Defining Metric Thresholds for Software Product Lines: A Comparative Study

Gustavo Vale[1], Danyllo Albuquerque[2], Eduardo Figueiredo[1], Alessandro Garcia[2]

[1]Software Engineering Laboratory (Labsoft) – Department of Computer Science
Federal University of Minas Gerais (UFMG) – Belo Horizonte – MG – Brazil
[2]OPUS Research Group – Software Engineering Lab – Informatics Department
Pontifical Catholic University of Rio de Janeiro (PUC-Rio) – Rio de Janeiro – RJ – Brazil
{gustavovale, figueiredo}@dcc.ufmg.br; {dwalbuquerque, afgarcia}@inf.puc-rio.br

ABSTRACT

A software product line (SPL) is a set of software systems that share a common and variable set of features. Software metrics provide basic means to quantify several modularity aspects of SPLs. 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 feature implementation. There are several methods to derive thresholds for software metrics. However, there is little understanding about their appropriateness for the SPL context. This paper aims at comparing three methods to derive thresholds based on a benchmark of 33 SPLs. We assess to what extent these methods derive appropriate values for four metrics used in product-line engineering. These thresholds were used for guiding the identification of a typical anomaly found in features' implementation, named God Class. We also discuss the lessons learned on using such methods to derive thresholds for SPLs.

Categories and Subject Descriptors

D.2.3 [Coding Tools and Techniques]: Object-Oriented Programming, D.2.8 [Metrics]: Complexity Measures.

General Terms

Measurement, Design, Experimentation, Verification.

Keywords

Metrics; Thresholds; Software Product Lines.

1. INTRODUCTION

With software-intensive systems growing in size and complexity, better support is required for measuring and controlling the software quality [24]. These tasks become harder in software projects constituted of several varying features, such as Software Product Lines (SPLs) [28]. SPLs are been increasingly adopted in software industry to support coarse-grained reuse of software assets [13]. Each SPL is a configurable set of systems that shares a common, managed set of features in a particular market segment [39]. Features can be defined as modules with consistent, well-defined, independent, and combinable functions [4].

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA
 ISBN 978-1-4503-3613-0/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2791060.2791078

However, the success of an SPL requires the effective measurement of several quality aspects of all its comprising features. Quality aspects can be measured, for example in terms of maintainability and changeability. Software metrics are the pragmatic means for assessing these quality aspects [8][16]. Certain metric values can help to reveal specific design flaws of a SPL that should be closely monitored [13]. For instance, such measures can be used to indicate whether a critical anomaly (or smell) is affecting a feature structure. Then, developers may suspect that something is wrong in the feature implementation. Typical examples include cases of large classes indicating they are realizing non-cohesive features of the SPL [1][25].

Nevertheless, as in every software project, the effectiveness of SPL measurement is directly dependent on the definition of appropriate thresholds. Thresholds allow to objectively characterize or classify each component according to one of the quality metrics. The definition of appropriate thresholds needs to be tailored to each metric. Thus, as there are several features and metrics being analyzed in the context of a SPL, it is not possible to derive thresholds in an *ad hoc* fashion [3][35]. Several general methods [3][9][14][18][23][34][35][40][42] have been proposed to derive metric thresholds. The use of appropriate methods is required in the context of SPLs in order to identify which features need closer attention. Unfortunately, there is limited empirical knowledge about the appropriateness of such threshold derivation methods in the context of SPLs.

In particular, some of these methods [3][18][35] consider the *skewed distribution* of software measurements. For instance, 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. Similarly, Ferreira et al. [18] 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. Additionally, Oliveira et al. [35] 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 [35].

This paper aims at comparing methods to derive thresholds in the context of SPLs. Given the heterogeneity of metrics in SPL assessment, we focus on methods that also consider the skewed distribution of software metrics. We designed and performed a comparative study aiming to investigate in which context each

method succeeds. This study involved four main steps. First, we built a benchmark of SPLs to explore the characteristics of each analyzed method. We then chose four metrics, which are present in a detection strategy to identify God Class code smell [1][2]; most of these metrics are typically used in software measurement process [38]. After, the number of systems was re-grouped, based on their size, to compose two additional benchmarks. The methods were used to derive thresholds for the four metrics in each of the three benchmarks. In the final step, we verified if the derived thresholds were appropriated, for example to God Class detection.

We built the three different benchmarks of SPLs developed with Feature-Oriented Programming (FOP) [6]. The four chosen metrics were applied in SPLs implemented with FOP techniques to collect the value of these metrics for each entity of each SPL. In this work, entities are AHEAD [5] and FeatureHouse [4] code files, such as constants and refinements. With these values, we run the three methods and derived thresholds for each benchmark. We performed a threefold analysis of threshold obtained by (i) analyzing the appropriateness of the thresholds individually, (ii) using those thresholds to support the detection of God Class, and (iii) reporting the strengths and weaknesses of each method by discussing our lessons learned.

As results, Alves' method was better at individual analysis, because it presented more representative thresholds given the inputs for three out of four metrics. In the evaluation of effectiveness and comprehensiveness, Ferreira' and Oliveira's methods achieved better precision and recall, respectively. After all comparison and analyzes, eight desirable points in methods to derive thresholds were described by following our experience in applying the methods.

The remainder of this paper is organized as follows. Section 2 sets the context of this study by discussing some related work. Section 3 reports the study settings. Section 4 describes and presents the study analysis and evaluation realized. Section 5 presents the lessons learned. Section 6 discusses limitations and threats to validity of this study. Section 7 concludes and presents suggestions for future work.

2. RELATED WORK

In this section, we review previous attempts to define values of 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 [7]. 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 [7]. The literature review of this work followed similar steps to the protocol of a Systematic Literature Review (SLR) [26]. 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 [26]. After applying the review protocol, we selected 50 primary studies in order to extract information related to methods to derive thresholds. The primary studies can be found on the project website [42]. Some of these primary studies are discussed in this section. The three selected methods explored in this study are described next (Section 3.1).

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

Thresholds derived from programming experience – Many authors defined values of metric thresholds according to their programming experience. For example, for the *McCabe* metric, the value 10 was defined as the threshold [33], and for the *NPATH* metric, the value 200 was defined as threshold [34]. McCabe counts the number of linearly independent paths through a program's source code and NPATH computes the number of possible execution paths through a function. The aforementioned values are used to indicate the presence or absence of code smells. Regarding *Maintainability Index* (MI), previous work [9] defined the threshold value 65 and 85. When MI values are higher than 85, the modules are considered as highly-maintainable. In the case that MI values are between 85 and 65 they are considered as moderately-maintainable, and when MI values are smaller than 65 they are considered as difficult to maintain. These thresholds rely on programming experience and it is difficult to reproduce or generalize these results. Thus, unlike the works cited above [9][33][34], our research aims to compare the methods to derive metric thresholds not based on programming experience in the context of SPLs.

Thresholds derived from measurement analysis - Erni et al. [14] 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 [27] use a similar strategy in their research for 45 Java projects and 37 C++ projects. Nevertheless, they use an additional label, very high. The very high label is calculated as $T = (\mu + \sigma) \times 1.5$. These methodologies were common statistical techniques to derive thresholds few years ago. However, Erni et al. [14] and Lanza and Marinescu [27] do not analyze the underlying distribution. The problem with these methodologies is that the assumption of metrics that are normally distributed is not justified, invalidating the use of these methodologies. Several studies [3][10][18][29][35] clearly demonstrate that most software metrics follow other distribution, such as *heavy-tailed distribution*. In contrast, our research focuses on methods that do not make assumption about data normality.

French [23] also proposes a methodology 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 or outliers. For metrics with high range or high variation, this technique identifies a smaller percentage of observations than its theoretical maximum. In contrast, our comparative study was designed to derive thresholds from benchmark data and, as such, it is resilient to high variation of outliers. While French applies the technique to Ada95 and C++ systems, we apply methods to derive thresholds in FOP-based SPLs.

Methodologies for characterizing metric distributions – Chidamber and Kemerer [8] use histograms to characterize and

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

analyze data. For each of their 6 metrics (i.e., WMC, DIT, NOC, CBO, RFC, and LCOM), they plotted histograms per programming language to discuss metric distribution and spot outliers in C++ and Smaltalk systems. Spinellis [40] 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, 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. [42] 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.

Several studies [3][10][18][29][35] clearly demonstrate that most software metrics follow *heavy-tailed distribution*. For instance, Concas et al. [10] show that for a large Smalltalk system most of the Chidamber and Kemerer's metrics [8] follow heavy-tailed distribution. Louridas et al. [29] show dependencies of different software artifacts also follow heavy-tailed distribution. These affirmations invalidated the use of any statistical method that uses average 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. On the other hand, our research is focused on analyzing methods to derive thresholds which somehow use information on metrics distribution to the identification of code smells in SPL.

3. STUDY SETTINGS

This section presents the configuration of this comparative study. Section 3.1 describes the methods to derive thresholds compared in this study. Section 3.2 presents a set of four software metrics for which we have applied the methods. Section 3.3 explains how we built three SPL benchmarks, which are used in this study. Section 3.4 explains how the oracle was generated.

3.1 Methods to Derive Metric Thresholds

This section describes the three methods to derive thresholds that are compared in this study. We decided to reduce the scope by focusing only on methods that explicitly consider that software metrics usually do not follow a normal (or symmetric) distribution [3]. Our goal is to select methods which somehow use information on metrics distribution for calculating thresholds. These methods are ordered by publication year and described below.

Alves's Method – The method proposed by Alves and his colleagues in 2010 [3] is divided into six steps: (1) measurement extraction, (2) weight ratio calculation, (3) entity aggregation, (4) system aggregation, (5) weight ratio aggregation, and (6) thresholds derivation. In this method, the metric values are collected for each system (SPL in our case) – each system values should be in a different file (step 1). It then computes the weight percentage of lines of code (LOC) within each system entity; in other words, it is necessary to know the total LOC of the target system. The LOC of each entity should be divided by the total LOC of the target system and, then multiplied by 100. This need to be done for each system (step 2). Equal measures of each system are then grouped by adding up the percentage. This may be done for each system (step 3). The obtained values are grouped in the same file and are divided for the number of systems which compose the benchmark. In other words, all data should be placed in a same spreadsheet, for example. Then, the percentage column should be divided by the number of systems – observes that if the data in that column were added the result should be 100 – (step 4). Equal measures of this file are also grouped and the percentage calculated, like step 3 (step 5). Finally, the percentage is defined and, the thresholds can be extracted. 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%). For example, if it is required values of high label it is necessary to add the percentages until get 80%, the upper metric value is the threshold.

Ferreira's Method – This method was proposed by Ferreira and her colleagues in 2011 [18]. It is relatively simple and can be divided in 4 steps: (1) measurement extraction, (2) grouping metrics, (3) boxplot representation, and (4) threshold derivation. The metric values are first collected for each system (step 1) and grouped into a unique file (step 2). Using manual analysis or with a supporting tool, three boxplots should be created (step 3). These boxplots represent values with a high, medium, and low frequency in the systems which are classified as *good*, *regular*, and *bad* values, respectively. Hence, each label represents an interval (step 4). The reasoning is that the lower the frequency, the far from the common metric value. In the method description, it is not clear how to extract the three boxplots.

Oliveira's Method – This method was proposed by Oliveira and her colleagues in 2014 [35] it relies on a formula for calculating thresholds. This formula is called *Compliance Rate* and can be expressed as follows: *p% of the entities should have* $M \leq k$, where M is a given source code software metric calculated for a given software entity (e.g., features or classes), k is the upper limit of M, and p is the minimal percentage of entities that should follow this upper limit k. Therefore, this relative threshold tolerates (100 - p)% of classes with $M > k$. The values of p and k are based in two constants, *Min* and *Tail*. These constants are used to drive the method towards providing some quality confidence to the derived thresholds. More specifically, these constants are used to convey the notions of real and idealized design rules, respectively. The values of these two constants are in a range between 0 and 100. This method also relies on three additional formulas beyond the *Compliance Rate*. Two of these formulas involve penalties for the *Min* and *Tail* constants, respectively. The third formula sums up these penalties. The combined pair of p and k with minor penalty is chosen to compose the *Compliance Rate*. In case of ties, it chooses the pair with highest p and then the one with the lowest k.

3.2 Selected Software Metrics

This section briefly describes four software metrics used in this study, namely *Lines of Code* (LoC) [16], *Coupling between Objects classes* (CBO) [8], *Weighted Method per Class* (WMC) [8], and *Number of Constant Refinements* (NCR) [1]. These metrics were chosen because they are part of a detection strategy [1], they can be collected automatically by VSD tool [2], and previous studies [1][2][38] relied on them to detect design problems in SPLs. They capture different attributes of a SPL design, i.e. size, coupling, complexity, and refinement [1]. For all four metrics described below, classes with high values are more likely to be worse in the SPL quality. In other words, higher values might be an indicative of a smell affecting the SPL design.

LOC [16] counts the number of uncommented lines of code per class. The value of this metric indicates the size of a class. CBO

[8] counts the number of classes called by a given class. This metric measures the degree of coupling among classes. *Weight* WMC [8] weights the classes by the number of methods present in them. This metric can be used to estimate the complexity of a class. 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) [6]. That is, refinements can change the behavior of a constant if a certain feature is included in a product. This work considers constants and refinements as entities for all metrics, except NCR. For this metric, the number of entities is lower than the evaluated by other metrics because only constants are considered as entities.

3.3 Software Product Line Benchmarks

This section presents three benchmarks of Software Product Lines (SPLs). To build these benchmarks, we focus on SPLs developed using FOP [6]. 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. It is very difficult find compositional feature-oriented SPLs, this benchmark can be important to SPL community.

We selected 47 SPLs from repositories, such as SPL2go [41] and FeatureIDE examples [15], 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 [15] has 4 different versions implemented in AHEAD, FH-C#, FH-Java and FH-JML. FH stands for FeatureHouse [4] 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 1.The step-to-step filtering is further explained on the project website [42].

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 1 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.

3.4 Oracle of Code Smells

This section introduces the main steps required to produce the oracle of code smells that guides our evaluation. The oracle can be understood as the reference model of the actual smells found in a SPL. The reference model is used for evaluating the methods to derive thresholds (Section 3.1). In particular, the oracle is the basis for determining whether the derived thresholds (computed by each method) are effective on the identification of code smells in a SPL. In order to compose the oracle, we (i) choose the target SPL to be assessed, (ii) define the code smell investigated in the study, (iii) select the detection strategy used, and (iv) validate the results with software experts.

Table 1. Software Product Lines Benchmarks

	Id	SPL	Tech.	LOC
Benchmarks 1, 2, and 3	1	BerkeleyDB [41]	FH-Java	37247
	2	AHEAD-Java [1]	AHEAD	16719
	3	AHEAD-guidsl [1]	AHEAD	8738
	4	TankWar [15], [41]	AHEAD	4670
	5	AHEAD-Bali [1]	AHEAD	3988
	6	Devolution [15]	AHEAD	3913
	7	MobileMedia v.7 [19]	AHEAD	2691
	8	WebStore v.6 [19]	AHEAD	2082
	9	DesktopSearcher [15], [41]	AHEAD	1858
	10	GPL [15]	AHEAD	1824
	11	Notepad v.2 [41]	FH-Java	1667
	12	Vistex [41]	FH-Java	1480
	13	GameOfLife [41]	FH-Java	1047
	14	Prop4J [[41]	FH-Java	1047
Benchmarks 1 and 2	15	Elevator [41]	FH-Java	728
	16	ExamDB [41]	FH-JML	568
	17	PokerSPL [41]	FH-JML	461
	18	EmailSystem [41]	FH-Java	460
	19	GPLscratch [41]	FH-JML	405
	20	Digraph [41]	FH-JML	374
	21	MinePump [41]	FH-JML	367
	22	Paycard [41]	FH-JML	319
Benchmark 1	23	IntegerSet [41]	FH-JML	225
	24	UnionFind [41]	FH-JML	194
	25	NumberContractOverrinding [41]	FH-JML	165
	26	NumberConsecutiveContractRef [41]	FH-JML	148
	27	NumberExplicitContractRef [41]	FH-JML	143
	28	BankAccount [41]	FH-JML	122
	29	EPL [15]	AHEAD	98
	30	IntList [41]	FH-JML	94
	31	StringMatcher [41]	FH-JML	45
	32	Stack [41]	FH-Java	22
	33	HelloWorld [15]	AHEAD	22

Choosing the Target SPL - We choose an SPL, called MobileMedia [20], which is a SPL for manipulating photos, music, and videos on mobile devices [20]. It is an open source SPL implemented in several programming languages, such as Java, AspectJ, and AHEAD. We selected MobileMedia version 7 - AHEAD implementation [19]. This SPL was chosen because: (i) it was successfully used in other previous empirical studies [11][17][19][20][30][37], (ii) it is present on the three benchmarks of this study, and (iii) we have access to its software developers and maintainers.

Defining the Code Smell – We choose the God Class smell [38][22] to be investigated in the context of this study. God Class is defined as a class that knows or does too much in the software system [38]. In particular, the occurrences of this smell in MobileMedia were indicators of two design problems: (i) large classes indicating they were realizing non-cohesive features of the SPL [25], and (ii) complex interfaces of components exposing unrelated features. In addition, we should mention that God Class is a strong indicator that a software feature is accumulating the implementation of many other ones (captured by NCR metric).

Selecting the Detection Strategy - A detection strategy is a logical condition, based on metrics and threshold values, which is used as mean for indicating the presence of code smells [31]. In this work, we decided to use the detection strategy below for the following reasons. First, it has been evaluated in other studies and presented good results for the detection of God Class [1][2]. Second, this detection strategy uses a straightforward way for

identifying instances of God Class using 4 different metrics which are presented in our study (Section 3.2). Third, this detection strategy is automated by VSD tool. We also believe that this strategy is better than traditional ones [27][31] because it was adapted for SPL by using NCR (a FOP-specific metric). This metric is able to fit complexity properties of SPLs that traditional metrics cannot fit. The proposed detection strategy for God Class can be formulated as follows [1]:

if ((LOC> threshold) and (CBO> threshold) and ((WMC/LOC)> threshold)) or (NCR> threshold) then Class is a God Class

Validating the Oracle – In order to provide a reliable oracle, we analyzed the source code and identified some God Class instances. This preliminary oracle has been validated by experts (maintainers and developers of MobileMedia SPL), and the final version of the oracle was produced as a joint decision. The final oracle includes the following seven instances of God Class: *MainUIMidlet (Base), MediaAccessor (Base), MediaController (MediaManagement), MediaListController (MediaManagement), MediaListScreen (MediaManagement), AlbumData (AlbumManagement),* and *SmsMessaging (SMSTransfer)*. The first word refers to constant or refinement and the word in parenthesis is the name of feature which this constant or refinement implements.

4. EVALUATION

We evaluate different aspects of the methods used to derive thresholds for SPLs. Before we present the derived thresholds we explore two varying characteristics of the studied methods for the four selected metrics: (i) correlation with LOC (Section 4.1), and (ii) distribution of software metrics (Section 4.2). Our goal is to reveal whether these varying characteristics influence the derived thresholds and can provide support to discuss some variations on the derived thresholds. Then, Section 4.3 presents the threshold values derived by the three methods. Section 4.4 investigates whether the derived thresholds contribute to the recall and precision measures in the identification of a specific smell.

4.1 Correlation with LOC

Alves's method assumes that all software metrics correlate with LOC. In order to verify if this is true, we use the *Pearson's correlation coefficient* . Pearson's correlation is +1 in the case of a perfect direct linear correlation, -1 in the case of a perfect decreasing linear correlation, and some values around 0 implies that there is no linear correlation between the variables [12]. We apply this coefficient to identify the correlation of LOC with the other selected metrics (CBO, WMC, and NCR).

Table 2 shows the coefficient of correlation of LOC and other metrics for the three benchmarks. It can be observed that for all benchmarks the metrics CBO and WMC have high correlation (values above 0.75) with LOC. However, NCR has low correlation with values closer to 0.3. A metric has correlation 1 with itself (case of LOC with LOC) and, therefore, this correlation was not presented in Table 2. The goal of investigating the correlation of the selected metrics with LOC is to investigate if this correlation impacts on the calculated thresholds.

Table 2. Correlation of metrics with LOC

Benchmark	Metrics		
	CBO	WMC	NCR
1	0.751621	0.976406	0.28995
2	0.753825	0.97711	0.295099
3	0.757247	0.97896	0.292746

4.2 Distribution of Software Metrics

All three selected methods to derive thresholds (Section 3.1) claim to take the distribution of metrics into account. Therefore, this section analyzes the distribution of each software metric (Section 3.2) based on the SPL benchmarks (Section 3.3).

According to the classification schema suggested by Foss et al. [21], the metric has *heavy-tailed distribution* when the best distribution of a particular measure is one of the following: *Weibull, Lognormal, Cauchy, Pareto,* or *Exponential*. We decided to use Weibull distribution because its versatility and relative simplicity. This distribution has two main probability distribution functions: (i) *probability density function* (pdf), f(x), which expresses the probability the random variable takes a value x, and (ii) *cumulative distribution function* (cdf), F(x), which expresses the probability the random variable takes a value less than or equal to x [32]. These functions have parameters α and β, defined by equations (Eq1) and (Eq2):

$$(\mathbf{Eq1})\ fw(x) = P(X = x) = \frac{\alpha}{\beta}\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-(x/\beta)^{\alpha}},\ \ \alpha > 0, \beta > 0$$

$$(\mathbf{Eq2})\ Fw(x) = P(X \leq x) = 1 - e^{-(x/\beta)^{\alpha}},\ \ \alpha > 0, \beta > 0$$

The parameter β is called by *scale parameter*. Increasing the value of β has the effect of decreasing the height of the curve and stretching it. The parameter α is called by *shape parameter*. If the shape parameter is less than 1, Weibull is a heavy-tailed distribution [32]. A heavy-tailed distribution means that a small number of entities has high values and a large number of cases has low values. In this distribution, the mean is not representative [3][18][32].

Table 3 presents the values of α and β, for each metric and benchmark. For example, LOC has values of α = 0,95201 and β = 26,858 for the benchmark 1. Based on the analysis of the parameter α, we can observe that the metrics LOC, WMC and NCR follow a heavy-tailed distribution. According to Table 3, CBO does not follow a heavy-tailed distribution for FOP-based SPL implementation because it presents α values higher than 1. By analyzing the parameters values, we can be more confident about the metric distribution because the plotted graph, in some cases (e.g., depending of the scale) may give the wrong impression that the metric follows a heavy-tailed distribution.

Table 3. Weibull Values for Each Metric per Benchmark

Metric	Benchmark	α	β
LOC	1	0.95201	26.858
	2	0.72892	28.37
	3	0.93964	27.232
CBO	1	1.1253	5.2798
	2	1.1693	5.4341
	3	1.2244	5.6196
WMC	1	0.7277	6.5979
	2	0.72748	6.6063
	3	0.72041	6.5979
NCR	1	0.98339	3.9359
	2	0.97109	3.9679
	3	0.96919	4.0734

4.3 Derived Thresholds

This section presents the derived thresholds that were obtained using the three methods to derive thresholds (Section 3.1) according to each benchmark (Section 3.3). The process was performed with the four metrics used in this study. Only the key values of each method are presented. For example, Alves's

method presents four labels, but these labels are established in three percentages. Hence, just the values that represent the percentages are shown. This presentation strategy is also applied to Ferreira's method, because the range of values of boxplot 2 is equals to the ranges of boxplots 1 and 3. In addition, although Ferreira's method definition does not provide how to extract the three boxplots, we extracted by our own knowledge (better explained on Section 6). The three box plot has 70% (*good*), 20% (*regular*) and, 10% (*bad*) of data, respectively.

The obtained values from the methods are presented by Tables 4, 5, and 6. We present in separate tables because each method has a different output. These tables should be read as follows: the first column represents the benchmarks, and the second column indicates the different labels in the case of Alves's and Ferreira's methods. The other columns determine the thresholds of LOC, CBO, WMC, and NCR, respectively. For example, Alves's method labels are defined as: *low* (0-70%), *moderate* (70-80%), *high* (80-90%) and *very high* (90-100%) that are represented by the intervals 0-8; 9-12; 13-20 and >21, respectively for CBO in benchmark 1.

Table 4. Threshold Values from Alves's 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

Table 5. Threshold Values from Ferreira's Method

Benchmark	Boxplot	LOC	CBO	WMC	NCR
1	1	27	5	6	0
	3	77	11	17	3
2	1	27	6	6	0
	3	78	11	17	3
3	1	27	6	6	0
	3	79	12	18	3

Table 6. Threshold Values from Oliveira's Method

Benchmark	LOC	CBO	WMC	NCR
1	91	6	11	1
2	86	9	14	2
3	78	13	21	2

It should be observed that there is a difference between the thresholds from the same method (varying the benchmark) and between methods. The variation was sharper in the case of Alves's method. This variation happens because the smallest (22 LOC) and the largest (37,247 LOC) SPLs have the same weight (step 4 of this method description). Hence, a higher variation across benchmark was observed in the case of metrics with high correlation with size. We did not have a high variation in the case of NCR, which has low correlation with size. The other methods do not correlate metrics and do not weight metrics by the number of systems. Then, in almost all cases, the thresholds remained the same or have a slight growth. A peculiar case occurred in Oliveira's method (Table 6), in which LOC had a small decrease. This decrease is probably impacted by the penalties applied to derive metric thresholds; we did not expect for it.

In order to have an objective discussion, we focus now on the analysis of the high values (considered). For Alves's method, it is clear that high values are upper than 80%; for the Ferreira's method, we have considered values of the third boxplot (*bad label*); and, for Oliveira's method, we use the returned values. Figure 1 presents the thresholds for LOC, CBO, WMC, and NCR. In addition, each letter presents the difference of each method for the three benchmarks. For example, Figure 1(a) refers to LOC and, for benchmark 1 the thresholds for methods of Alves, Ferreira, and Oliveira are 151, 77, and 91.

Figure 1 indicates Alves's method returned higher values for three of the four metrics. These three metrics (LOC, CBO, and WMC) have high correlation with LOC metric. In contrast, NCR has low correlation with LOC and it has the smallest values in two out of three cases (benchmarks 2 and 3) as presented in Figure 1(d). NCR values vary from 0 to 28, however, approximately 90% of the entities has NCR lower than 4. This phenomenon can be an evidence for the low thresholds derived by three methods.

Analyzing the derived thresholds and considering the correlation of the metrics with LOC (Section 4.1), it is possible to see that Alves's method has undergone a major change in its behavior, probably, because this method uses this correlation to calculate thresholds. Related to Ferreira' and Oliveira's methods, it is not possible to percept a clear change in its behavior.

By looking at the derived thresholds and considering the software metrics distribution (Section 4.2), we did not observe a big difference on the behavior of the three methods to calculate thresholds. For example, even though CBO does not have a heavy-tailed distribution (Section 4.2), the threshold values were not too different between the methods. As far as the values of the other metrics are concerned, the behavior was close in different methods. We observed some variation in threshold values of all metrics. For example, the largest class of our benchmarks has 1,686 LOC and the thresholds of this metric derived by methods of Alves, Ferreira, and Oliveira were 293, 79, and 78, in this order to benchmark 3. Similarly, CBO, in the same case, ranges from 0 to 68 and the derived thresholds of this metric was 22, 12 and 13. Based on these two examples (in the context of benchmark 3) the thresholds seem to be relatively low. Due to this assumption, the number of outliers tends to be large.

The threshold 6 to CBO, defined by Oliveira's method for the benchmark 1 has 639 outliers out of 2,700 entities. This number of outliers represents 23.67% of the entities. We consider that this percentage is not reasonable, as we do not want a very large number of outliers. Based on this, we can conclude that: (i) this calculation is extremely dependent from the benchmark quality to the three methods and (ii) Oliveira's method is more rigid to define thresholds than the others. Considering these two points, we believe Alves's method fared better with the majority of the analyzed metrics, especially with the one highly correlated with LOC.

It is clear that some metrics are correlated to others and, this is natural. However, we considered that correlating metrics to calculate thresholds may not be a good practice, when this correlation is low. Since it is not always easy to know if a metric correlates with other metrics; it is better to not correlate metrics to derive thresholds. In the case of the distribution of the software metrics, the three methods seem to work well because with heavy-tail distribution. Even if the distribution is not heavy-tailed, the three methods presented similar derived thresholds.

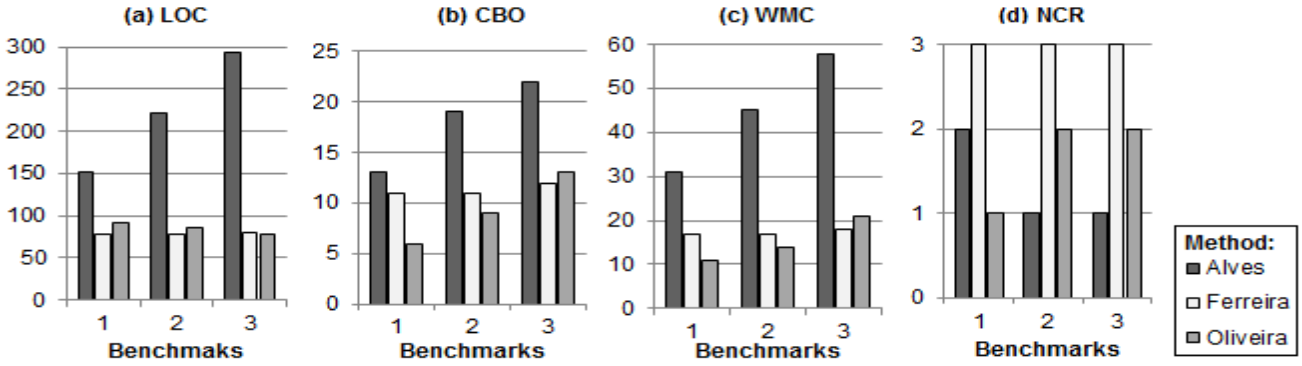

Figure 1. Metric Thresholds Side by Side

4.4 Identification of Code Smells

This section presents a comparative analysis of code smell identification using threshold values obtained from three methods (Section 3.1). It is important to mention that each method was applied using the same detection strategy (Section 3.4) for God Class. Table 7 describes the results per method, summarizing the detected code smells (DCS), true positives (TP), false positives (FP), and false negatives (FN). TP and FP quantify the number of correctly and wrongly identified God Classes by the detection strategy. FN, on the other hand, quantifies the number of God Classes that the detection strategy missed out. Additionally, the thresholds were derived for the benchmarks 1, 2, and 3. For instance, by using the thresholds derived by Alves's method in benchmarks 1, 2, and 3 the number of DCS (i.e., God Class candidates) was 8, 13, and 13, respectively. In all three cases, 5 correct God Class instances (TP) were found.

Table 7. Identification of God Classes Based on Derived Thresholds from Each Method

#	Alves			Ferreira			Oliveira		
	Benchmark			Benchmark			Benchmark		
	1	2	3	1	2	3	1	2	3
DCS	8	13	13	7	7	7	15	11	8
TP	5	5	5	6	6	6	7	7	5
FP	3	8	8	1	1	1	8	4	3
FN	2	2	2	1	1	1	0	0	2

Aiming to provide an additional perspective of the effectiveness on identification of code smells, we also analyzed precision and recall measures. Precision (P) quantifies the rate of TP by the number of *detected code anomalies* (TP + FP). Recall (R) quantifies the rate of TP by the number of e*xisting code anomalies* (TP + FN). Precision and recall are represented by Equations 3 and 4, respectively.

$$(\mathbf{Eq3})\ Precision = \frac{TP}{TP + FP} \qquad (\mathbf{Eq4})\ Recall = \frac{TP}{TP + FN}$$

Table 8 presents recall and precision of the detection strategy applied to MobileMedia using the thresholds derived by the three methods considered in this study. We can observe that recall is always higher (or equal) than precision in all cases. In addition, recall is considered more useful than precision in the context of code smells usually [37] as 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.

Table 8. Recall and Precision Based on Derived Thresholds from Each Method

Benchmark	Alves		Ferreira		Oliveira	
	P	R	P	R	P	R
1	62,5	71,4	85,7	85,7	46,6	100
2	38,5	71,4	85,7	85,7	63,6	100
3	38,5	71,4	85,7	85,7	62,5	71,4

In this evaluation, Oliveira's method achieved the best performance in terms of recall measure. This is due to the fact that the thresholds values from Oliveira's method are the lowest ones in a half of metrics considered in the detection strategy (see Figure 1). Consequently, the detection strategy tends to identify a larger number of instances of God Class. Similarly, Alves's method achieved the worst performance in terms of recall because this method has the highest threshold values for 3 out of 4 metrics used in the detection strategy (Figure 1). In Section 4.3, we presented a more detailed discussion of the threshold values that were obtained according to each method to derive metric thresholds. It is important to mention that the thresholds used in a detection strategy directly impact on DCS obtained by each method. Accordingly, recall and precision values are also impacted by the threshold values used in the detection strategy.

Regarding precision values, we observed Ferreira's method has good rates in the three benchmarks. This is due to the fact that the NCR thresholds values from Ferreira's method are high in the three benchmarks (Figure 1(d)). Similarly, Alves's method achieved the worst performance in terms of precision because this method has the lowest NCR thresholds values (Figure 1(d)). Then, we believe that NCR thresholds were responsible by low precision for all methods because this metric is used in an OR expression in the detection strategy. In other words, NCR has too much importance in the God Class detection strategy. A solution to minimize this problem would be the use higher thresholds to NCR (for example, very higher label). However, in an attempt to be fair to all methods, we use only the values considered high. This strategy may have benefited the Ferreira's method and/or hindered Alves's method.

5. LESSONS LEARNED

This section discusses characteristics of each method for threshold derivation based on the lessons we learned during this study. Regarding the use of the methods, we enumerate seven questions for discussion: (i) Is it a deterministic method? (ii) Are step-wise outliers identified? (iii) Does the number of systems impact on the

derived thresholds? (iv) Does the number of entities impact on the derived thresholds? (v) Is some metric correlated with another? (vi) Are lower bound thresholds calculated? (vii) Does the method have tool support? It is important to mention that each method has different strengths and weaknesses. This does not necessarily mean that a method is better (or worst) than the others. Table 9 shows the answer for each question in which are discussed on the five topics bellow, based both on the characteristics of the proposed method and on our empirical experience by applying the methods in this study.

Deterministic - If we replicate this study, are the results going to be exactly the same? Alves's method is deterministic, but the chosen percentage can vary. Therefore, we considered it as partially deterministic. In the case of Ferreira's method, it does not describe how the three boxplots should be extracted and, so, this makes it not deterministic. Oliveira's method is highly deterministic. If someone uses the same input, the same results are obtained.

Step-wise Outliers - Outliers deeply impact on the definition and use of thresholds. In addition, developers may want to be aware of metrics with outlier values. Therefore, it is desirable that the methods can identify and handle the outliers (i.e., highest and lowest values) associated with a given metric. The Alves' and Ferreira's methods describe labels to identify outliers; *low*, *moderate*, *high*, and *very high* in the Alves's method case and *good*, *regular*, and *bad* in Ferreira's method. Oliveira's method has only returned one value for each metric. Moreover, detection strategies can be interest to consider different labels for different metrics [27]. For example, if one metric has too much importance in the detection strategy, the *very high* label can be used instead of *high* label. This action can minimize the number of false positives.

Number of Systems and Entities - Thresholds are extracted from software entities (e.g., features and classes). Therefore, the main information used for calculating thresholds is expected to be metrics collected from these entities instead of the number of systems, for example. Although the number of systems can be considered important in terms of representativeness, we believe that the number of entities is more important. Hence, a method is expected to derive thresholds weakly dependent on the number of systems and strongly dependent on the number of entities. Alves's method calculates thresholds by using, essentially, the number of systems. Both Ferreira's and Oliveira's methods use the number of entities to derive thresholds. However, Oliveira's method uses the median of entities of the systems. Therefore, Alves's and Oliveira's methods can be considered as strongly dependent to the number of systems and Ferreira's method as strongly dependent to the number of entities.

Correlation of metrics - Alves's method uses LOC to weight other metrics and generate percentages. Due to high correlation of LOC and many other software metrics, the reasoning applied in Alves's method is usually useful. However, it fails when the metric (e.g., NCR) does not correlate with LOC (Section 4.1). The method does not make explicit whether or not to weight measurements by LOC. The other two methods do not consider the correlation of metrics with LOC. As explained in the end of Section 4.3, we believe it is better to not correlate metrics to calculate thresholds.

Lower bound thresholds and tools support - Thresholds are often used to filter upper bound outliers. However, in some cases, it may make sense to identify lower bound outliers. For instance, classes with low values of LOC can be an indicative of the Lazy Class code smell [22]. A Lazy Class is defined as a class that knows or does too little in the software system [22]. None of analyzed methods calculate lower bound thresholds. Tool support is not essential, but it can facilitate the use of a method because it easier their systematic application. Among the analyzed methods, we only found a tool to support the Oliveira's one [36].

6. 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 larger number of SPLs. This limitation has implication in the amount of analyzed entities, which is particularly relevant to NCR. This factor can influence the defined thresholds as the number of entities for NCR analysis is further reduced. Therefore, in order to mitigate this limitation, we created different benchmarks for comparison of the derived thresholds.

Metric Distribution – In this work we identified that LOC, WMC and NCR has heavy-tailed distribution and CBO does not have. Using a different benchmark (e.g., composed by other programming technologies) the distribution of these metrics may be different. The detection strategy chosen in this work might have influenced the results. For example, as NCR has greater influence than other metrics, it may be interesting to define higher thresholds (very high instead of high) for this metric. Nonetheless, as not all methods have labels, we decided to use a default label (that we considered high) for all metrics.

Measurement Process – The SPL measurement process in our study was automated based on the use of existing tooling support. However, as far as we are concerned, there is 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 automated tests with a few SPLs. In fact, we observed all software proprieties were preserved after the transformation.

Table 9. Comparative Evaluation of the Method for Calculating Thresholds

Method	Is it deterministic?	Are step-wise outliers identified?	Does the number of systems impact?	Does the number of entities impact?	Does it correlate with other metrics?	Lower bound thresholds?	Tool support?
Alves	Partially	Yes	Strong	Weak	Yes	No	No
Ferreira	No	Yes	Weak	Strong	No	No	No
Oliveira	Yes	No	Strong	Weak	No	No	Yes

Metric Labels – The non-systematization of Ferreira's method to extract the three boxplots required us to define three boxplots with approximately 70% (*good*), 20% (*regular*) and, 10% (*bad*) of data, respectively, totalizing the 100%. Like other methods are full or partially deterministic, we did not have this problem with them.

Selected code smell: We focused the identification of code smells on discussing only one type of code smell (i.e., God Classes). Fowler [22] has cataloged a list of 22 code smells. Therefore, the God Class smell used to evaluate the effectiveness of threshold calculation methods may not necessarily be a representative sample of code smells found in certain SPL. However, the God Class was chosen because it is a well-known code smell in the context of SPL [38]. In addition, God Classes have been frequently used to indicate problems in the structure of SPL components, such as *Complex Interfaces* and *Overloaded Components* [25]. Related to God Class strategy detection, we choose this one [1] because it aims to identify this type of anomaly in the context of SPL and it is automated by VSD tool. Nevertheless, we assume that results are limited using only this smell in the comparison. Maybe using other smells the results can be different.

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, VSD tool [2] was used both to collect the metric values and to identify God Class instances. 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 very high complexity, may result in higher threshold values [3]. As previously stated, for deriving thresholds we removed all *useless code* (e.g., generated code and test cases) from our analysis.

Oracle Generation – An oracle had to be created in order to calculating recall and precision measures. Several precautions were taken. In spite of that, we can have omitted some God Class or chosen a God Class 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.

7. CONCLUSION AND FUTURE WORKS

This paper describes the importance of software metrics; the use of a set of metrics used together to measure some quality attribute; and, the calculation of representative thresholds. In order to focus in the last point, three methods to derive metric thresholds were described and compared using as input data metrics collected from SPLs. We believe that the methods were reasonable evaluated because we provide as comparison with: (i) three different benchmarks; (ii) metrics with different distribution; (iii) metrics with different correlation with LOC; and, (iv) the derived thresholds were analyzed individually and using the same detection strategy.

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 (to exclude duplicates). In addition, we applied two refinements to extract the benchmarks 2 and 3. These two refinements consist in keeping only SPLs with more than 300 and 1,000 LOC, respectively. Providing the benchmarks as input for the methods was perceived that the Alves's method is a little more sensitive to the benchmark quality. It happened because this method weight by each SPL and SPLs of different sizes receives the same weight.

Regarding distribution of the metrics used in this study, the metrics LOC, WMC, and NCR have *heavy-tailed distribution* and CBO has a different distribution. In spite of that, CBO apparently did not present a different behavior of other metrics. Although we have a small sample, the methods seem to behave well with different distributions. Regarding metrics correlation, we noticed that Alves's method correlates metrics to derive thresholds. As can be seen in section 4.1, correlating metrics can be danger, when we have metrics with low correlation. The other methods did not correlate metrics, and hence this was not a problem for them.

In order to provide reliable outcomes, these outcomes were analyzed in two ways: (i) individually and (ii) identifying code smell. We considered that the Alves's method was better in the individually evaluation because it presented more representative thresholds given the inputs for three of four metrics. In addition, the thresholds are the higher related on the other methods. It results in a smaller number of outliers when related on the number of outliers detected by the other methods. Using the identification of code smell, Ferreira's method presented higher precision values. We trust that this result was strongly influenced by NCR threshold, because of the importance that this metric has on the detection strategy. Additionally, Oliveira's method presented higher recall values. This is due to the fact that the thresholds values from Oliveira's method in a half of metrics considered in the detection strategy are the lowest ones.

After all comparison and analyzes, it can be observed some desirable points in methods to derive thresholds. For instance, it is desirable that a method to derive metric thresholds should: (i) to be Systematic and deterministic; (ii) derive thresholds in a step-wise format; (iii) to be weakly dependent on the number of systems; (iv) to be strongly dependent on the number of entities; (v) Not correlate metrics; (vi) Calculate upper and lower thresholds; (vii) Provide representative thresholds independent of metric distribution and (viii) Provides Tool support. These desirable points will be explored in futures work. In addition, we plan to develop a tool to measure FH code to avoid the transformation from FH code to AHEAD code and avoid the risk of losing behavior or syntax of FH. Another proposed future work is exploring how to build representative benchmarks.

8. ACKNOWLEDGMENTS

This work was partially supported by CAPES, CNPq (grants 485907/2013-5, 483425/2013-3, and 309884/2012-8), FAPEMIG (grant PPM-00382-14), and FAPERJ.

9. REFERENCES

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

[2] Abilio, R., Vale, G., Oliveira, J., Figueiredo, E. and Costa, H. 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] Alves, T.L., Ypma, C. and Visser, J. Deriving Metric Thresholds From Benchmark Data. *In Proc. of 26th Int. Conf. on Software Maintenance (ICSM)*, pp. 1–10, 2010.

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

[5] Batory, D. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. *In Proc. of the Int. Conf. on Gen. and Transf. Tech. In Soft. Eng. (GTTSE)*, pp.3-35, 2005.

[6] Batory, D. Sarvela, J., Rauschmayer, A. Scaling step-wise refinement, *IEEE Trans. on Sof. Eng.*, 30 (6) pp. 355-371, 2004.

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

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

[9] Coleman, D., Lowther, B. and Oman, P. The Application of Software Maintainability Models in Industrial Software Systems. *Journal on System Software*, vol. 29, Issue 1, pp. 3–16, Feb., 1995.

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

[11] Conejero, J.M. *et al*. On the Relationship of Concern Metrics and Requirements Maintainability. *Inf. and Soft. Technology (IST)*, Vol. 54, Issue 2, pp.212-238, February, 2012.

[12] Dowdy, S. and Wearden, S. Statistics for Research, Wiley, pp. 230, 1983.

[13] Dumke, R.R. and Winkler, A.S. Managung The Component-Based Software Engineering with Metrics. *In proceedings of International Symposium on Assessment of Software Tools and Technologies*, pp.104-110, 1997.

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

[15] FeatureIDE – Available in: <http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/>. Access in Dec., 2014.

[16] Fenton, N.E. and Pfleeger, S.L. Software Metrics: A Rigorous and Practical Approach. 2nd, Publishing Co. Boston, p. 656, 1998.

[17] Ferrari, F. *et al*. An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs. *In Proceeding of International Conference on Software Engineering (ICSE)*, pp. 65-74, 2010.

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

[19] Ferreira, G.C., Gaia, F.N., Figueiredo, E. and Maia, M.A. 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.

[20] Figueiredo, E. *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.

[21] Foss, S., Korshunov, D. and Zachary, S. An Introduction to Heavy-Tailed and Subexponential Distributions. Springer-Verlag, 2011.

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

[23] French, V.A. Establishing Software Metric Thresholds. *In Proc. of the International Workshop on Software Measurement (IWSM'99)*, 1999.

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

[25] Garcia, J., Popescu, D., Edwards, G. and Medvidovic, N. Identifying Architectural Bad Smells. *In Proceedings of Conference on Software Maintenance and Reengineering (CSMR)*, pp.255-258, 2009.

[26] Kitchenham, B. and Charters, S. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Software Engineering Group, School of Computer Science and Mathematics, Keele University, *EBSE Technical Report* Version 2.3, 2007.

[27] Lanza, M., Marinescu, R. Object-Oriented Metrics in Pratice. Springer-Verlag, p. 205, 2006.

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

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

[30] Macia, I. *et al.*, Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity?. *In Proceedings of International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 167-178, 2012.

[31] Marinescu, R. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. *In Proceedings of 20th International Conference on Software Manutenace (ICSM)*, pp. 350-359, 2004.

[32] Mathwave – Available in: <http://migre.me/nOM3b> Access in December, 2014.

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

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

[35] Oliveira, P., Valente, M.T. and Lima, F.P. Extracting Relative Thresholds for Source Code Metrics. *In Proceedings of the Conference on Software Maintenance, and Reengineering (CSMR)*, pp.254-263, 2014.

[36] Oliveira, P., Lima, F., Valente, M.T. and Serebrenik, A. RTTOOL: A Tool for Extracting Relative Thresholds for Source Code Metrics. *In Proc. of the 30th Int. Conf. on Software Maintenance and Evolution (ICSM)*, pp. 1-4, 2014.

[37] Padilha, J., Pereira, J., Figueiredo, E., Almeida, J. Garcia, A. and Sant'Anna, C. On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study. *In Proc. of 26th International Conference on Advanced Information Systems Engineering (CAISE)*, pp. 656-671, 2014.

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

[39] Software Engineering Institute. Software product line. Available in: <http://migre.me/nOM7f>. Access in December, 2014.

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

[41] SPL2GO - Available in: <http://spl2go.cs.ovgu.de/>. Access in December, 2014.

[42] SPL Metric Threshold – Available in: <http://labsoft.dcc.ufmg.br/doku.php?id=%20about:spl_list>. Access in April, 2015.

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