A BENCHMARK-BASED METHOD TO DERIVE METRIC THRESHOLDS

GUSTAVO ANDRADE DO VALE

A BENCHMARK-BASED METHOD TO DERIVE METRIC THRESHOLDS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Eduardo Magno Lages Figueiredo

Belo Horizonte

Fevereiro de 2016

GUSTAVO ANDRADE DO VALE

A BENCHMARK-BASED METHOD TO DERIVE METRIC THRESHOLDS

Dissertation presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

Advisor: Eduardo Magno Lages Figueiredo

Belo Horizonte
February 2016

Vale, Gustavo Andrade do

V149b A Benchmark-based Method to Derive Metric Thresholds / Gustavo Andrade do Vale. — Belo Horizonte, 2016
xx, 76 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação

Orientador: Eduardo Magno Lages Figueiredo

1. Computação – Teses. 2. Engenharia de software – Teses. 3. Software – Reutilização – Teses. I. Orientador. II. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

A benchmark-based method to derive metric thresholds

GUSTAVO ANDRADE DO VALE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ALESSANDRO FABRICIO GARCIA
Departamento de Informática - PUC-RJ

PROFA. KÉCIA ALINE MARQUES FERREIRA
Departamento de Computação - CEFET-MG

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 22 de fevereiro de 2016.

Acknowledgments

This work would not have been possible without the support of many people.

I thank God to provide me the discipline and persistence to reach a Master degree.

I thank my dear wife Fernanda, who has had patience in difficult moments and has always been by my side.

I thank my whole family — especially my father Fernando, my mother Marcia, my sister Leticia, my father-in-law Tom, my mother-in-law Regina and my sister-in-law Paula — for having always supported me.

I thank my advisor E. Figueiredo for his attention, motivation, patience, dedication, and immense knowledge. I certainly could not complete my Master study without him.

I thank L. Veado, E. Fernandes, R. Abilio, H. Costa, D. Albuquerque for the valuable collaboration in the case studies.

I thank the members of the LabSoft research group for the friendship and technical collaboration.

I would like to express my gratitude to the members of my dissertation defense — A. Garcia (PUC-Rio), K. A. M. Ferreira (CEFETMG) and M. T. Valente (UFMG).

I thank Capes and PPGCC-UFMG for the financial support.

Resumo

Com o crescimento em tamanho e complexidade dos sistemas de software, melhores suportes são requeridos para medir e controlar a qualidade de software. Métricas de software são um caminho prático para avaliar diferentes atributos e características de qualidade, como tamanho, complexidade, manutenibilidade e usabilidade. Apesar disso, apenas os valores de métricas não são suficientes. A medição efetiva de sistemas de software é diretamente dependente da definição de valores limiares apropriados. Valores limiares permitem caracterizar objetivamente ou classificar cada componente de acordo com uma métrica de software. A definição de valores limiares apropriados precisa ser calculada para cada métrica. Com o objetivo de investigar este tópico, uma revisão sistemática da literatura de métodos para calcular valores limiares foi realizada. Nesta revisão, foi analisada a evolução de tais métodos e percebeu-se que pesquisadores e profissionais da indústria não possuem um consenso sobre as características de tais métodos. De fato, muitos métodos têm sido propostos e utilizados nos últimos anos. Após a revisão da literatura, foi realizado um detalhado estudo comparativo de três métodos recentemente propostos para calcular valores limiares (métodos de Alves, Ferreira e Oliveira). Nessa comparação são destacadas as principais características de cada método e, como lições aprendidas, baseado no conhecimento teórico e prático adquirido, são apresentados oitos pontos desejáveis para este tipo de método. Almejando cobrir todos os pontos desejáveis e capturar o melhor de cada método, um método é proposto para calcular valores limiares para métricas, chamado Vale's method. Este método foi devidamente descrito, cada passo do método foi justificado e uma ferramenta para apoiar o método proposto e os métodos comparados foi desenvolvida. No total derivaram-se valores limiares para 8 métricas de software. Para avaliar o método proposto, (i) analisaram-se os valores limiares individualmente e utilizando uma estratégia baseada em métricas, (ii) analisaram-se os resultados utilizando duas bases de dados com métricas de diversos sistemas de dois diferentes tipos, e (iii) forneceu-se uma visão geral do método proposto comparado com outros métodos presentes na literatura. Em resumo, todos os métodos estudados parecem ser justos para

calcular valores limiares para métricas de software, no entanto, o método proposto se saiu melhor nas avaliações.

Palavras-chave: Qualidade de Software, Métricas de Software, Valores Limiares, Métodos para Calculo de Valores Limiares.

Abstract

With software-intensive systems growing in size and complexity, better support is required for measuring and controlling the software quality. Software metrics are the practical means for assessing different quality attributes and characteristics, such as size, complexity, maintainability, and usability. In spite of that, only the values of metrics are not enough. The effective measurement of software systems is directly dependent on the definition of appropriate thresholds. Thresholds allow to objectively characterize or to classify each component according to one of the software metrics. The definition of appropriate thresholds needs to be tailored to each metric. Aiming to investigate this topic, we first performed a literature review of methods to derive thresholds. In this review, we analyzed the evolution of such methods and realized that researchers and practitioners do not have a consensus about the characteristics of these methods. In fact, many methods have been proposed and have been used in the lasts years. After the literature review, we present a detailed comparison of three recently proposed methods to derive metric thresholds (Alves's, Ferreira's, and Oliveira's methods). This comparison highlights the main characteristics of each method and, as lessons learned, we present eight desirable points for this kind of method based on our theoretical and practical knowledge. Trying to fit all desirable points and getting the best of each method, we propose our own method to derive metric thresholds, named Vale's method. We explain our method, justifying each of its steps, and develop a tool to support the method, called TDTool. In the total we provide thresholds for 8 metrics. To evaluate Vale's method, we (i) analyzed the derived thresholds individually and using a metric-based detection strategy, (ii) analyzed the results using two different types of benchmarks, and (iii) provided an overview of the method compared to other methods in the literature. In summary, all the compared methods seem to be fair to derive metric thresholds, but our method fared better in the evaluations.

Keywords: Software Quality, Metric, Thresholds, Method to Derive Thresholds.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

With software-intensive systems growing in size and complexity, better support is required for measuring and controlling the software quality [Gamma et al., 1995]. Software metrics are the practical means for assessing different quality attributes, such as maintainability and usability [Chidamber and Kemerer, 1994; Lorenz and Kidd, 1994]. Certain metric values can help to reveal specific components (or modules) of a software system that should be closely monitored [Dumke and Winkler, 1997]. For instance, such measures can be used to indicate whether a critical anomaly (or smell) is affecting a component structure [Riel, 1996].

Although software metrics are the pragmatic means for assessing different quality attributes, only the values of metrics are not enough. The effective measurement of software systems is directly dependent on the definition of appropriate thresholds. Thresholds allow to objectively characterize or to classify each entity (e.g. module, class, method) according to one of the quality metrics. The definition of appropriate thresholds needs to be tailored to each metric.

1.1 Motivation, Problem Description, and Goal

Thresholds have a high influence in the software quality measurement. Additionally, as software systems have been increasing in size and complexity in the past few years, thresholds must be calculated in a specific context, avoiding generic or global thresholds. However, it is necessary methods easy to use, simple, and fair to derive thresholds. Given the necessity we want to know:

- *What are the methods to derive metric thresholds?*
- *What are the main characteristics of methods to derive thresholds?*

- *What is desirable for methods to derive thresholds?*

- *Do a method better than another? Based on its characteristics?*

Given these four questions, we start a literature review to answer them. We could see that in the past few years, thresholds were calculated based on software engineers' experience or by using a single system as reference [Chindamber and Kemerer, 1994; Coleman et al., 1995; Erni and Lewerentz, 1996; French, 1999; McCabe, 1976; Nejmeh, 1988; Spinellis, 2008; Vasa et al., 2009]. Recently, it has been changing and thresholds have been calculated considering three key points [Alves et al., 2010; Ferreira et al., 2012; Oliveira et al., 2014]: (i) well-defined methods, (ii) methods that consider the skewed distribution of software measurements, and, (iii) methods which use data from benchmarks.

Although we have found many studies about thresholds calculation, the recent improvements in methods to derive thresholds indicate an open field to explore this research topic and we did not find any comparative study of methods to derive thresholds. Additionally, methods previously proposed in literature do not address important aspects when metric-based strategies are used, such as the lower bound thresholds. Lower bound thresholds can be useful for identifying lazy class, for example. Lazy class is a bad smell defined as a class that knows or does too little in the software system [Fowler et al., 1999].

Figure 1.1 presents a detection strategy to identify lazy class instances [Munro, 2005] which combine three metrics (Lines of Code (LOC) [Fenton and Pfleeger, 1998], Weight Method per Class (WMC) [Chidamber and Kemerer, 1994], and, Coupling between Objects (CBO) [Chidamber and Kemerer, 1994]) with logical operators (AND and OR). Note that for each metric a specific threshold is required. According to this detection strategy, for a class be a lazy class instance, it should have LOC and WMC smaller than threshold of these metrics; or CBO smaller than threshold of this metric.

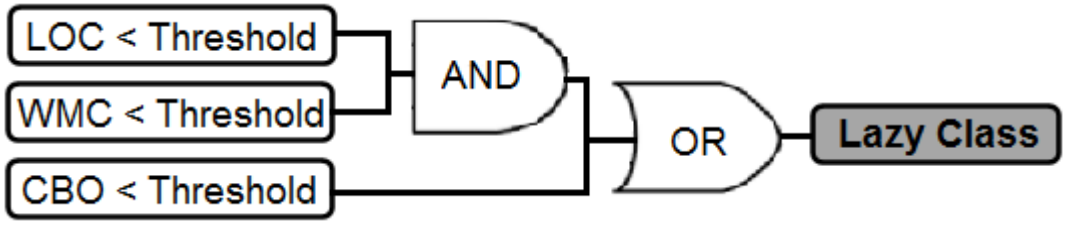

Figure 1.1: Lazy Class Detection Strategy (Adapted from [Munro, 2005])

Additionally to lower bound thresholds, as we are going to see in this dissertation, we compared three methods recently proposed, which address the three key points previously mentioned. With this comparison, we described eight desirable points extracted based on our theoretical and practical experience. Following these desirable points, the methods should: (i) be well-defined and deterministic; (ii) derive thresholds in a step-wise format; (iii) be weakly dependent on the number of systems; (iv) be strongly dependent on the number of entities; (v) not correlate metrics; (vi) calculate upper and lower thresholds; (vii) provide representative thresholds independent of metric distribution and (viii) provide tool support. These desirable points were used as motivation to propose our own method to derived metric thresholds, called Vale's method.

Our method is organized in the five following steps: (i) metric extraction, (ii) weight ratio calculation, (iii) sort in ascending order, (iv) entity aggregation, (v) thresholds derivation. In summary, we need to extract the metric values of the target entities (which composes the benchmark) to give the same weight for each entity. The sum of all entities represents 100%. After, we should organize the entities by the value of the selected metric. Then, we should sum up the entities with same value. Finally, we should to derive the thresholds for each one of the labels of the method (*verylow*, *low*, *moderate*, *high*, and *veryhigh*). It is important to highlight that for each metric these steps should be followed. Additionally, we provide a tool to support our method and, the proposed method was evaluated in different ways, it is better explained on the next section.

The main goal of this dissertation is to propose a method to derive metric thresholds addressing the eight desirable points that previous proposed methods do not address, but these desirable points are important to derive appropriate thresholds.

1.2 Methodological Procedures and Contributions

Given the importance of metrics as well as thresholds to measure software quality, we explore this topic in this dissertation. To achieve our main goal, we realized four main tasks. The four tasks are: literature review, comparative study, proposal of a method, and evaluation of the proposed method. First, we performed a literature review about methods to derive thresholds. With this literature review, we found many methods to derive metric thresholds. In special, we highlight three methods that fit the three key

points previous mentioned: Alves's [Alves et al., 2010], Ferreira's [Ferreira et al., 2012] and Oliveira's [Oliveira et al., 2014] methods.

Then, we provide a comparative study with the highlighted methods using 3 benchmarks composed by software product lines (SPLs). An SPL is a configurable set of systems that shares a common, managed set of features in a particular market segment [SEI, 2016]. Features can be defined as modules of an application with consistent, well-defined, independent, and combinable functions [Apel et al., 2009]. We decided to build SPL benchmarks because of SPLs tend to be systems more modularized than single systems and they are been increasingly adopted in software industry to support coarse-grained reuse of software assets [Dumke and Winkler, 1997]. To build these benchmarks, we looked for papers and repositories about SPLs. Chapter 3 gives more details about the SPL benchmarks.

With the comparative study, we pointed out some desirable points based on theoretical and practical knowledge applying the three methods. With these desirable points, we find the opportunity to propose a method (third task). For example, Alves's, Ferreira's, and Oliveira's methods do not present lower bound thresholds. Additionally, we can find aspects addressed by one method, but not addressed by other methods, such as, to present thresholds in a step-wise format.

After the comparative study, we propose our own method, called Vale's method. The proposed method is descripted and we present a complete example of use. Additionally, in the third task, we provide a tool, called TDTool, to support the proposed method and other three methods (Alves's, Ferreira's, and Oliveira's methods).

Finally, on the forth task, we evaluate the proposed method using different contexts, benchmarks, and in different ways, such as the effectiveness in detecting bad smells and analyzing the values individually. In the total, we derive thresholds for eight different metrics. In the case of benchmarks, we use three benchmarks composed by SPLs and one composed by single systems developed using Java. Differently to the SPL benchmarks, the Java benchmark was previously proposed and used in another studies.

1.3 Dissertation Outline

This master dissertation is organized in 6 chapters, as follows.

Chapter 2 introduces background concepts about metrics, software product lines, and the evolution of methods to derive thresholds.

Chapter 3 presents a comparative study of methods to derive thresholds highlighting desirable points pointed out in this comparison.

Chapter 4 describes the proposed method, called Vale's method, based on the desirable points, presented in previous chapter, and a tool to support the proposed method and other three methods, called TDTool.

Chapter 5 evaluates the proposed method considering different aspects, such as using different benchmarks and benchmarks from different contexts.

Finally, Chapter 6 concludes this dissertation, presents some contributions of our work, publication results, and directions for future work. Additionally, Appendix A, presents primary studies of our literature review.

Chapter 2

Background

This chapter presents important concepts to understand this dissertation. These main concepts involve three main topics: software product lines (SPLs), metrics, and methods to derive metric thresholds. Section 2.1 starts with some important concepts about SPLs and feature-oriented programming because we build a benchmark composed by SPLs on the next chapter and, we use a metric specific to SPLs. Then, Section 2.2 introduces the concept of metrics and the metrics used in this dissertation. After, the next sections are related to the literature review and concepts about metrics to derive thresholds. Therefore, Section 2.3 presents our protocol to get methods to derive thresholds. Section 2.4 presents the results of our literature review and the different types of these methods. Section 2.5 discusses the importance of three fundamental points of methods to derive thresholds. We called these points of three key points. These key points are: (i) methods well-defined, (ii) methods that consider the skewed distribution of software measurements, and, (iii) methods that use benchmarks as database to derive thresholds. Section 2.6 describes three methods which address these three key points. These methods are explored in this dissertation and, because of that we present them in a separate section. Finally, Section 2.7 summarizes Chapter 2.

2.1 Software Product Lines

Software Product Line (SPL) is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment [Pohl and Metzer, 2006]. The systematic and large scale reuse adopted in SPLs aim to reduce time-to-market and improve software quality [Pohl et al., 2005]. The software products derived from an SPL share common features and differ themselves by their specific features [Pohl et al., 2005]. A feature represents an increment in functionality

or a system property relevant to some stakeholders [Kastner et al., 2007]. And, features can be defined as modules with consistent, well-defined, independent, and combinable functions [Apel et al., 2009]. The possible combinations of features to build a product are called SPL variability [Weiss and Lai, 1999] and it can be represented in a feature model [Kang et al., 1990]. Feature model is a formalism to capture and to represent the commonalities and variabilities among the products in an SPL [Asikainen et al., 2006].

In order to develop an SPL, we can use different approaches, such as, annotative [Liebig et al., 2010] and compositional [Apel and Kastner, 2009]. For these approaches, there are several techniques, for example, preprocessors [Liebig et al., 2010], virtual separation of concerns [Kastner et al., 2008], aspect-oriented programming [Kiczales et al., 1997], delta-oriented programming [Schaefer et al., 2011], and feature-oriented programming [Batory et al., 2003]. These approaches and techniques aim to support configuration management at source code level and improve the software quality.

Feature-oriented programming (FOP) is a compositional technique to develop SPLs. There are many feature-oriented languages and tools aiming at feature modularity, e.g., AHEAD/Jak [Batory et al., 2004], FeatureC++ [Apel et al., 2005], and FeatureHouse [Apel et al., 2009]. In this section, we use AHEAD as a representative for FOP compositional approaches. AHEAD is based on the concept of step-wise refinements. Step-wise refinement is a paradigm to develop a complex program from a simple program by incrementally adding details [Batory et al., 2003]. The program increments and original fragments are called refinements and constants, respectively [Batory et al., 2003]. Classes (constants) implement basic functions of a system and extensions in these functions constitute the class refinements. The AHEAD Tool Suite (ATS) was developed to support FOP in AHEAD and it has tools for realization and composition of features [Batory, 2004]. ATS relies on the Jakarta (Jak) programming language (superset of Java) [Batory, 2004]. Constants and refinements are defined in Jak files, but constants are pure Java-code and refinements are identified by the keyword $refines$.

Figure 2.1 depicts the concept of constants and refinements into features. In this figure, three features are represented (i, j, and k). Feature i has 4 class constants (ai, bi, ci, di); feature j has 3 class refinements aj, cj, and dj and 1 class constant ej; feature k has 2 class refinements ck and dk. However, the refinements ck and dk cross-cuts 3 features, i.e., it encapsulates fragments of i, j, and k. In general, a forest of inheritance hierarchies is created as features are composed, and this forest grows progressively broader and deeper as the number of features increases [Batory et al., 2002].

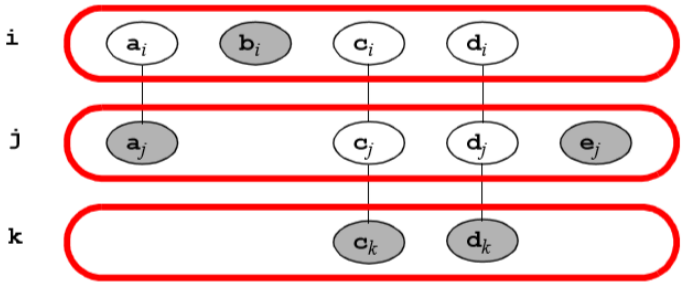

Figure 2.1: Features, Constants, and Refinements relationship

Figure 2.2 depicts a code example in AHEAD. The ai class implements a stack with two methods: push and pop. The refines keyword in aj class indicates that aj refines ai. In this example, the aj class adds new methods (`backup()` and `restore()`) and extends the behavior of the `push()` method (in the ai class) by adding the calling of `backup()` method. The calling of `push()` method in the refinement chain is performed using the `Super` keyword. We use ATS to compose base code and different feature modules. Different products are generated according to inclusion of features in the composition process [Kastner and Apel, 2008].

Fature i

```
class a { //stack
   void push(Object o) { … }
   Object pop() { … }
}
```

Feature j

```
refines class a { //stack
   void backup() { … }
   void restore() { … }
   void push(Object o) {
      backup();
      Super.push(o);
   }
}
```

Figure 2.2: Example of code (adapted from [Kastner and Apel, 2008])

2.2 Software Metrics

Developers, project managers, clients, and software maintainers are interested in measuring different properties of software projects, processes, and products. For instance, developers might measure software properties aiming at checking functional requirements or quality. Regarding software internal quality, we can measure properties, such

as size, coupling, cohesion, and complexity using metrics. In this work, we use the eight following metrics: *Coupling between Objects*, *Depth of Inheritance Tree*, *Lack of Cohesion in Methods*, *Number of Children*, *Response for a Class*, *Weight Method per Class*, *Lines of Code*, and *Number of Constant Refinements*. Figure 2.3 provides examples of applying each of the eight metrics:

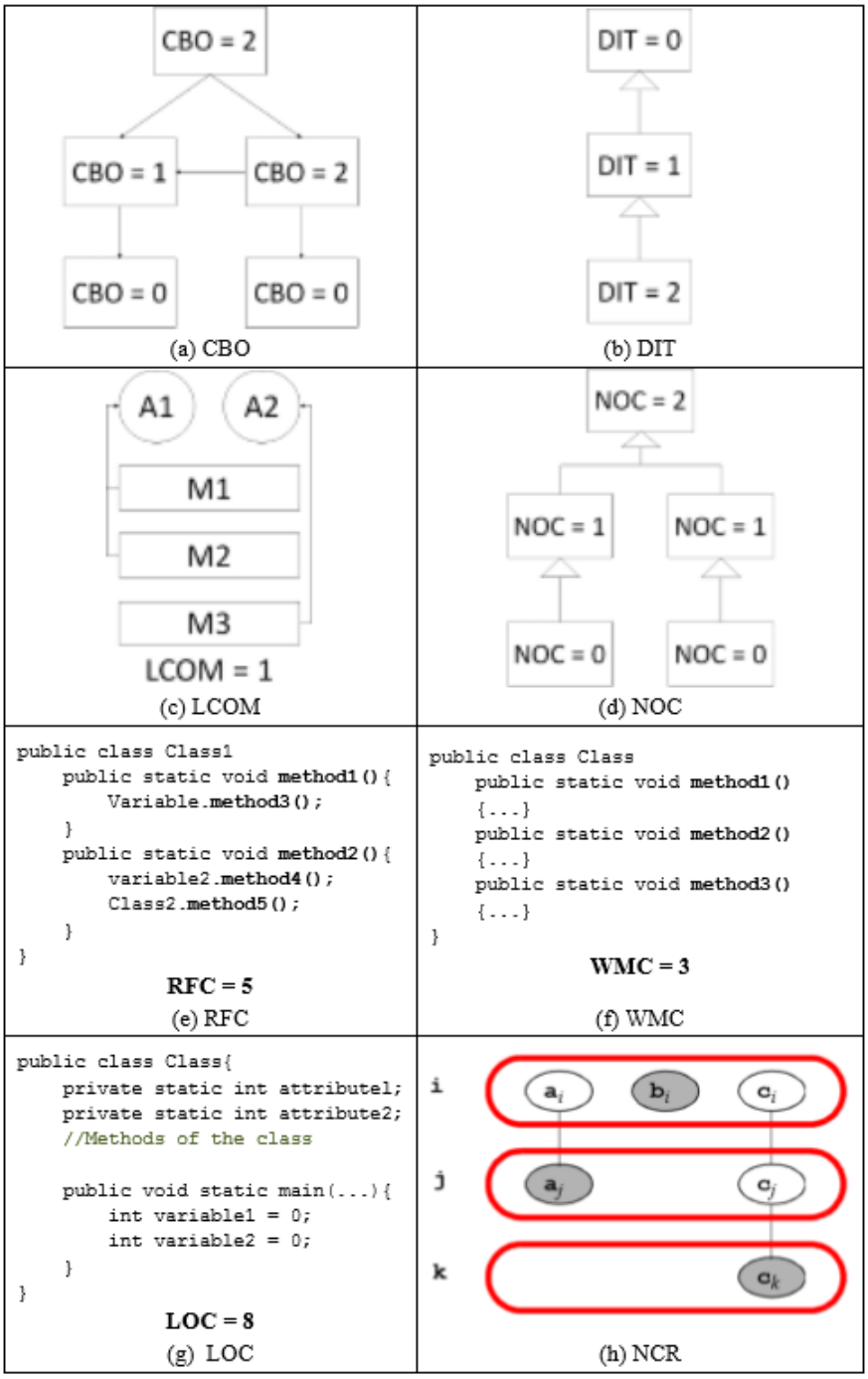

Figure 2.3: Examples of Computing Metrics

- Coupling between Objects (CBO) (Chidamber and Kemerer 1994) counts the number of classes called by a given class. CBO measures the degree of coupling among classes. Figure 1.3(a) illustrates an example of how CBO is calculated. In that example, each box represents classes and each arrow represents a relation between two classes.

- Depth of Inheritance Tree (DIT) (Chidamber and Kemerer 1994) counts the number of levels that a subclass inherits methods and attributes from a superclass in the inheritance tree of the system. This is another metric to estimate the class complexity/coupling. Figure 1.3(b) presents an example of DIT computation. As can be seen in Figure 1.3(b), a class has DIT = 0, the subclass of this class has DIT = 1 and, successively.

- Lack of Cohesion in Methods (LCOM) (Chidamber and Kemerer 1994) counts the number of method pairs whose access non-common attributes, minus the count of method pairs whose access common attributes. The larger the number of similar methods, the more cohesive the class. For instance, in Figure 1.3(c), M1, M2, and M3 are methods. The pairs M1-M3 and M2-M3 do not access the attribute A1. On the other hand, the pair M1-M2 accesses the attribute A1. Therefore, LCOM = 1.

- Number of Children (NOC) (Chidamber and Kemerer 1994) counts the number of direct subclasses of a given class. This metrics indicates code reuse. Figure 1.3(d) presents an example of computed NOC. For instance, the class higher up in inheritance tree has two derived classes at the immediately below level, that extends the parent class. Therefore, this class has NOC = 2. In turn, the classes at the lowest levels have no derived classes that extend them. Therefore, NOC = 0.

- Response for a Class (RFC) (Chidamber and Kemerer 1994) is a set of methods that can potentially be executed in response to a message received by an object of that class. This metric supports the assessment of class complexity. Figure 1.3(e) presents an example of RFC computing. In this figure, the sample class implements two methods, and it calls three methods from other classes. These five methods may be called depending on the use of instantiate objects of the illustrative class. Therefore, RFC = 5.

- Weight Method per Class (WMC) (Chidamber and Kemerer 1994) weights the methods of a class. Particularly, in this work it weights the method of a class

counting the number of methods in a class. This metric can be used to estimate the complexity of a class. Figure 1.3(f) illustrates how WMC is computed when considering the number of method as a weight. For each method present in a class, we increment the value of WMC. Therefore, in case of Figure 1.3(f) in which there are three methods, WMC = 3.

- Lines of Code (LOC) (Lorenz and Kidd 1994) counts the number of uncommented lines of code per class. The value of this metric indicates the size of a class. Figure 1.3(g) presents an illustrative example of computed LOC. LOC counts code lines, but LOC does not count neither comment lines nor blank lines. In Figure 1.3(g), LOC is equals to 8.

- Number of Constant Refinements (NCR) (Abilio et al. 2015) 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) (Batory, 2004). That is, refinements can change the behavior of a constant if certain feature is included in a product (see Section 2.1). Figure 1.3(h) presents an example of computed NCR. In Figure 1.3(h), we have the features i, j, and k; classes a, b, and, c; ai, bi, ci are constants of feature i; aj, cj, ck are refinements of feature j and k. Therefore, constants ai, bi, and ci have NCR equal to 1, 0, and, 3, respectively.

We chose LOC, CBO, WMC, and, NCR because of the metric-based detection strategies that we are going to use in this dissertation. Moreover, we included other metrics to cover all the six well-known object-oriented software metrics proposed by Chidamber and Kemerer (1994): CBO, DIT, LCOM, NOC, RFC, and WMC. For all eight metrics described, classes with higher values are more likely to be worse in the software systems quality.

2.3 Literature Review Protocol

One of the goals of Software Engineering is to manage and control the quality of software systems [Sommerville, 2011]. Software metrics are the practical means for assessing different quality aspects, such as maintainability and usability [Chidamber and Kemerer, 1994; Lorenz and Kidd, 1994]. As important as metrics, thresholds allow to objectively characterize or to classify each component according to one of the quality metrics. Given the importance of metrics and thresholds, we can find many studies

that propose metrics and thresholds [McCabe 1976; Nejmeh, 1988; Chidamber and Kemerer, 1994; Erni et al., 1996; Lanza and Marinescu, 2006; Ferreira et al., 2012]. Generally, studies that propose thresholds present a method or strategy to derive such thresholds.

This section summarizes how we got the papers related to methods to derive thresholds. First, we started an ad-hoc literature review held in four different electronic databases: IEEExplore [1], Science Direct [2], ACM Digital Library [3], and El Compendex [4]. With this ad-hoc literature review, we found a systematic literature review (SLR) [Lima, 2014] with a similar, but different purpose of ours. An 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 [Kitchenham and Charters, 2007]. This existing SLR of Lima [2014] aims to group metric thresholds reported in the literature, but it does not focus on methods to derive thresholds.

The previous SLR was used as a starting point for our work. It selected 19 papers to get and report information. We read these 19 selected papers and other papers that we found in the ad-hoc literature review. Then, we performed the snowballing technique in those papers [Brereton et al., 2007]. This technique consists of investigating the references retrieved in electronic databases in order to find additional relevant papers to increase the scope of the search, providing broader results [Brereton et al., 2007].

The literature review of this dissertation followed similar steps to the protocol of a Systematic Literature Review (SLR) [Kitchenham and Charters, 2007]. The inclusion criteria are: (i) paper must be in computer science area; (ii) paper must be written in English; (iii) paper must be completely in electronic form; and, (iv) paper must propose or use at least one method to derive metric thresholds. Following these inclusion criteria and applying the snowballing technique in the papers of our literature review, we selected 50 primary studies in order to extract information related to methods to derive thresholds. These primary studies are listed in Appendix A. We summarize the main idea of these primary studies in the next section.

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

2.4 Types of Methods to Derive Thresholds

The 50 primary studies mentioned in the previous section propose or use a strategy or method to derive thresholds. The primary studies were published between 1976 and 2015. This range shows that software engineers started to worry about thresholds a long time ago. Additionally, we can see that methods to derive thresholds are still an open issue and different strategies have been proposed over time.

To summarize our review, we start by describing papers where thresholds are defined by programming experience. Then, we analyze in details methods that derive thresholds based on data analysis, which are directly related to our research. After, we discuss techniques to analyze and summarize metric distributions. Finally, we present the three key points and well-defined methods that consider the three key points.

2.4.1 Thresholds Derived from Programming Experience

Many authors defined metric thresholds according to their programming experience [McCcabe, 1976; Nejmeh, 1988; Coleman et al., 1995]. For example, the values 10 and 200 were defined as thresholds for Cyclomatic Complexity of McCabe [1976] and NPATH [Nejmeh, 1988], respectively. McCabe Cyclomatic Complexity 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. Code smells describe a situation where there are hints that suggest a flaw in the source code [Riel, 1996]. Regarding Maintainability Index (MI), the values 65 and 85 are defined as thresholds [Coleman t al., 1995]. 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 these results are difficult to reproduce or generalize. Additionally, the lack of scientific support can lead to contest the derived values.

2.4.2 Thresholds Derived from Metric Analysis

Erni et al. [1996] 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 $T = \mu$ - σ when high and low values of a metric indicate potential design problems, respectively. Lanza and Marinescu [2006] use a similar method in their research for 45 Java projects and 37 C++ projects. Nevertheless, they use four labels: low, mean, high, and very high. Labels low, mean, and high is calculated in the same way as Erni [1996]. Labels

very high is calculated as $T = (\mu+\sigma)$ x 1.5. Abilio et al. [2015] use the same method of Lanza and Marinescu, but they derive thresholds based on eight Software Product Lines (SPLs). These methods are a common statistical technique few years ago. However, Erni et al. [1996], Abilio et al. [2015], and Lanza and Marinescu [2006] do not analyze the underlying distribution of metrics. The problem with these methods is that they assume metrics are normally distributed, limiting the use of these methods.

French [1999] also proposes a method based on the mean and standard deviation. However, French used the Chebyshev's inequality theorem (whose validity is not restricted to normal distributions). A metric threshold 'T' can be calculated by $T = \mu+k$ x σ, 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.

2.4.3 Methods for Characterizing Metric Distributions

Chidamber and Kemerer [1994] use histograms to characterize and analyze data. For each of their 6 metrics (e.g., WMC and CBO), they plotted histograms per programming language to discuss metric distribution and spot outliers in C++ and Smaltalk systems. Spinellis [2008] compares metrics of four operating system kernels (i.e., Windows, Linux, FreeBSD, and OpenSolaris). For each metric, boxplots of the four kernels are put side-by-side showing the smallest observation, lower quartile, median, mean, higher quartile, and the highest observation and identified outliers. The boxplots are then analyzed by the author and used to give ranks, + or -, to each kernel. However, as the author states, the ranks are given subjectively.

Vasa et al. [2009] propose the use of Gini coefficients to summarize a metric distribution across a system. The analysis of the Gini coefficient for 10 metrics using 50 Java and C# systems revealed that most of the systems have common values. Moreover, higher Gini coefficient values indicate problems and, when analyzing subsequent releases of source code, a difference higher than 0.04 indicates significant changes in the code.

Other three papers [Alves et al., 2010; Ferreira et al., 2012; and, Oliveira et al., 2014] propose methods to derive thresholds characterizing metric distributions. These papers consider it and other two points fundamental for this kind of method. Hence, we explain these fundamental points (called key points) first in Section 2.5, to after in Section 2.6 describes Alves', Ferreira', and Oliveira's methods. Additionally, these three methods are highlighted because they were used in other chapters of this dissertation.

2.5 Key Points of Methods to Derive Thresholds

Previously, we have mentioned three key points for methods to derive thresholds: (i) methods well-defined, (ii) methods that consider the skewed distribution of software measurements, and, (iii) methods that use benchmarks as database to derive metric thresholds. This section discusses the importance of each one of these key points. But before it, we would like to highlight our point of view. We think that thresholds should be derived for specific contexts (benchmarks), rather than for universal contexts.

2.5.1 Well-defined Methods

This key point is related to the steps of a method. In the key point it is taken into account how well structured and described, a method is. Using the same input and following the method description, the same thresholds for a metric must be obtained.

2.5.2 Consider the Skewed Distribution of Software Metrics

The second key point is related to the statistical approach used by the method to derive thresholds. This is an important point because software metrics can have different distributions, such as normal, power law, and common values (like Poisson distribution) [Louridas et al., 2008]. A method using only mean and standard derivation can provide invalid or non-representative thresholds. Some studies assume that software metrics have a normal distribution [Louridas et al., 2008]. In spite of that, several studies clearly demonstrate that most software metrics do not follow normal distributions [Alves et al., 2010; Concas et al., 2007; Ferreira et al., 2012; Louridas et al., 2008; Oliveira et al., 2014], limiting the use of any statistical method that relies on mean to derive thresholds, for example.

Hence, some studies fall short in concluding how to use these distributions, and their coefficients, to establish baseline values for 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 [Concas et al., 2007; Louridas et al., 2008] show that different software metrics follow heavy-tailed distribution. For instance, Concas et al. [2007] show that most of the Chidamber and Kemerer's metrics [Chidamber and Kemerer, 1994] follow heavy-tailed distribution for a large Smalltalk system.

2.5.3 Benchmark-based

This key point is related to the confidence of the derived thresholds. In the past, software engineers derived thresholds by their subjective opinion. Then, they started to discuss in groups and get a consensus about the thresholds [Coleman et al., 1995]. After, in a third moment, they started to analyze systems to help deriving thresholds. Finally, as from the first to the second case, they started to derive thresholds using a group of systems, commonly called benchmarks. In this work, we consider a benchmark as a set of systems in which the source code and metrics used are available online.

The idea of using benchmark-based methods is to collect information from similar systems to help derive thresholds. For example, if almost all classes of a benchmark have cyclomatic complexity of McCabe [McCabe, 1976] smaller than 10, the minority of classes with cyclomatic complexity greater than 10 are outliers. It does not mean that these outliers are worse than the other classes, but they are different. On the other hand, it is known that the greater the cyclomatic complexity of a class is, the more difficult it is to be understood and consequently smaller its quality is. Summarizing, the idea of using benchmark-based methods is to get common behaviors of the majority of entities. Hence, we assume that it is better than outliers.

2.6 Methods to Derive Thresholds that Fit the Key Points

This section describes three methods that fit the three key points described in the previous section. These methods are compared in the next chapter. Hence, we describe these methods with more details.

Alves's Method – The method proposed by Alves and his colleagues in 2010 [Alves et al., 2010] 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 – 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 needs 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 is proposed by Ferreira and her colleagues in 2012 [Ferreira et al., 2012]. It is relatively simple and can be divided in 4 steps: (1) measurement extraction, (2) grouping metrics, (3) group representation, and (4) threshold derivation. The metric values are first collected for each system (step 1) and organized into a unique file (step 2). Using manual graphic analysis or with a supporting tool, three groups should be created (step 3). These groups represent values with a high, medium, and low frequency in the systems which are classified as *good*, *regular*, and *bad* labels, 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 groups.

Oliveira's Method – This method is proposed by Oliveira and her colleagues in 2014 [Oliveira et al., 2014] it relies on a formula for calculating thresholds. This formula is called *ComplianceRate* 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.

2.7 Final Remarks

This chapter provides an overview of methods to derive metric thresholds, metrics and feature-oriented software product lines. We saw that thresholds have been explored since a long time ago. Additionally, software engineers still do not achieve a consensus on which method to use because new methods have been recently proposed. Although, it is notable an evolution in the different types of methods to derive thresholds proposed since 2010. The recently proposed methods addressed the three key points, described on Section 2.5. These key points are fundamental when appropriate thresholds are required.

To help software engineers in the choice of methods to derive thresholds, it is required a comparison to highlight the strengths and weaknesses of each one of the recently proposed methods. Therefore, in the next chapter, we compare the three methods to derive metric thresholds, presented on Section 2.6, highlighting different aspects. Our comparison covers different aspects, such as, we analyze the skewed distribution of some target metrics and correlate the target metrics with one specific metric.

Chapter 3

A Comparison of Methods to Derive Metric Thresholds

In the previous chapter, we presented an overview about methods to derive thresholds. We saw that the recent proposed methods addressed three key points: well-defined methods, methods that consider the skewed distribution of software measurements, and methods that use benchmarks as database to derive metric thresholds. We discussed the importance of these three key points in Section 2.5. Additionally, we described in Section 2.6 Alves's, Ferreira's and Oliveira's methods which address these three key points.

This chapter presents a comparative study of these three methods to derive thresholds in the light of benchmarks of software product lines and four metrics also presented in Chapter 2. The metrics used in this study are LOC, CBO, WMC, and NCR. As we could see in previous chapters, thresholds are also important to evaluate quality of software systems. Therefore, the idea is highlighting strangeness and weakness of methods to derive thresholds aiming to make easy the choice of one method that fits better with the user needs. Section 3.1 describes how we built our benchmark. Section 3.2 presents the method comparison. Section 3.3 presents lessons learned with the comparison. Section 3.4 presents threats to validity of the comparative study. Finally, Section 3.5 summarizes this chapter.

3.1 SPL Benchmarks

This section presents three benchmarks of Software Product Lines (SPLs). To build these benchmarks, we focus on SPLs developed using FOP [Batory and Sarvela, 2004]. The main reason for choosing FOP is because this technique aims to support modular-

ization of features - i.e., the building blocks of an SPL (See Section 2.1). In addition, we have already developed a tool, named Variability Smell Detection (VSD) [Abilio et al., 2014], which is able to measure FOP code. Since it is very difficult to find compositional feature-oriented SPLs, this benchmark by itself can be considered an important contribution to the SPL community.

We selected 47 SPLs from repositories, such as SPL2go [SPL2GO, 2015] and FeatureIDE examples [FeatureIDE, 2015], 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 [FeatureIDE, 2015] has 4 different versions implemented in AHEAD, FH-C#, FH-Java and FH-JML. FH stands for FeatureHouse [Apel et al., 2009] 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 3.1. The step-to-step filtering of SPLs is further explained on the supplementary website [SPL Repository, 2016].

In order to generate different benchmarks for comparison, we split the 33 SPLs into three benchmarks according to their size in terms of LOC. Table 3.1 presents the 33 SPLs ordered by their value of LOC, implementation technology (Technology), and grouped by their respective benchmarks. Benchmark 1 includes all 33 SPLs. Benchmark 2 includes 22 SPLs with more than 300 LOC (SPLs 1-22). Finally, Benchmark 3 is composed of 14 SPLs with more than 1,000 LOC (SPLs 1-14). The goal of creating three different benchmarks is to analyze the results with varying levels of thresholds.

3.2 Comparative Study

We evaluate different aspects of 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 (LOC, CBO, WMC, and NCR): (i) correlation with LOC (Section 3.2.1) and (ii) distribution of software metrics (Section 3.2.2). Our goal is to reveal whether these varying characteristics impact on the derived thresholds and if they can provide support to discuss the derived thresholds. Section 3.2.3 presents the derived threshold by Alves', Ferreira's, and, Oliveira's methods.

Table 3.1: Software Product Lines Benchmarks

Id	SPL	Technology	LOC
1	BerkeleyDB [SPL2GO, 2015]	FH-Java	37247
2	AHEAD-Java [Abilio et al., 2015]	AHEAD	16719
3	AHEAD-guidsl [Abilio et al., 2015]	AHEAD	8738
4	TankWar [FeatureIDE], [SPL2GO, 2015]	AHEAD	4670
5	AHEAD-Bali [Abilio et al., 2015]	AHEAD	3988
6	Devolution [FeatureIDE, 2015]	AHEAD	3913
7	MobileMedia v.7 [Ferreira et al., 2014]	AHEAD	2691
8	WebStore v.6	AHEAD	2082
9	DesktopSearcher [FeatureIDE, 2015], [SPL2GO, 2015]	AHEAD	1858
10	GPL [FeatureIDE, 2015]	AHEAD	1824
11	Notepad v.2 [SPL2GO, 2015]	FH-Java	1667
12	Vistex [SPL2GO, 2015]	FH-Java	1480
13	GameOfLife [SPL2GO, 2015]	FH-Java	1047
14	Prop4J [SPL2GO, 2015]	FH-Java	1047
15	Elevator [SPL2GO, 2015]	FH-Java	728
16	ExamDB [SPL2GO, 2015]	FH-JML	568
17	PokerSPL [SPL2GO, 2015]	FH-JML	461
18	EmailSystem [SPL2GO, 2015]	FH-Java	460
19	GPLscratch [SPL2GO, 2015]	FH-JML	405
20	Digraph [SPL2GO, 2015]	FH-JML	374
21	MinePump [SPL2GO, 2015]	FH-JML	367
22	Paycard [SPL2GO, 2015]	FH-JML	319
23	IntegerSet [SPL2GO, 2015]	FH-JML	225
24	UnionFind [SPL2GO, 2015]	FH-JML	194
25	NumberContractOverrinding [SPL2GO, 2015]	FH-JML	165
26	NumberConsecutiveContractRef [SPL2GO, 2015]	FH-JML	148
27	Number ExplicitContractRef [SPL2GO, 2015]	FH-JML	143
28	BankAccount [SPL2GO, 2015]	FH-JML	122
29	EPL [FeatureIDE, 2015]	AHEAD	98
30	IntList [SPL2GO, 2015]	FH-JML	94
31	StringMatcher [SPL2GO, 2015]	FH-Java	22
32	Stack [SPL2GO, 2015]	FH-Java	22
33	HelloWorld [FeatureIDE, 2015]	AHEAD	22

3.2.1 Correlation with LOC

Alves's method assumes that all software metrics correlate with LOC. In order to verify if this assumption 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 [Dowdy and Wearden, 1983]. We apply this coefficient to identify the correlation of LOC with the other selected metrics (CBO, WMC, and NCR).

Table 3.2 shows the coefficient of correlation of LOC and other metrics for the three benchmarks. It can be observed that CBO and WMC metrics have high correlation (values above 0.75) with LOC for all benchmarks. However, NCR has no linear correlation with LOC since the values are 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 3.2: 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

3.2.2 Distribution of Software Metrics

All three selected methods to derive thresholds (Section 2.6) claim to take the distribution of metrics into account. Therefore, this section analyzes the distribution of each software metric (Section 3.1.1) based on the SPL benchmarks (Section 3.1.2).

According to the classification schema suggested by Foss et al. [2011], 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. Weibull has two main probability distribution functions: (i) *probability density function* (pdf) – f(x), and (ii) *cumulative distribution function* (cdf) – F(x). The first function expresses the probability the random variable takes a value x. On the other hand, cdf expresses the probability the random variable takes a value less than or equal to x [Mathwave, 2015]. These functions have parameters α and β, defined by equations (Eq1) and (Eq2):

$$fx(x) = P(X = x) = \frac{\alpha}{\beta}(x/\beta)^{\alpha-1}e^{-(\frac{x}{\beta}^{\alpha})}, \alpha > 0, \beta > 0$$

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

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 [Mathwave, 20015]. A heavy-tailed distribution means that a small number of entities have high values and a large number of cases have low values. In this distribution, the mean is not representative [Alves et al., 2010; Ferreira et al., 2012; Mathwave, 2015].

Table 3.3 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 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 3.3, CBO does not follow a heavy-tailed distribution for FOP-based SPL implementations because it presents α values higher than 1. By analyzing the parameters values, we can be more confident about the metric distribution because, in some cases (e.g., depending of the scale), the plotted graph may give the wrong impression that the metric follows a heavy-tailed distribution.

Table 3.3: 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

3.2.3 Derived Thresholds

This section presents the derived thresholds that were obtained using the three methods to derive thresholds (Section 2.6) according to each benchmark (Section 3.1). The process was performed with the four metrics used in this comparative 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 (key values). Hence, only

the values that represent the percentages are shown. This presentation strategy is also applied to Ferreira's method, because the range of values of group 2 is equals to the ranges of groups 1 and 3. In addition, although Ferreira's method definition does not provide how to extract the three groups, we extracted them by our own knowledge about the method. The three groups have 50% (*good*), 25% (*regular*) and, 25% (*bad*) of data, respectively.

Tables 3.4, 3.5, and 3.6 present the obtained values from the three methods, respectively. 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 defined labels as: *low* (0-70%), *moderate* (70-80%), *high* (80-90%), and *veryhigh* (90-100%). These labels are represented for CBO by the intervals 0-8, 9-12, 13-20, and >21, respectively in Benchmark 1.

Table 3.4: 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 3.5: Threshold Values from Ferreira's Method

Benchmark	**Group**	**LOC**	**CBO**	**WMC**	**NCR**
1	1	12	3	2	0
	3	33	6	7	1
2	1	12	3	2	0
	3	34	6	7	1
3	1	12	4	2	0
	3	34	7	7	1

Table 3.6: 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 Alves's method description). Hence, a higher variation across benchmarks 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 (LOC). 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 in the system size. A peculiar case occurred in Oliveira's method (Table 3.6), 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 because with larger SPLs and consequently lager class it was expected higher thresholds. It does not mean a problem only a particularity of such metric for this method.

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%. However, such percentage is not clear for Ferreira's method. Therefore, we have considered values of the third group (bad label). Finally, we use the returned values for Oliveira's method. Figure 3.1 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 3.1(a) refers to LOC and, for Benchmark 1, the thresholds for methods of Alves, Ferreira, and Oliveira are 151, 33, and 91, respectively.

Figure 3.1 indicates that 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 3.1(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 of the low thresholds derived by three methods.

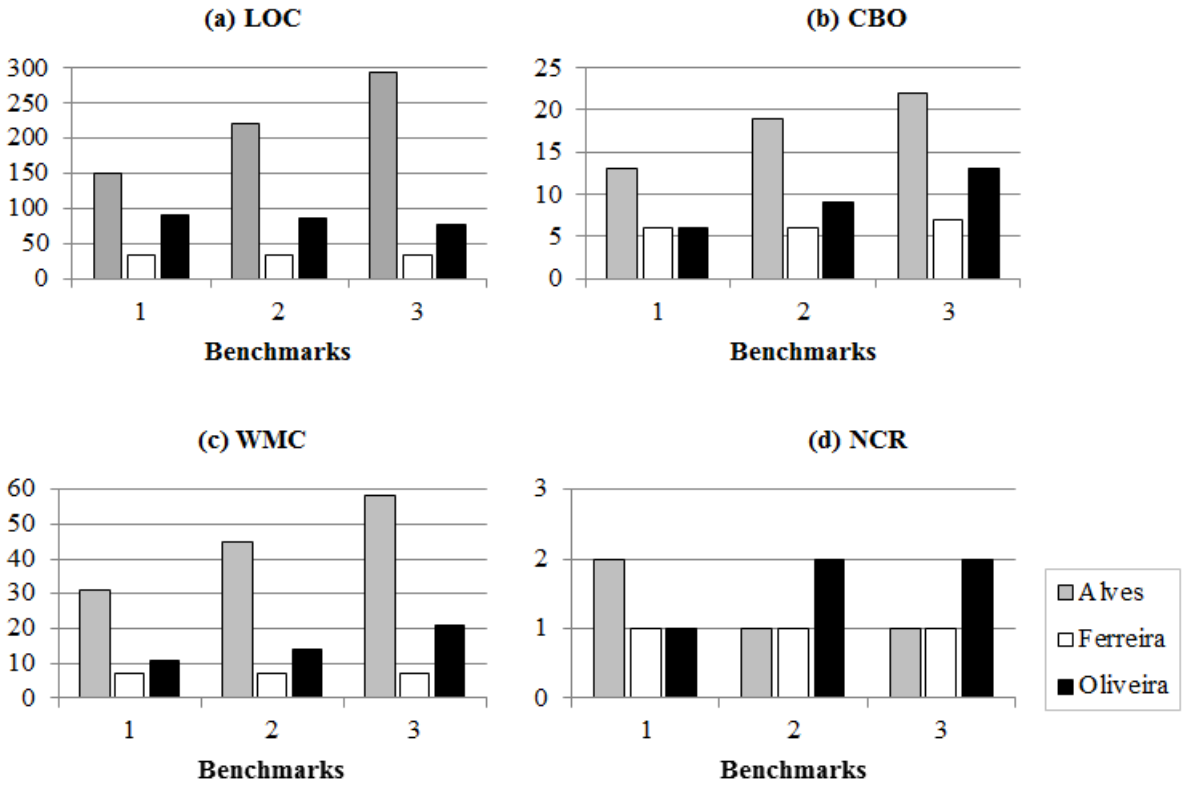

Figure 3.1: Metric Thresholds Side by Side

By analyzing the derived thresholds and considering the correlation of the metrics with LOC (Section 3.2.1), it is possible to see that Alves's method has undergone a major change in its behavior. This major change is probably, due to the method correlate LOC to calculate thresholds. With respect to Ferreira' and Oliveira's methods, it is not possible to observe a clear change in their behavior like Alves's method.

By looking at the derived thresholds and considering the software metrics distribution (Section 3.2.2), 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 3.2.2), the threshold values were not too different between the methods. As far as the values of the other metrics are concerned, the behavior was similar 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, 34, 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, 7, 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.

To illustrate this point, the threshold 6 to CBO defined by Oliveira's method for 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 observation, we can conclude that: (i) this calculation is extremely dependent from the benchmark quality to the three methods and (ii) Alves's method is more rigid to define thresholds than the other methods. Considering these two points, we believe Alves's method fared better with the majority of the analyzed metrics, especially with the ones highly correlated with LOC.

It is clear that some metrics are correlated to others and this is expected. 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 might be 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 with heavy-tail distributions. Even if the distribution is not heavy-tailed, the three methods presented similar derived thresholds.

3.3 Desirable Points of Threshold Derivation Methods

This section discusses characteristics of each method for threshold derivation based on the lessons we learned during this study. In other words, after all our theoretical analysis in and running these three methods in Benchmark 1, 2, and 3 we observed some points that increase or decrease the reliability of the method. We called these points of desirable points. Regarding the use of each method, we enumerate eight questions for discussion: (i) Is it a well-defined method (ii) Is it a deterministic method? (iii) Are step-wise outliers identified? (iv) Does the number of systems impact on the derived thresholds? (v) Does the number of entities impact on the derived thresholds? (vi) Is some metric correlated with another? (vii) Are lower bound thresholds calculated? (viii) Does the method have tool support? It is important to mention that each method has different strengths and weaknesses. These strengths and weaknesses do not necessarily mean that a method is better (or worst) than the other methods. Table 3.7 shows the answer for each question in which are discussed on the five topics bellow, based on our empirical experience by applying the methods in this study.

Table 3.7: Comparative Evaluation of the Method for Calculating Thresholds

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

Well-defined and Deterministic – We consider well-defined methods when it is possible to define steps based on the method description. As we did it for the three methods, we considered all these three methods well-defined. If we replicate this study, are the results going to be exactly the same? Alves's method is well-defined, 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 groups should be extracted and, so, this subjectiveness makes it well-defined, but not deterministic. Oliveira's method is highly deterministic and well-defined. If someone uses the same input, the same results are expected to be obtained.

Number of Systems and Entities - Thresholds are extracted from software entities (e.g., features, modules 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 3.2.1). 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 3.2.1, we believe that, in the general case, 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 [Fowler et al., 1999]. A Lazy Class is defined as a class that knows or does too little in the software system [Fowler et al., 1999]. 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 it's systematic application. Among the analyzed methods, we only found a tool to support the Oliveira's method [Oliveira et al., 2014].

Given the answer of these question we think that it is desirable that a method should: (i) be well-defined and deterministic; (ii) derive thresholds in a step-wise format; (iii) be weakly dependent on the number of systems; (iv) be strongly dependent on the number of entities; (v) not correlate metrics; (vi) calculate upper and lower thresholds; (vii) provide representative thresholds independent of metric distribution, and (viii) provide tool support.

3.4 Threats to Validity

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

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. In order to mitigate this limitation, we created different benchmarks for comparison of the derived thresholds.

Metric Distribution and Metric Label – In this work, we identified that LOC, WMC, and NCR have 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 non-systematization of Ferreira's method

to extract the three groups required us to define three groups with approximately 50% (*good*), 25% (*regular*) and, 25% (*bad*) of data, respectively, totalizing the 100%. We know that Ferreira's method needs a graphic analysis, but to make it more systematic we decided to derive the thresholds with those percentages. Since the other methods are full or partially deterministic, we did not have this problem with them.

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 that this transformation preserves all properties of FH [Apel et al., 2009]. 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.

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 [Alves et al., 2010]. To overcome this problem, the VSD tool [Abilio et al., 2014] 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 [Alves et al., 2010]. On the other hand, the existence of generated code, which normally has very high complexity, may result in higher threshold values [Alves et al., 2010]. As previously stated, for deriving thresholds, we removed all *supplementary code* (e.g., generated code and test cases) from our analysis.

3.5 Final Remarks

This chapter discussed the calculation of representative thresholds in the light of three methods to derive metric thresholds. These methods were described and compared using as input data metrics collected from 33 SPLs. We believe that the methods were reasonable evaluated because we provided a comparison with respect to: (i) three different benchmarks, (ii) metrics with different distributions, (iii) metrics with different degrees of correlation with LOC, and (iv) an analysis of the derived thresholds.

We created a repository with 64 SPLs, in spite of that only 33 SPLs were used (Benchmark 1) following some restrictions. For instance, we picked up only the most

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

Regarding distribution of the metrics used in this study, the metrics LOC, WMC, and NCR have $heavy - tailed\ distributions$. In spite of following a different distribution, CBO apparently did not present a different behavior of the other metrics. Although we have a small sample, the methods seemed 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 3.2.1, correlating metrics can be danger, when we have metrics with low correlation. The other methods did not correlate metrics and, hence, this problem did not impact them.

In order to provide reliable outcomes, we analyzed the thresholds individually. We considered that Alves's method was better in the individual evaluation because it presented more representative thresholds given the inputs for three out of four metrics. In addition, the thresholds are higher compared to the other methods. It resulted in a smaller number of outliers compared to the number of outliers detected by the other methods. On the other hand, Alves's method seems to be more instable when metrics have low correlation with LOC.

After all comparisons and analyses, it can be observed eight desirable points in methods to derive thresholds. It is desirable that a method to derive metric thresholds should: (i) be well-defined and deterministic; (ii) derive thresholds in a step-wise format; (iii) be weakly dependent on the number of systems; (iv) be strongly dependent on the number of entities; (v) not correlate metrics; (vi) calculate upper and lower thresholds; (vii) provide representative thresholds independent of metric distribution, and (viii) provide tool support. These desirable points are explored in the next chapter where we propose a new method to derive thresholds based on them.

Chapter 4

The Proposed Method

Based on the comparison of methods presented in the previous chapter, we see the opportunity to propose a method to derive thresholds with the strangeness and avoiding the weaknesses of the compared methods. However, this chapter proposes a method to derive thresholds based on the eight desirable points described in the previous Chapter.

Section 4.1 presents the proposed method. Section 4.2 describes how we believe that our method addresses the eight desirable points. Section 4.3 presents the tool to support the method. Section 4.4 provides an example of use of the proposed method using a Software Product Line (SPL) benchmark. Section 4.5 presents some threats to validity to the proposed method and it example of use. Section 4.6 summarizes this chapter.

4.1 Method Description

We propose a method with 5 well-defined steps. With the proposed method, we try to get the best of each compared method avoiding the points that we saw are not adequate for methods to derive thresholds, such as, metrics' correlation. Figure 4.1 summarizes the 5 steps of the proposed method. Each step is described as follows.

1. Metric extraction: in the first step, metrics have to be 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 in the second step. That is, we divide the entity weight by the total number of entities and, then, it is multiplied by one hundred. All

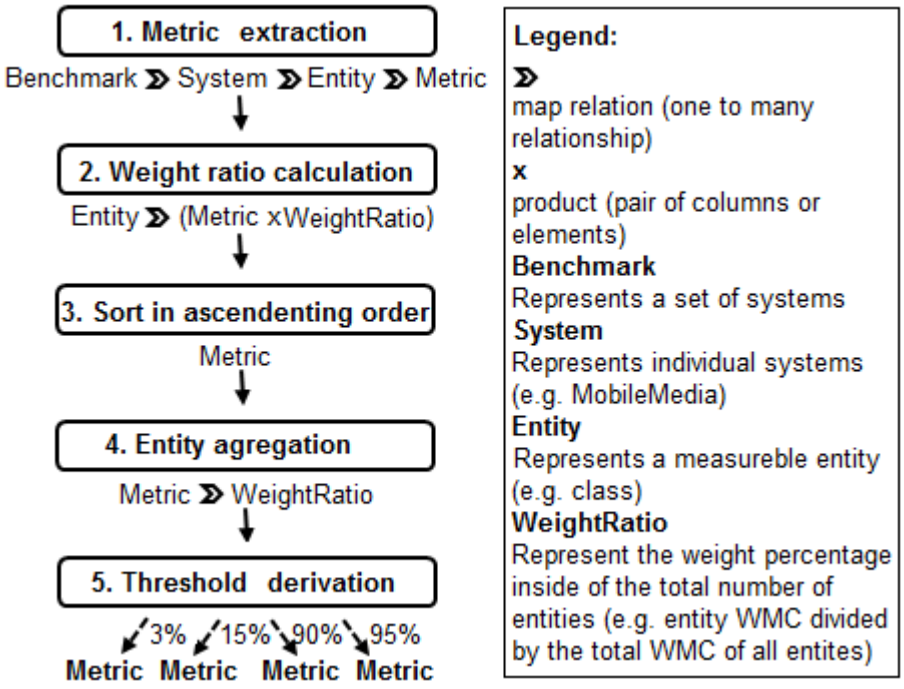

Figure 4.1: Summary of the Method Steps

entities have the same weight and the sum of all entities must be 100%. As an example, if one benchmark has 10,000 entities, each entity represents 0.01% of the overall (0.01% x 10,000 = 100%).

3. Sort in ascending order: we sort the metric values in ascending order and take the maximal metric value that represents 1%, 2%, ..., 100%, of the weight. This step is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale. For instance, all entities that WMC value is 4 must come first that all metrics which WMC value is 5.

4. Entity aggregation: we aggregate all entities per metric value in the step. This aggregation is equivalent to computing a weighted histogram (the sum of all bins must be 100%). As an example, if we have four entities with WMC value of 4 and each entity representing 0.01%, it corresponds to 0.04% of all entities.

5. Threshold derivation: finally, 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 X. This threshold is meaningful, since not only 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 believe 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, where X1 $\leq$ X2 $\leq$ X3 $\leq$ X4. This step allows identifying metrics value to be defined in long-term, medium-term, and short-term. Furthermore, these percentiles can be 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%).

4.2 Addressing the Eight Desirable Points

This section discusses the decisions we made to propose our method, called Vale's method. This discussion mainly justifies why we believe the proposed method addresses all the eight desirable points. Following the eight desirable points a method should: (i) be well-defined and deterministic; (ii) derive thresholds in a step-wise format; (iii) calculate upper and lower thresholds; (iv) be weakly dependent on the number of systems; (v) be strongly dependent on the number of entities; (vi) not correlate metrics; (vii) provide representative thresholds independent of metric distribution, and (viii) provide tool support.

First, we think that the proposed method is well-defined because we have well descripted and structured steps. On the other hand, despite of recommending the default percentages, we give freedom for the users to change the percentages if they want. It makes the proposed method partially deterministic, like Alves's Method. Addressing the second desirable point, our method presents the thresholds in a step-wise format with the labels *very low*, *low*, *moderate*, *high*, and *very high*. Additionally, addressing the third desirable point and, differently of the related methods, our method calculates lower bound thresholds captured by *very low* and *low* labels.

The fourth and fifth desirable points are related to the dependence with the number of systems and the number of entities. We consider that our method has a strong dependence with the number of entities because it uses this information to derive thresholds. On the other hand, the proposed method has a weak dependence with the number of systems because it is only a consequence related to the number of entities. Related to the sixth desirable point, our method does not correlate metrics; therefore, it fits this desirable point.

The seventh is the most difficult desirable point to fit, but we are going to present some arguments to explain why we think that our method provide representative thresholds independent to the metric distribution. The thresholds are derived to find outliers in a system. If the statistical properties of metrics are changed, the derived thresholds can include wrong outliers. Hence, it is believed that a good method

should analyze the metric distributions without changing anything in their statistical properties. It implies in neither correlating metrics nor weighting entities differently (all entities of all systems should have the same importance). Therefore, the analysis of the metric distributions can be done by viewing the metrics distribution in different ways. For example, an alternative way to examine a distribution of values is to plot its Probability Density Function (PDF).

Figure 4.2 depicts the distribution of the CBO and WMC values for Benchmark 3 (Section 3.1.2), 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 the percentage of observations (percentage of entities). The use of 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 the size of the benchmark making it possible to compare different distributions [Alves et al., 2010]. In Figure 4.2(a), we can observe that 68.35% of entities have a CBO value ≤ 5. In Figure 4.2(b), we can observe that 90% of entities have a WMC value < 19. Nevertheless, 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 rigid, but as can be seen, it is not.

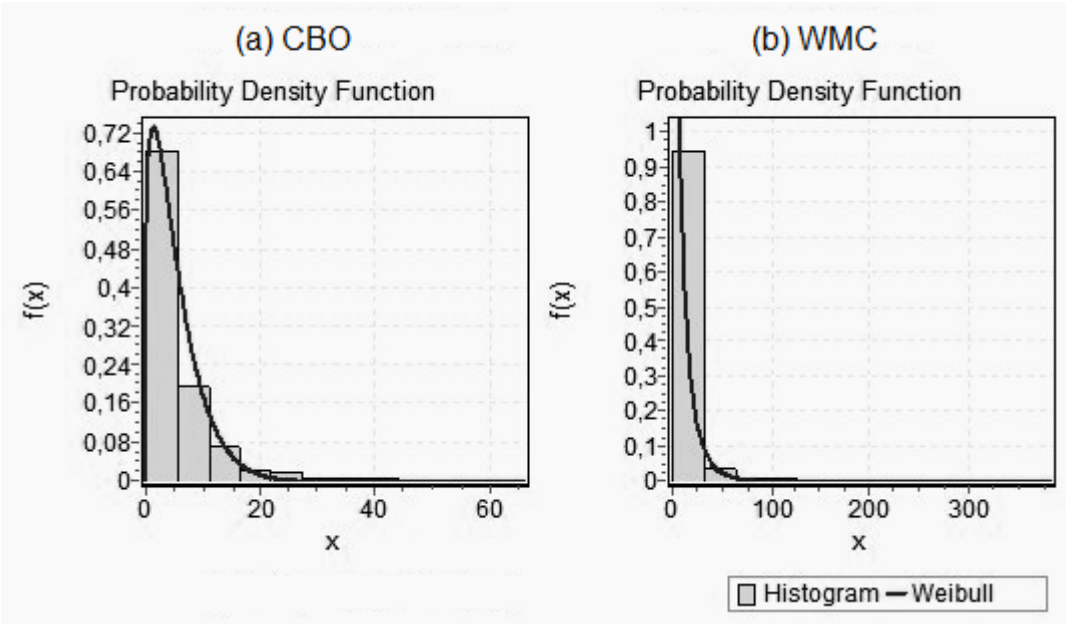

Figure 4.2: Probably Density Function

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. Similarly it 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 have (or close to have) a heavy-tailed distribution (with shape parameter equals to 1.2244 and 0.72041, respectively to CBO and WMC). By analyzing these data, it 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 the selected metrics do not follow a normal distribution (see Table 3.3). Additionally, several studies show that different software metrics follow heavy-tailed distribution [Alves et al., 2010; Concas et al., 2007; Louridas et al., 2008]. Nevertheless, if a metric does not follow a heavy-tailed distribution, are the derived thresholds from the proposed method valid? The proposed method takes into account the metric distributions focused on how to identify outliers. Hence, if the metric follows a normal distribution or a common value distribution, for example, the outliers will be identified using the proposed labels. As a concrete example, Figure 4.3 presents the DIT (Depth of Inheritance Tree) metric from the benchmark of Ferreira et al. [2012] which has a common value distribution, like Poisson, (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 identifying discrepant values (outliers). Therefore, we consider that our method derives reliable thresholds.

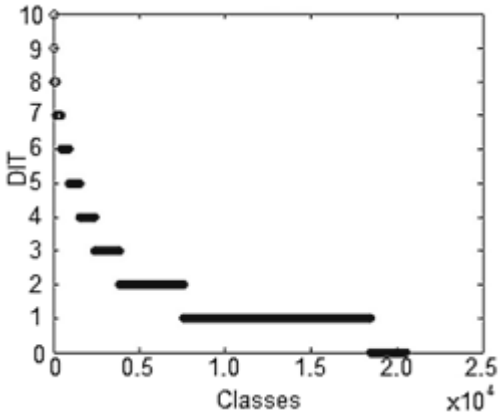

Figure 4.3: Probably Density Function [Ferreira et al., 2012]

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 the proposed method is a good way to derive thresholds with different metric distributions.

Finally, the eighth desirable point is related to tool support. The next section presents a tool to support the proposed method, called TDTool. Therefore, the proposed method fits all the eight desirable points at least partially.

4.3 Tool Support

We provide a tool to derive metric thresholds, called TDTool. The proposed tool is able to execute not only our method, but also three other methods (Alves's, Ferreira's, and, Oliveira's Methods). Figure 4.4 presents the architecture of TDTool. The tool expects a set of CSV files which each file must represent the measures of the entities of a system. For a benchmark with 100 systems, 100 CSV files are expected. The results of TDTool are presented on screen and they also can be exported in CSV format.

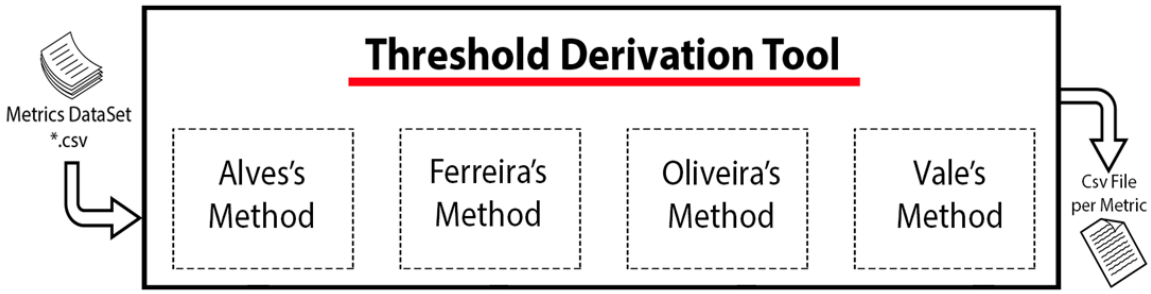

Figure 4.4: TDTool Overview

Figure 4.5 presents the main four stages of the execution of TDTool. The stages are: *method selection*, *configuration*, *processing*, and *presentation*. In the method selection stage, it is shown to the tool user a summary of all methods available and for each method an execution button. After choosing a method, the tool opens another screen which the method can be executed. We chose the proposed method (Vale's method) to illustrate the use of TDTool. The same view is presented for the other three methods.

In the *configuration* stage, the tool user must select the files which compose the benchmark and, after, the user should select metrics that the user wants to derive thresholds. Each file must represent a software system. TDTool accepts CSV files and

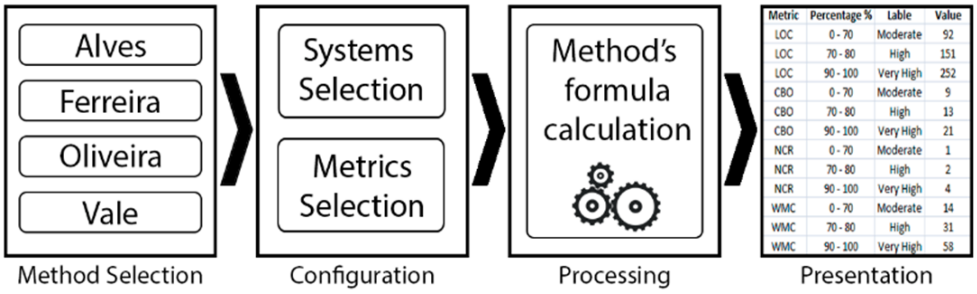

Figure 4.5: TDTool Stages

it is expected files in the following structure: each column must represent a metric and each row must represent an entity. Figure 4.6 shows an example of structure for the expected file to derive thresholds for four metrics (LOC, CBO, WMC, and NCR). TDTool does not depend on the way the metrics have been calculated. It only needs to receive as input the file in the expected format.

Component	LoC	CBO	WMC	NCR
Maler.jak	8	2	1	0
Option.jak	27	2	3	0
Maler.jak	300	12	63	16
TankManager.jak	11	2	4	0
Menu.jak	254	7	66	0
KeyMonitor.jak	63	4	28	0
GameManager.jak	2	1	0	0
Missile.jak	6	2	1	0
Tank.jak	6	2	1	0
ExplodierenEffekt.jak	126	3	18	0
TankManager.jak	6	2	2	0
Maler.jak	321	8	104	0
Tank.jak	15	3	3	0
Missile.jak	13	4	3	0
Tank.jak	41	4	13	0
TankManager.jak	18	4	5	0
SoundPlayer.jak	51	5	11	0
Tank.jak	49	3	16	0
Sprach.jak	18	1	0	0
Sprach.jak	18	1	0	0
Maler.jak	8	2	1	0
InfoPanel.jak	37	6	6	0
Option.jak	27	2	3	0
Maler.jak	304	18	58	16
Menu.jak	293	12	67	0
MIDlet.jak	1	0	0	0

Figure 4.6: Example of Input File for TDTool

Figure 4.7 presents the selection metrics view. TDTool identified Component, LOC, CBO, WMC, and NCR as metrics. In spite of that, the first column represents the entities' name. Therefore, we did not want to derive thresholds for this column. Hence, we selected only LOC, CBO, WMC, and NCR.

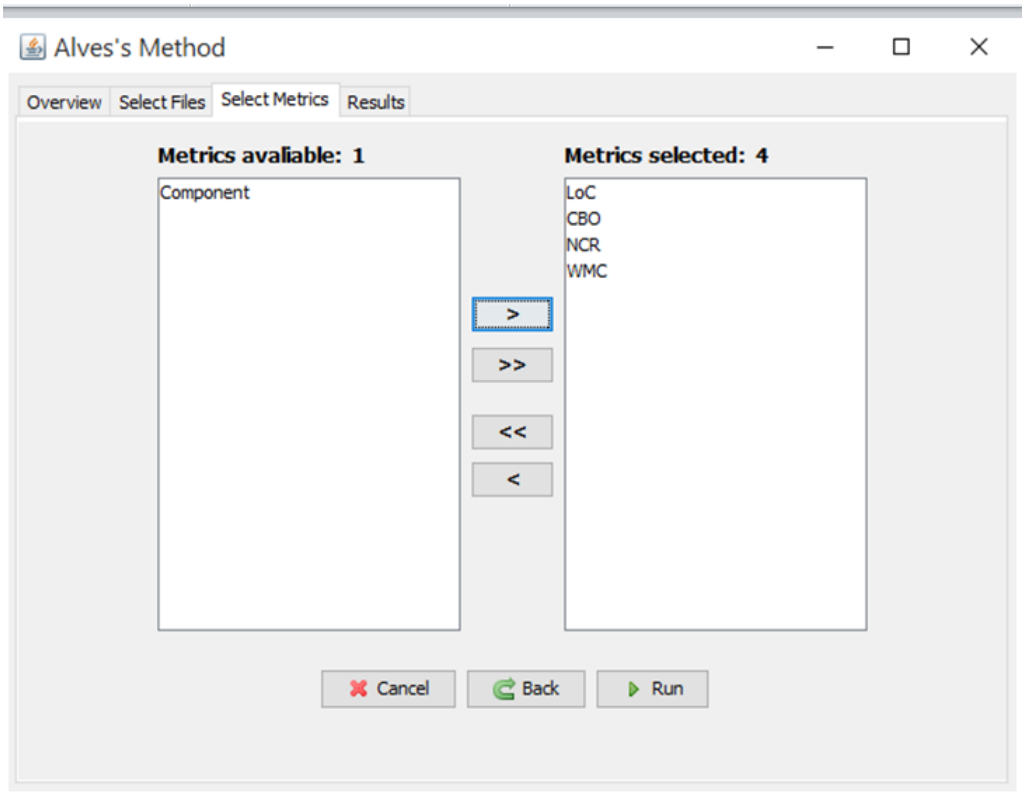

Figure 4.7: Metrics Selection

The *processing* stage is responsible for deriving the thresholds. Each method derives following its descriptions. The descriptions of Alves's, Ferreira's, and Oliveira's methods are in Section 2.6 and the description of the proposed method is in Section 4.1.

Finally, in the *presentation* stage, the results are shown in a table. The table summarizes the derived thresholds for a method. In particular, Figure 4.8 presents the results of Vale's Method for CBO, LOC, NCR, and WMC. As described in Section 4.1, Vale's Method provides thresholds in step-wise format and because of that, more than one threshold is presented for each metric.

For each stage the user can close the tool, return to the previous stage or go to the next one. In the final stage, the user can export the results clicking in the Save button. The results are saved in a CSV file containing all the metrics with their respective percentage. TDTool is an open-source project and available at our research group website[1].

[1]http://labsoft.dcc.ufmg.br/doku.php?id=about:tdtool

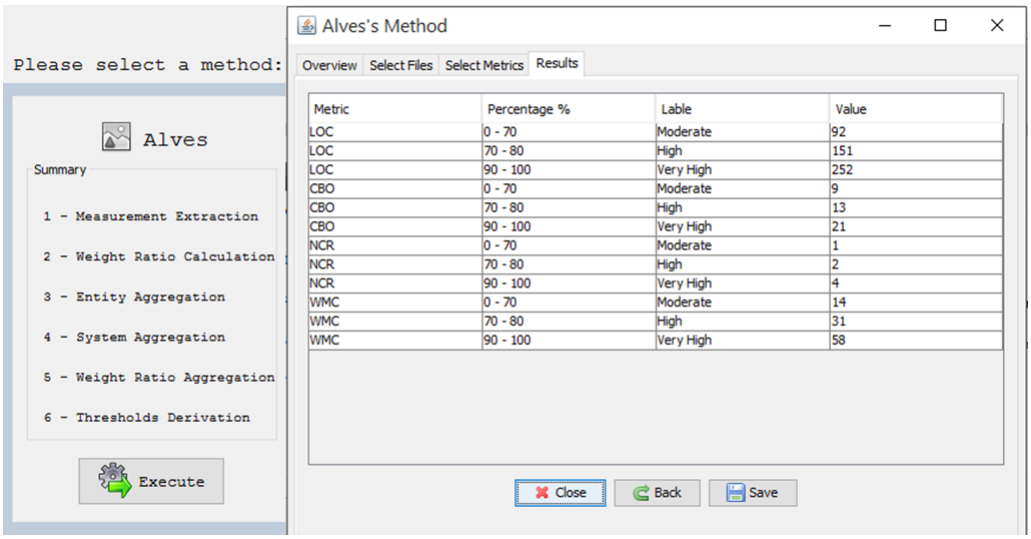

Figure 4.8: Final Results

In order to evaluate the thresholds provided by the TDTool, we derive thresholds for the benchmarks previously presented in Section 3.1. We compared the thresholds obtained manually with the thresholds obtained by TDTool, the thresholds are the same. Therefore, we believe that TDTool works correctly.

4.4 Example of Use

The proposed method can be applied in different ways, such as using SIG quality model [Heitlager et al., 2007], using metrics individually, or using metric-based detection strategies [Fowler et al., 1999]. Generally, it is necessary six steps to derive metric thresholds in the proposed method: (i) to have a benchmark composed by software systems, (ii) to choose a set of metrics to derive thresholds, (iii) to choose a tool able to extract these metrics value from each system of the target benchmark. Additionally, as we decided to exemplify our method using metric-based detection strategies, it is still necessary: (iv) to select metric-based detection strategies, (v) to select a system to identify anomalies, and, (vi) to have an oracle of the bad smells instances, if it is required to know the effectiveness of the detection.

We are going to present a complete example. Hence, we are going to use the SPL benchmark (Section 3.1), the set of four metrics (LOC, WMC, CBO, NCR) described in Section 2.2, and TDTool (4.3). The rest of this section is organized as follows. Section 4.4.1 presents the metric-based detection strategies used to exemplify our method. Section 4.4.2 presents the target system, why we selected it and the oracle of the target bad smells. Section 4.4.3 presents the derived thresholds using the chosen benchmarks

to the selected metrics. Finally, Section 4.4.4 evaluates the effectiveness of the detection strategies using the derived thresholds and the target metric-based detection strategies.

4.4.1 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 [Lanza and Marinescu, 2006]. In order to overcome this limitation, metric-based detection strategies have been proposed [Marinescu, 2004]. Detection strategies are a composed logical condition, based on metrics and threshold values, which detect design fragments with specific code smells [Lanza and Marinescu, 2006]. Code smells describe a situation where there are hints that suggest a flaw in the source code [Riel, 1996]. This section illustrates detection strategies for 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 [Fowler et al., 1999]. 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 [Fowler et al., 1999]. As can be seen by this definition, Lazy Class is the opposite of God Class.

In this work, we selected detection strategies in the literature to identify God Classes [Abilio et al., 2015] and Lazy Classes [Munro, 2005] for the following reasons. First, they have been evaluated in other studies and presented good results for the detection of God Class and Lazy Class [Abilio et al., 2015; Abilio et al., 2014]. Second, these detection strategies use a straightforward way for identifying instances of God Class and Lazy Class by combining 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 4.9 shows the God Class and Lazy Class detection strategies adapted from Abilio et al. [2015] and Munro [2005], respectively. LOC, CBO, WMC, and NCR refer to the metrics used in these detection strategies. The original detection strategies rely on absolute values. However, to provide strategies more dependent of the derived thresholds, we substitute absolute values by labels, such as Low and High.

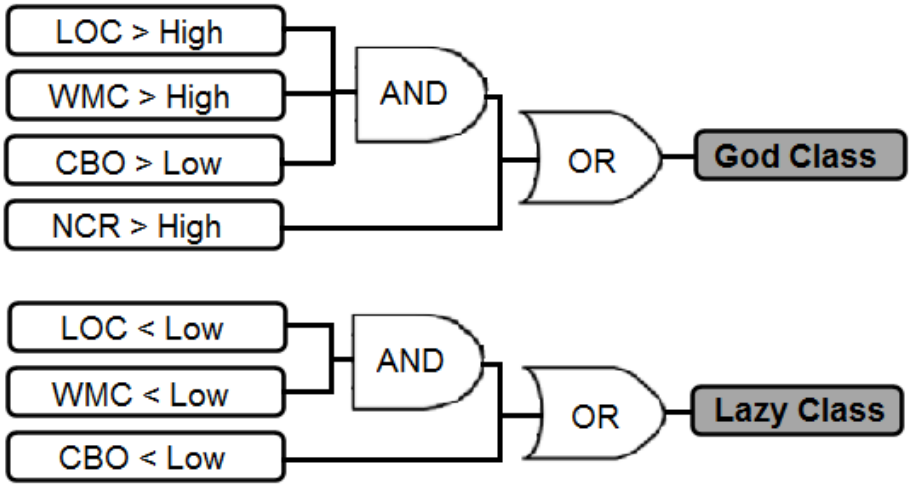

Figure 4.9: Code Smells Detection Strategy

4.4.2 Target System and Oracle of Code Smells

We choose an SPL, called MobileMedia [Figueiredo et al., 2008] to exemplify the method in practice. MobileMedia is an SPL for manipulating photos, music, and videos on mobile devices [Figueiredo et al., 2008]. It is an open source SPL implemented in several programming languages, such as Java, AspectJ, and AHEAD. We selected MobileMedia version 7 - AHEAD implementation [Ferreira et al., 2014]. This SPL was chosen because: (i) it was successfully used in other previous empirical studies [Ferreira et al., 2014; Figueiredo et al., 2008; Padilha et al., 2014], (ii) it is part of the benchmarks presented in Section 3.1, and (iii) we have access to its software developers.

The oracle can be understood as the reference model of the actual smells found in an 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 are effective on the identification of code smells in a specific SPL. In order to provide a reliable oracle, we analyzed the source code of the target SPL and the oracle of other versions of the MobileMedia developed using other technologies and languages. Only after that we defined some God Class and Lazy Class instances. This preliminary oracle has been validated by experts. The experts are two other researches that know the MobileMedia source code and the main developer of this system. The final version of the oracle was produced as a joint decision of us and the experts. Table 4.1 presents the final version of the oracle that includes seven God Class instances and ten Lazy Class instances.

Table 4.1: 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.

4.4.3 Derived Thresholds

This section presents the derived thresholds that were obtained using the proposed method to derive thresholds (Section 4.1) according to each benchmark presented in Section 3.1. The process was performed with the four metrics (LOC, WMC, CBO, and NCR) presented in Section 2.2. 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 4.2 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%). These labels are represented by the intervals 0-3, 4, 5-76, 77-138, and >138, respectively for LOC in Benchmark 1.

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.

Table 4.2: Threshold Values from the Proposed Method

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

4.4.4 Evaluation of the Derived Thresholds in Detecting Bad Smells

This section presents the results of a preliminary evaluation of effectiveness applied to the derived thresholds of the proposed method. This evaluation relies on the MobileMedia SPL and its oracle of code smells (Section 4.4.2). It is important to mention that the thresholds were derived based on the three benchmarks presented in Section 3.1.

Table 4.3 presents the results of the proposed method, summarizing the true positives (TP), false positives (FP), false negatives (FN), precision, and recall. TP and FP quantify the number of correctly and wrongly identified code smell instances by the detection strategies. FN, on the other hand, quantifies the number of code smell instances that the detection strategies missed out. Precision quantifies the rate of TP by the number of detected code anomalies (TP + FP). Recall quantifies the rate of TP by the number of existing code anomalies (TP + FN).

The derived thresholds were applied for the Benchmarks 1, 2, and 3. For instance, by using the thresholds derived in Benchmarks 1, 2, and 3 the number of TP for God Class candidates was 7 for the three benchmarks. In all cases, one FP was found. In the case of FN, we found 0 for Benchmarks 1, 2, and 3. The derived thresholds found the same values for TP, FP, and FN for Benchmarks 1, 2, and 3. In spite of that, we found different thresholds for the chosen metrics. For instance, for Benchmarks 1 and 2, the proposed method derives 77 and 78 to LOC, respectively (see Table 4.2). It happens because the MobileMedia source code does not have any entity (class) in the interval of our thresholds. For example, we can see that the thresholds for Benchmarks

1 and 2 for LOC are 77 and 78. Applying the detection strategy the same instances were obtained because the MobileMedia source code does not have an entity with 78 Lines of Code.

Table 4.3: Identification of Code Smells Based on Thresholds Derived From the Proposed Method

Code Smell	#	**Benchmark**		
		1	**2**	**3**
God Class	TP	7	7	7
	FP	1	1	1
	FN	0	0	0
	Precision	87.5	87.5	87.5
	Recall	100	100	100
Lazy Class	TP	9	9	9
	FP	0	0	0
	FN	1	1	1
	Precision	100	100	100
	Recall	90.0	90.0	90.0

We can observe in Table 4.3 that the precision is 87.5% to Benchmarks 1, 2, and 3, for the detection strategy to identify God Class instances. For the detection strategy that aims to identify Lazy Class instances, the values of recall and precision are 100% and 90%, respectively, for the three benchmarks. We considered that the identification of God Class instances was better because recall is considered more useful than precision in the context of identification of code smells as recall is a measure of completeness [Padilha et al., 2014]. That is, high recall means that the detection strategy was able to identify a high number of code smells in software.

4.5 Threats to Validity

Even with the careful planning, this preliminary evaluation can be affected by different factors which, while extraneous to the concerns of the research, can invalidate its main findings. Some actions to mitigate the weakness of the proposed method, TDTool, and the example of use of the proposed method are described, as follows.

Metric Labels for the Proposed Method – 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 for all systems and benchmarks. But, to try generalizing and providing default labels, we decide to use these percentages. In addition, it can be seen that very low and low labels might be equals or similar values.

In spite of that, we prefer keeping both labels and increase their difference in terms of percentages. For most metrics, 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 percentages were defined based on our experience analyzing some metric distributions. If someone thinks that these values do not fit well in their metric distribution, other percentages can be used.

TDTool evaluation – We compared the results calculated manually with the thresholds automatically calculated by TDTool. If some mistake occurred in both cases (manually or automatically), we may have achieved wrong thresholds. To minimize this problem, we calculate the thresholds for three different benchmarks. Therefore, if some mistake happened, it happened in the three cases. We think that it is unlikely since we had calculated the thresholds for three benchmarks.

Code smell – We discuss only two code smells (i.e., God Classes and Lazy Classes). Fowler et al. [1999] has cataloged a list with twenty two code smells. Therefore, these smells used to evaluate the effectiveness of 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 labels of the used metric. It can have affected the evaluation, but we assume that we made a fair decision. The detection strategy chosen in this work might have influenced the results. For example, as NCR has greater influence in the strategy of God Class 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.

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.

4.6 Final Remarks

This chapter described our method to derive thresholds. Additionally, we discussed each point to justify our decisions, presented TDTool, and presented a complete example of use. We believe that the proposed method fits all the eight desirable points. In the example of use of our method, we derived thresholds from three SPL benchmarks

for four metrics and identified God Class and Lazy Class instances in an SPL, called MobileMedia. We also performed a preliminary evaluation of the proposed method in terms of its effectiveness. Our results indicate good (90% and 100%) recall and precision for both code smells (God Class and Lazy Class), being better to God Class.

The next chapter presents more consistent evaluations of the proposed method. While in this chapter we present an example of use, in the next chapter we compared the derived thresholds of the proposed method with other methods. Additionally, present a scalability study using a different benchmark composed by single software systems developed in Java and a summarization of our previous results.

Chapter 5

Evaluation of the Proposed Method

In Chapter 3, we compared three methods to derive metric thresholds. Based on desirable points extracted from this comparison, we proposed, described, and exemplify a method to derive metric thresholds in Chapter 4. Now, in this chapter, we evaluate and compare the proposed method with the three methods compared on Chapter 3.

This evaluation occurs in three different ways. First, in Section 5.1, we analyze the precision and recall (effectiveness) for detection of God Class instances using thresholds derived by the four methods (Alves's, Ferreira's, Oliveira's, and Vale's methods). Second, in Section 5.2, we derive the thresholds for a Java benchmark using the four methods and analyze the derived thresholds. Finally, in Section 5.3, we put the thresholds derived for these four methods side by side (like figure 3.1) for two benchmarks, one composed by SPLs (Benchmark 3) and another composed by Java systems (Benchmark 4). This chapter still describes some threats to validity of our work (Section 5.4) and final remarks (Section 5.5).

5.1 Comparison of God Class Instances

This section presents a comparison on the effectiveness of the four methods explored in this dissertation (Alves's, Ferreira's, Oliveira's, and Vale's methods). For this comparison, we need: (i) to derive thresholds for each method, (ii) to select a smell and its metric-based detection strategy, (iii) to select a system, and (iv) to have an oracle of bad smell instances. Therefore, we use the thresholds presented in Section 3.2.3 for Alves's, Ferreira's, and Oliveira's methods and the thresholds presented in Section 4.3.3 for Vale's method. Additionally, we use the metric-based detection strategy of God Class presented in Section 4.4.1, the MobileMedia SPL and the Oracle of God Class instances presented in Section 4.4.2. In other words, we get the results of previ-

ous chapters, put these results together and realize this first evaluation. We would like to highlight that the metric-based detection strategy of God Class used was selected from a previous study [Abilio et al., 2015].

Table 5.1 describes the results per method, summarizing true positives (TP), false positives (FP), false negatives (FN), precision, and recall. 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. Precision (P) quantifies the rate of TP by the number of *detected code anomalies* (TP + FP). Finally, recall (R) quantifies the rate of TP by the number of *existing code anomalies* (TP + FN).

Table 5.1: Identification of God Classes based on Derived Thresholds of Each Method

#	**Method**											
	Alves			**Ferreira**			**Oliveira**			**Vale**		
	Benchmark			**Benchmark**			**Benchmark**			**Benchmark**		
	1	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
TP	6	5	5	7	7	7	7	7	6	7	7	7
FP	3	8	8	12	12	12	8	4	3	1	1	1
FN	1	2	2	0	0	0	0	0	1	0	0	0
P	66.7	38.5	38.5	36.8	36.8	36.8	46.7	63.8	66.7	87.5	87.5	87.5
R	85.7	71.4	71.4	100	100	100	100	100	85.7	100	100	100

The thresholds were derived from Benchmarks 1, 2, and 3 in the other chapters of this dissertation. For instance, by using the thresholds derived by Alves's method in Benchmarks 1, 2, and 3 the number of FP was 3, 8, and 8, respectively. Our oracle has 7 instances (see Table 4.1) and in the case of Alves's method we found 6, 5, and 5 true positive instances to Benchmarks 1, 2, and 3, respectively. We found 1, 2, and 2 false negatives. Looking at the results of the same method we found 66.7%, 38.5%, and 38.5% of precision and 85.7%, 71.4%, and 71.4% of recall for Benchmarks 1, 2, and 3, respectively.

In this evaluation, Vale's and Ferreira's methods achieved the best performance in terms of recall measure. This result is due to the fact that the thresholds values from these two methods are not too high when compared to Alves's thresholds. Similarly, Alves's thresholds are not too bad being 85.7% in the worst case. Ferreira's method derived the lowest thresholds for the majority of metrics and, because of that it got one of the higher recall values to the three benchmarks. On the other hand, Alves's method provide the highest thresholds for the majority of metrics, consequently, this method got the lowest recall.

Regarding precision values, we observed Vale's method has good rates in the three benchmarks. For this method, we found precision higher than 85%. Alves's method achieved the worst performance in terms of precision. In the case of Alves's method, NCR metric was responsible for the lowest precision to Benchmarks 1, 2, and 3. For instance, if the thresholds were three to Benchmark 2 and 3, like Vale's method, the precision would have increased to 71.4% in both cases (rather than 38.4%). NCR was not the unique responsible for the low values of precision for Ferreira's method, but this metric has a strong influence in such measures. Given these two examples, we assume that NCR has too much importance in the selected god class detection strategy, because LOC, CBO, and WMC are combined, but NCR is not (see Figure 4.9). This is the main fact that we get the low values of precision to Alves's and Ferreira's methods.

Summarizing, NCR has too much importance in the used God Class detection strategy. A solution to minimize this problem would be to use higher labels to NCR (for example, very high label instead of high label). However, in an attempt to be fair to all methods, we use only the values considered high in the proposed method. This strategy may have favored the Ferreira's method and/or affected Alves's method in terms of recall and precision, respectively. This argument is not valid to Oliveira's because this method provides a unique threshold (not thresholds in a step-wise format). In the case of Vale's method change to very high label would reduce recall values, but increase the precision to 100%. Given the limitations of three out of these four methods do not derive lower bound thresholds and one out of these four methods do not derive thresholds in a step-wise format; we did not calculate the effectiveness of Lazy Class instances.

It is important to mention that the thresholds used in a detection strategy directly impact on detected code smells obtained by each method. Accordingly, recall and precision values are also impacted by the threshold values used in the detection strategy. On the other hand, the detection strategy directly impact on the detected code smells obtained by each method. Hence, we provided an analysis difficult to be generalized, but in this case Vale's method presented overall better results. Probably for another detection strategy another method can be better. Given the importance of the detection strategy and the thresholds, it is important to select a method to derive thresholds and a detection strategy previously evaluated and reliable to get correct results.

5.2 Scalability Study

We also conducted a scalability study to evaluate our method using another benchmark (Benchmark 4). We chose a well-known benchmark of Java systems, called Qualitas Corpus [Oliveira et al. 2014]. We chose this benchmark because only "real" software systems compose it (not toy or prototype systems). This benchmark has more than a hundred systems and most of these systems are larger and more complex than the SPLs of the previously used benchmarks. In addition, systems from Qualitas Corpus were developed in another programming language (Java). We planned this evaluation to see if the thresholds derived from the four methods follow a behavior, for example with larger systems they are higher, to analyze the results in another scale (larger benchmark with larger systems, analyzing more metrics, etc.), and if the proposed method works perfectly out of the context of SPLs.

We used the 20101126 release, composed by 106 open source Java software systems. For each system, the corpus presents a set of 21 software metrics, 20 of them are numeric values. In this study, we aim to derive thresholds for a subset of seven metrics: LOC, CBO, WMC, DIT, LCOM, NOC, and RFC. These metrics were described on Section 2.2. From the 106 systems available in Qualitas Corpus, only three of them do not have all these metrics computed. These systems are Eclipse 3.7.1, JRE 1.6.0, and Netbeans 7.3. Therefore, we discarded them from this study. We then derived thresholds using the four methods (Alves, Ferreira, Oliveira, Vale's methods) to 103 systems. The 103 systems have 94,393 entities together. These entities are used to derive thresholds for the four methods.

Table 5.2 shows the obtained threshold values for the four methods. For example, for LOC using Alves's method, we found thresholds equals to 565, 901, and 1650 to the percentages 70, 80 and, 90, respectively. Note that, we present the key values for each method, like we did in Chapters 3 and 4.

It is difficult comment the thresholds of these seven metrics for these four methods because each one present the thresholds in a different format. In spite of that, we can note that the thresholds do not vary too much for metrics with a common value distribution, such as NOC and DIT. In the case of NOC, 87.4% of the entities have NOC equals to 0. In the case of DIT, 50.2% of the entities have DIT equals to 1. These numbers of equal values characterizes a common distribution for these metrics. NOC varies from 0 to 670 and DIT from 0 to 12. Note that despite of NOC has a greater variation than DIT, the thresholds vary from 0 to 2 in the maximum case.

Table 5.2: Threshold Value from Four Studied Methods

Method	Label	LOC	CBO	DIT	LCOM	NOC	RFC	WMC
Alves	70	565	29	2	90	0	84	75
	80	901	39	2	240	0	119	123
	90	1650	59	3	851	1	191	233
Ferreira	1	19	3	1	0	0	6	3
	3	131	14	3	14	0	30	19
Oliveira	-	222	21	3	39	1	50	41
Vale	3	3	0	1	0	0	1	1
	15	11	2	1	0	0	3	2
	90	308	24	4	66	1	58	42
	95	510	33	5	186	2	85	70

To present a more direct discussion, we are going to use the thresholds with label *high* for Alves's and Vale's methods, with label *bad* for Ferreira's method, and the obtained thresholds for Oliveira's method. The *high* label represents 80% of the entities in the case of Alves's method and 90% of the entities in the case of Vale's method. It is important to highlight that Alves's method weight the entities' values with LOC and it changes a little the metric distribution. The *bad* label of Ferreira's method represents the third quartile (75%) of the entities in this section. Oliveira's method uses a defined formula. We are going to represent these labels as *high* for the four methods. For more details about how the methods work, see the methods descriptions on Sections 2.6 and 4.1.

Figure 5.1 depicts the thresholds considered *high* for the four studied methods. We can see that Alves's method provides the highest thresholds in the majority of the cases. Ferreira's method provides the lower thresholds. Oliveira's and Vale's methods provide the intermediate thresholds. The greater difference between the derived thresholds from Alves' and the other methods in absolute and percentage is to LOC and LCOM metrics, respectively. The derived threshold of Alves's method for LOC is 901 while the derived threshold of Vale's method, the second higher, is 308. It represents almost three times more. LCOM, on the other hand, represents a difference of almost four times more (240 to 66).

Another point to highlight here is that, despite of NOC has almost 90% of data equals to 0, Alves's and Ferreira's failed to determine threshold 0 for this metric. We said this, because 0 is the lowest value possible for this metric. Therefore, it is not expected these threshold for a high label.

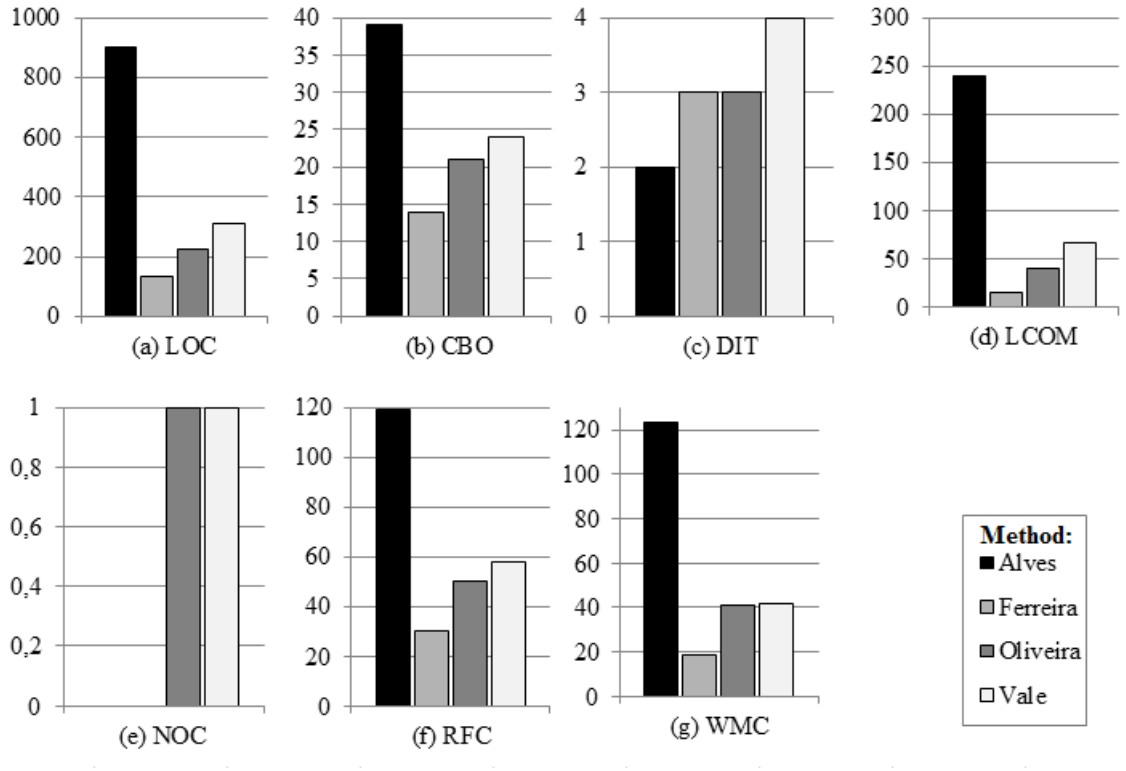

Figure 5.1: Derived Thresholds for 7 Metrics Using Benchmark 4

5.3 Discussing Previous Results

We can see that using different methods, generally, different thresholds are derived. In this section, we discuss the derived thresholds of three metrics (LOC, CBO, WMC) using two different benchmarks (an SPL benchmark and a Java benchmark). The first benchmark has fewer systems and the systems are smaller than the second benchmark. Additionally, the benchmarks have systems developed by different programming languages. While in the first one, systems are developed using AHEAD and FeatureHouse, in the second one the systems are developed in Java. Despite of AHEAD and FeatureHouse are Java-based languages, they have different functions (such as stepwise refinement) and are feature-oriented programming languages. With this study we aim to show the difference and a common behavior for the thresholds derived from different benchmarks.

These different functions and programming languages justify the different thresholds. However, we believe that even using the same language different thresholds can be obtained, such as the derived thresholds from benchmark 1, 2, and 3, but it is dependent to the target benchmark. In other words, we want to highlight that we did not believe in universal thresholds even using the same programming language. We

believe that thresholds are dependent on the used benchmarks and, it is better when it is possible to build a benchmark with systems similar to target system that is going to be evaluated, for example, systems in a same domain and similar sizes. This is one of the reasons we believe that this task should be easier as possible to do. Because of that we provide tool support for the studied methods.

This section presents the thresholds using the same method side-by-side, but derived from different benchmarks. We aim to highlight the difference of the derived thresholds using different benchmarks. In Chapter 3, we derived thresholds using different subgroups of an SPL benchmark. In this section, we put the derived thresholds from benchmarks composed by different types of systems (SPLs and single systems) side-by-side. It is important to highlight that despite of SPLs give the idea that are larger than single systems, in our case, the majority of single systems are bigger than the SPLs of our SPL benchmark.

Figure 5.2 presents thresholds of LOC, CBO, and WMC for an SPL benchmark with 14 feature-oriented SPLs (Benchmark 3) and another benchmark composed by 103 Java systems (Benchmark 4). We can see that for all metrics and methods the thresholds derived from the Java benchmark are higher than the SPL benchmark. In some cases, we found a large difference, such as the case of LOC, and in other cases we found a smaller difference, such as the cases of CBO and WMC. It is justified, mainly, because the systems in the Java benchmark are larger than the systems of the SPL Benchmark. In Chapter 3, we could see that the thresholds tend to increase when a benchmark with larger systems is selected. Now, we can see in a greater scale that it also happens.

5.4 Threats to Validity

Even with the careful planning, the evaluations presented in this chapter can be affected by different factors which, while extraneous to the concerns of the research, can invalidate its main findings. Some actions to mitigate the threats of our evaluations are described, as follows.

Related to the first evaluation, we took some cares described in Section 4.5. In addition to these cases, we compare the effectiveness of the thresholds derived from four different methods, but using the same detection strategy. In that analysis, a method fared better. Probably, if we use another detection strategy or another bad smell other method can fare better. Our results cannot be generalized for all bad smells.

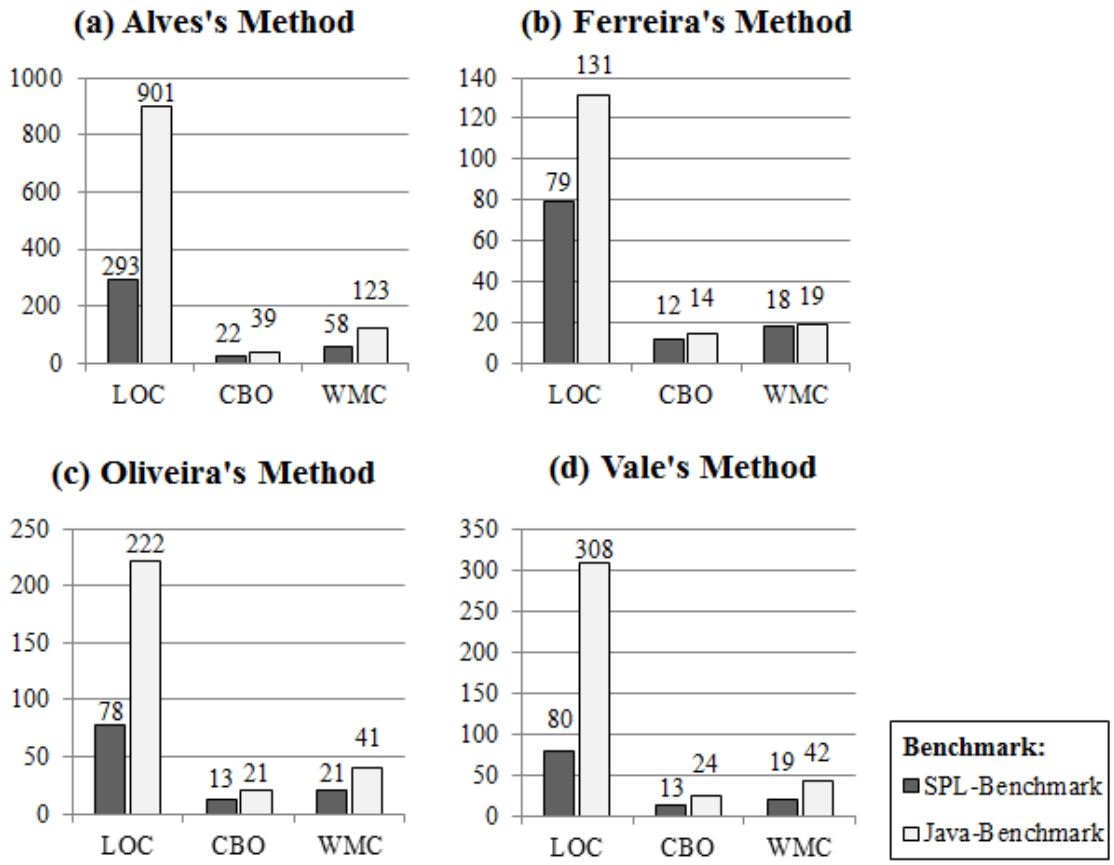

Figure 5.2: Derived Thresholds from SPL and Java Benchmark Side-by-Side

In the second evaluation, we got the Qualitas Corpus metrics already calculated. All metrics are available online and we believe that they are correct, but we are not sure. The third evaluation strengthens some results of our dissertation. As we only discuss previous results, some threats to validity this evaluation can already have been mentioned on previous sections (such as, Sections 3.4 or 4.5).

5.5 Final Remarks

This chapter compared the four methods to derive thresholds explored in this dissertation (Alves's, Ferreira's, Oliveira's, and Vale's methods). We conducted three evaluations. In the first evaluation, we compared the derived thresholds using a metric-based detection strategy. This metric-based detection strategy aims to identify a bad smell called God Class. This strategy is composed by four metrics (LOC, CBO, WMC, and NCR). Vale's method fared better in that evaluation of effectiveness (precision and recall). We also could conclude that the selected detection strategy gives too much

importance to NCR metric. We suggested changing the label of this metric to *very high* instead of *high* to reduce the importance of such metric in the detection strategy.

In the second evaluation, we provided a scalability study. Previously, we presented three benchmarks composed by software product lines (Benchmarks 1, 2, and 3). In the second evaluation, we derived thresholds for another benchmark composed by Java systems (Benchmark 4). We considered a scalability study because all systems from the Java benchmark are, in general, bigger than the systems from the SPL benchmarks and, the systems were developed in another technology and language (Java – object-oriented programming instead of AHEAD and FeatureHouse – feature-oriented programming). With this evaluation, we can see that the methods work similarly to the other benchmark and the derived thresholds are different from one to other methods. Alves's method carries on deriving higher thresholds and metrics such as DIT and NOC, which have a common value distribution the thresholds from the different methods, are more similar.

In the third evaluation, we compared the derived thresholds of Benchmarks 3 and 4. We could see that the derived thresholds from Benchmark 4 (Java benchmark) are higher than the thresholds from Benchmark 3. In the case of LCOM the method which derives the highest thresholds (Alves's method), derived thresholds almost four times higher than the second higher thresholds.

The next chapter concludes this dissertation by highlighting our main findings. It also describes our intended contributions, gives directions to future work, and summarizes some published papers in the master period.

Chapter 6

Final Considerations

This chapter presents our conclusions, contributions, published papers directly or indirectly related to this dissertation, and directions for future work.

6.1 Conclusion

In this dissertation, we can see that metrics are a practical means in the process of measurement of software quality. Additionally, we can see that to this process be effective, it is directly dependent on the definition of appropriate thresholds. Starting to study this topic of threshold calculation, we can see that it is an open topic to explore, because, despite of many methods have been proposed, we did not find a consensus of researches and practitioners. Additionally, we can see that this topic have being studied since a long time ago. To get a consensus of them, in the recent years, more reliable methods have been proposed. We believe on that, because recent proposed methods consider three points (well-defined methods, methods which consider the skewed distribution of software metrics, and benchmark-based). We consider these three points fundamental for this kind of method and, because of that we called them of three key points.

Aiming to make easier the choice of a method to derive metric thresholds, we performed a comparison of method which addresses the three key points. Hence, we compared Alves's, Ferreira's, and Oliveira's methods. To compare these methods we built three benchmarks composed by software product lines. Additionally, we explore each characteristic of these methods, such as correlate metrics and provide thresholds in a step-wise format. As main results, we can see that Alves's method derives higher thresholds and, consequently, the number of outliers is smaller than the other methods. With this perspective we consider the thresholds of this method better than the

thresholds than other methods. Additionally, we described eight desirable points for this kind of method and we saw the opportunity to propose a method getting the best of each method.

Therefore, we propose Vale's method. Additionally to Vale's method, we propose TDTool. TDTool a tool able to support the proposed method (Vale's method) and other three methods previously compared (Alves's, Ferreira's, and Oliveira's methods). The idea of proposing a tool able to support the four methods is to make easier the threshold calculation independent of the used method. The proposed method and TDTool were descripted, exemplified, and evaluated. In our description, we justified each decision point of our method, such as why we chose that percentages to represent the labels *very low*, *low*, *moderate*, *high*, and *very high*. We exemplified our method with two metric-based detection strategies, one for detect God Class instances and another to detect Lazy Class instances.

The evaluation of the proposed method occurred in three ways. First, we compared the effectiveness of the derived thresholds from the SPL benchmarks using a detection strategy of God Class. Second, we derived the thresholds using a different type of benchmark composed by single-systems developed in Java. Third, we discuss the derived thresholds of Benchmark 3 and 4, discussing our previous results.

As results of the evaluations of this dissertation, we can see that Vale's method fared better in the first evaluation. In the second evaluation, we can see that the studied methods work similarly to another benchmark. Discussing previous results, in the third evaluation, we showed that the derived thresholds from the Java benchmark are higher than the thresholds from the SPL benchmarks for all methods and metrics. Therefore, as the Java benchmark is larger than the SPL benchmarks and the systems from the Java benchmark are higher than the systems from the SPL benchmark the thresholds seem to be reliable. In addition, TDTool works correctly when compared to the thresholds derived manually.

Summarizing the research questions, we find many methods to derive metric thresholds, such as Erni et al. [1996], Lanza and Marinescu [2006], Chidamber and Kemerer [1994], Spinellis [2008], Alves et al., 2010, Ferreira et al., 2012, Oliveira et al., 2014, and other method, see Section 2.4. The more reliable methods follow three key points: well-defined methods, methods that consider the skewed distribution of software measurements, and benchmark-based. With the comparative study, we can see eight desirable points for this kind of method: (i) be well-defined and deterministic; (ii) derive thresholds in a step-wise format; (iii) be weakly dependent on the number of systems; (iv) be strongly dependent on the number of entities; (v) not correlate metrics; (vi) calculate upper and lower thresholds; (vii) provide representative thresh-

olds independent of metric distribution, and (viii) provide tool support. And finally, given that in the evaluations our method fared better we recommend use it to derive thresholds.

To conclude our dissertation and highlight our increments in the literature, we provide an overview of the four studied methods in this dissertation. In Section 3.3, we presented a comparative table summarizing the strangeness and weakness of Alves's, Ferreira's, and Oliveira's methods (Table 3.7). Now, we provide such table actualized. Table 6.1 shows the strangeness and weakness of Alves's, Ferreira's, Oliveira's, and Vale's method. When comparing Tables 3.7 and 6.1, we add Vale's method and tool support for Alves's and Ferreira's methods (underlined lines means contribution to the state of art).

Table 6.1: Evolution in the Threshold Calculation Literature

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

Therefore, we believe that we achieved all our goals because we answered the research question, provide an overview of method to derive thresholds, compared three methods and, given the needs to propose a method with address all the eight desirable points, we proposed our own method, called Vale's Method. Additionally, we provided a tool, called TDTool, able to run all four studied methods in this dissertation.

6.2 Contribution

We consider four main contributions in this dissertation, as follows.

- The literature review and overview of methods to derive thresholds;
- The proposal of a new method to derive thresholds;
- The comparison of four methods to derive metric thresholds, and;

- The tool, called TDTool, to support our method (Vale's method) and other three methods (Alves's, Fereira's and Oliveira's methods).

6.3 Publication Results

This section presents 9 published papers directly and indirectly related to this dissertation. The first two papers are part of this dissertation. The other papers are related, but are not part of this dissertation. Nevertheless, they were fundamental to increase the quality of this dissertation.

1. Vale, G.; Albuquerque, D.; Figueiredo, E.; Garcia, A. **Defining metric thresholds for software product lines**. *In: the 19th International Conference, 2015, Nashville. Proceedings of the 19th International Conference on Software Product Line - SPLC '15.* New York: ACM Press. p. 176.

2. Vale, G. A. and Figueiredo, E. **A Method to Derive Metric Thresholds for Software Product Lines.** *In: Proceedings of 29th Brazilian Symposium on Software Engineering (SBES)*, Belo Horizonte, 2015. `* 2nd Best Paper *`

3. Vale, G.; Abilio, R.; Freire, Andre; Costa, H. **Criteria and Guidelines to Improve Software Maintainability in Software Product Lines.** *In: 2015 12th International Conference on Information Technology New Generations (ITNG)*, 2015, Las Vegas. 2015 12th International Conference on Information Technology - New Generations. p. 427.

4. Vale, G. A. ; Borges, H. ; Figueiredo, E. ; Padua, C. **Ferramentas de Medição de Software: Um Estudo Comparativo.** *In: Workshop on Experimental Software Engineering - CIbSE*, 2015, Lima. ESELAW,

5. Reis, J. ; Vale, G.; Costa, H. **Manutenibilidade de Tecnologias para Programação de Linhas de Produtos de Software: Um Estudo Comparativo.** *In: Simpósio Brasileiro de Qualidade de Software*, 2015, Manaus. XIV Simpósio Brasileiro de Qualidade de Software, 2015. `* Best paper *`

6. Vale, G.; Figueiredo, E. **Detection and Description of Variability Smells.** *In: V Workshop de Teses e Dissertações do CBSoft (WTDSoft)*, 2015, Belo Horizonte.

7. Abilio, R.; Vale, G.; Oliveira, J.; Figueiredo, E. ; Costa, H. **Code Smell Detection Tool for Compositional-based Software Product Lines.** *In: Session tools - CBSoft, 2014, Maceio. v. 2. p. 109-116.*

8. Vale, G.; Ferreira, L; Figueiredo, E. **On the Detection of God Class in Aspect-Oriented Programming: An Empirical Study**. *In: Workshop on Software Modularity (WMod)*, Maceio, 2014. v. 2. p. 27-38.

9. Vale, G.; Figueiredo, E.; Abilio, R.; Costa, H. **Bad Smells in Software Product Lines: A Systematic Review.** *In: 2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, 2014, Maceio, p. 84-94.

6.4 Future Work

With this dissertation, we found many directions for future work. With our literature review and the comparison of methods to derive thresholds, we proposed a method. Other ways could be followed in the proposal of a method, for example, using fuzzy logic to get the groups which represent the labels. We knew a tool to measure AHEAD code (VSD tool), but we did not find any too to measure FeatureHouse code. Aiming to avoid the transformation of FeatureHouse code to AHEAD code, when it is necessary to measure FeatureHouse code (as we did in this dissertation), we plan to develop a tool to measure FeatureHouse code.

We know that benchmarks are a practical means to derive thresholds and, we know many methods to do this task using benchmarks. Despite of it, we did not find any work providing guidelines or helping software engineers to build a benchmark. In addition, we could see that the thresholds are dependent to benchmarks. Hence, low quality benchmarks can provide low quality thresholds. Thereby, we want to explore how to build representative benchmarks for software systems.

As other future work and ways to evaluate the proposed method, we intent to: use the studied methods using benchmarks composed by systems developed in other languages; analyze other bad smells and other detection strategies for the studied bad smells, and; use other quality models, such as the SIG quality model to measure the quality of software systems using the derived thresholds.

Bibliography

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

Abilio, R. , Vale, G. , Oliveira, J. , Figueiredo, J.and Costa, H. (2014). Code Smell Detection Tool for Compositional-based Software Product Lines. In *Proceedings of 21th Brazilian Conference on Software, Tools Session*. pages 109-116.

Alves, T.L. , Ypma, C. and Visser, J. (2010). Deriving Metric Thresholds From Benchmark Data. In *Proceedings of 26th International Conference on Software Maintenance (ICSM)*. pages 1-10.

Apel, S. and Kästner, C. (2009). An Overview of Feature-Oriented Software Development. In *Journal of Object Technology*. volume 8, pages 49-84.

Apel, S. , Kästner, C. and Lengauer, C. (2009). Language-Independent, Automated Software Composition.. In *Proceedings of the International Conference on Software Engineering (ICSE)*. pages 221-231.

Apel, S. , Leich, T. , Rosenmuller, M. and Saake, G. (2005). FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. pages 125-140.

Asikainen, T. , Mannisto, T. and Soininen, T. (2006). A Unified Conceptual Foundation for Feature Modelling. In *Proceedings of the International Software Product Line Conference (SPLC)*. pages 31-40.

Batory, D. (2004). Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the International Conference on Software Engineering (ICSE)*. pages 702-703.

Batory, D. (2005). A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the International Conference on Generative and Transformational Techniques in Software Engineering (GTTSE)*. pages 3-35.

Batory, D. , Johnson, C. , MacDonald, B. and von Heeder, D. (2002). Achieving extensibility through product-lines and domain-specific languages: a case study. In *ACM Transactions on Software Engineering and Methodology*. volume 11,pages 191-214.

Batory, D. , Sarvela, J. and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. In *IEEE Transactions on Software Engineering*. pages 335-371.

Brereton, P. , Kitchenham, B. , Budgen, D. , Tumer, M. and Khalil, M. (2007). Lessons From Applying the Systematic Literature Review Process within the Software Engineering Domain. In *Journal of Systems and Software*. volume 80,pages 571-583.

Chidamber, S. R. and Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. In *IEEE Transactions on Software Engineering*. volume 20, pages 476-493.

Coleman, D. , Lowther, B. and Oman, P. (1995). The Application of Software Maintainability Models in Industrial Software Systems. In *Journal on System Software*. volume 29, pages 3-16.

Concas, G. , Marchesi, M. , Pinna, S.and Serra, N. (2007). Power-Laws in a Large Object-Oriented Software System. In *IEEE Transactions on Software Engineering*. volume 33, pages 687-708.

Conejero, J.M. , et al. (2012). On the Relationship of Concern Metrics and Requirements Maintainability. In *Information and Software Technology (IST)*. volume 54, pages 212-238.

Dowdy, S. and Wearden, S. (1983). Statistics for Research. In *Wiley*. page 230.

Dumke, R.R. and Winkler, A.S. (1997). The Component-Based Software Engineering with Metrics. In *Proceedings of International Symposium on Assessment of Software Tools and Technologies*. pages 104-110.

Erni, K. and Lewerentz, C. (1996). Applying Design-Metrics to Object-Oriented Frameworks. In *Proceedings of the 3rd International Symposium on Software Metrics (METRICS)*. pages 64-72.

FeatureIDE (2016). <http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/>. In *Access in January, 2016*.

Fenton, N.E. and Pfleeger, S.L. (1998). Software Metrics: A Rigorous and Practical Approach. In *Publishing Co. Boston*. page 656.

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

Ferreira, K. , Bigonha, M. , Bigonha, R. , Mendes, L. and Almeida, H. (2012). Identifying Thresholds for Object-Oriented Software Metrics. In *Journal of Systems and Software*. volume 85, pages 244-257.

Ferreira, G.C. , Gaia, F.N. , Figueiredo, E. and Maia, M.A. (2014). On the Use of Feature-Oriented Programming for Evolving Software Product Lines – A Comparative Study. In *Science of Computer Programming*. volume 93, pages 65-85.

Figueiredo et al. (2008). Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability. In *Proceedings of the International Conference on Software Engineering (ICSE)*. pages 261-270.

Foss, S. , Korshunov, D. and Zachary, S. (2011). An Introduction to Heavy-Tailed and Subexponential Distributions. In *Springer-Verlag*.

Fowler, M. , Beck, K. , Brant, J. , Opdyke, W. and Roberts, D. (1999). Refactoring: Improving the Design of Existing Code. In *Addison-Wesley Professional*.

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

Gamma, E. , Helm, R. , Johnson, R. and Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. In *Addison-Wesley*.

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

Heitlager, I. , Kuipers, T. and Visser, J. (2007). A Practical Model for Measuring Maintainability. In *Proceedings of International Conference on the Quality of Information and Communications Technology (QUATIC'07)*. pages 30-39.

Kang, K. , Cohen, S. , Hess, J. , Novak, W.and Peterson, A. (1990). Feature-Oriented Domain Analysis (FODA) – Feasibility Study. In *SEE Technical report CMU/SEI-90-TR-021*.

Kästner, C. and Apel, S. (2008). Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proceedings of GPCE – Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*. pages 35-40.

Kästner, C. , Apel, S. and Batory, D. (2007). A Case Study Implementing Features Using AspectJ. In *Proceedings of International Software Product Line Conference (SPLC)*. pages 223-232.

Kästner, C. , Apel, S. and Kuhlemann, M. (2008). Granularity in Software Product Lines. In *Proceedings of International Conference on Software Engineering (ICSE)*. pages 311-320.

Kiczales, G. , Lamping, J. , Mendhekar, M. , Maeda, C. , Lopes, C.V. , Loingtier, J. M. and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming*. pages 220-242.

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

Lanza, M. and Marinescu, R. (2006). Object-Oriented Metrics in Pratice. In *Springer-Verlag*. page 205.

Liebig, J. , Apel, S. , Lengauer, C. , Kästner, C. and Schulze, M. (2010). An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of International Conference on Software Engineering (ICSE)*. pages 105-114.

Lima, E. (2014). Uma Análise dos Valores de Referência de Algumas Medidas de Software. In *Master Dissertation in Computer Science, Computer Science Department of Federal University of Lavras (UFLA)*. page 192.

Lorenz, M. and Kidd, J. (1994). Object-oriented Software Metrics. In *New York: Prentice Hall*. page 146.

Louridas, P. , Spinellis, D. and Vlachos, V. (2008). Power Laws in Software. In *ACM Transactions on Software Engineering and Methodology*. volume 18.

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

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

Mathwave (2016). `https://netbeans.org/`. In *Access in January, 2016*.

McCabe, T.J. (1976). A Complexity Measure. In *IEEE Transactions on Software Engineering*. volume 2, pages 308-320.

Munro, M. J. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of 11th IEEE International Software Metrics Symposium (METRICS)*.

Nejmeh, B.A. (1988). NPATH: A Measure of Execution Path Complexity and its Applications. In *Magazine Communications of the ACM*.

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

Oliveira, P. , Lima, F.P. , Valente, M.T. and Serebrenik, A. (2014). RTTOOL: A Tool for Extracting Relative Thresholds for Source Code Metrics. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSM)*. pages 1-4.

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

Pohl, K. , Bockle, G. and Linden, F.J.V. (2005). Software Product Line Engineering: Foundations, Principles, and Techniques. In *Berlin Springer*. page 490.

Polh, K. and Metzger, A. (2006). Software Product Line Testing. In *Communications of the ACM*. pages 78-81.

Riel, J. (1996). Object-Oriented Design Heuristics. In *Addison-Wesley Professional*. page 400.

Schaefer, I. , Bettini, L. and Damiani, F. (2011). Compositional Type Checking for Delta-Oriented Programming. In *Proceedings of International Conference on Aspect-oriented Software Development (AOSD)*. pages 43-56.

Schulze, S. , Apel, S. and Kästner, C. (2010). Code Clones in Feature-Oriented Software Product Lines. In *Proceedings of International Conference on Generative Programming and Component Engineering (GPCE)*. pages 103-112.

SEI - Software Engineering Institute (2016). <http://migre.me/nOM7f>. In *Access in January, 2016*.

Sommerville, I. (2011). Software Engineering, 9ª Edition. In *Pearson Education*.

Spinellis, D. (2008). A Tale of Four Kernels. In *Proceedings of the International Conference on Software Engineering (ICSE)*. pages 381-390.

SPL2GO (2016). <http://spl2go.cs.ovgu.de/>. In *Access in January, 2016*.

SPLRepository (2016). <goo.gl/BQUxJU>. In *Access in January, 2016*.

Vasa, D. , Lumpe, M. , Branch, P. and Nierstrasz, O. (2009). Comparative Analysis of Evolving Software Systems Using the Gini Coefficient. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. pages 179-188.

Weiss, D.M. and Lai, C.T.R. (1999). Software Product-Line Engineering: A Family-Based Software Development Process. In *Addison-Wesley*.

Appendix A

Primary Studies

This appendix presents the list of papers which we analyze to extract the methods to derive metric thresholds explored in this dissertation. Table A.1 presents the list of 50 papers selected in our literature review in alphabetical order, with the year of publication.

Table A.1: List of Papers of Our Literature Review

Title/Authors	**Venue/Publication Year**
A Complexity Measure / McCabe	TSE/1976
A Measure of Execution Path Complexity and its Applications / B. Nejmeh	Commun. ACM / 1988
A Metrics Suite for Measuring Reusability of Software Components / H. Washizaki et al.	METRICS / 2003
A Metrics Suite for Object Oriented Design / S. Chidamber and C. Kemerer	TSE / 1994
A Practical Model for Measuring Maintainability / I. Heitlager et al.	QUATIC / 2007
A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems / R. Shatnawi	TSE / 2010
Assessing the impact of bad smells using historical information / A. Lozano et al.	IWPSE / 2007
A Tale of Four Kernels / D. Spinellis	ICSE / 2008
An adjusted boxplot for skewed distributions / M. Hubert and E. Vandervieren	JCSDA / 2008
An Empirical Exploration of the Distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite / G. Succi et al.	JESE / 2005

Table A.2: List of Papers of Our Literature Review (Cont.1)

Title/Authors	**Venue/Publication Year**
An Outlier Detection Algorithm Based on Object-Oriented Metrics Thresholds / O. Alan and C. Catal	ISCIS / 2009
Applying Design-Metrics to Object-Oriented Frameworks / K. Erni and C. Lewerentz	METRICS / 1996
Benchmark-based Aggregation of Metrics to Ratings / T. Alves et al.	IWSM-MENSURA / 2011
Can We Avoid High Coupling? / C. Taube-Schock et al.	ECOOP / 2011
Calculation and optimization of thresholds for sets of software metrics / S. Herbold	JESE / 2011
Class noise detection based on software metrics and ROC curves / C. Catal et al.	JIS / 2011
Comparative Analysis of Evolving Software Systems Using the Gini Coefficient / R. Vasa et al.	ICSM / 2009
Clustering and Metrics Thresholds Based Software Fault Prediction of Unlabeled Program Modules / C. Catal et al.	ITNG / 2009
Deriving Metric Thresholds from Benchmark Data / T. Alves et al.	ICSM / 2010
Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Systems / R. Queiroz et al.	FOSD / 2014
Establishing Software Metric Thresholds / V. French	IWSM/ 1999
Estimation of Software Reusability: An Engineering Approach / T. Nair and R. Selvarani	SIGSOFT / 2010
Extracting Relative Thresholds for Source Code Metrics / P. Oliveira et al.	CSMR-WCRE / 2014
Faster Defect Resolution with Higher Technical Quality of Software / B. Luijten and J. Visser	TDU-SERG / 2010
Faster Issue Resolution with Higher Technical Quality of Software / D. Bijlsma et al.	SQJ / 2012
Finding software metrics threshold values using ROC curves / R. Shatnawi et al.	JSME / 2010

Table A.3: List of Papers of Our Literature Review (Cont.2)

Title/Authors	**Venue/Publication Year**
Getting what you measure: four common pitfalls in using software metrics for project management / E. Bouwers et al.	SR / 2012
Identifying thresholds for object-oriented software metrics / K. Ferreira et al.	JSS / 2012
Improving the applicability of object- oriented class cohesion metrics / J. Dallal	JIST / 2011
Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis / S. Chidamber et al.	TSE / 1998
Mining the impact of evolution categories on object-oriented metrics / H. Rocha et al.	SQJ / 2013
Observing Distributions in Size Metrics: Experience from Analyzing Large Software Systems / R. Ramler et al.	COMPSAC / 2007
Power Law Distributions in Class Relationships / R. Wheeldon and S. Counsell	IWSCAM / 2003
Power-Law Distributions in Empirical Data / A. Clauset et al.	SIAM / 2007
Power Laws in Software / P. Louridas et al.	TOSEM / 2008
Power-Laws in a Large Object-Oriented Software System / G. Concas et al.	TSE / 2007
Quantifying Maintainability in Feature Oriented Product Lines / G. Aldekoa et al.	CSME / 2008
Reference Values for Object-Oriented Software Metrics / K. Ferreira et al.	SBES / 2009
RTTOOL: A Tool for Extracting Relative Thresholds for Source Code Metrics / P. Oliveira et al.	ICSME / 2014
Scale-free Geometry in Object-Oriented Programs / A. Potanin et al.	Commun. ACM / 2005
Software metrics for object-oriented systems / J. Coppick and T. Cheatham	Commun. ACM / 1992
Software Reuse Metrics for Object-Oriented Systems / K. Aggarwal et al.	ACIS / 2005
Software Metrics Model for Quality Control / N. Schneidewind	METRICS / 1997
Standardized Code Quality Benchmarking for Improving Software Maintainability / R. Baggen et al.	SQJ / 2012

Table A.4: List of Papers of Our Literature Review (Cont.3)

Title/Authors	**Venue/Publication Year**
The Application of Software Maintainability Models in Industrial Software Systems / D. Coleman et al.	JSS / 1995
The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies / E. Tempero et al.	APSEC / 2010
The Optimal Class Size for Object-Oriented Software / K. Emam et al.	TSE / 2002
Thresholds for Object-Oriented Measures / S. Benlarbi et al.	ISSRE / 2000
Understanding the Shape of Java Software / G. Baxter et al.	OOPSLA / 2006
You Can't Control the Unfamiliar: A Study on the Relations Between Aggregation Techniques for Software Metrics / B. Vasilescu et al.	ICSM / 2011