PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n - Ondina
Salvador, BA, Brasil, 40170-110

http://pgcomp.dcc.ufba.br
pgcomp@ufba.br

State-of-the-art techniques and Automated Static Analysis Tools (ASATs) for identifying code smells rely on metric-based assessment. However, most of these techniques have low accuracy. One possible reason is that source code elements, such as methods implemented according to different design decisions, are assessed through the same generic threshold for each metric. Other possible reason is that these metric thresholds are usually derived from classes driven by different design decisions. Using generic metric thresholds that do not consider the design context of each evaluated class can generate many false positives and false negatives for software developers. Our goal is to propose design-sensitive techniques to derive contextual metric thresholds. Our primary hypothesis is that using the design role played by each system class to define this context may point out more relevant code smells to software developers. We conducted some empirical studies to define the proposed techniques. Firstly, we performed a large-scale survey that showed that practitioners recognize difficulties in fitting ASATs into the software development process. They also claim that there is no routine for application. One possible reason practitioners recognize that most of these tools use a single metric threshold, which might not be adequate to evaluate all system classes. Secondly, we conducted an empirical study to investigate whether fine-grained design decisions also influence the distribution of software metrics and, therefore, should be considered to derive metric thresholds. Our findings show that the distribution of metrics is sensitive to the following design decisions: (i) design role of the class (ii) used libraries, (iii) coding style, (iv) exception handling, and (v) logging and debugging code mechanisms. We used these findings to propose two new techniques to derive design-sensitive metric thresholds using the class design role as context. Then, we carried out two large-scale empirical studies to evaluate them. The first study showed that our proposed techniques improved precision according to developers' perceptions. Since it is impossible and tiring to perform a complete source code quality assessment with developers, we conducted a second study mining the evolution of software projects from popular architectural domains. We found that our techniques improved recall to point out methods effectively refactored during software evolution.

Keywords: Software Design, Metric Thresholds, Design Decisions, Design Role, Class design role, Code Analysis, Static Analysis Tools, Software Quality, Empirical Studies.

DSC | 21 | 2021

Design-Sensitive Metric Thresholds based on Design Roles

Marcos Dósea

UFBA

Design-Sensitive Metric Thresholds based on Design Roles

Marcos Barbosa Dósea

Tese de Doutorado

Universidade Federal da Bahia

Programa de Pós-Graduação em Ciência da Computação

Agosto | 2021

MARCOS BARBOSA DÓSEA

DESIGN-SENSITIVE METRIC THRESHOLDS BASED ON DESIGN ROLES

Esta Tese de Doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Dr. Cláudio Nogueira Sant'Anna

Salvador
16 de Agosto de 2021

Sistema de Bibliotecas - UFBA

Dosea, Marcos Barbosa.

Design-Sensitive Metric Thresholds based on Design Roles / Marcos Barbosa Dósea – Salvador, 2021.

155p.: il.

Orientador: Prof. Dr. Dr. Cláudio Nogueira Sant'Anna.

Tese (Doutorado) – Universidade Federal da Bahia, Instituto de Matemática, 2021.

1. Metric Threshold. 2. Software Design. 3. Design Role. I. Sant'Anna, Cláudio Nogueira. II. Universidade Federal da Bahia. Instituto de Matemática. III Título.

CDD – XXX.XX

CDU – XXX.XX.XXX

MARCOS BARBOSA DÓSEA

"DESIGN-SENSITIVE METRIC THRESHOLDS BASED ON DESIGN ROLES"

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da UFBA.

Salvador, 16 de agosto de 2021

Prof. Dr. Cláudio Nogueira Sant`Anna (Orientador-UFBA)

Prof. Dr. Manoel Gomes de Mendonça Neto (UFBA)

Profª. Drª. Christina von Flach Garcia Chavez (UFBA)

Prof. Dr. Marcelo de Almeida Maia (UFU)

Prof. Dr. Maurício Finavaro Aniche (TU Delft)

RESUMO

O estado da arte das técnicas para identificação de anomalias de código são baseadas na análise de métricas do código. Entretanto, muitas dessas técnicas possuem baixa acurácia. Uma possível razão é porque os elementos do código-fonte, tais como métodos, implementados com diferentes decisões de design, são avaliados usando os mesmos valores limiares para cada métrica. Outra possível razão é que estes valores limiares são geralmente derivados de classes que foram desenvolvidas usando decisões de design distintas. Usar valores limiares genéricos e que não consideram o contexto de design de cada classe avaliada pode gerar muitos falsos positivos e falsos negativos para os desenvolvedores de software. Nosso objetivo é considerar o papel de design desempenhado por cada classe do sistema como contexto para derivar valores limiares e detectar anomalias de código. Nossa principal hipótese é que valores limiares que consideram o papel de design da classe como contexto, podem apontar anomalias de código mais relevantes para os desenvolvedores de software. Nós conduzimos alguns estudos experimentais para definição de duas técnicas que consideram o papel de design como contexto. Primeiro, nós executamos uma survey de larga escala que mostrou que profissionais possuem dificuldades para ajustar ferramentas de análise estática de código nos seus processos de desenvolvimento de software. Uma possível razão reconhecida pelos profissionais é que muitas dessas ferramentas usam um valor limiar genérico para cada métrica que pode não ser adequado para avaliar todas as classes do sistema. Segundo, nós definimos uma heurística para identificar o principal papel de design de cada classe e conduzimos um estudo empírico para investigar a influência de algumas decisões de design na distribuição dos valores das métricas de software. Nossas descobertas mostram que as decisões de design que mais influenciaram na distribuição dos valores das métricas foram: (i) papel de design de classe (ii) bibliotecas utilizadas, (iii) estilo de codificação, (iv) estratégia de codificação usada para tratamento de exceção, e (v) estratégia para realizar log e depuração do código. Portanto, essa decisões de design poderiam ser consideradas como contexto para derivar valores limiares. Nós usamos essas descobertas para propor duas novas técnicas para derivar valores limiares sensíveis ao design do sistema usando os papéis de design da classe como contexto. Em seguida, realizamos dois estudos experimentais para avaliar as técnicas propostas. O primeiro estudo mostrou que as técnicas melhoraram a precisão de acordo com as percepções dos desenvolvedores. Como é impossível e cansativo fazer uma avaliação completa da qualidade do código-fonte com os desenvolvedores, conduzimos um segundo estudo explorando a evolução dos projetos de software de domínios arquiteturais populares. Descobrimos que nossas técnicas melhoraram o *recall* para apontar métodos efetivamente refatorados durante a evolução do software.

Palavras-chave: Software Design, Limiares de Métricas, Decisões de Design, Papel de Design, Papel de Design da Classe, Análise de Código, Ferramentas de Análise Estática, Qualidade de Software, Estudos Empíricos

ABSTRACT

State-of-the-art techniques and Automated Static Analysis Tools (ASATs) for identifying code smells rely on metric-based assessment. However, most of these techniques have low accuracy. One possible reason is that source code elements, such as methods implemented according to different design decisions, are assessed through the same generic threshold for each metric. Other possible reason is that these metric thresholds are usually derived from classes driven by different design decisions. Using generic metric thresholds that do not consider the design context of each evaluated class can generate many false positives and false negatives for software developers. Our goal is to propose design-sensitive techniques to derive contextual metric thresholds. Our primary hypothesis is that using the design role played by each system class to define this context may point out more relevant code smells to software developers. We conducted some empirical studies to define the proposed techniques. Firstly, we performed a large-scale survey that showed that practitioners recognize difficulties in fitting ASATs into the software development process. They also claim that there is no routine for application. One possible reason practitioners recognize that most of these tools use a single metric threshold, which might not be adequate to evaluate all system classes. Secondly, we conducted an empirical study to investigate whether fine-grained design decisions also influence the distribution of software metrics and, therefore, should be considered to derive metric thresholds. Our findings show that the distribution of metrics is sensitive to the following design decisions: (i) design role of the class (ii) used libraries, (iii) coding style, (iv) exception handling, and (v) logging and debugging code mechanisms. We used these findings to propose two new techniques to derive design-sensitive metric thresholds using the class design role as context. Then, we carried out two large-scale empirical studies to evaluate them. The first study showed that our proposed techniques improved precision according to developers' perceptions. Since it is impossible and tiring to perform a complete source code quality assessment with developers, we conducted a second study mining the evolution of software projects from popular architectural domains. We found that our techniques improved recall to point out methods effectively refactored during software evolution.

Keywords: Software Design, Metric Thresholds, Design Decisions, Design Role, Class design role, Code Analysis, Static Analysis Tools, Software Quality, Empirical Studies

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Automated static code analysis have become an important pillar of modern code review (MCR) practices next to testing and manual code review (BELLER et al., 2016). The main objectives are finding code anomalies (code smells), and performing code improvements in terms of readability, commenting, consistency, and dead code removal (BACCHELLI; BIRD, 2013). Many open-source software projects and companies such as Microsoft, Google, Facebook use automated static code analysis prior to merging new code into the main project codebase (RIGBY; BIRD, 2013; BALACHANDRAN, 2013).

Automated static analysis tools (ASATs) (e.g. PMD[1], Checkstyle[2], SonarQube[3] and NDepend[4]) are among the most popular static code analysis tools in industry to scan predefined problems and perform source code improvements in software systems (BESSEY et al., 2010; AYEWAH; PUGH, 2010; ZHENG et al., 2006). State-of-the-art techniques implemented by current ASATs rely on metric-based detection strategies (MARINESCU, 2004; ARCOVERDE et al., 2012; BALACHANDRAN, 2013; OIZUMI et al., 2016). A detection strategy uses logical operators to combine metrics and theirs respective thresholds to identify source code elements (usually classes or methods) with structural characteristics that correspond to a certain code smell (MARINESCU, 2004).

However, the accuracy of a detection strategy is heavily influenced by the calibration of the used metric thresholds (SHARMA; SPINELLIS, 2018). Using a single metric threshold may be too restrictive to cope with a wide range of specific contexts (SOBRINHO; LUCIA; MAIA, 2018). The overload of alarms caused by inaccurate metric thresholds and the way in which they are presented to software developers have been pointed out as the main barriers to the consistent and widespread use of ASATs (AYEWAH; PUGH, 2010; JOHNSON et al., 2013). In addition, some studies have reported that many of these alarms are false-positives since manual inspection reveals they have no effect on software

[1] http://pmd.sourceforge.net
[2] http://checkstyle.sourceforge.net/
[3] https://www.sonarqube.org/
[4] http://www.ndepend.com/

quality or maintenance effort (OLBRICH; CRUZES; SJØBERG, 2010; KHOMH et al., 2011; SJOBERG et al., 2013; YAMASHITA, 2013; PALOMBA et al., 2014; HOZANO et al., 2018).

1.1 PROBLEM STATEMENT

Most of the metric-based detection strategies used by current ASATs to detect code smells adopt generic metric thresholds that disregard the design context of the evaluated class. We have a generic threshold for a given metric when we use the same single value for classifying into categories (such as low or high) every class (or every method) of one or more systems regardless of the design decisions considered by the development team. For instance, Lanza e Marinescu (2006) classify as *long* any method that has more than 20 lines of code (LOC) in Java systems. In this case, they used 20 as a generic threshold for LOC. Different approaches for calculating generic thresholds have been proposed (LANZA; MARINESCU, 2006; ALVES; YPMA; VISSER, 2010; FERREIRA et al., 2012; OLIVEIRA; VALENTE; LIMA, 2014; FONTANA et al., 2015; VALE; FIGUEIREDO, 2015). These approaches calculate thresholds based on the distribution of metrics obtained from measurement data over sets of software systems as benchmarks.

However, some studies suggest the primary reason for the occurrence of false-positive and false-negatives alarms on smell detection methods is the lack of context for metric thresholds (ZHANG et al., 2013; ANICHE et al., 2016; SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018). Using generic thresholds to source code evaluation may not make sense for the entire set of classes in a system (LAVAZZA; MORASCA, 2016). In fact, Zhang et al. (2013), for instance, show that application domain and programming language are some design decisions to be considered as context and that have a strong influence on the derived metric thresholds. Aniche et al. (2016) show that the architectural role played by classes is another design decision to take into account to derive metric thresholds. For instance, in an MVC-based system, a generic threshold value might be too low for classes playing the *View* architectural role or too high for classes playing the *Controller* architectural role. This might, for instance, lead to false code smell alarms or hide potential code smells. Both situations may hinder maintenance activities and developers' perception of the quality of the source code (SJOBERG et al., 2013; YAMASHITA, 2013; HOZANO et al., 2015; PALOMBA et al., 2014; HOZANO et al., 2018).

We hypothesize, however, that the design decisions used so far as context to derive metric thresholds are too coarse-grained to explain the differences in the metrics distributions. For instance, several classes are not bound to any predefined architectural framework (e.g. Spring MVC, Android). Many systems are developed using proprietary reference architectures, not bounded to available architectural frameworks or architectural patterns. For instance, in MVC-based systems, there are many classes that do not assume the roles defined by this architectural pattern (e.g. *Controller* or *View*). Therefore, guiding the source code analysis based only on architectural roles to derive contextual metric thresholds might not cover a reasonable number of classes. Moreover, classes implemented with the same programming language (e.g. Java) and playing the

same architectural role (e.g. *Repository*) in different systems may use distinct persistence libraries (e.g. JDBC or JPA for the Java platform). Thus, the source code of classes playing the same architectural role and using the same programming language in different systems may require distinct metric thresholds to source code evaluation.

To illustrate that the design decisions studied so far (e.g. architectural roles and programming language) may not be enough to define the class context used to derive contextual metric thresholds, we selected two real-world MVC-based web applications: LibrePlan[5] and WebBudget[6]. LibrePlan is a project management, monitoring, and controlling tool, whereas WebBudget is a personal financial management tool. We downloaded the source code of both system from Github: WebBudget on October 20th 2016 and LibrePlan on November 9th 2016.

Firstly, we identified that only considering architectural roles bound to reference architectures, as Aniche et al. (ANICHE et al., 2016) do, may hinder to associate a context to many system classes. For instance, we identified that 83.3% of LibrePlan classes and 79,8% of WebBudget classes do not play any of the MVC architectural roles (e.g. Model, View, or Controller). They are classes designed to solve specific problems of each system specific context. These classes can play important design roles in the system. For instance, Libreplan has a set of 40 classes with the design role of providing additional features to HTML components. All these classes extend the HtmlMacroComponent abstract class. This group of classes may have specific design characteristics and, as a consequence, their metrics distributions may be different from the metrics distributions of other classes of the system. Therefore, it may be important to consider this group of classes as context to group classes to analyze metrics distributions and derive contextual metric thresholds.

Afterwards, we computed the Lines of Code per Method (LOC/Method) metric for methods of classes playing the Repository architectural role in both systems. We manually identified those classes. Both systems use the Repository design pattern (ALUR; CRUPI; MALKS, 2003) and the Hibernate framework (BAUER; KING; GREGORY, 2015) for implementing persistence. We assigned the Repository architectural role to all system classes that: (i) have the @Repository annotation or (ii) extend classes or implement interfaces with the "Repository", "DAO" or "Store" tokens in their names. This approach extends the technique proposed by Aniche et al. (2016) because we identified distinct ways to assign the same architectural role in these systems, not covered by Aniche's approach.

Then we applied the Mann-Whitney U statistical test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) to compare the distribution of LOC/Method metric from WebBudget and LibrePlan samples. The test result showed differences between the two samples. We then conducted a manual analysis on both samples and noticed that methods from WebBudget use at least 50% more lines of code than similar methods in LibrePlan system.

[5]https://github.com/LibrePlan/libreplan

[6]https://github.com/arthurgregorio/web-budget

Listing 1.1: Query Example in Libreplan system.

```
1 public List<MaterialAssignment> getByMaterial(Material material) {
2         return (List<MaterialAssignment>) getSession().
              createCriteria(MaterialAssignment.class). add(Restrictions.
              eq("materialInfo.material", material)).list();
3 }
```

Listing 1.2: Query Example in WebBudget system.

```
1 public List<Movement> listByCardInvoice(CardInvoice cardInvoice) {
2         final Criteria criteria = this.getSession().
              createCriteria(this.getPersistentClass());
3         criteria.createAlias("cardInvoice", "ci");
4         criteria.add(Restrictions.eq("ci.id", cardInvoice.getId()));
5          criteria.addOrder(Order.desc("inclusion"));
6         return criteria.list();
7 }
```

Why does this happen if they are assigned to the same architectural role (Repository) and framework (Hibernate)? We manually analyzed their source code and found another design decision responsible for that difference: coding style. To illustrate that, on the one hand, we show on Listing 1.1 a method extracted from LibrePlan with a single-line statement to query a database and return a list of objects. On the other hand, Listing 1.2, extracted from WebBudget, shows a similar query, but using five lines of code. One may argue about the quality of both source code fragments, but both coding styles are quite common in applications that use the Hibernate framework and thus represent a design decision usually obeyed by other classes on the same system (YANG, 2010). In summary, this is an example that different choices related to a single design decision (coding style) may contribute to different metrics distributions of the same architectural role across different systems. In this way, the coding style can be another design decision to be taken into account to define the class context used to derive contextual metric thresholds.

1.2 MAIN GOAL AND RESEARCH QUESTIONS

This research proposes design-sensitive metric thresholds using the class design role as context to derive metric thresholds. The design role is a set of related responsibilities assumed by an object to fit into a community, such as a framework or an enterprise architecture. Modern object-oriented systems assigned a design role to one or more classes through inheritance, interface implementation, or class annotations. For instance, in the Libreplan system, we discussed that 40 classes were assigned with the same design role because they extend HtmlMacroComponent class. Our main hypothesis is that metric thresholds, defined by taking class design role as context, may detect more relevant code smells. Therefore, our overarching research question is:

Do design-sensitive metric thresholds based on class design roles improve code smell detection strategies' accuracy?

Our general research method uses a three-step approach proposed by Wohlin et al. (WOHLIN et al., 2012) to improve a process or practice: (1) understand the current

process to identify improvement opportunities; (2) evaluate the current process and new ideas; (3) improve the process by incorporating suggestions. We defined a set of more specific research questions that guided the proposed empirical studies to apply this method. In this section, we define our three general research questions and the reasoning and assumptions that motivate them.

RQ1: *How do practitioners perceive automated static analysis for code smell identification?*

We designed a large-scale survey to understand software developers' current process to apply automated static analysis tools (ASTs). Our initial hypothesis is that, in general, practitioners have difficulties using AST in their software development processes due to the high number of false alarms. Some studies capture the perception of developers who already use ASTs regularly. However, to capture other issues, challenges, and opportunities of improvements, our goal was to conduct a large-scale study without limiting the target audience to developers who regularly use ASTs. Also, we aim to understand the practitioners' perception about using metric thresholds in ASTs that take into account the design role played by the evaluated class. Using design-sensitive metric thresholds is a possible way to avoid the high number of false alarms generated by current tools.

RQ2: *Are there statistically significant differences between measures obtained from classes developed with different design decisions?*

To evaluate the current process and new ideas, we conducted an empirical study to investigate how some fine-grained design decisions, not considered in previous studies, impact software metrics distributions. Source code analysis techniques usually rely on metric-based assessment. However, most of these techniques have low accuracy. We hypothesize that this occurs because metric thresholds are derived from classes driven by different design decisions. Previous studies have already shown that classes implemented according to some coarse-grained design decisions, such as programming languages, impact the distribution of metric values. Therefore, these design decisions must be taken into account when using benchmarks for metric-based source code analysis. However, in this research question, our goal is to investigate whether other design decisions, in particular, fine-grained design decisions such as the class' design role, also impact the distribution of software metrics and, therefore, should take into account as context to derive metric thresholds used to evaluate system classes.

RQ3 Are design-sensitive metric thresholds more accurate to detect code smells prone to be refactored?

Finally, to improve the current process by incorporating suggestions, we propose two novel techniques to derive design-sensitive metric thresholds. We hypothesize that design-sensitive metric thresholds are more accurate than state-of-the-art metric thresholds used by most existing detection strategies to detect code smells. Additionally, we propose an automated and flexible heuristic to identify the class design role, not binding to specific reference architectures. Our techniques use this heuristic to derive design-sensitive metric thresholds from benchmarks of systems developed with high similarity on design decisions. To answer this research question, we performed empirical studies comparing the thresholds derived from our techniques and from state-of-art techniques based on: (1) practitioners' perception about if they would refactor the code anomalies detected; and

(2) the accuracy to point out methods that suffered refactoring along software evolution.

Each general research question derives more specific research questions detailed in the following chapters.

1.3 CONTRIBUTIONS

The contributions of this thesis are:

- A survey on code analysis practices with Brazilian practitioners engaged in the software industry with not-so-well established practices.
- A heuristic to automatically identify the design role played by each system class. We use this heuristic in our techniques for deriving metric thresholds based on design roles. However, any other technique can take advantage of it to use design roles as context information.
- A novel approach to measure the level of similarity of design decisions between two systems. The approach helps to identify systems with similar design decisions to compose benchmarks to be used to derive design-sensitive metric thresholds.
- An empirical study to assess whether fine-grained design decisions affect the distribution of four method-level metrics. The study involves fifteen real-world systems from three distinct architectural domains.
- Two novel techniques to derive design-sensitive metric thresholds that takes into account class design roles. Both techniques use design roles to compose the benchmark from systems with high similarity to each other. In addition, one of the techniques derives distinct metric thresholds for each class design role identified in the system to be evaluated.
- An empirical study evaluating practitioners' perception about code smells pointed out by metric thresholds derived from five distinct benchmark-based techniques. We use Web-based real-world software projects. The practitioners also are familiar with design decisions impacting the evaluated source code.
- An empirical study evaluating the accuracy of our techniques to identify refactored methods during the software evolution. We compare these results with those obtained using metric thresholds derived from other state-of-art benchmark-based techniques.
- All material from the empirical studies was made publicly available on the web[7] so that the studies can be replicated or extended in further investigations.
- An open-source tool, namely DesignRoleMiner[8], that extends MetricMiner tool (SOKOL et al., 2013), adding method-level metrics and the proposed design role identification heuristic.

[7] https://github.com/marcosdosea/thesis
[8] https://github.com/marcosdosea/DesignRoleMiner

- An open-source tool, namely SystemSimilarity[9], that allows calculating the level of similarity between two systems based on design decisions extracted from these systems.

- An open-source tool, namely ThresholdTool[10], that implements our two proposed techniques and others three state-of-the-art techniques to derive metric thresholds.

- An open-source Eclipse plugin, namely ContextSmell[11] that extends ContextLongMethod tool (SANTOS; DÓSEA; SANT'ANNA, 2016). The tool identifies the design role played by each system class and detects code smells using metric thresholds derived from distinct techniques. For each smelly method, the tool shows the techniques that derived the metric threshold assigned to smelly detection. This data is useful to carry out a comparison of techniques like the one performed in this thesis. In addition, the Eclipse plugin generates just-in-time recommendations of identified smelly methods during the source-code development.

- An open-source tool, namely SmellRefactored[12], that uses the RefactoringMiner tool (TSANTALIS et al., 2018) to detect smelly methods refactored during the software evolution.

1.4 PUBLICATIONS

We published the papers listed below in chronological order.

1. DOSEA, M.; SANT'ANNA, C.; SANTOS, C.. Towards an Approach to Prevent Long Methods Based on Architecture-Sensitive Recommendations. In: IV Workshop on Software Visualization, Maintenance, and Evolution (VEM 2016), 2016, Maringá. Anais do Congresso Brasileiro de Software: Teoria e Prática, 2016. p. 73-80.

2. SANTOS, C.; DOSEA, M.; SANT'ANNA, C.. ContextLongMethod: Uma Ferramenta Sensível à Arquitetura para Detecção de Métodos Longos. In: Sessão de Ferramentas do Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2016, Maringá. Anais do Congresso Brasileiro de Software: Teoria e Prática, 2016. p. 25-32.

3. DOSEA, M.; SANT'ANNA, C. Uma Abordagem para Prevenir a Erosão do Design baseada em Recomendações Sensíveis à Arquitetura. In: VI Workshop de Teses and Dissertações do CBSoft (WTDSoft 2016), 2016, Maringá. Anais do Congresso Brasileiro de Software: Teoria e Prática, 2016. p. 19-27.

4. DOSEA, M.; SANT'ANNA, C.; DA SILVA, B. C. How do design decisions affect the distribution of software metrics?. In: the 26th Conference, 2018, Gothenburg.

[9] https://github.com/marcosdosea/SystemSimilarity
[10] https://github.com/marcosdosea/ThresholdTool
[11] https://github.com/marcosdosea/ContextSmellEclipse
[12] https://github.com/marcosdosea/SmellRafactored

Proceedings of the 26th Conference on Program Comprehension - ICPC '18. New York: ACM Press, 2018. p. 74-85.

5. DOSEA, M.; SANT'ANNA, C. Um Método para Detectar Similaridade entre Sistemas baseado em Decisões de Design: um Estudo Preliminar. In: VI Workshop on Software Visualization, Evolution and Maintenance, 2018, São Carlos. Anais do Congresso Brasileiro de Software: Teoria e Prática, 2018.

6. LIMA, R.; DOSEA, M.; SANT'ANNA, C. Comparando Técnicas de Extração de Valores Limiares para Métricas: Um Estudo Preliminar com Desenvolvedores Web. In: VI Workshop on Software Visualization, Evolution and Maintenance, 2018, São Carlos. Anais do Congresso Brasileiro de Software: Teoria e Prática, 2018.

1.5 DOCUMENT OUTLINE

We divided this document as follows:

- **Chapter 2** provides an overview of the background of this research. In particular, we present the notion of code smells, design decisions, and techniques to derive metric thresholds and factors that can impact metric thresholds.

- **Chapter 3** aims to answer RQ1 showing the results of a survey conducted with Brazilian software practitioners that investigated issues and challenges they face to apply automated code analysis practices.

- **Chapter 4** propose an automatic heuristic to identify the design role played by each class in a system. We use the class design role as context to derive design-sensitive metric thresholds. Additionally, we propose an approach using the heuristic to identify system similar to each other in terms of design decisions.

- **Chapter 5** aims to answer RQ2 showing the results of an empirical study to investigate whether fine-grained design decisions also influence the distribution of software metrics and, therefore, should be taken into account to describe the context of system classes. We evaluate the distributions of four metrics applied over fifteen real-world systems based on three different domains.

- **Chapter 6** aims to propose two novel design-sensitive techniques based on class design role as context to derive metric thresholds.

- **Chapter 7** discussed an industrial multi-project study analyzing developers' perception of code smells detected in source code they maintain. We aim to answer RQ3 by identifying which technique derived the most precise metric thresholds to detect smelly methods according to developers' perceptions.

- **Chapter 8** aims to complement the RQ3 answer from another perspective. We carry out a large-scale retrospective study over the commit history of 20 Web-based and 26 Android-based software projects observing the impact of four method-level code smells in effective refactorings performed during the software evolution.

- **Chapter 9** summarizes our work by describing what we have done in the context of this research. In addition, we point out perspectives on future research directions.

Chapter 2

BACKGROUND

Code smells are symptoms in the source code that may indicate the possibility of refactoring due to deeper maintainability problems (FOWLER; BECK, 1999). They are particularly harmful when contributing to architectural degradation (BASS; CLEMENTS; KAZMAN, 2012). Code smells are considered a poor solution that violates best practices to source code development (KHOMH et al., 2011) and impact the quality of the system making it more difficult to evolve and maintain (YAMASHITA, 2013; KHOMH et al., 2012; SOH et al., 2016).

Software developers have reported that finding code smells is one of the main motivations to source code review (BACCHELLI; BIRD, 2013; BOSU et al., 2017). Therefore, many research efforts have been made to automate the code smell detection process aiming to reduce the number of errors and the required time to perform code reviews. However, software developers have pointed out many obstacles to use tools to automatically detect code smells due mainly to the high number of false alarms (JOHNSON et al., 2013; CHRISTAKIS; BIRD, 2016).

In this chapter, we deepen this discussion used throughout this research. Thereby, we discuss in Section 2.1 types of code smells and detailed the four code smells selected to carry out our empirical studies. Section 2.2 shows strategies to detect code smells. The techniques to identify metric thresholds are discussed in Section 2.3, and we also detailed the ones used to conduct our empirical studies. Finally, Section 2.4 discusses some design decisions that could impact metric thresholds derived from the metric analysis.

2.1 TYPES OF CODE SMELLS

Smells are discussed in different domains. For example, Hermans et al. (HERMANS; PINZGER; DEURSEN, 2015) investigate the applicability of code smells to spreadsheet formulas as a means to assess and improve spreadsheet quality. In other study, Hermans et al. (HERMANS; AIVALOGLOU, 2016) examine code smells in the context of block-based Scratch programs. They evaluated Scratch smells in a controlled experiment with 61 high-school kids. In software system domain, we identified code smells focusing in

architecture, design, implementation, test, energy, and performance. Sobrinho et al. (SOBRINHO; LUCIA; MAIA, 2018) indicate fragmentation of code smell definitions due to the lack of systematic or/and formal taxonomies for code smells. Sharma and Spinellis (SHARMA; SPINELLIS, 2018) have compiled an extensive catalog including many types of smells. This catalog can be also accessed online[1].

In this research, we focused in four code smells related to method-level implementation: long methods, complex method, long parameter list, high efferent coupling method. We selected these method-level smells because we can manually find them without tool support. This criterion is essential for conducting the manual analysis planned for our proposed empirical studies. Also, these smells are available in many tools (PAIVA et al., 2017) and have been successfully used for fault-proneness prediction (FONTANA et al., 2013; GIL; LALOUCHE, 2017; BOUCHER; BADRI, 2018), for instance. They are used by quality models for measuring and rating the technical quality of a software system in terms of the quality characteristics of ISO/IEC 9126 (HEITLAGER; KUIPERS; VISSER, 2007). Finally, smells related to complex or long source code are generally perceived as an important threat by developers (PALOMBA et al., 2014). We detailed these smells as follow:

- **Long Methods** (FOWLER; BECK, 1999) occurs when a method is too long to understand. Fontana et al. (FONTANA et al., 2013) show long methods are among the most common code smells in different application domains. According Fowler et al. (FOWLER; BECK, 1999), the longer a procedure (method) is, the more difficult it is to understand. We use the Lines of Code (LOC) (LANZA; MARINESCU, 2006) metric to identify this smell. It counts the number of executable statements of each method, excluding comments and blank lines.

- **Complex Method** (SHARMA; FRAGKOULIS; SPINELLIS, 2017) occurs when a method has high cyclomatic complexity. Methods of high cyclomatic complexity may be more difficult to maintain and understand. We use the McCabe's Cyclomatic Complexity (CC) (MCCABE, 1976) to identify this smell. It counts number of branching points of each method. This metric is widely recognized and used as indicator of the source code maintainability (LANZA; MARINESCU, 2006; ZHANG et al., 2013; RADJENOVIć et al., 2013).

- **High Efferent Coupling Method** occurs when a method has high efferent coupling. We use the efferent coupling (EC) metric (MARTIN, 1995) to indicate this code smell. This metric counts the number of classes from which each method calls methods or accesses attributes. A large number of calls to external libraries can also hinder the understanding and maintenance.

- **Long Parameter List** (FOWLER; BECK, 1999) occurs when a method accepts a long list of parameters. Long parameter lists are hard to understand, because they become inconsistent and difficult to use, and because we are forever changing them as you need more data (FOWLER; BECK, 1999). To identify this smell, we use

[1] http://www.tusharma.in/smells/index.html

the metric Number of Method Parameters (NMP) (FOWLER; BECK, 1999) which counts the number of parameters of each method.

Other more complex strategies could be used to identify these code smells. However, we selected strategies based on a single metric because our goal is to individually assess the impact of the metric threshold proposed by different techniques. Strategies involving other variables, such as other metrics or other source code information, could hamper the individual analysis of metric threshold proposed by the techniques.

2.2 STRATEGIES TO DETECT CODE SMELLS

Many strategies to automatically detect code smells have been proposed. Sharma and Spinellis (SHARMA; SPINELLIS, 2018) classify these strategies in five categories: history-based, metric-based, rules/heuristic-based, optimization-based and, machine learning-based detection strategies.

History-based strategies detect smells using source code evolution information. Palomba et al. (PALOMBA et al., 2013), for example, propose an approach to detect five different code smells by exploiting change history information mined from versioning systems. However, history-based strategies are limited to a few code smells associated with evolutionary changes.

Metric-based strategies are the most common strategy adopted by state-of-the-art tools to detect code smells. They allow to identify design problems in an object-oriented software system using metrics based on *filtering* and *composition* mechanisms (MARINESCU, 2004). The *filtering* mechanism intend to select design fragments captured by a metric. A common way to define filters is specifying explicit thresholds. The *composition* mechanism is based on a set of *AND* and *OR* operators that compose different metrics together to form a composite rule. Figure 2.1, for example, shows a God Class detection strategy composed by three filtering mechanisms and one composition operator (AND). God Class is defined as a class that knows or does too much in a software system (FOWLER; BECK, 1999). Each filtering mechanism compare a metric to a threshold value. The established thresholds for each metric are applied to all system classes. Thus, the accuracy of metric-based strategies are thresholds dependent.

Rules/heuristic-based strategies detect smells when the defined rules or heuristics are satisfied. They are used to find smells that cannot be detected by metrics alone. Usually, these strategies take source code model and sometimes additional software metrics as inputs. For example, Moha et al.(MOHA et al., 2010) propose a strategy for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Rules and heuristics are usually combined with metric-based strategies to reveal a high proportion of known smells. The accuracy of these strategies also depends on metric thresholds.

Optimization-based strategies apply optimization algorithms, such as genetic algorithms, to detect code smells. Ghannem et al. (GHANNEM; BOUSSAIDI; KESSENTINI, 2016), for example, discuss it is difficult to find the best threshold values because the rules do not take into consideration the programming context. As an alternative, they

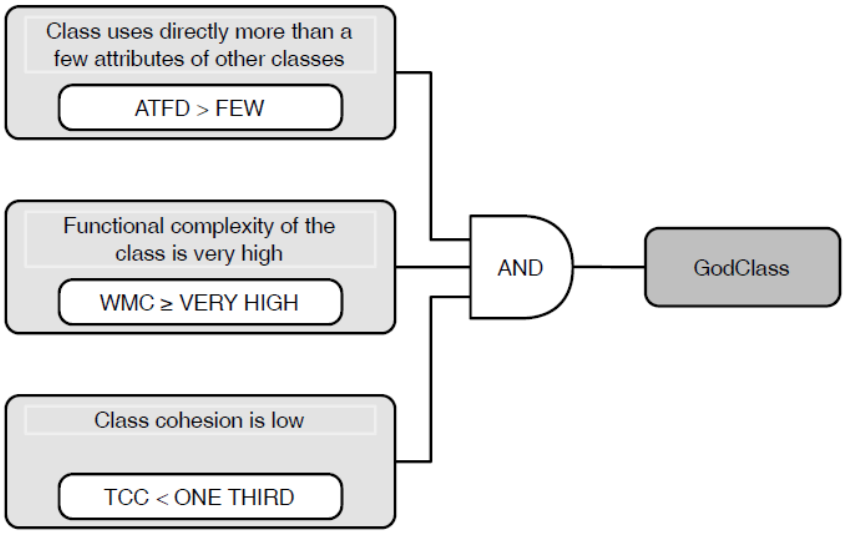

Figure 2.1: Detection Strategy of God Class
(LANZA; MARINESCU, 2006)

propose to identify design defects using a genetic algorithm based on the similarity/distance between the system under study and a set of defect examples without the need to define a heuristic-based strategy. However, many these strategies also depends heavily on metric data and corresponding metric thresholds (SHARMA; SPINELLIS, 2018).

Machine learning-based strategies detect smells using mathematical probabilistic models that represent the smell detection problem. Source code examples are used to train a proposed probabilistic model to detect code smells. For example, Khomh et al. (KHOMH et al., 2009) propose an approach to convert existing state-of-the-art detection rules into a probabilistic model. They use this approach to generate a model to detect occurrences of the Blob code smells. Mansoor et al. (MANSOOR et al., 2017) use multi-objective genetic programming (MOGP) to find the best combination of metrics that maximizes the detection of code-smell examples and minimizes the detection of well-designed code examples. However, these strategies depends heavily on training data and the lack of such training datasets can be a challenge to apply them (KHOMH et al., 2009). In addition, it is unknown whether machine learning-based strategies can scale to the large number of known smells (SHARMA; SPINELLIS, 2018). Studies conducted by Pecorelli et al. (PECORELLI et al., 2019, 2020) compare heuristic-based and machine-learning-based techniques for code smell detection. They emphasize the need of further research to improve the effectiveness of both machine learning and heuristic approaches. However, heuristic-based approach generally achieves better performance and precision.

The selection of proper thresholds is one of the hardest issues faced by various detection strategies relied on metrics (MARINESCU, 2004). The metric threshold dependence can be a challenge in rules/heuristic-based, optimization-based and metric-based detection strategies. On one hand, a low metric threshold could lead to many false code smell alarms, on the other hand, a high metric threshold could hide potential code smells. Both situations may hinder maintenance activities and developers' perception about the

quality of the source code (SJOBERG et al., 2013; YAMASHITA, 2013; HOZANO et al., 2015; PALOMBA et al., 2014; HOZANO et al., 2018). In addition, detection strategies to identify code smells that relies on metric thresholds are used by most of the state-of-the-art tools and, therefore, they are easier to implement in real-world software development process. In Section 2.3, we discuss some techniques to identify metric thresholds and we detailed the ones used to conduct our empirical studies to compare with the proposed design-sensitive technique to identify contextual metric thresholds.

2.3 TECHNIQUES TO DERIVE METRICS THRESHOLDS

Initial works propose to derive metric thresholds from programming experience (MCCABE, 1976; NEJMEH, 1988; COLEMAN; LOWTHER; OMAN, 1995). For instance, McCabe (MCCABE, 1976) suggested the value 10 as a threshold for the McCabe Cyclomatic Complexity metric derived from his experience. This metric counts the number of linearly independent paths through a program's source code. Coleman et al. (COLEMAN; LOWTHER; OMAN, 1995) propose metric thresholds for the maintainability index metric. All components above the 85 maintainability index are highly maintainable, components between 85 and 65 are moderately maintainable, and components below 65 are difficult to maintain. These values indicate the quality cutoff established by Hewlett-Packard developers experience. Metric thresholds derived from experience are difficult to reproduce or generalize to other systems or contexts due to lack of scientific support.

Some studies propose to derive metric thresholds using error models (SHATNAWI et al., 2009; BENLARBI et al., 2000). For example, Shatnawi et al. (SHATNAWI et al., 2009) identify metric thresholds using receiver operating characteristic (ROC) curves. The study used three releases of the Eclipse project and found threshold values for some OO metrics that separate no-error classes from classes that had high-impact errors. The technique propose different metric thresholds for each Eclipse release. Benlarbi et al. (BENLARBI et al., 2000) test threshold effects in a subset of the Chidamber and Kemerer (CK) suite of measures (CHIDAMBER; KEMERER, 1994). The results indicated that there are no threshold effects for any of the measures studied. This means that there is no value for the studied CK measures where the fault-proneness changes from being steady to rapidly increasing. Thresholds derived using error models are only valid for the specific error prediction model and for the evaluated metrics. Other models can give different results. Although these thresholds cannot predict whether a class will definitely have errors in the future, they can provide a more scientific method to assess class error proneness.

Yoon et al. (YOON; KWON; BAE, 2007) propose to derive metric thresholds using cluster techniques. They suggest an approach to outlier detection of software measurement data using the k-means clustering method. However, the process to identify outliers is manual and influenced by input parameters that can affect both performance and accuracy of the results.

Finally, most studies propose to derive metric thresholds from metric analysis. Initial works propose techniques relied on the use of the mean and standard deviation (ERNI; LEWERENTZ, 1996; LANZA; MARINESCU, 2006). For example, Erni and Lewerentz

(ERNI; LEWERENTZ, 1996) propose a multi-metrics approach for the design and improvement of a framework for industry. They propose to use mean (μ) and standard deviation (σ) to produce $Tmin = \mu - \sigma$ and $Tmax = \mu + \sigma$, the lower and the higher thresholds, respectively. Lanza and Marinescu (LANZA; MARINESCU, 2006) proposed some statistics-based thresholds extracted from a sample of 37 C++ system and 45 Java systems. They proposed lower and higher thresholds calculated in a similar way as Erni and Lewerentz technique (ERNI; LEWERENTZ, 1996). In addition, they propose a very high threshold 50% higher than the threshold for a high value. However, several studies show that most software metrics do not follow normal distributions (ALVES; YPMA; VISSER, 2010; FERREIRA et al., 2012; OLIVEIRA; VALENTE; LIMA, 2014), limiting the use of any statistical technique that relies on mean and standard derivation to derive metric thresholds. These techniques can derive invalid or non-representative metric thresholds.

To avoid the discussed problems in previous techniques, Alves et al. (ALVES; YPMA; VISSER, 2010) propose three core principles to be followed by metric thresholds techniques: (1) The technique should not be driven by expert opinion but by measurement data from a representative set of systems (benchmark); (2) The technique should respect the statistical properties of the metric, such as metric scale and distribution and should be resilient against outliers in metric values and system size (robust); (3) The technique should be repeatable, transparent and straightforward to carry out (pragmatic).

We have identified five techniques to derive thresholds that meet these principles (ALVES; YPMA; VISSER, 2010; FERREIRA et al., 2012; OLIVEIRA; VALENTE; LIMA, 2014; VALE; FIGUEIREDO, 2015; ANICHE et al., 2016). However, we disregard the Ferreira et al. (FERREIRA et al., 2012) technique and Oliveira et al. (OLIVEIRA; VALENTE; LIMA, 2014) to carry out our empirical studies.

Ferreira et al. (FERREIRA et al., 2012) propose three ranges of reference values for the metrics based on the most common values found in practice: *good*, which refers to the most common values of the metric; *regular*, which is an intermediate range of values with low frequency, but not irrelevant; and *bad*, that refers to values with quite rare occurrences. We disregard this technique because the authors do not explicitly describe how these ranges are determined. These ranges are not automatically defined and based on manual metric distribution analysis. This non-determinism to define the ranges which may lead to misinterpretations. Also, the manual analysis of the distribution of values may hinder to apply this technique to derive threshold values in real-world development environments.

Oliveira et al. (OLIVEIRA; VALENTE; LIMA, 2014) instead of using the threshold as a hard filter, they proposed a minimal percentage of classes that should be above this limit. The relative thresholds are based on a statistical analysis of a benchmark of systems and attempting to balance two forces. First, the derived relative thresholds should reflect real design rules, widely followed by the systems in the considered corpus, based on widely accepted quality principles (LANZA; MARINESCU, 2006). Second, the derived relative thresholds should not be based on rather lenient upper limits. We disregard this technique because to reduce this problem, others techniques are more flexible, proposing many metric thresholds for each metric aiming to minimize the number of false positives

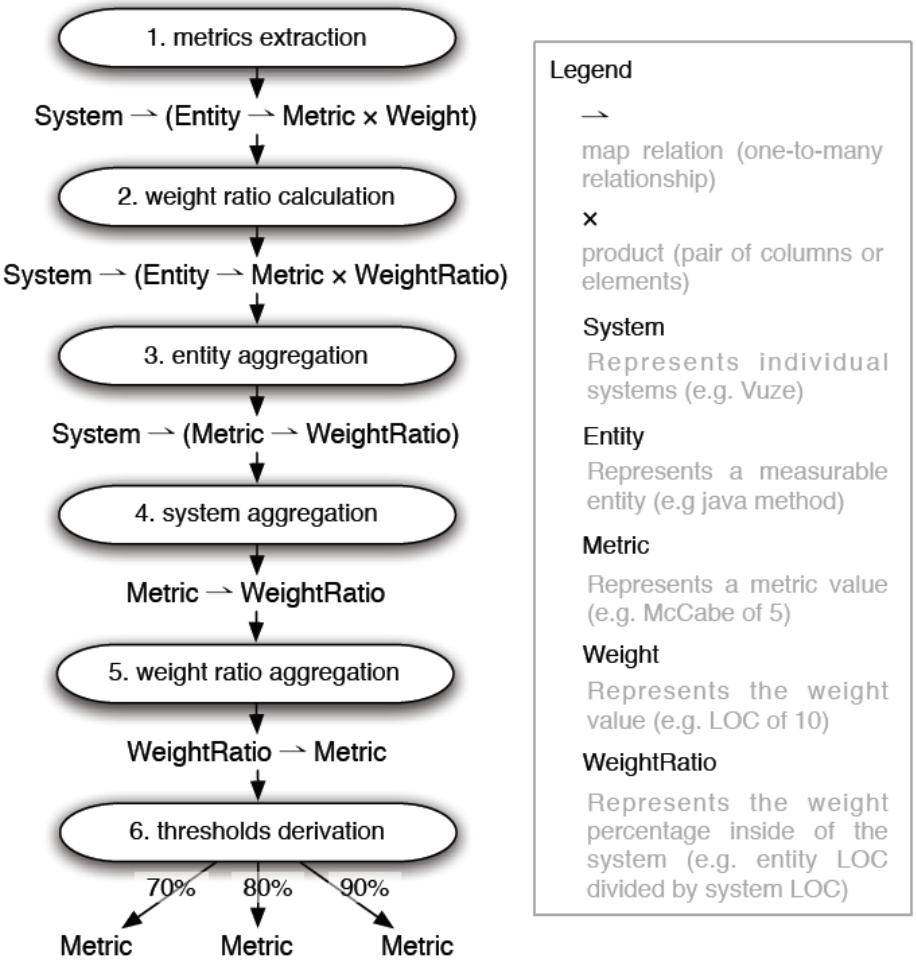

Figure 2.2: Alves et al. technique steps
(ALVES; YPMA; VISSER, 2010)

and streamline the code review process when there is not much time for further review. In addition, many metric-based detection strategies not only use the upper limit, but also need medium and lower limits. The technique does not make it clear how these other limits could be found. Thus, we detail in the next subsections the other three techniques used in our empirical studies.

2.3.1 Alves et al. Technique

Alves et al. (ALVES; YPMA; VISSER, 2010) proposed a technique to derive metric thresholds based on weighted functions. Using lines of code as weight, they select the code metric values relative to the 70%, 80%, and 90% percentiles of the accumulated weight, and uses these as thresholds. The values are extracted from the measurement data of a benchmark of software systems. Figure 2.2 summarizes the six steps of the their technique:

- *metrics extraction*: metrics are extracted from a benchmark of software systems. For each system *System*, and for each entity *Entity* belonging to *System* (e.g. method), they record a metric value, *Metric*, and weight metric, *Weight* for that system's entity. As weight they consider the source lines of code (LOC) of the entity.

- *weight ratio calculation*: for each entity, the technique compute the weight percentage within its system, i.e., it divide the entity weight by the sum of all weights of the same system.

- *entity aggregation*: the technique aggregate the weights of all entities per metric value, which is equivalent to computing a weighted histogram (the sum of all bins must be 100%).

- *system aggregation*: the technique normalize the weights for the number of systems and then aggregate the weight for all systems. Normalization ensures that the sum of all bins remains 100%, and then the aggregation is just a sum of the weight ratio per metric value.

- *weight ratio aggregation*: they order the metric values in ascending way and take the maximal metric value that represents 1%, 2%, ..., 100% of the weight. This is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale.

- *thresholds derivation*: thresholds are derived by choosing the percentage of the overall code we want to represent.

The technique uses LOC as a measure of size and use it to have a better representation of the part of the system to be characterized. Instead of assuming every unit (e.g. method) of the same size, they take its size in the system measured in LOC. They emphasize the variation of the metric allowing a more clear distinction between software systems. Hence, the correlation between LOC and other metrics poses no problem. In fact, Gil and Lalouche (GIL; LALOUCHE, 2017) discuss that many metrics are good predictors of external features, such as correlation to bugs, due its correlation to the size of the code artifact. Hence, the more a metric is correlated with size, the more able it is to predict external features values, and vice-versa.

2.3.2 Vale and Figueiredo' Technique

Vale and Figueiredo (VALE; FIGUEIREDO, 2015) propose a technique to derive thresholds in the software product line (SPL) domain and discuss the use of distinct benchmark datasets to extract thresholds. The technique was applied for God Class and Lazy Class detection strategies. They used three benchmarks of SPLs composed by subsets of 33 SPLs splitted up according to their size in terms of Lines of Code (LOC). They observed the thresholds are sensitive to benchmarks. The threshold values are higher in benchmarks composed by larger systems. The authors advocate that deriving thresholds from

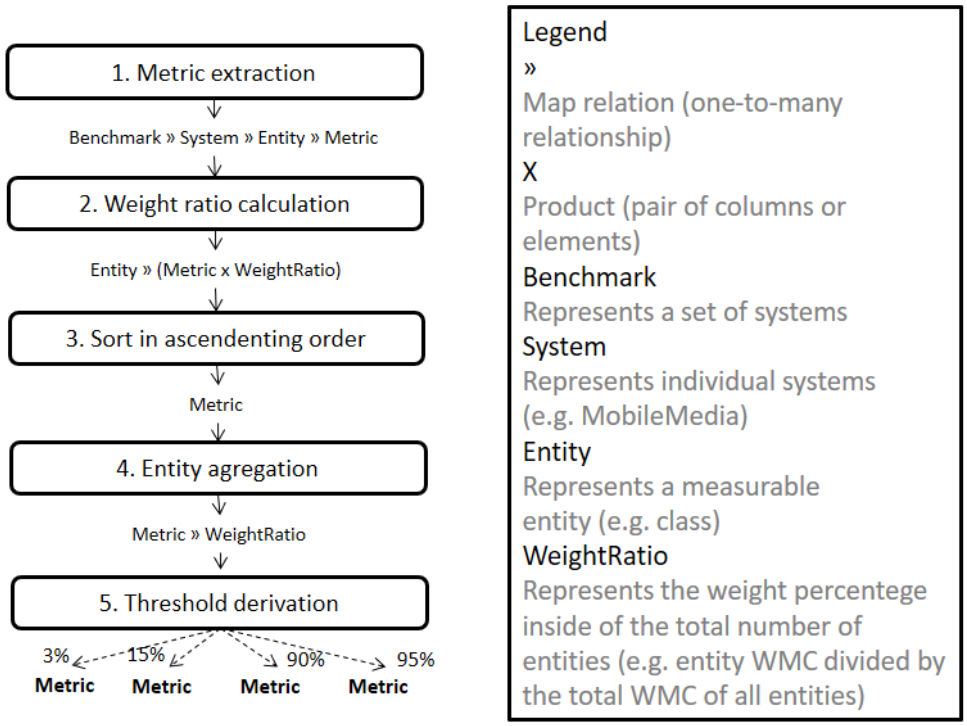

Figure 2.3: Vale and Figueiredo Technique Steps
(VALE; FIGUEIREDO, 2015)

a benchmark is more accurate than deriving thresholds from a single system. However, they did not address the issues of having applications of different domains within the analyzed benchmark dataset.

In contrast to Alves' technique the authors propose different labels to lower bound thresholds. Also, they do not use LOC to calculate, for each entity, the weight ratio. Figure 2.3 summarizes the five steps of the technique.

- *Metrics extraction*: metrics are extracted from a benchmark of software systems. For each system, and for each entity belonging to the system (e.g., class or method), they record a metric value.

- *Weight ratio calculation*: for each entity, they compute the weight percentage within the total number of entities, i.e., they divide the entity weight by the total number of entities, and then it is multiplied by one hundred. All entities have the same weight and the sum of all entities must be 100%.

- *Sort in ascending order*: the technique order the metric values in ascending order and take the maximal metric value that represents 1%, 2%, ..., 100%, of the weight. This is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale.

- *Entity aggregation*: the technique aggregates all entities per metric value, which is equivalent to computing a weighted histogram (the sum of all bins must be 100%).

- *Thresholds derivation*: thresholds are derived by choosing the percentage of the overall metric values to be represented. They propose different thresholds derived by choosing 3%, 15%, 90% and 95% of the overall metric value. These percentiles 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%).

Both Alves et al. technique, discussed in Section 2.3.1, as Vale and Figueiredo technique propose generic metric thresholds to be used to evaluate all system classes. The techniques disregard the context of the evaluated classes. In the next section, we discuss the Aniche et al. technique that proposes to consider architectural roles played by classes as context to derive metric thresholds.

2.3.3 Aniche et al. Technique

The discussion of considering some context information to use metrics is not a new idea (MARINESCU, 2006; ZHANG et al., 2013; GIL; LALOUCHE, 2016). Marinescu (MARINESCU, 2006) increases the accuracy of the detection of code smells (Data Class and Feature Envy) by making them take into account the identified design roles as factor. They identify automatically roles which design entities (classes and methods) might have within a particular enterprise application. Zhang et al. (ZHANG et al., 2013) show that metric values can be affected by factors, such as programming language, age and lifespan. Souza and Maia (SOUZA; MAIA, 2013) investigate the impact of software categories on the coupling level of software systems. They suggest that different categories may have different levels of coupling. Thereby, software systems should use distinct metric thresholds according to this category. Gil and Lalouche (GIL; LALOUCHE, 2016) discuss that metric values vary among projects, and they mean nothing when examined out of their context. In this perspective, Aniche et al. (ANICHE et al., 2016) propose SATT (Software Architecture Tailored Thresholds) that provides specific threshold for each architectural role whether it is considerably different from others in the system in terms of code metric values. They defined architectural role as a particular role that classes can play in a system architecture. For instance, CONTROLLERS in Spring MVC applications coordinate the flow between the user interface and the domain layer. In the following, we detail the eight steps of the proposed method:

- *Dataset creation*: select systems that follow the analyzed architecture, e.g., Spring MVC applications. The technique performs this step only once and use the same benchmark to calculate the thresholds for all other architectural roles.

- *Architectural roles extraction*: The technique identify each class' architectural role in the benchmark. To determine the architectural role for classes in Spring MVC applications, they analysed their annotations. If a class contains one of the following annotations, they consider that class as playing that role. The name of the annotation matches with the name of the architectural role: @CONTROLLER, @SERVICE, @ENTITY, @REPOSITORY, and @COMPONENT. Android applications make use of inheritance to determine the roles. Thus, if the class inherits

from one of following classes (or its sub-classes), they consider that class to play a specific role: ASYNCTASK, ACTIVITY, and FRAGMENT.

- *Metrics calculation*: The technique calculates code metrics for all classes in the benchmark, regardless of their architectural role.

- *Statistical measurement*: They perform a statistical test to measure the difference between the code metric values in that architectural role (group 1) and the other classes (group 2). They suggested the use of non-paired Wilcoxon test and Cliff's Delta between the two groups. Bonferroni correction should be applied, as the approach is performed for all combinations of architectural roles and code metrics.

- *Analysis of the statistical tests*: If the difference is significant and the effect size ranges from medium to large, they continue the approach. Otherwise, they stop.

- *Weight ratio calculation.* They use lines of code (LOC) as a weight of all classes. Thus, the technique calculate LOC for all classes and normalize it for all classes that belong to that architectural role in the benchmark. Normalization ensures that the sum of all weights will be 100%.

- *Weight ratio aggregation*: the technique orders classes according to their metric values in an ascending way. For each class, it aggregate the weights by summing up all the weights from classes that have smaller metric values, i.e., classes that are above the current class.

- *Thresholds derivation*: the technique extracts the code metric value from the class that has its weight aggregation closest to 70% (moderate), 80% (high), and 90% (very high).

Although the proposed technique takes a step towards to improve the identification of the context played by each system class, assigning the architectural role as context to derive metric thresholds, the proposed heuristic to identify architectural roles were able to identify and associate architectural roles to only 17.5% of the classes in MVC-based systems and 10.5% of the classes in Android applications. Consequently, metric-based assessments disregard many roles played by the rest of the classes. In the next section, we deepen this discussion by showing that other design decisions also impact on metrics.

2.4 FACTORS IMPACTING ON METRIC THRESHOLDS

The source code quality is usually compared to industry standards. A common way to define these standards is extracting them from systems benchmark. For example, Section 2.3 shows some benchmark-based techniques to extract metric thresholds used to source code assessment. However, studies report that the resulting metric thresholds are benchmark dependent (VALE; FIGUEIREDO, 2015). Some factors must be taken into account in selecting the systems to compose the benchmark due to their influence on the resulting metrics thresholds.

The application domain and system size are factors pointed out by some studies (FERREIRA et al., 2012; SOUZA; MAIA, 2013; ZHANG et al., 2013; MORI et al., 2018). For example, Ferreira et al. (2012) propose to consider the application domain and system size as factor to build benchmarks used to derive metric thresholds. However, they do not examine whether these factors affects the distribution of metric values. Souza e Maia (2013) claim that distribution of coupling metrics values vary according to application domain, called by them as category. Thus, different software categories should use distinct threshold for coupling metrics. For instance, Games category has a high level of class coupling in comparison to Development category. Mori et al. (MORI et al., 2018) investigate whether and how thresholds vary across domains by presenting a large-scale study on 3,107 software systems from 15 domains. The results indicate that metric thresholds are sensitive to software domain. For example, some metrics may vary across domains from 1.5x to 4.8x. Moreover, they observed that not only the domains, but also the size of the systems that compose the benchmark is a factor that affect the metric thresholds. Finally, they also discussed that domain-specific metric thresholds are more appropriated than generic ones for code smell detection.

However, we believe that the application domain and system size alone do not explain the differences in the derived metric thresholds. We hypothesize that the design decisions made over each system class are the main factors that influence the obtained values. For example, the threshold value used to assess the quality of a health system (health domain) developed for a mobile platform can be quite different from other developed for a Web platform. Therefore, the development platform could be an important design decision influencing the derived metric thresholds.

Zhang et al. (ZHANG et al., 2013) show that programming language is one the most influential design decisions to be considered as context and that have a strong influence on the derived metric thresholds. Aniche et al. (ANICHE et al., 2016) show that the architectural role played by classes is another design decision to take into account to derive metric thresholds. However, we hypothesize that the design decisions studied so far to derive contextual metric thresholds are too coarse-grained to explain the differences in the distribution of the metrics. In this thesis, we carry out an in-depth analysis evaluating the class design role as a fine-grained design decision used as a new context factor that could influence the derived metric thresholds.

To obtain the practitioners' perception using the class design role to improve code analysis practices, Chapter 3 shows the results of a large-scale survey with Brazilian practitioners. In Chapter 5, we show the results of an empirical study conducted to evaluate the impact of class design role and other design decisions discussed in software metrics.

Chapter 3

A SURVEY OF SOFTWARE CODE ANALYSIS PRACTICES IN BRAZIL

This chapter investigates the difficulties developers face in applying automated static analysis practices without limiting the target audience to developers who already use these practices regularly. We carry out a more comprehensive study at the beginning of this research, encompassing research questions related to software code review and containing a deeply statistical analysis (DÓSEA et al., 2020). However, this chapter reports only the research questions and the main results that motivated the studies in this thesis. We conducted a survey that aims to answer our second general research question:

RQ1: How do practitioners perceive automated static analysis for code smell identification?

To answer this research question, we conducted a large-scale web-based survey with 350 Brazilian practitioners that most often do not have well-established practices to use automated static analysis. The main goal is to assess whether ongoing researches are addressing the same issues and challenges practitioners face to apply static code analysis in their development process. Some studies suggest that only very few software projects adopt these tools because programmers seem to not fully benefit from them (KUMAR; NORI, 2013; JOHNSON et al., 2013; BELLER et al., 2016).

An initial hypothesis for the low use of Automated static analysis tools (ASATs) could be the lack of importance development teams and their companies assign to static code analysis practices. Benefits of code analysis for software quality are already long recognized (FAGAN, 1976). Surveys conducted with developers who regularly perform code analysis indicate that developers spend 10-15 percent of their time in code analysis. Finding defects, performing code maintainability improvements, and knowledge transfer are the main reasons to perform static code analysis (BACCHELLI; BIRD, 2013; BOSU; CARVER, 2013; BOSU et al., 2017). Additionally, other studies recognize that code

anomalies impact the effort of different activities, such as editing, navigating, and reading of source code (SOH et al., 2016). Although empirical evidence on the benefits of static code analysis practices encourages their use, many software companies do not apply them regularly (JOHNSON et al., 2013). Previous surveys addressing difficulties to apply static code analysis limited their target audience to software developers that regularly use these practices(KONONENKO; BAYSAL; GODFREY, 2016; MACLEOD et al., 2018; CHRISTAKIS; BIRD, 2016). Therefore, the issues and challenges faced by developers who do not apply these practices regularly are unclear.

Another possible concern faced by developers is to fit static code analysis practices in their particular development process. Many studies about practices and tools focus on their correctness, completeness or performance in companies that already use these practices (CHRISTAKIS; BIRD, 2016; BOSU et al., 2017). But when an organization needs to integrate these practices in their development process other considerations need to be taken into account. For example, Ayewah et al. (AYEWAH et al., 2008) conducted some interviews to get qualitative feedback about systematic policies used by FindBugs users. They report that 76% from users do not have systematic policies for using FindBugs and 81% do not have a policy on how soon each FindBugs issue must be human-reviewed. Therefore, when code analysis practices are unclear into the development process they could be easily disregarded (LAVALLÉE; ROBILLARD, 2015). The issues and challenges faced by development teams to integrate software code analysis practices into the software development process are unclear. A deeper understanding of these problems could guide researchers' future work to offer alternatives and improve guidelines.

Finally, some studies have reported that current ASATs are prone to false-positive alarms. These false-positives correspond to warnings that manual inspection reveals no effect on the software quality and on maintenance effort (OLBRICH; CRUZES; SJØBERG, 2010; KHOMH et al., 2011; SJOBERG et al., 2013; YAMASHITA, 2013; PALOMBA et al., 2014; HOZANO et al., 2018). Despite developers being able to eliminate many defects using the warnings produced by these tools, the overload of warnings and the way in which they are presented are pointed out as the main barriers to the consistent and widespread use of ASATs (AYEWAH; PUGH, 2010; JOHNSON et al., 2013). Some studies indicate that only 6% to 22% of warnings are removed in the context of code analysis (KIM; ERNST, 2007; PANICHELLA et al., 2015). Analyzing warnings is a time-consuming activity. For instance, a study conducted at Google indicated that, on average, eight minutes are required to manually triage each static analysis warning (RUTHRUFF et al., 2008). A line of research to reduce these false alarms is improving the accuracy of metric thresholds used by popular metric-based ASATs (MARINESCU, 2004; ARCOVERDE et al., 2012; BALACHANDRAN, 2013; OIZUMI et al., 2016). These tools usually use generic metric thresholds for classifying source code elements (such as classes and methods) of one or more systems into categories (e.g. low or high) (LANZA; MARINESCU, 2006; ALVES; YPMA; VISSER, 2010; FERREIRA et al., 2012; OLIVEIRA; VALENTE; LIMA, 2014; FONTANA et al., 2015; VALE; FIGUEIREDO, 2015). For instance, Lanza and Marinescu (LANZA; MARINESCU, 2006) classify as *long* any method that has more than 20 lines of code (LOC) in Java systems. In this case, 20 is used as a generic threshold for LOC.

However, some studies suggest the major reason for the occurrence of false positive and false negatives warnings is the lack of context for metric thresholds (ZHANG et al., 2013; ANICHE et al., 2016; SHARMA; SPINELLIS, 2018; DÓSEA; SANT'ANNA; SILVA, 2018; SOBRINHO; LUCIA; MAIA, 2018). For example, for systems that follow the layered architectural style, there might be differences between the average source code complexity of classes belonging to the View layer and the Business layer. The business logic implementation is often more complex than View logic implementation. A generic threshold value might be low for classes in a layer or high for classes in another layer. Too low or too high thresholds may lead to false code smell alarms (false positives) or may hide potential code smells (false negatives). Therefore, applying a single generic threshold to evaluate classes in these distinct layers may not make sense. An obvious solution would be to propose multiple threshold values for each metric. In our example, we could offer a distinct metric threshold for each architectural layer of the system. In this case, architectural layers would be used as context to define multiple metric thresholds. However, the practitioners' perception about multiple metric thresholds for source code evaluation is also unclear.

Our results show that, although 84.85% of respondents claimed to use at least one code analysis practice and 54.85% declared to use tools that support code analysis, they do not apply these practices and tools regularly. The results also show that we can not justify this lack of regularity to the level of importance developers and companies give to code analysis practices. The respondents also pointed out that fitting these practices to the software development process and configuring the tools are key challenges they face. Another finding is that developers agreed that using multiples metric thresholds for each metric, configured according to the class design context, could decrease the number of false alarms.

The remainder of this chapter is structured as follows. Section 3.1 presents details of the methodology used to perform the survey, including purpose, instrument, and data collection. Section 3.2 discusses results of the survey, including background information of the respondents and issues and problems reported by practitioners to apply code analysis practices. Section 3.3 discusses threats to validity. Section 3.4 brings the related works. Finally, section 3.5 brings the summary, remarks, and future work.

3.1 STUDY SETTINGS

In this section we present the goal and research questions of our survey (Section 3.1.1). We then discuss the statistics on the sample and the design of the survey questionnaire in Section 3.1.2. Section 3.1.3 discusses the survey execution and data collection procedures. In Section 3.1.4 we discuss the analysis methodology used to answer the proposed research questions.

3.1.1 Goal and Research Questions

The goal of this survey is to characterize code analysis practices that software developers in Brazil use to ensure source code quality. Our purpose is identifying trends, difficulties

and challenges to use these practices which could be addressed by researchers both in Brazil and worldwide. Based on this goal, we raise the following research questions (RQ).

- *RQ1: What are the code analysis practices adopted by developers in Brazil to evaluate source code quality?* With this research question we aim to gain a broader view of the most used code analysis practices. We also intend to assess how often these practices are used.

- *RQ2: How important do developers in Brazil perceive code analysis practices?* With this research question, we aim to understand which level of importance developers in Brazil assign to code analysis practices, since this perception could affect the adoption of such practices.

- *RQ3: What difficulties do developers in Brazil face to use automated static analysis tools?* Several studies discussed the completeness of ASATs to point out relevant alarms, but only few discuss issues and challenge to fit them in the software development process (VASSALLO et al., 2018). In this research question, our goal is identifying problems that may make developers in Brazil avoid using automated static analysis tools.

- *RQ4: What is the developers' perception about evaluating source code using multiple threshold values for each metric?* Most ASATs rely on metrics and thresholds to point out pieces of low-quality source code. For a metric, ASATs usually use a single generic metric threshold to evaluate all system classes. However, calibrating metric thresholds without taking context into account can increase the number of false-positive warnings (SHARMA; SPINELLIS, 2018). In fact, recent studies claim that considering context factors to define multiple thresholds could improve accuracy and reduce false-positive alarms (ZHANG et al., 2013; ANICHE et al., 2016; DÓSEA; SANT'ANNA; SILVA, 2018). In this research question, we aim to evaluate developers' perception about single and multiple metric thresholds. To make the notion of context more concrete to the respondents, in our questionnaire we suggested two context factors that could be used to define multiple metric thresholds: (i) the architectural layer of a class, and (ii) the main business entity handled by a class. The hypothesis about business entities is that some of them may be simpler to be handled than others. For example, considering a library management system, the business entity *author of books* is usually simpler to be handled by the system than *books* and *borrowing of books* entities that usually involve more data and more complex business operations. We suggested these two factors based on factors considered in previous studies (ANICHE et al., 2016; DÓSEA; SANT'ANNA; SILVA, 2018), but they may not be the only factors that affect metric thresholds.

We used each RQ to derive one or more "survey questions" detailed in the following section. The exploratory nature of the survey questions aimed to clarify the current issues faced by practitioners to perform source code analysis. Additionally, they enabled us to seek new insights and generating hypotheses for future studies.

3.1.2 Survey Design

We used guidelines reported by (KITCHENHAM; PFLEEGER, 2002; KASUNIC, 2005; LINAKER et al., 2015) to design the survey. Surveys have been used in empirical software engineering investigations to learn about the state of the practice, identify improvement potentials, or investigate the acceptance of a technology insight (PUNTER et al., 2003). We describe below the sampling method, survey questions, execution process, and analysis methodology.

3.1.2.1 Population and Sampling Method In our study, the target population is formed by Brazilian software practitioners engaged on the software development industry. We did not find official data in Brazil about our target population. For this reason, we use a non-probabilistic sampling, used by researchers when systematic probabilistic sampling is not possible. However, a non-official estimate published by SOFTEX (Association to Promote the Excellence of Brazilian Software)[1] and widely used by Brazilian press estimates 570 thousand Information Technology (IT) Brazilian professionals in 2018, working in a wide range of areas, including telecommunications, networks and software development. A total of 411 respondents started the questionnaire, whereof 350 (85,15%) completed all mandatory questions. We then used the R tool to determine the level of confidence and the margin of error of the sample considering this total estimate of IT professionals. We obtained a confidence level of 95% and a margin of error of 5.24%. This means that, if we undertake 100 surveys for the same purpose and with the same methodology, in 95 of them the results would be within the margin of error. The size of our sample is considerable when compared with previous surveys in software engineering, especially Brazilian surveys (AGNER et al., 2013), which reach much smaller numbers of respondents and, therefore, much larger margin of error.

We decided to use a self-recruited survey, in which the respondents get to know somehow about the survey and decide to participate. The main advantage of a self-recruited survey is that respondents are attracted already by the topic of the survey (PUNTER et al., 2003). We used an online questionnaire created by means of the SurveyMonkey tool[2] to collect the data. The motivation for using an online questionnaire was to maximize coverage and participation. It also allows an easier data entry from the respondent perspective, a simpler data collection from the researcher perspective and is less error-prone (PUNTER et al., 2003).

3.1.2.2 Survey Questions We designed our survey questionnaire aiming to answer the research questions (Section 3.1.1) and to characterize the respondents. We were also concerned to avoid a large questionnaire, which would take a long time from the respondents. Studies showed that short questionnaires have a higher response rate in comparison to long ones (PUNTER et al., 2003; SMITH et al., 2013). Thus, we considered five minutes as target time for the participants to answer all questions.

[1] https://www.softex.br/
[2] http://www.surveymonkey.com

To achieve this goal, we elaborated clear and objective questions. Additionally, we conducted two pilot studies before coming up with the final version of the questionnaire. The first one involved five undergraduate students attending the last period of an information systems course. The second one involved 15 experienced professionals (5 to 20 years working in software development industry) who attended a course given by the author of this thesis. The pilot studies involved these two group of developers because we also aimed to assess whether the proposed questionnaire was suitable for both new developers and expert developers. During the pilot studies, we monitored time, questions the participants made, and registered misunderstandings due to question formulation. Everyone was able to answer the questionnaire within 4 to 5 minutes. Then we discussed with them their understanding about each question. As a result of the pilot studies, we made some adjustments in the vocabulary used in the questions. Some participants raised doubts about practices or tools mentioned in some questions. Thus, we added examples to illustrate them and make the questions clearer.

The questionnaire is organized into four sections. Each section was showed to the respondent in a distinct Web page. All the pages had the same title that reflected our main research goal: "Which practices do you use to evaluate the quality of source code?". The first section informs to the respondents that the survey would required around five minutes from them. It also informs that the survey has eleven questions. In addition, the first section gives a brief explanation about source code analysis practices and tools.

Table 3.1 lists the survey questions. For each question, it informs the questionnaire section where the question appears, the related research question, the question itself (translated from Portuguese into English), the type of allowed answer and the number of respondents. Overall, the questionnaire consists of seven questions, plus four background questions. Some questions allow the respondent to select a single answer (S) and other questions allow the respondent to select multiple answers (M). Additionally, some questions allow an open answer (O) where the respondent can write a free text response to add an answer not included in the list of answer options. For instance, the first question is related to research question RQ1 and appears in the second section of the survey. It allows multiples answers and an open answer. And 411 respondents answered it.

The second section of the survey has four questions related to the first two research questions. To answer RQ1 it includes two questions about the code analysis practices the respondent uses and with which frequency he or she uses them. To answer RQ2 the questionnaire contains two questions about the importance given by the respondent and the company to the use of code analysis practices. All 411 respondents who started the survey advanced through the first and second sections, answering all the questions. We elaborated the simplest questions in the second section aiming to encourage the progress to the other survey questions.

The third section contains two questions aiming to identify issues and challenges faced by practitioners to use code analysis techniques. Each question is associated to one research question (RQ3 and RQ4). The third section contains more reflective questions and obtained high dropout rate 56 out of 411 (13.62%) of the respondents.

Finally, the four section is concerned with the background of the participants, including questions about experience with software development as well as about the number of

Table 3.1: Questionnaire - Questions (S/M/O Stands for Single, Multiple, or Open answer).

Section	Research Question or Background	Question	S/M/O	#Respondents
2	RQ1	What code analysis practices do developers in your company use to analyze the quality of source code?	M, O	411
2	RQ1	How often do you apply code analysis practices?	S	411
2	RQ2	What importance do you give to code analysis practices?	S	411
2	RQ2	What importance does your company give to code analysis practices?	S	411
3	RQ3	What difficulties do you have to use automated static analysis tools?	M, O	355
3	RQ4	Automated Static Analysis tools usually use a single metric threshold to evaluate all system classes. Considering a three-tier system (GUI, Business and Persistence), what is your opinion about the threshold values that should be used to evaluate classes in each of these three tiers?	S	355
4	Background	What is your highest academic degree?	S, O	350
4	Background	What is your current role in the company?	S,O	350
4	Background	How much experience do you have in software development?	S	350
4	Background	How many systems have you developed or performed maintenance tasks?	S	350

developed systems that they already worked with. We structured the background section at the end aiming the respondent to focus from the early stages on the main objectives of the survey (SEAMAN, 1999). We believe that this approach also influenced the high rate of responses we obtained. In the last section, there were only four dropouts. In summary, the median number of responses per question was 411 for RQ1 and RQ2 questions, 355 for RQ3 and RQ4 and 350 for background questions.

3.1.3 Execution

We made the survey available through an online questionnaire created by means of the SurveyMonkey[3] tool. The sampling considered was 350 respondents. Given the lack of any reliable data about the population of Brazilian software developers, we selected developers to participate in the survey as follows:

- We sent invitations through Google, Linkedin and Facebook software developers online forums and private groups, such as, C# Brazil [4], Java Brazil[5], Web Development Brazil[6] and Android Brazil[7]. We were also invited to publish the survey on the main page of an important Brazilian software quality website[8]. The publication on the site was available for approximately one month.

- We sent invitation emails to the main researchers associated with this field of study

[3]http://www.surveymonkey.com
[4]https://www.facebook.com/groups/csharpbrasil
[5]https://www.facebook.com/groups/JavaBr/
[6]https://www.facebook.com/groups/desenvolvimentoweb/
[7]https://www.facebook.com/groups/androidbrasiloficial/
[8]http://qualidadedesoftware.com.br/

in Brazil. We requested that they submit the survey to students and professionals engaged in the area of software development.

The survey was conducted from January 2015 until September 2018. Most of the questionnaires (363 out of 411) were responded by accessing the online survey in January 2015, which was the most intense period of dissemination of the survey. Aiming to improve the sample reliability, we performed a new disclosure in August 2018, totaling 411 forms. We noticed that the new responses did not significantly alter the results obtained with the first responses, demonstrating that the sample would already be large enough to obtain reliable results.

3.1.4 Analysis methodology

We address the research questions discussed in Section 3.1.1 through descriptive statistics and using statistical hypothesis testing to conduct a cross-factor analysis of source code analysis practices and practitioners' background. We aim to understand the challenges of using code analysis practices associated with each class of practitioners. However, this section reports only the results of descriptive statistics. The results from statistical hypothesis testings are available in (DÓSEA et al., 2020).

3.2 RESULTS AND DISCUSSION

From 411 professionals who accessed the Web questionnaire, 350 filled the entire questionnaire, yielding a 85.15% response rate. This rate is higher than other on-line surveys in software engineering (PUNTER et al., 2003). We considered that the following reasons were determinant for this high rate: (i) the short time required to answer the survey (5 minutes), emphasized in the first screen, (ii) the use of simple and objective language to invite the respondents, and (iii) the use of examples to explain some terms the respondents might not be familiar with. Due to the type of sampling, we are not able to determine the number of respondents who received the questionnaires, therefore, the response rate takes into account the respondents who really opened the questionnaire.

This section initially shows the background of the respondents, and it presents and discusses results regarding the four research questions.

3.2.1 Respondent background

Regarding the predominant role, Table 3.2 shows that 96 (27.43%) respondents are programmers and 97 (27.71%) are software engineers. Thirty-three (9.43%) respondents are software architects. Fifty-four (15.43%) declared themselves as quality analysts, role which usually has source code quality analysis as one of its tasks. Programmers, software engineers, software architects and quality analysts are roles concerned somehow with source code maintenance. Finally, 70 (20.00%) respondents indicated other roles, for example, technical leader of programmers. We observed that 28 out of 70 respondents that indicated other roles also performed tasks related to source code maintenance. In summary, 308 (88%) respondents play a role associated with source code maintenance

and thus, should concern with source code quality. Other roles cited by respondents are related to software testing, project management and company direction.

Table 3.2: Predominant role of the respondents.

Role	**Respondents**	**(%)**
Programmer	96	27.43%
Software engineer	97	27.71%
Software architect	33	9.43%
Quality analyst	54	15.43%
Other (please specify)	70	20.00%

In order to understand the respondents' educational background, we asked them to inform their highest academic degree. Table 3.3 shows that 175 out of 350 (50%) respondents are bachelor and 105 (30%) has postgraduate diploma. We also observed 34 (9.71%) respondents have master degree and seven have doctoral degree (2%). Only 29 (8.29%) respondents claim to have associate degree. In Brazil, associate degree varies between 2 to 3 years of full-time studies. This degree provides highly specialized knowledge (e.g. Web developer). In Brazil, postgraduate diploma requires a previous bachelor degree for admission and performs a specialization course in one area of study, mostly addressed to professional practice. The results indicate a high educational level of the respondents of which 91.71% have at least bachelor degree.

Regarding work experience, Table 3.4 shows that 112 (32%) respondents has 2 to 5 years of work experience. Others 87 (24.86%) respondents claim to have 5 to 10 and 92 (26.29%) more than ten years of work experience. Finally, 59 (16.86%) respondents have less than two years of experience. That means the majority of respondents are practitioners who have a reasonable level of experience in software development tasks.

To be sure about the level of experience level with development tasks, we also asked about the number of systems the respondents work on. Table 3.5 shows that 134 (38.51%) respondents claimed to have contributed with more than ten software projects and 75 (21.55%) between 6 to 10 software projects. Also, 75 (21.55%) respondents claimed to have contributed with 3 to 5 software projects, and only 64 (18.39%) respondents claimed to participate in less than three software projects. These results illustrate that most of

Table 3.3: Respondents' highest academic degrees.

Academic Degree	**Respondents**	**(%)**
Associate	29	8.29%
Bachelor	175	50.00%
Postgraduate diploma	105	30.00%
Master	34	9.71%
Doctoral	7	2.00%

Table 3.4: Work experience.

Years	Respondents	(%)
0 - 2	59	16.86%
2 - 5	112	32.00%
5 - 10	87	24.86%
> 10	92	26.29%

Table 3.5: Number of Developed Systems.

Systems	Respondents	(%)
0 - 3	64	18.39%
3 - 5	75	21.55%
6 - 10	75	21.55%
> 10	134	38.51%

the respondents have a reasonable experience of which 81.61% contribute with at least three software projects.

We used the data obtained from the background questions to evaluate whether respondent's profile influenced the result about the research questions presented in the next subsections.

3.2.2 RQ1: What are the code analysis practices adopted by developers in Brazil to evaluate source code quality?

To answer this research question, we analyze data from the first two survey questions (Table 3.1). First, we asked the respondents about the practices they use to analysis source code quality in their companies. Respondents could select one or more answer options. Figure 3.1 shows that 226 (64.57%) respondents claim to use manual code reviews, and of those, we observed that 96 (27.42% of 350) use manual code review exclusively. We observed that 139 (39.71%) respondents declared to use automated static analysis tools (e.g. PMD, CheckStyle, ReSharper, FxCop), 87 (24.86%) use code metric tools and 36 (10.29%) use tools developed by their company itself. Finally, 56 (16.00%) respondents declared that do not perform source code analysis and 17 (4.86%) respondents claimed to use others practices. Some respondents indicated to use some specific code analysis tools (e.g. SonarQube and CodeClimate). Others also indicated the use of unit and integration test tools, which are usually not considered as source code analysis tools. We also identified 67 (19.14%) respondents that claimed to use more than one tool to source code analysis (e.g. code metrics tool and tool developed by the company). Therefore, although manual code reviews are still the practice most often cited (64.57% from respondents), we identified 192 (54.85%) respondents that use at least one tool to perform code analysis. Finally, the results indicated that 297 (84.85%) respondents use at least manual or automated practice to source code analysis, showing that respondents

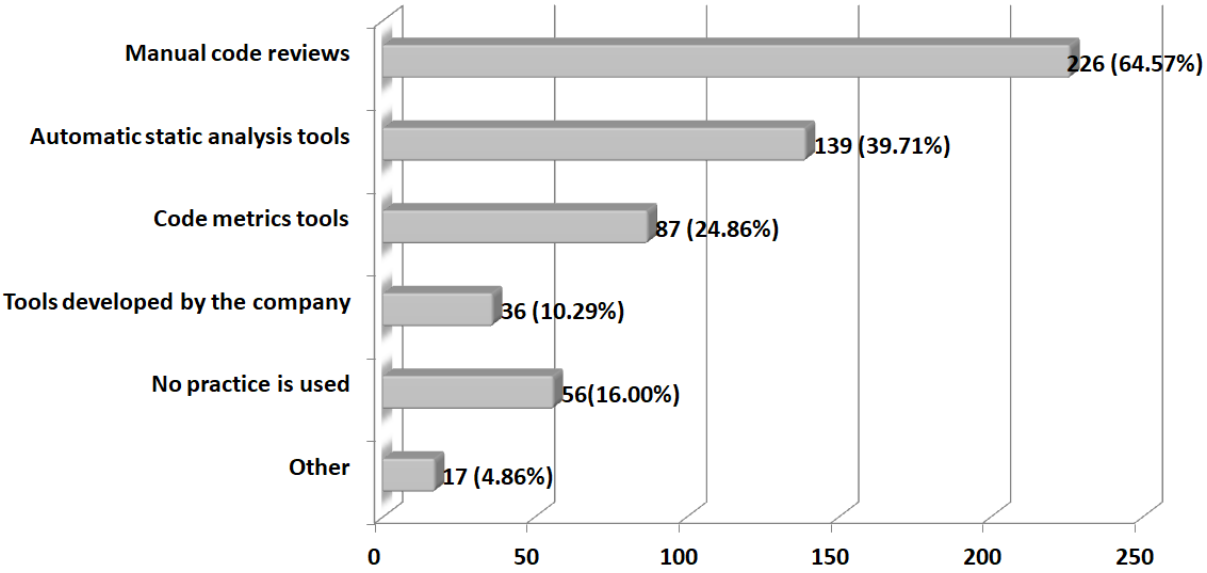

Figure 3.1: Code analysis practices adopted by the Brazilian companies to assess the quality of source code.

have reasonable level of knowledge about code analysis practices.

Secondly, we asked respondents how often they use code analysis practices. Our goal is to complement the results of the previous question by verifying how often these practices are actually applied in the development process. Figure 3.2 illustrates that 185 out of 350 (52.86%) respondents declared that there is not a well-defined time to analysis source code, but occasionally revise it. We observed 36 (10.29%) respondents that declared reviewing source code at least once a month and 46 (13.14%) that claimed never reviewing the source code. Only 83 (23.71%) respondents declared reviewing source code at least once a week. These results are not aligned the idea that software teams should meticulously review each change to source code to ensure quality standards (TANAKA et al., 1995). A survey conducted with Microsoft and OSS developers that adopt code analysis practices regularly found that they spend approximately six hours per week in code review (BOSU; CARVER, 2013). Comparing these results with our survey results, we can say that 76.29% of Brazilian practitioners do not perform code analysis regularly.

In summary, based on the responses to the first two questions of our survey, we can answer RQ1 as follows:

Software code analysis practices are well disseminated among Brazilian practitioners. However, they are not applied regularly in the software development process. Thus, the positive impact of these practices on source code quality may not be perceived by development teams and companies.

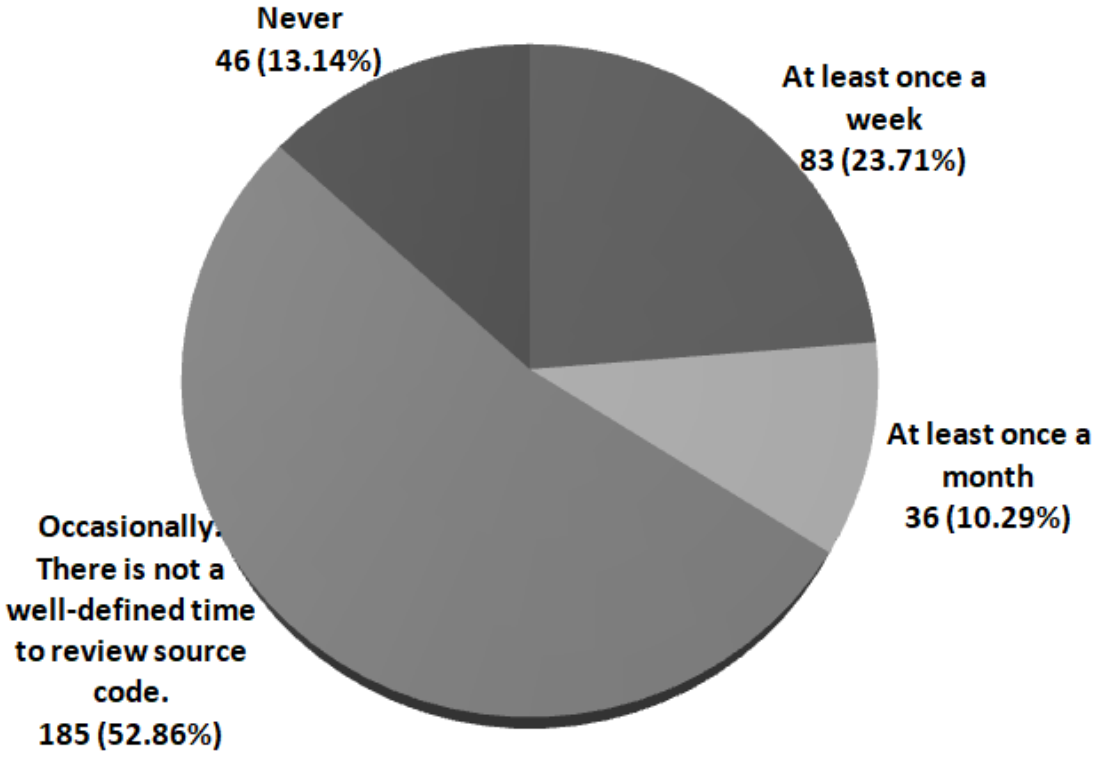

Figure 3.2: Frequency of use of each code analysis practice.

3.2.3 RQ2: How important do developers in Brazil perceive code analysis practices?

For code analysis practices to be applied regularly, recognizing its importance is crucial. In this sense, our second research question aims to figure out what level of importance Brazilian practitioners and their companies give to code analysis practices.

Figure 3.3 illustrates the level of importance both practitioners and companies give to code analysis practices. It puts together the answers for the third and forth questions of our survey, which are: *What importance do you give to code analysis practices?* and *What importance does your company give to code analysis practices?* We can notice that, for 239 (68.29%) respondents, code analysis practices are very important and, for 98 (28%), they are important. Only 11 (3.14%) and 2 (0.57%) respondents declared to consider code analysis as slightly important or as not important, respectively. Regarding the level of importance attributed by companies, 97 (27.71%) and 137 (39.14%) respondents declared to believe that their companies consider code analysis practices as very important or as important, respectively declared their companies give. On the other hand, 95 (27.14%) respondents declared their companies give slightly importance. Finally, 21 (6.00%) claimed to believe that their companies do not give any importance to code analysis practices.

This data allows us to raise the hypothesis that developers believe that companies give less importance to code analysis practices than they do. Thereby, the analysis of

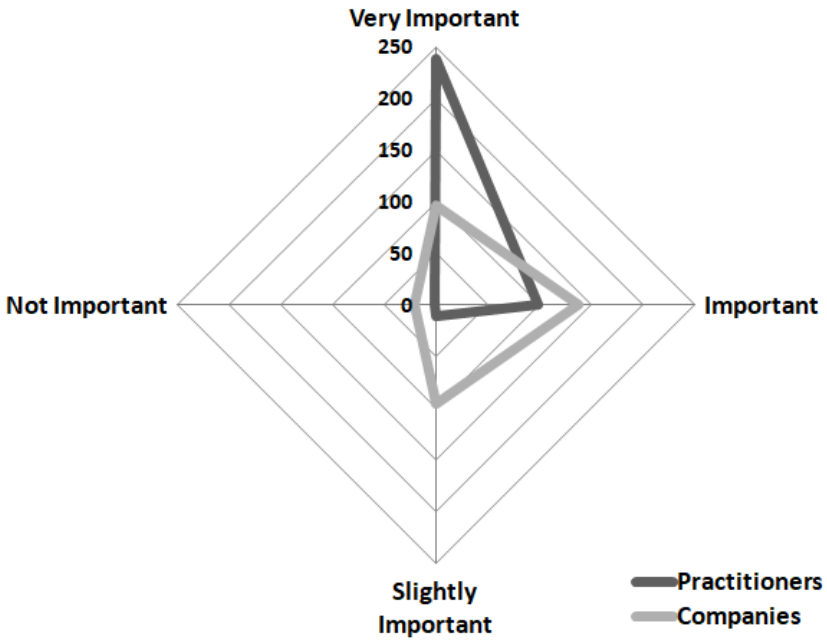

Figure 3.3: Importance of code analysis practices.

these results allows us to answer RQ2 as follows:

> *Practitioners recognize code analysis practices as important to the software development process. However, their perception is that companies give less importance to code analysis practices than they do. This perception may discourage the regular use of these practices.*

3.2.4 RQ3: What difficulties do developers in Brazil face to use automated static analysis tools to support code analysis?

A lot of research effort has been put into improving automated static analysis tools (ASATs) aiming to make code analysis more objective and standardized. However, a study conducted with 168,241 OSS projects showed that, despite 60% of the projects make use of ASATs, they typically only use one ASAT in an ad-hoc fashion and not integrated with the flow of development. In addition, the configurations of the ASATs used in those projects barely deviate from the default or introduce custom checks (BELLER et al., 2016). Our third research question aimed to evaluate the issues and challenges Brazilian practitioners face to use automated static analysis tools.

To answer this research question, we only use the fifth question of our survey, which is *What difficulties do you have to use code analysis tools?* It is a multiple choice question, so that respondents could select more than one difficulty that they consider to hinder their regular use of ASATs. Figure 3.4 illustrates the obtained results. We discussed in RQ1 that 54.85% of the respondents claim to use at least one tool to perform code analysis. We found in RQ3 that 149 out of 350 (42.57%) respondents declared lack of knowledge about

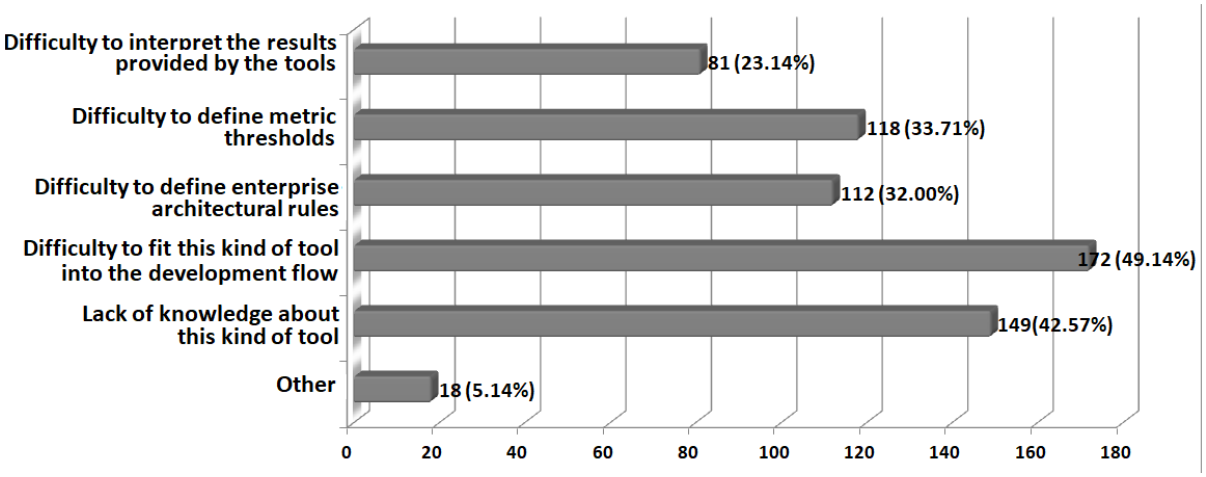

Figure 3.4: Difficulties to use automated code analysis tools.

these tools. These results mean that almost everyone who knows ASATs tools uses them in some way. Interestingly, respondents who are unaware of ASATs are scattered across all levels of experience. For instance, regarding the respondents who claimed to have less than two years of experience, 28 out of 112 (25%) are unaware of ASATs. Other levels of experience have similar rates. For example, among the 216 respondents with two to five years of experience, 50 (23.14%) are unaware of ASATs. These numbers demonstrate that ASATs need to be better disseminated among practitioners.

Still regarding difficulties to use ASATs, also a high number of respondents, 172 (49.14%), claimed to have difficulties to fit the tool into the development flow. Difficulties to fit ASATs into the development flow are also reported by Microsoft developers that use ASATs more regularly (CHRISTAKIS; BIRD, 2016). Developers point out insufficient training and problems to manage large reviews as challenges to fit these tools in development flow (MACLEOD et al., 2018). These results show that improving ASATs may not be enough. To reach the benefits promised by the use of ASATs, companies and researchers also need to invest in education and guidelines to fit ASATs into their software development flow. In RQ5 we deep the discussion about the best time of the development flow to apply ASATs.

In addition, a considerable number of respondents, 112 (32%), claimed to have difficulties to define enterprise architectural rules of the system under review (e.g. rules of communication between architectural layers). Defining the key architectural rules is an essential task for setting parameters, rules and metric thresholds used by many ASATs to perform code analysis process. This is a challenge also cited by other studies conducted with Microsoft and Mozilla core developers (KONONENKO; BAYSAL; GODFREY, 2016; MACLEOD et al., 2018). These discussion highlight the need for studies and tools to help developers quickly understand the key architectural decisions of the system under review, such as, architectural layers and their rules of communication.

A similar number of respondents, 118 (33.71%), declared difficulties to define metric thresholds (e.g. maximum number of lines of code per method). A number of ASATs

allow users to adjust metric thresholds used to identify code anomalies. The accuracy of metric-based assessment is heavily influenced by the calibration of metric thresholds (SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018). Threshold selection is a challenge because of the proneness to false positives (KESSENTINI et al., 2014). A threshold that points out code smells that hold good in the context of an application module may not necessarily make sense for other applications or other modules of the same application (FONTANA et al., 2015). Previous works suggest that deriving metric thresholds according to the application design context might reduce false code smell alarms (ZHANG et al., 2013; ANICHE et al., 2016; DÓSEA; SANT'ANNA; SILVA, 2018). In RQ4 we discussed the developers' perception about considering the context to derive metric thresholds.

Finally, 81 (23.14%) respondents declared problems to interpret the results pointed out by the tools. This rate seems to be low but we need take into account a high rate of respondents that declared lack of knowledge about ASATs. In fact, the way ASATs present results is considered as one of the main barriers to the consistent and widespread use of ASATs by many studies (JOHNSON et al., 2013; BACCHELLI; BIRD, 2013; KONONENKO et al., 2015; CHRISTAKIS; BIRD, 2016). The high number of false alarms, which is also considered as one of the main barriers to ASATs use (JOHNSON et al., 2013; BACCHELLI; BIRD, 2013; KONONENKO et al., 2015; CHRISTAKIS; BIRD, 2016), may hinder ASATs results interpretation too. Eighteen respondents (5.14%) cited other difficulties to use ASATs, including: (i) lack of culture and knowledge of developers and (ii) difficulty use ASATs in legacy code, because it usually comprises design rules different from newer applications. These results also show that training is needed and that ASATs should be adapted to particular contexts.

Therefore, the analysis of these results allows us to answer RQ3 as follows:

> *Many Brazilian practitioners are still unaware of ASATs or have difficulty to adapt them to their development flow or have difficulty to interpret their results. Thus, research involving ASATs should not only be limited to improving accuracy but must also be concerned with providing guidelines for developers to use them and to adjust them to their particular software development processes.*

3.2.5 RQ4: What is the developers' perception about evaluating source code using multiple threshold values for each metric?

A major reason for the occurrence of false positive and negatives on metric-based code smells detection is the lack of context for metric thresholds (SHARMA; SPINELLIS, 2018). Nevertheless, popular ASATs use generic metric thresholds for the metrics used for detecting code smells. We have a generic threshold for a given metric when we use the same single value for classifying into categories (such as low or high) every class (or every method) of one or more systems. For instance, Lanza and Marinescu (LANZA; MARINESCU, 2006) classify as long any method that has more than 20 lines of code (LOC) in Java systems. In this case, 20 is used as a generic threshold for LOC. Using a generic metric threshold for each metric to evaluate all system classes ends up disregarding con-

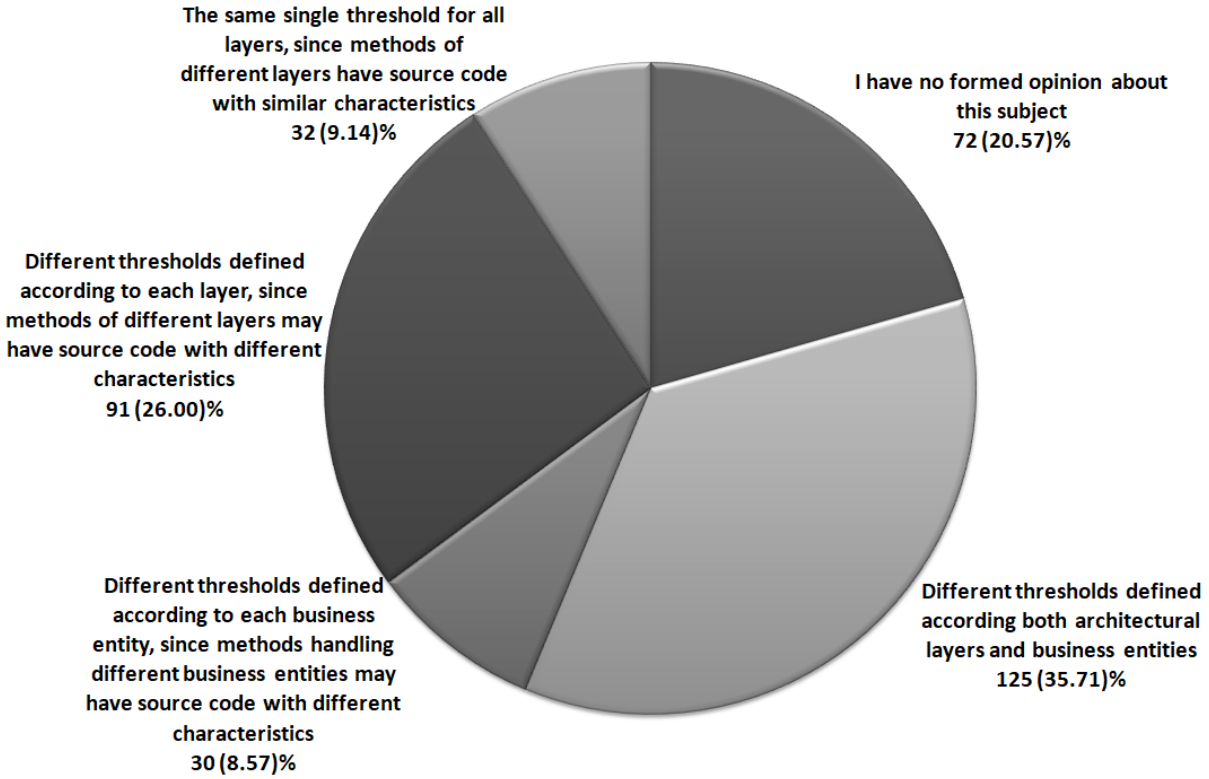

Figure 3.5: Practitioners' perception of the influence of class context in the selected metric thresholds.

textual information of each evaluated class. Some studies have shown that generic metric thresholds might not make sense for the entire set of classes in a system (LAVAZZA; MORASCA, 2016) and taking into account context factors to define multiple context-sensitive thresholds could improve accuracy and reducing false-positive alarms (ZHANG et al., 2013; ANICHE et al., 2016; DÓSEA; SANT'ANNA; SILVA, 2018). RQ4 aims to evaluate the practitioners' perception about context-sensitive metric thresholds to source code evaluation.

To answer RQ4, we rely on the the sixth question of our survey, which is: *Automated Static Analysis tools usually use a single metric threshold to evaluate all system classes. Considering a three-tier system (GUI, Business and Persistence), what is your opinion about the threshold values that should be used to evaluate classes in each of these three tiers?* It is a single question, so that the respondent is only allowed to choose one answer. The question asks the respondent to consider, as example, a system developed according a three-tier architecture (GUI, Business and Persistence). Then, it asks the opinion of the respondents about whether a single metric thresholds should be used to evaluate the entire source code or different thresholds should be used for different architectural layers or business entities. By business entities we mean the main entities handled by the system. For instance, in a library management system, some examples of business entities are book, book author, publisher, book borrowing, and book return. We hypothesize that some business entities may be more complex (e.g. book borrowing) to be handled by the

system and therefore should be evaluated with threshold values different from the ones used for simpler business entities (e.g. book author).

Figure 3.5 shows that only 32 (9.14%) respondents declared given a metric, the same single threshold should be used to analyze source code in all three layers. This result shows that practitioners' perception is different from the strategy adopted by most ASATs, which allow only a single threshold for each supported metric. On the other hand, 91 (26.0%) respondents claimed that metric thresholds should be different and defined according each architectural layer, since the source code in methods of one layer may have distinct characteristics from methods in the other layers. Also, 30 (8.57%) respondents declared that metric thresholds must be different and defined according to each business entity, since methods that handle different business entities may have different characteristics. Moreover, 125 (35.71%) respondents believe that different metric thresholds should be defined according both architectural layer and business entities. In summary, 246 out of 350 (70.28%) respondents do not agree with the use of single generic thresholds. Finally, 72 (20.57%) respondents did not have a formed opinion regarding this subject. These results motivate future research for investigating whether using multiple metric thresholds reduces the number of false alarms current ASATs return.

In summary, we observed only few respondents that agree with single generic metric thresholds usually adopted by the most popular ASATs. This allow us to answer RQ4 as follows:

> *Instead of a single generic metric threshold, practitioners believe that ASATs should use multiple thresholds, calibrated according contextual design information, such as architectural layers. Thus, future research should further investigate the use of multiple thresholds.*

3.3 THREATS TO VALIDITY

In general on-line surveys are considered to have lower internal validity and stronger external validity in comparison with other means of empirical investigation, such as case-studies or experiments (PUNTER et al., 2003). We discuss the threats to the validity of our study according to four categories (WOHLIN et al., 2012):

Construct validity. In a survey, such threats may mainly occur because respondents could possibly interpret a question in a different way than it has been conceived, possibly producing misleading results. To minimize this threat, as explained in Section 3.1, we tested the questionnaire, by means of two pilot studies, to check possible problems related to ambiguity, missing response options, and lack of clarity in our questions. Also, to make important concepts and terms (e.g., static analysis tools) clear, we included examples of them in the questions or answer options.

Internal validity. Threats to internal validity are related to issues that may affect the causal relationship between treatment and outcome. In general, it is hard to control these factors since survey is an unsupervised study and the level of control is very low. To avoid apprehension, we guaranteed the respondents their complete anonymity. Also, there is always a risk that different respondent backgrounds (e.g., experience) influence

the experiment results. However, due to the large sample, as well as the range of competence and experience levels, this risk was limited. Similarly, the large sample mitigated any threat potentially caused by respondents with different personalities (FELDT et al., 2010).

External validity. Despite the size of our sample being large enough to enable statistically significant results, we are also aware that the sample size could not be large enough for generalization purposes given the lack of accurate data about the target population. However, our sample is similar to other surveys conducted on different software engineering subjects (CHRISTAKIS; BIRD, 2016; BOSU et al., 2017). In addition, we do not claim that our conclusions can be generalized outside the scope of our study.

Conclusion validity. Threats to conclusion validity are concerned whether correct conclusions are reached through rigorous and repeatable treatment (WOHLIN et al., 2012). To minimize possible errors related the target audience sampling, we used, for each research question, non-parametric statistical tests and measured the effect size to discuss our findings. Therefore, all the conclusions that we drew in this survey are strictly traceable to data. Moreover, to increase transparency, the survey data is available online[9] so that other researchers can validate it or replicate the study.

3.4 RELATED WORKS

Some studies capture the perception of practitioners who already use code analysis practices regularly. Differently, we conducted a large scale survey without limiting the target audience to code analysis experts. With this, we expected to capture others issues and challenges about code analysis practices adoption. In addition, we also attempted to capture the practitioners' perception about multiple metric thresholds that take the context of source code elements into account. Using design-sensitive metric thresholds is a possible way to avoid what many software developers consider as a key problem on the use of ASATs: the high number of false alarms. Previous studies do not address this point.

An initial small-scale study conducted by Johnson *et al.* (JOHNSON et al., 2013) investigated 20 developers using semi-structured interviews to know why static analysis tools are not widely used and how these tools could be improved to increase usage based on developer feedback. The study focused on static analysis tools that have well-defined programming rules to find defects (e.g. FindBugs, Lint, IntelliJ, and PMD). Most of the developers (19 of 20) claimed that static analysis tools do not present their results with enough information that allows them to clearly understand the problem and know what they should be doing differently. The same number of developers expressed the importance of offering different ways to fit a tool into the software development process. Some developers prefer finding a "stopping point" in their code to run the tool (LAYMAN; WILLIAMS; AMANT, 2007). Other developers prefer the tool running in the background. Our study also investigated how developers prefer to use source code analysis tools, but based on the opinion of a considerably higher number of developers. In addition, we also investigated other issues that may discourage developers to adopt

[9]https://github.com/marcosdosea/thesis/tree/main/survey

static analysis tools.

Bacchelli and Bird (BACCHELLI; BIRD, 2013) conducted an exploratory study following a mixed approach and collecting data from different sources for triangulation. They (i) observed 17 industrial developers performing code analysis; (ii) interviewed these developers using a semi-structured interview; (iii) manually inspected and classified the content of 570 comments in discussions about code reviews; and (iv) surveyed 165 managers and 873 programmers. The results showed that finding defects and code improvement are the primary motivations to code review, although participants believe that code review brings other benefits, for example, knowledge transfer and proposition of alternative solutions. They also identified when the business context of the software is clear and understanding is very high, as in the case when the reviewer is the owner of changed files, code review comments have better quality. Bosu et al. (BOSU et al., 2017) conducted a survey with 416 Microsoft developers aiming to provide additional insight into similarities or differences between OSS and Microsoft developers. They aim to verify if Microsoft developers who work on distributed projects would have similar views about code as OSS developers (whose projects are also distributed). The results show a large amount of similarity between the Microsoft and OSS respondents and a little difference between distributed and co-located Microsoft teams. They also verify that developers spend approximately 10-15 percent of their time in code reviews, with the amount of effort increasing with experience. Our survey complements these two studies as it identifies additional issues and challenges developers face for adopting code review practices, such as problems to fit code review tools in their software development process, difficulties to define metric thresholds and difficulties to interpret ASAT results.

Beller at al. (BELLER et al., 2016) conducted a study to understand the prevalence of ASATs, their configuration in real software projects, and how those configurations evolve over time. Firstly they analyzed the use of nine popular ASATs in 122 Open-Source Software (OSS) projects. Then, they analyzed how ASATs were configured and how their configuration settings evolved in 168,241 OSS projects. The results show that 60% of the most popular and (therefore arguably) most advanced projects make use of ASATs, although they typically use only one ASAT in an ad-hoc fashion, not integrated with the flow of development. Regarding to ASAT configurations, the study showed that, after an one-week period of changes, the ASAT configurations usually remain unchanged along the rest of the project. Additionally, ASAT configurations barely deviate from the default settings and rarely comprise custom checks. Our study evaluated developers' perception about new ideas related to ASAT configurations, such as, the use of multiple metric thresholds.

MacLeod et al. (MACLEOD et al., 2018) conducted semi-structured interviews with 18 developers from four teams at Microsoft. The initial findings about tool use, developer motivations, and the challenges developers face were validated through a survey with 911 Microsoft developers. The study revealed that Microsoft developers recognize manual code reviews' value and importance. They appreciate reviewer feedback and they develop more thorough artifacts when they know that someone will revise them. The study showed that 87% of the respondents acted as a code reviewer during the previous week of the research. Improve the code, find defects, transfer knowledge and explore alterna-

tive solutions are ranked as the main motivations for code reviews. Reviewers point out the main challenge to conducted code reviews are the difficulty to manage large reviews, finding time to do reviews, understanding a changing and its motivation, finding relevant documentation and understanding the history of changes and decisions. They also complained about insufficient training for reviews. Christakis and Bird (CHRISTAKIS; BIRD, 2016) interviewed and surveyed developers across Microsoft to understand their needs and how ASATs can or do fit into their process. They also examined many corrected defects to understand what types of issues occur most and least often. They received 375 responses to the survey, yielding a 19% response rate. Developers point out the main pain points, obstacles, and challenges to use ASATs are (i) wrong checks are on by default, (ii) bad warning messages, (iii) too many false positives, (iv) too slow, (v) no suggested fixes, (iv) difficult to fit into the workflow. Developers also prefer that ASATs show alerts in the code editor followed by the build output. In addition, developers suggested that ASATs should have a false positive rate no higher that 15% to 20%. These results are in line with the findings of previous works (JOHNSON et al., 2013; AYEWAH et al., 2008). Our target audience also recognizes code analysis practices as important, but only very few respondents claimed to perform code analysis regularly. Also, as recent studies showed that context-sensitive metric thresholds could decrease the number of false positives (ZHANG et al., 2013; ANICHE et al., 2016), our study goes further on this point by investigating developers' perception about multiple metric thresholds.

In summary, as our survey target audience was not limited to source code analysis experts, we were able to identify some challenges different from those related work identified. These additional challenges give new insights for studies conducted in this doctoral thesis.

3.5 SUMMARY

We conducted a web-based survey with 350 Brazilian practitioners engaged in the software industry whose code analysis practices are not so well established. This chapter discussed the survey featured questions about (i) the practices used to code analysis, (ii) the importance given to such practices, (iii) the issues about the automation of these practices (iv) practitioners' perception about multiples metric thresholds. We summarize the results and their implications for research and practice as follows.

Code analysis practices are known but applied irregularly. Our results showed that Brazilian practitioners know code analysis practices (RQ1) and recognize their importance (RQ2). However, development teams do not apply these practices regularly (RQ1). We use these results as motivation to conduct studies to clarify the impact and benefits of applying code analysis practices.

Practitioners are unaware or have many issues with using automated static analysis tools. Our results showed that 54.85% of respondents use at least one automated code analysis practice (RQ1). However, many respondents (42.85%) stated unaware of these tools (RQ3). Additionally, practitioners reported many issues and challenges to adopt these tools regularly (RQ3). We use these results to propose solutions for integrating such tools in software development processes more efficiently.

Practitioners' perception is that multiple metric thresholds could help in source code quality analysis. Current static analysis tools use a single metric threshold value for each metric. State-of-art tools use detection strategies to identify code anomalies based on metric thresholds. However, our results showed that only 9.14% of the respondents agreed with a single metric threshold to evaluate all system classes (RQ4). On the other hand, 70.28% of them believe that contextual factors, such as architectural layers or business entities, influence metric values so that multiple thresholds for each metric could help improve accuracy (RQ4). We use this result to propose techniques for defining multiple design-sensitive metric thresholds and investigate whether they improve static analysis tool accuracy.

Finally, as lessons learned, our study illustrates that surveys with a well-defined focus, a language close to the respondents, and questions designed to be answered quickly have a high response rate. Also, using examples to clarify terms and concepts that are not always clear to the respondents proved to be an essential tool for the developers' comprehension of survey questions.

Chapter 4

HEURISTIC FOR IDENTIFYING DESIGN ROLES

This chapter describes the design role concept and the proposed heuristic to identify and assign design roles to classes. We also present the DesignRoleMiner tool that implements the heuristic we propose to identify each class's design role automatically in a system. We discuss two empirical studies conducted to evaluate the proposed heuristic. We also propose an application of the proposed heuristic to find out systems implemented with similar design roles.

4.1 DESIGN ROLE CONCEPT

Many different design role concepts have been proposed and discussed from different viewpoints (RIEHLE; GROSS, 1998; EADDY; AHO; MURPHY, 2007; ZHU; ZHOU, 2008; WANG et al., 2011). We consider the concept Wirfs-Brock *et al.* defined for software component design (WIRFS-BROCK; MCKEAN, 2003). According to them, a design role is a set of related responsibilities assumed by an object to fit into a community, such as a framework or an enterprise architecture. We decided to use this concept because many modern object-oriented systems are developed based on reference architectures (BASS; CLEMENTS; KAZMAN, 2012). Design roles are assigned to one or more classes through inheritance, interface implementation, or class annotations in such systems.

In this context, we defined a heuristic to automatically identify the main design role played by each system class. The proposed heuristic uses a customizable token-based method that considers the syntactic structures of the class. Some classes may accommodate more than one design role, but the proposed heuristic assigns only the most prominent design role. Although this overlapping of responsibilities is not considered adequate in object-oriented systems (BOOCH, 1986), future studies could assess the impact of having classes with multiple design roles. The automatic identification of the design role a class plays is not a trivial task for the following reasons:

a) the same design role, even in the same system, can be assigned to a class by different mechanisms. For instance, in Spring MVC-based systems, a class playing the

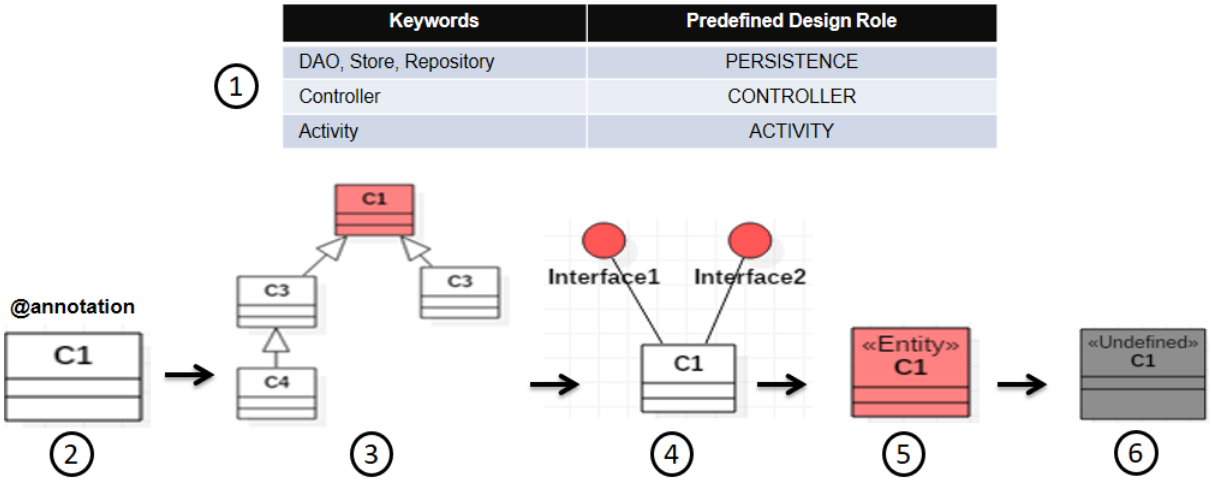

Figure 4.1: Design Role Heuristic

Controller design role can get the @Controller annotation or extend the *Abstract-Controller* class.

b) the same design role can be assigned to a class using different levels of inheritance in the same hierarchy of classes. For instance, in Android applications, a class that extends the Activity superclass plays the *Activity* design role. Consider that the name of this class is *PreferenceActivity*. In addition, classes that extend the class *PreferenceActivity* also play the *Activity* design role.

c) the same design role may follow different naming patterns, usually related to used design patterns or frameworks. For instance, Repository and DAO (Data Access Object) are two common design patterns used to implement the *Persistence* design role. Superclasses or interfaces defining this design role usually have the keywords "DAO" or "Repository" in their names.

4.2 PROPOSED HEURISTIC

Based on previous assumptions, we propose a keyword-based heuristic to assign design roles to classes. Figure 4.1 illustrates a high-level overview of the proposed heuristic six steps.

Step 1: Preparing the table of keywords and corresponding predefined design roles. The heuristic receives as input a table that associates keywords with corresponding design roles. We call the design roles in this table as *predefined design roles*. Predefined design roles are design roles that, based on our previous knowledge of the domain, we know are present in a system, and also we know keywords related to them. Step 1 of Figure 4.1 shows examples of keywords and their corresponding design roles. For instance, classes implementing interfaces that contain keywords such as "Repository", "DAO", or "Storage" in their names usually play the *Persistence* design role. Based on the analysis of the three studied domains and our previous experience as developers of

systems based on them, we built Table 4.1 that shows all keywords and predefined design roles. This table can be reused as it is or refined before starting the next steps, which are, in fact, the automatic ones. It is essential to note that the predefined design roles are not the only ones assigned to classes in the following steps. The heuristic discovers other non-predefined design roles during the process.

Step 2: Assigning design roles by means of annotations. Some architectures use class annotations to define the design role a class plays. For example, in MVC-based systems, developers use the *@Service* annotation to define that a class implements the *Service* design role. However, class annotations can be placed with other goals. For instance, the *@deprecated* annotation is used to indicate a deprecated class. For this reason, our heuristic considers class annotation to assign a design role to a class only if the annotation is included in the set of keywords defined in Step 1.

Step 3: Assigning design roles by means of inheritance: This step only applies for classes without a design role assigned to them in the previous step. It assigns to a class the design role associated with the superclass's name at the top of the inheritance tree where that class is. This step considers two possibilities. The first possibility holds if the superclass's name contains a keyword that matches with a keyword in the table defined in Step 1. In this case, the heuristic selects the corresponding predefined design role to assign to all subclasses. For instance, suppose a superclass called *AbstractController* with different levels of subclasses. Moreover, suppose that *"Controller"* is a keyword corresponding to the *Controller* predefined design role (Step 1). All direct or indirect subclasses of *AbstractController* would have the *Controller* design role assign to them. When there is no keyword matching with the superclass's name, the second possibility holds: our heuristic creates a new design role (non-predefined) named after the superclass and associates it to its subclasses. According to the example, this step would create an *AbstractController* design role. This step does not consider Java platform classes as superclasses.

Step 4: Assigning design roles by means of implemented interfaces: Again this step applies only for classes without a design role assigned to them in the previous steps. It considers that classes implementing the same set of interfaces should be grouped in the same design role, as each possible set of interfaces can assign different responsibilities to a class. Again, there are two possibilities. First, if at least one of the interfaces contains a keyword that corresponds to a predefined design role (Step 1), then the predefined design role is assigned to all classes implementing the set of interfaces. On the other hand, if there is no keyword matching with any interface's name, this step creates a new design role and names it by using the names of the implemented interfaces separated by a comma and bounded by square brackets. For instance, this step assigns the design role named [IRepository, Comparable] to classes that implement *IRepository* and *Comparable*.

Step 5: Assigning the Entity design role: When the previous steps fail to identify a class' design role, this step applies. It assigns the *Entity* design role to classes with non-static attributes and with at least 90% of its methods starting with "get" or "set". These classes are responsible for encapsulating business models, including rules, data, relationships, and sometimes persistence behavior. We can adjust this percentage

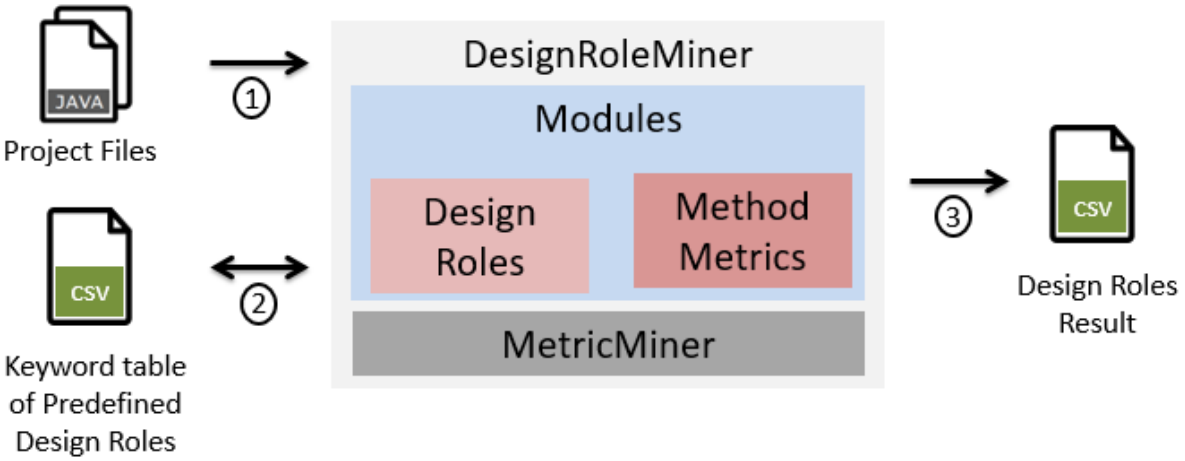

Figure 4.2: High-level Architecture of the Design RoleMiner Tool

according to the evaluated system.

Step 6: Assigning the Undefined design role: When all previous steps fail to define a class' design role, the heuristic assigns to it a general design role called *Undefined*. This step is performed when the class does not contain any structural elements that allow the heuristic to associate other design roles. For instance, the *Undefined* design role is usually assigned to utility classes because they generally do not use structural elements like inheritance or annotations. Classes with the *Undefined* design role are classes our heuristic could not cover. In Section 5.4 we discuss how our heuristic increases the number of covered classes compared to previous works. We also make some suggestions for future works that might reduce the number of *Undefined* classes.

To support the proposed heuristic, we developed a tool called DesignRoleMiner[1]. Figure 4.2 illustrates the high level architecture of the tool that extends the MetricMiner tool (SOKOL et al., 2013). The proposed tool identifies all the design roles played by classes in a software project. In (1), the tool receives the project files of the evaluated software. In (2), the tool uses the keywords table to define predefined design roles. Finally, in (3), the tool generates a file containing the list of classes and the design role assigned by the heuristic to each class. Also, the tool calculates the method-level metrics used in our empirical studies.

4.3 EVALUATION

We try to carry out an initial evaluation of the proposed heuristic with the 15 open-source systems selected for our study on metric distributions discussed in Chapter 5. We sent the heuristic results to four active developers of each system (60 requests). We sent two e-mails for each selected developer containing a short message explaining the purpose of the research, a form listing the design roles assigned to each class and asking them to say

[1]https://github.com/marcosdosea/DesignRoleMiner

it they agreed or not with the design role of each class. However, after 90 days, only two developers replied saying they would answer the survey, but we never did that. For this reason and convenience, we perform evaluations of the proposed heuristic in two software development organizations with easy access to conduct proposed evaluations.

4.3.1 Evaluation with Single Developers

We carried out a first evaluation of the heuristic and tool with single developers. The first evaluation involved five developers and five governmental Web systems of a Secretary of State Treasury. Due to confidentiality reasons, we did not make the data from these systems available. We selected this organization for convenience because the author of this thesis had already worked there.

Study Settings We invited 40 developers to take part in the evaluation. They are the most experienced of their development teams. Five developers of the organization accepted the invitation. Each developer evaluated the results of the heuristic for only one system, the one he led. Each developer received a worksheet with the design role identification results generated by DesignRoleMiner. Each row of the worksheet included: class name, the design role the heuristic assigned to the class, and a field for the developer to answer if he agreed or not with the assignment. Each developer did that for all the system classes, except for classes the heuristic classified as *Undefined*.

We used the default table of keywords available at DesignRoleMiner to determine the predefined design roles. Table 4.1 shows all keywords and predefined design roles. The keywords available in this table are based on the authors' experience and are usually found on systems architecture based on Web, Android, and Eclipse plugins. If necessary, we can configure it to fit the systems' architecture evaluated.

Results and Discussion The five developers of the organization that accepted the invitation have at least 12 years of experience as software developers and ten years working with Java. Besides, each of them is the most experienced and leading developer of one of the five systems. They have been leading maintenance and evolution tasks concerning the systems' design for at least five years. Here we call the systems as S1, S2, S3, S4, and S5. They have 47, 70, 99, 181, and 808 classes, respectively.

Each developer evaluated the design role assignment of one system. They took between 30 to 120 minutes to finish it. The number of classes of each system influenced the time each developer spent conducting the analysis. For instance, the analysis of S5, which is the largest system, required the longest time. During the evaluation, the developers had access to the source code of the system. They checked the source code of some classes, but they did not find it necessary for every class.

The developers agreed with the design role the heuristic assign to 1039 classes, which represents 86.2% of the total number of classes (1205 classes). On the other hand, according to them, the heuristic failed only for 15 classes (1.2%). The other 12.5% of the classes (151 classes) received the *Undefined* design role. Figure 4.3 shows the results per system. All the 15 misclassified classes belong to the system S5. Some misclassification occurred due to programmer mistakes in the use of the enterprise architecture. For instance, some classes were misclassified because they extended a class responsible for defining the ap-

Table 4.1: Keywords and Proposed Predefined Design Roles

Keywords	**Predefined Design Role**
comparable, parcelable, clonable	ENTITY
content, asynctaskloader	PERSISTENCE
controller	CONTROLLER
model	MODEL
service	SERVICE
component	COMPONENT
adapter	ADAPTER
dialogfragment	FRAGMENT
dialog, presentation	DIALOG
activity	ACTIVITY
fragment	FRAGMENT
view, listener, layout, wizard, page	VIEW
widget	WIDGET
notification	NOTIFICATION
action	ACTION
thread, throwable, runnable, asynctask	ASYNCTASK
exception	EXCEPTION
test	TEST

plication constants. Programmers use this approach, which is not recommended, as a shortcut to access constants. Other errors could be avoided with adjustments in the table of keywords.

The design roles assigned to the highest number of classes were: Transaction (67.14%), Persistence (4.23%), Entity (5.98%), BackgroundProcess (3.40%) and Abas (2,07%). Transaction is an implementation of the Command design pattern (GAMMA et al., 1993), in which each action provided by the system is implemented in a class. This explains the high number of *Transaction* classes. Most of the classes with the *Undefined* design role are concerned with the application business logic. In fact, these classes did not use annotations, inheritance or implement interfaces.

4.3.2 Evaluation with Pairs of Developers

We carried out a second evaluation that involved eight developers and four governmental Web systems of the Federal University of Bahia. In addition to the previous evaluation, we aimed to verify whether different system developers would confirm the same design role proposed by the heuristic. For the same reasons of confidentiality, we did not make the data obtained from these systems available. We also selected this organization for convenience because a member of our research group work there.

Study Settings: We invited the two most experienced developers in each system to take part in the study. Each developer evaluated the results of the heuristic for only

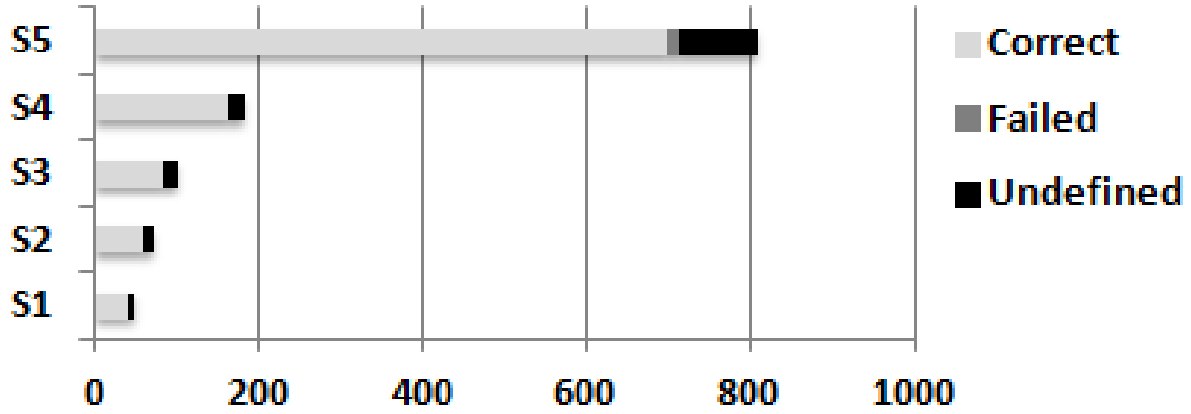

Figure 4.3: Evaluation of the Proposed Design Roles by System

Table 4.2: New Keywords and Proposed Predefined Design Roles

Keywords	**Predefined Design Role**
ManagedBean	MBEAN
ActionForm	FORM
DefaultValidatorTO, DefaultTO	TO

one system, the one he had maintained. Similarly, each developer received a worksheet with the design role identification results generated by DesignRoleMiner. Each row of the worksheet included: class name, the design role the heuristic assigned to the class, and a field for the developer to answer if he agreed or not with the assignment. Each developer did that for all the system classes, except for classes the heuristic classified as *Undefined*.

We conducted a preliminary adjustment of the keywords proposed in Table 4.1 with an experienced developer of the team. This developer did not participate in the evaluation. This adjustment is optional, but we recommend doing it at least once to adapt to the evaluated systems. Table 4.2 shows the keywords added to the predefined design role proposed in Table 4.1.

The keyword ManagedBean aims to identify Java Bean classes from the Java Server Faces framework. These classes usually use the *@ManagedBean* annotation. The keyword ActionForm ais to identify Java Bean classes from the Struts framework. These classes usually extends the class *org.apache.struts.action.ActionForm*. Finally, DefaultValidatorTO, DefaultTO are specific keywords to the enterprise architecture evaluated. The inclusion of these tokens is not mandatory for the heuristic operation. However, its inclusion aims to improve the accuracy to identify the class design role played by system classes.

Results and Discussion Five developers had more than ten years of experience as software developers, and three had at least five years of experience. Regarding the working time in the evaluated project, six developers had more than five years working on the

project, and only two had less than a year. Each developer assessed the results of the heuristic for only one system, the one he worked with. The two most experienced developers evaluated each system. Here we call the systems as S6, S7, S8, and S9. They have 158, 348, 457, and 949 classes, respectively.

Developers took between 30 to 60 minutes to finish it. The number of classes of each system also influenced the time each developer spent conducting the analysis. As classes follow a pattern of names, and developers work with these classes regularly, they practically did not need to look out for the source code to confirm the association proposed by the tool.

The developers agreed with the heuristic design role to 1724 classes, which represents 90.16% of the total number of classes (1912 classes). There was an agreement between pairs of developers on 100% of the design role correctly assigned to classes. On the other hand, according to them, the heuristic failed only for six classes (0.31%). We consider that the proposed heuristic failed whenever at least one developer did not agree with the design role assigned to the class. Interestingly, there was also agreement among developers in four of the six identified failures. In five of the misclassified classes, developers did not agree with the assigned design role's name. For example, the *MatriculaUtil* class was associated with *Observable* design role because it implements an interface with this same name. However, developers disagreed with the name of the design role proposed by the heuristic. They claim that the name of the design role should be *Util*. We could correct these naming errors including a new keyword *Observable* to a predefined design role *Util* they suggest.

One misclassification occurred due to programmer mistakes in the use of the enterprise architecture. The developer make a business class implements an interface, but business classes should not implement any interface of the systems. The proposed heuristic usually assigned the business class to the *Undefined* design role because these classes did not use annotations, inheritance, or implement interfaces. Finally, the other 9.51% of the classes (182 classes) received the *Undefined* design role. Most of the classes assigned to *Undefined* design role are business, utility, or constant classes. Figure 4.4 shows the results per system.

The heuristic identified 11 distinct design roles in S6 and S8 systems, 13 in S9 system, and 20 in S7 system. The design roles assigned to the highest number of classes were: Transfer Object (31.7%), Action (19.1%), Persistence (12.8%), Entity (11.5%), Exception (2.5%) and Test (2,1%). Transfer Object is a typical pattern in web applications, used to pass data with multiple attributes in one shot from client to server. Actions are a core design role in Struts-based Web applications. Each URL mapped a specific action, which provides the processing logic to the requested service. This explains the high number of these design roles in evaluated applications.

Although we need to perform a broader study to generalize these results, we know that systems widely use the structural mechanisms used by the proposed heuristic.

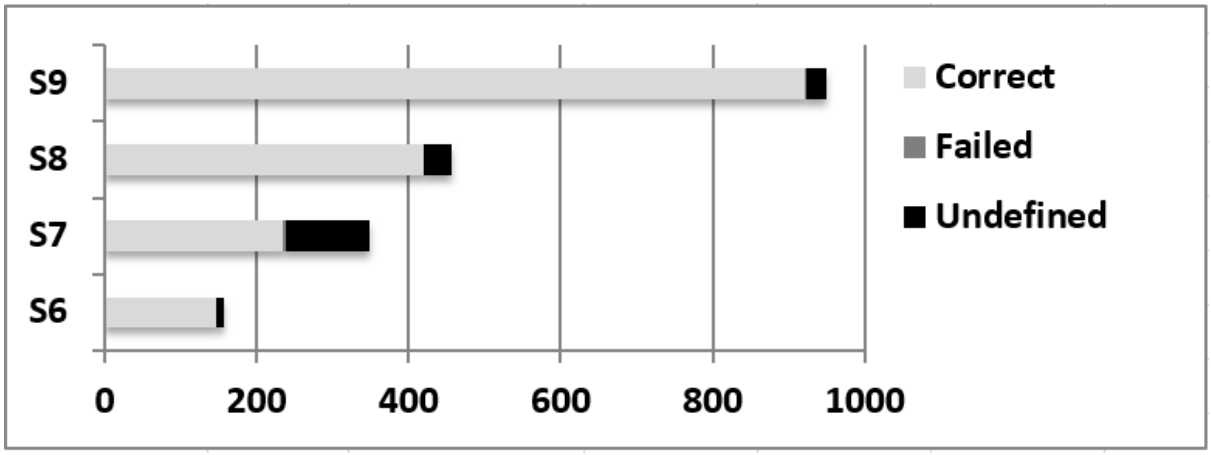

Figure 4.4: Evaluation of the Proposed Design Roles by System

4.4 IDENTIFYING SIMILAR SYSTEMS WITH DESIGN ROLES

This section proposes an approach that uses the heuristic to identify the design role played by each system class, detailed in Section 4.2, to select systems developed with similar design decisions. We hypothesize that deriving metric thresholds from a benchmark of systems developed with similar design decisions can improve metric thresholds accuracy.

Stable systems with recognized quality are used as design models to develop, maintain, and ensure other systems' quality developed with similar design decisions. For example, considering a system developed in a layer architectural style and using the Hibernate framework [2], it is common software developers to look for design solutions in other systems developed with similar design decisions (e.g., architectural style, frameworks, and similar libraries) (SINGER et al., 2010; MARINESCU, 2006). Identifying systems developed with similar design decisions can also help to build benchmarks to derive metric thresholds that consider the design context of classes (DÓSEA; SANT'ANNA; SILVA, 2018).

The similarity between code snippets to search for code examples and clone detection is extensively explored in the literature (HOLMES; MURPHY, 2005; ROY; CORDY; KOSCHKE, 2009; WU; MAR; JIAU, 2010). However, identifying design similarity between complete software systems is a topic still little explored. The search for systems developed with similar design decisions can be even more challenging when we need to perform this searching in large public systems repositories, such as, GitHub[3], or even enterprise repositories with many available systems.

In this context, this section proposes an approach to calculate the level of similarity between systems. The approach uses an abstract representation of each system based on the design roles assigned to its classes by the heuristic discussed in Section 4.2. The proposed approach relies on the hypothesis that systems implemented with the same design decision must have high similarity. To support the proposed heuristic, we developed a

[2]http://hibernate.org/orm/
[3]https://github.com/

tool called SystemSimilarity[4].

Also, we conducted an exploratory study applying the proposed approach to calculate the similarity among 15 systems from three distinct architectural domains. We consider that two systems belong to the same architectural domain when they run on the same platform. The results show that the approach was able to identify the design similarity between most systems of the same architectural domain.

Section 4.4.1 describes the proposed approach for calculating the similarity between systems. Section 4.4.2 details the configuration of the exploratory study carried out to evaluate the proposed approach. Section 4.4.3 presents the results and Section 4.4.4 discusses threats to validity. Section 4.4.5 presents the related works. Finally, Section 4.5 presents conclusions and future works.

4.4.1 Proposed Approach

The primary objective of the proposed approach is to find out systems developed with similar design decisions. The method calculates the similarity between systems using an abstract representation created for each system, based on the identified design roles and the percentage of lines of code (LOC) associated with each design role. We aim to evaluate if the LOC metric is enough to represent other design decisions. We have this hypothesis because design decisions (e.g., architectural style, frameworks, and used libraries) may impact the number of lines of code. The approach has four steps:

Step 1) Identify the design role of each system class: To perform this step we use the DesignRoleMiner (DÓSEA; SANT'ANNA; SILVA, 2018) tool that implements the heuristic to identify the design roles detailed in Section 4.2. The heuristic uses structural information about inheritance, annotations, and implementation of interfaces to associate a design role with each system class.

Step 2) Disregard design roles that do not describe the system design: When the heuristic cannot identify the class design role, it associates a generic design role called *Undefined*. Many systems, regardless of their architectural domain, have classes assigned to the *Undefined* design role. We disregarded this design role because it would artificially increase the similarity value between most systems. We also ignored the design role *Test*, assigned to test classes, because this role does not compose the design of a system.

Step 3) Create the abstract representation of the system: The proposed abstract representation describes each system as a vector that considers (i) the identified design roles in each system and (ii) the percentage of source code associated with each design role. For example, if half of the lines of code of the considered design roles is assigned to *Service* design role, then we consider 50% weight for this design role in the vector. We perform similar calculation for the other design roles. This abstract representation allows comparing systems with different sizes. Additionally, it is intended to evaluate if the percentage of lines of code is enough to represent other design decisions. For example, the use of different libraries and coding style to implement the *Persistence* design role in distinct systems can influence the percentage of code assigned to this role in each system.

[4]https://github.com/marcosdosea/SystemSimilarity

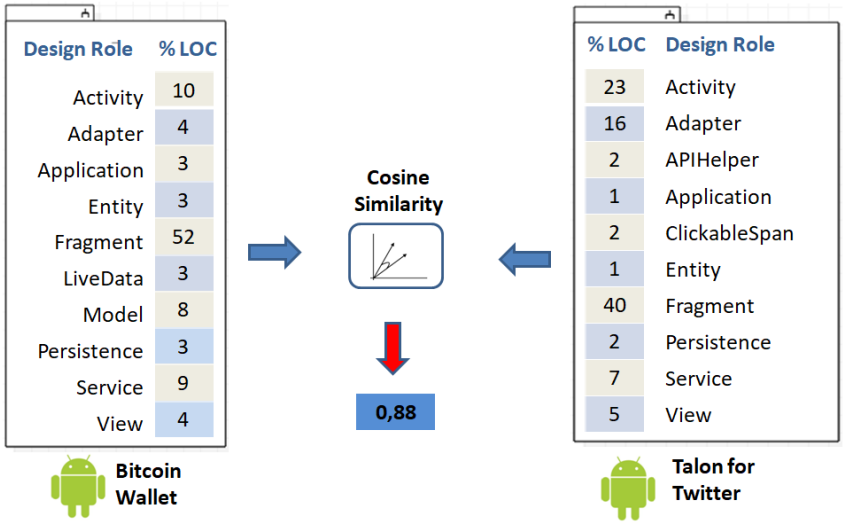

Figure 4.5: Level of Similarity computed by the Proposed Approach between Systems Bitcoin Wallet and Talon for Twitter.

Step 4) Calculate the similarity between systems using their abstract representations: We use the cosine measure to compare the similarity between the two vectors that represent each system. The resulting value ranges from 0 to 1. The closer to 1, the more similar systems are. We use the cosine measure because it is an effective similarity measure used in applications for natural language processing and information retrieval (WILKINSON; HINGSTON, 1991). It calculates the relevance of the tokens by calculating the cosine between two vectors. In the proposed approach, we consider that the tokens are the design roles and the percentage of lines of code associated with each design role is the weight.

Figure 4.5 illustrates the approach using Android Bitcoin Wallet and Talon for Twitter systems, presented in Section 5.1. Each system uses the proposed abstract representation, which uses the identified design roles (step 1) and the percentage of lines of code assigned with each design role (step 3). The representation removes the design roles *Test* and *Undefined* (step 2). Finally, the proposed approach calculates the similarity using the cosine function (step 4). The result (0.88) is very close to the value one, indicating that the evaluated systems have high similarity in design decisions.

4.4.2 Study Settings

We conducted an exploratory study to evaluate whether the proposed approach for calculating the similarity between systems, detailed in Section 4.4.1, is effective to identify systems developed with similar design decisions. We have formulated the following research question:

RQ: *Is the proposed abstract representation able to detect the similarity between systems of the same architectural domain developed with similar design decisions?*

The research question aims to evaluate quantitatively if systems of the same archi-

Table 4.3: Target Systems Summary

		BigBlueButton	OpenMRS	Heritrix	Qalingo	LibrePlan	Bitcoin	K-9 Mail	Exoplayer	SMS Backup	Talon	Activiti	AngularJS	Arduino	DroolsJBPM	Sonarlint
Web Applications	BigBlueButton	1.00	0.17	0.23	0.28	0.19	0.08	0.41	0.33	0.28	0.11	0.31	**0.56**	**0.48**	**0.51**	**0.56**
	OpenMRS	0.17	1.00	0.10	**0.37**	**0.34**	0.10	0.24	**0.29**	**0.31**	0.08	0.06	0.07	0.07	0.13	0.08
	Heritrix	**0.23**	0.10	1.00	**0.48**	**0.28**	0.03	**0.15**	0.10	0.10	0.01	0.06	0.05	0.05	0.07	0.05
	Qalingo	**0.28**	**0.37**	**0.48**	1.00	**0.64**	0.11	0.18	0.15	0.18	0.06	0.14	0.01	0.03	0.08	0.03
	LibrePlan	0.19	**0.34**	**0.28**	**0.64**	1.00	0.11	0.16	0.15	**0.21**	0.06	0.06	0.02	0.03	0.08	0.03
Android Applicartions	Bitcoin	0.08	0.10	0.03	0.11	**0.11**	1.00	**0.41**	0.08	**0.16**	**0.88**	0.04	0.07	0.06	0.08	0.08
	K9 Mail	**0.41**	0.24	0.15	0.18	0.16	**0.41**	1.00	0.04	**0.65**	**0.53**	0.15	0.33	0.25	0.32	0.33
	Exoplayer	**0.33**	0.29	0.1	0.15	0.15	0.08	0.04	1.00	0.28	0.10	0.15	**0.34**	0.24	**0.36**	**0.35**
	SMS Backup	**0.28**	**0.31**	0.10	0.18	0.21	0.16	**0.65**	**0.28**	1.00	0.03	0.03	0.03	0.03	0.09	0.04
	Talon	**0.11**	0.08	0.01	0.06	0.06	**0.88**	**0.53**	0.10	**0.30**	1.00	0.03	0.08	0.07	0.08	0.10
Eclipse Plugins	Acitiviti	0.31	0.06	0.06	0.14	0.06	0.04	0.15	0.15	0.03	0.03	1.00	**0.42**	**0.38**	**0.47**	**0.40**
	AngularJS	**0.56**	0.07	0.05	0.01	0.02	0.07	0.33	0.34	0.03	0.08	0.42	1.00	**0.77**	**0.74**	**0.89**
	Arduino	**0.48**	0.07	0.05	0.03	0.03	0.06	0.25	0.24	0.03	0.07	0.38	**0.77**	1.00	**0.62**	**0.76**
	DroolsJBPM	**0.51**	0.13	0.07	0.07	0.08	0.08	0.32	0.36	0.09	0.08	0.47	**0.74**	**0.62**	1.00	**0.74**
	Sonarlint	**0.56**	0.08	0.05	0.01	0.03	0.08	0.33	0.35	0.04	0.10	0.4	**0.89**	**0.76**	**0.74**	1.00

tectural domain always have high similarity using the proposed approach. We intend to verify if the proposed abstract representation was enough to detect systems from the same architectural domain. We evaluate the impact of class design roles and other design decisions on the proposed similarity value.

Target Systems: Firstly, we searched on GitHub and selected fifteen real-world systems developed in Java from three distinct architectural domains: (i) Web Applications; (ii) Mobile Applications for Android platform; and (iii) Eclipse plugins. We chose three distinct domains implemented in the same language (Java) because application domain and programming language are recognized factors that impacts the distribution of metrics (ZHANG et al., 2013). The author of this thesis experience and knowledge on the selected domains was also a requirement to allow the manual analysis of the code planned for our empirical study. Moreover, the selected domains are popular in the software development industry[5] (mainly the first two) and follow well-defined reference architectures (MEDVIDOVIC; TAYLOR, 2010), which is an essential requirement to apply the heuristic proposed in Section 4.2.

To select the systems from GitHub we used the following search strings: "Eclipse plugin language:java", "android language:java" and "Web language:java". We ordered the resulting lists according to the number of repository forks. The goal was to select systems with as many contributions as possible. Additionally, we excluded frameworks, libraries, and systems not updated since January 2016. Frameworks and libraries are out of scope of our study because they usually follow very particular design decisions seldom shared with other systems. We also excluded systems with no release available because we consider software versioning a premise to develop high-quality software. In order to select widely used Android applications, we also only considered systems with at least 1,000 reviewers and 1,000 downloads. This information is only available for Android applications at Google Play store. Then, we selected the first five systems from the resulting list of each domain. The sample corresponded to approximately 5% of the resulting list of each domain. Finally, the criteria of selection excluded *frameworks* and libraries because they rarely share design decisions with other systems.

Table 4.3 summarizes the main characteristics of the fifteen selected systems. The #classes and #methods columns show the number of classes and methods in each system. The selected systems have between 12 and 282 thousand lines of code (#LOC column)

[5]https://octoverse.github.com/

Table 4.4: Similarity between Systems using the Proposed Approach

	BigBlueButton	OpenMRS	Heritrix	Qalingo	LibrePlan	Bitcoin	K-9 Mail	Exoplayer	SMS Backup	Talon	Activiti	AngularJS	Arduino	DroolsJBPM	Sonarlint
BigBlueButton	1.00	0.17	0.23	0.28	0.19	0.08	0.41	0.33	0.28	0.11	0.31	**0.56**	**0.48**	**0.51**	**0.56**
OpenMRS	0.17	1.00	0.10	**0.37**	**0.34**	0.10	0.24	**0.29**	**0.31**	0.08	0.06	0.07	0.07	0.13	0.08
Heritrix	**0.23**	0.10	1.00	**0.48**	**0.28**	0.03	**0.15**	0.10	0.10	0.01	0.06	0.05	0.05	0.07	0.05
Qalingo	**0.28**	**0.37**	**0.48**	1.00	**0.64**	0.11	0.18	0.15	0.18	0.06	0.14	0.01	0.03	0.08	0.03
LibrePlan	0.19	**0.34**	**0.28**	**0.64**	1.00	0.11	0.16	0.15	**0.21**	0.06	0.06	0.02	0.03	0.08	0.03
Bitcoin	0.08	0.10	0.03	0.11	**0.11**	1.00	**0.41**	0.08	**0.16**	**0.88**	0.04	0.07	0.06	0.08	0.08
K9 Mail	**0.41**	0.24	0.15	0.18	0.16	**0.41**	1.00	0.04	**0.65**	**0.53**	0.15	0.33	0.25	0.32	0.33
Exoplayer	**0.33**	0.29	0.1	0.15	0.15	0.08	0.04	1.00	0.28	0.10	0.15	**0.34**	0.24	**0.36**	**0.35**
SMS Backup	**0.28**	**0.31**	0.10	0.18	0.21	0.16	**0.65**	**0.28**	1.00	0.03	0.03	0.03	0.03	0.09	0.04
Talon	**0.11**	0.08	0.01	0.06	0.06	**0.88**	**0.53**	0.10	**0.30**	1.00	0.03	0.08	0.07	0.08	0.10
Acitiviti	0.31	0.06	0.06	0.14	0.06	0.04	0.15	0.15	0.03	0.03	1.00	**0.42**	**0.38**	**0.47**	**0.40**
AngularJS	**0.56**	0.07	0.05	0.01	0.02	0.07	0.33	0.34	0.03	0.08	0.42	1.00	**0.77**	**0.74**	**0.89**
Arduino	**0.48**	0.07	0.05	0.03	0.03	0.06	0.25	0.24	0.03	0.07	0.38	**0.77**	1.00	**0.62**	**0.76**
DroolsJBPM	**0.51**	0.13	0.07	0.07	0.08	0.08	0.32	0.36	0.09	0.08	0.47	**0.74**	**0.62**	1.00	**0.74**
Sonarlint	**0.56**	0.08	0.05	0.01	0.03	0.08	0.33	0.35	0.04	0.10	0.4	**0.89**	**0.76**	**0.74**	1.00

and 08 and 224 contributors contributors (#contributors column). The #releases column shows the number of stable releases available in each system. The last column shows the commit date corresponding to the source code we used in our study. Finally, the #design roles column shows the number of design roles identified by our heuristic (Section 4.2). For instance, SMS Backup+ system has six predefined design roles (eg. *Activity* and *Persistence*), three non-predefined design roles (eg. *BroadCastReceiver*) and the *Undefined* design role, totaling 10 design roles.

Study Procedures: To answer the research question (RQ), we calculate the similarity between the 15 considered systems, totaling 105 similarity values. Then, we identified the four most similar systems for each system, that is, with the highest values in the proposed measure of similarity. Our goal was that the approach was able to indicate only the four previously selected target systems that belong to the same architectural domain. Also, the source code of the four most similar systems was manually analyzed to justify the similarity value found. We aim to identify if the proposed abstract representation was enough to represent other design decisions not considered by it, for example, used libraries and coding style.

4.4.3 Results and Discussion

Table 4.4 shows the similarity values obtained using our proposed approach detailed in Section 4.4.1. Each value in Table 4.4 is the resulted of the similarity calculation between the systems that are in the row and the column, respectively. For example, the calculated value of similarity between the K-9 Mail and Bitcoin Wallet systems is equal to 0.41. The bold values highlight the four systems most similar to the system on the line. For example, the four most similar systems to the LibrePlan system are OpenMRS, Heritrix, Qalingo, and SMS Backup.

Our initial hypothesis was that systems associated with the same domain would have high similarity value to each other. This similarity would occur because of the high probability of systems assigned to the same domain use similar design decisions (e.g., design roles and libraries). The results obtained with the calculation of similarity and manual evaluation of the source code showed that this does not always occur. Some systems had low similarity values due to the use of design roles very different from those generally used by systems of the same domain. Also, systems that do not associate class

design roles through inheritance, interface, or annotations mechanisms can also generate an unrepresentative abstract representation for the similarity calculation.

In the Web systems architectural domain, the Qalingo system obtained the highest similarity values with the other four systems of the same domain. The Libreplan and Heritrix systems obtained the highest similarity values with three systems of the domain. The OpenMRS system with only two systems of the Web architectural domain. *Service*, *Entity*, *Component* and *Controller* are design roles usually found and representative in the Web domain. However, some systems obtained some of the highest similarity values with systems of the Android domain due to some common design roles into both domains. For example, the Android SMS Backup system was the fourth most similar system to the LibrePlan Web system. *Persistence* and *Service* are design roles common to both systems and that impacted the similarity calculation. However, in the LibrePlan system, *Persistence* is implemented using the Hibernate framework, whereas, the Android SMS Backup system uses Android persistence libraries. That is, we found the same design role (*Persistence*) implemented with distinct design decisions on both systems. Finally, the BigBlueButton system ended up not having high similarity neither with the systems of the Web domain nor with systems of the Android domain. We investigate the source code of this system and we noted that 45% of its classes did not have an associated design role, creating an unrepresentative abstract representation of this system within the Web domain.

In the Android domain, the systems Bitcoin, K-9 Mail, Talon for Twitter have three systems of the same domain with the highest similarity value. The SMS Backup system obtained high similarity with two systems of the same domain. Only the Exoplayer system did not achieve high similarity with any Android system. The Exoplayer system also had 56% of the classes assigned to *Undefined* and *Test* design roles. These design roles are not considered to build the proposed abstract representation used in the calculation of similarity. In addition, *Activity*, *Fragment*, and *Service* design roles, which are very common in Android systems, are unrepresentative in the Exoplayer system. Thus, the similarity calculation was able to reflect the distinct design decisions of Exoplayer compared to other evaluated Android systems.

Finally, in the plug-ins for Eclipse domain, we obtained the highest values of similarity. We explain these values by the little variation of the design roles that can be used to implement systems of this architetural domain. Design roles such as *Action*, *Dialog*, *Plugin* and *View* are well representative in this domain. However, BigblueButton Web system ended up having the highest similarity values with some Eclipse plug-ins due to some design roles which are more common in Eclipse plug-ins (e.g., *View*, *Action* and *Dialog*). These design roles are implemented using distinct design decisions (e.g., libraries). However, since the abstract representation does not consider this design decision, the proposed approach found high similarity between these systems.

The proposed abstract representation identified the similarity of design decisions among most systems in the same architectural domain. However, in some cases, the approach calculated a high similarity between systems assigned to distinct architectural domains. For example, the OpenMRS Web system achieved high similarity with two systems belonging to the Android domain (Exoplayer and SMS Backup) due to the common

use of some design roles. For example, *Persistence*, *Service*, *Entity* and *View* design roles are common to both domains, but are usually implemented with distinct design decisions in each domain. That is, in these cases, the approach found high similarity between systems that are implemented with distinct design decisions, although they have a similar set of design roles.

In summary, the proposed approach to compute the similarity between systems reflected the design similarity usually found in systems of the same architectural domain. It was also able to identify when the design decisions of a system were quite different from other systems of the same architectural domain, for example, because it used different design roles. Future work can evaluate whether considering other design decisions, for example, the libraries used by each design role, would improve the representativeness of proposed similarity values.

4.4.4 Threats to Validity

In this section we discuss the threats to the validity of the study and the actions that we take to minimize them.

Construct validity: There is a possible threat related to the systems used to conduct the study. The systems can be selected prioritizing the propose approach. To narrow this bias, we applied well defined criteria to select systems from the GitHub repository. Another threat was the choice of the measure of cosine similarity. Despite being one of the most used measures of similarity, this study was exploratory and future works can evaluate other measures of similarity..

Internal Validity: The main threat is related to the heuristic to identify the design roles used to create the abstract representation of the system. However, the heuristic has been used in other studies to identify design role with good precision (DÓSEA; SANT'ANNA; SILVA, 2018). Despite the *Undefined* design role is not considered by the proposed abstract representation, future improvements in the heuristic can also improve the similarity calculation results.

External Validity: The results obtained are valid for the fifteen evaluated systems and three domains. It is not suggested to generalize these results to other systems or domains.

4.4.5 Related Works

Few studies consider the design similarity between systems. Tibermacine et al. (TIBERMACINE; TIBERMACINE; CHERIF, 2014) propose an approach to measure the similarity of web services through their WSDL interfaces. The goal is to find the best replacement for a Web Service when it fails. Al-msie'deen et al. (R.AL-MSIE'DEEN et al., 2013) propose an approach to mining features by calculating lexical and structural similarity. Our proposed approach is also based on the structural similarity of the identified design roles. However, our approach uses a higher level of abstraction (design roles) that allows comparing the similarity between complete systems.

Nagappan et al. (NAGAPPAN; ZIMMERMANN; BIRD, 2013) propose an approach to evaluate if the coverage a sample of systems is representative to conduct an experi-

ment. They suggest a systems similarity function based on numerical dimensions (e.g., the number of developers and lines of code) and categorical dimensions (e.g., main programming language and domain). Our approach proposes a new numerical dimension of similarity based on the identified design roles, not limiting to generic information about the system.

4.5 SUMMARY

This chapter proposed a heuristic to assigned the design role played by each system class. We also reported two evaluations of the proposed heuristic that we carried out in two real-world software development environments. The heuristic assigned the design role correctly to 86,2% of the classes in the first case study and 90.16% of the classes in the second case study. Developers disagree with the design role assigned to 1.2% and 0.31%, respectively, of assigned design roles. Usually, the disagreements were problems in the software architecture implementation or divergence in the assigned design role nomenclature. The heuristic assigned *Undefined* design role, respectively, to 12.5% and 9.51% of the evaluated classes. In Chapter 5, we use these findings to investigate the impact of design role on distribution or metric values. Our central hypothesis is that the class design role is an important design decision to take into account to derive metric thresholds.

We also propose an approach to calculate the similarity between systems using the proposed heuristic to assigned design roles to system classes. Similar systems can be used as a design model to develop and maintain other systems. We also can use them to compose benchmarks used to extract metric thresholds to evaluate the software quality. We perform an evaluation using 15 systems from three different domains, and the results showed that the proposed approach was able to point out most systems developed with similar design decisions. Although the approach identified some level of similarity between systems from distinct architectural domains, this would not be its primary use since it is usually not difficult to separate systems into a repository (e.g., GitHub) from distinct architectural domains, for example, Android systems and Web systems. We aim to use the proposed approach to point out systems developed with similar design decisions assigned to the same architectural domain. We use these results to propose techniques, in Chapter 6, that help to select systems developed with similar design decisions to composing benchmarks to derive metric thresholds.

As future work, we intend to evaluate whether the design role assigned by the proposed heuristic improves the comprehension of software design by software developers. We also intend to conduct studies to assess whether the proposed similarity measure could detect the distancing of the planned design during the software development process. Finally, we intend to carry out studies using the design role assigned to classes to detect violations of planned design or planned architecture.

Chapter 5

HOW DO DESIGN DECISIONS AFFECT THE DISTRIBUTION OF SOFTWARE METRICS?

In this chapter, we carried out a study to investigate whether class design roles and other fine-grained design decisions affect metrics distributions and, therefore, should be taken into account when building benchmarks for metric-based analysis of source code. This study aims to answer our second general research question:

RQ2: Are there statistical significant differences between measures obtained from classes developed with different design decisions?

To answer this research question, we conducted an empirical study analyzing the source code of fifteen real-world open-source systems from three distinct architectural domains (Eclipse plugins, Android Applications and Web-based Systems). Over the selected systems, we compute four metrics commonly used to assess method maintainability and then we evaluate the effect of design decisions on their distributions. We analyze the distributions over class methods grouped according to what we call as *design roles*, discussed in Chapter 4. Design roles include architectural roles, but also include classes whose responsibility is application-specific and not bound to any particular reference architecture.

Our investigation has three main perspectives. First, we investigate whether metrics distributions vary between different design roles of the same system. Second, we compare the metrics distributions of classes from different systems but having the same design role. Our goal here is to verify whether other design decisions, besides the design role, also affect metrics distributions. Finally, we compare metrics distributions of classes from the same design role but over different stable releases of the same system. Since releases of the same system tend to comprise the same design decisions, our hypothesis here is that the distributions would not vary significantly. We summarize our findings as follows:

- We found that different design roles from the same system drive different metrics distributions. These findings extend the results obtained by Aniche *et al.*, which

considers only architectural roles. Using the design role concept (our approach) increases the number of classes that could be covered and assessed, for instance, by different thresholds.

- We then investigated whether the distribution of metric values of the same design role were similar across different systems of the same domain. If this occurs, we should consider building benchmarks with different systems that have similar sets of design roles. However, we found that the same design role (e.g. Persistence) can drive different distributions of software measures in different systems. Then, we conducted a manual and deep source code analysis to identify what design decisions made such distributions different.

- Finally, we investigated if the distribution of metric values of the same design role were similar across different releases of a system. If this occurs, we should consider building benchmarks with previous releases that underwent quality review. The results we obtained show that in most of the cases the same design role in a system did not vary significantly throughout different releases.

The remainder of this chapter is organized as follows. Section 5.1 describes the settings of our empirical study to identify the impact of design decisions on the distribution of metrics. Section 5.2 presents the results of the study. Section 5.3 discusses threats to validity and Section 5.4 discusses related work. Finally, Section 5.5 summarizes the results and discusses implications of our research.

5.1 STUDY SETTINGS

Our main hypothesis is that design role, proposed in Chapter 4, is an important design decision that may impact on the distribution of metric values. Therefore, this study takes into account the design role of every class of the analyzed systems. The main *goal* of this study is to evaluate the impact of fine-grained design decisions over distribution of metric values in systems of the following domains: Web, Android and Eclipse Plugins. A high impact may suggest that we should not overlook fine-grained design decisions when building metric-based benchmarks for assessing source code quality. For this purpose, we conceived the following research questions (RQs) to guide our study.

RQ1 *Is there a significant difference in the distribution of metric values of different design roles in the same system*

RQ2 *Is there a significant difference in the distribution of metric values of the same design role across different systems of the same domain?*

RQ3 *Is there a significant difference among metric distributions of the same design role across different releases of the same system?*

Through RQ1, we aim to investigate whether design role is a design decision that affects metric distributions. With RQ2, our goal is to evaluate whether other design decisions, besides design roles, also influence the distribution of metrics values. Finally,

through RQ3, we aimed to verify whether there are decisions along the releases of each system that change the design in a way that significantly impact metrics values. The goal with RQ3 is also verifying whether previous stable releases of the same system would be good candidates to compose benchmarks. In some cases, modules of a previous version, implemented or reviewed by a senior and experienced developer, for instance, may be the only source of design decisions developers consider that fit to their context. To answer these research questions we designed a study composed of three major steps described in the following subsections.

5.1.1 Selecting Target Systems

We used the same criteria and systems discussed in Section 4.4. The main characteristics of the fifteen selected systems are detailed in Table 4.3.

Design roles are usually domain-specific. For instance, *Activity* is a design role typically found only in Android applications. Some design roles are typically found in all systems of a domain. For instance, *Fragment* and *Service* are common in Android applications. However, each system is a unique design solution and normally has some particular design roles. For instance, Exoplayer has the *Buffer* and *SimpleDecoder* design roles for manipulation of audio files. These design roles are hardly found in other Android systems. This situation illustrates why the number of design roles are distinct even among applications of the same domain.

Compared to previous works (see Section 5.4), our heuristic improved the number of covered classes, i.e. classes the heuristic was able to assign a design role different of the *Undefined* one. In fact, our heuristic works better for systems whose classes are structurally bound to a reference architecture. The higher is the number of classes structurally bound to the reference architecture, the smaller is the number of classes associated to the *Undefined* design role. This is the reason for the differences on the number of *Undefined* classes among the systems. For example, the Qalingo system has only 5.6% of its classes associated with the *Undefined* design role. However, the SMSBackup system has 40.5% of its classes as *Undefined*. In Section 5.4, we detailed our plan to conduct future studies to further reduce the number of classes assigned to the *Undefined* design role.

5.1.2 Design Role Identification and Metric Computation

In this step, we used our heuristic (Section 4.2) for identifying and automatically assigning a design role to each class of the fifteen systems. We used the DesignRoleMiner to do that as well as to compute method-level metrics. We use Table 4.1 of keywords to assigned predefined design role. It is important to highlight that classes are grouped by design roles. As a consequence, methods are grouped by their classes' design roles. Therefore, each design role constitutes a sample of method-level metric values. Our study considered the following four metrics:

- **McCabe's Cyclomatic Complexity (CC) (MCCABE, 1976)**: It counts number of branching points of each method.

- **Number of Method Parameters (NMP) (FOWLER; BECK, 1999)**: It counts the number of parameters of each method.
- **Lines of Code (LOC) (LANZA; MARINESCU, 2006)**: It counts the number of executable statements of each method, excluding comments and blank lines.
- **Efferent Coupling (EC) (MARTIN, 1995)**: It counts the number of classes from which each method calls methods or accesses attributes.

We selected these method-level metrics because we can manually compute them without tool support. This criterion is essential for conducting the manual analysis planned for our study and identifying the factors that impact on the distribution of metrics. Also, these metrics are available in many tools (PAIVA et al., 2017) and have been successfully used for fault-proneness prediction(FONTANA et al., 2013; GIL; LALOUCHE, 2017; BOUCHER; BADRI, 2018), for instance.

5.1.3 Comparing Distributions of Metric Values

In this step, we compare the distribution of values of each metric according to the following configuration: (i) to answer RQ1, we compare different design roles within each system, (ii) to answer RQ2, we compare the same design role across different systems, and (iii) to answer RQ3, we compare the same design role across releases of each system.

To do that, we initially apply the Kruskal-Wallis test (SHESKIN, 2007) using the 5% significance level (i.e. p-value < 0.05). Kruskal-Wallis is a non-parametric statistical test used to evaluate whether three or more samples have similar distribution of values. When the null hypothesis is rejected, the test indicates that at least one of the samples has distribution of values different to the others. However, it does not indicate what is that sample.

Therefore, if Kruskal-Wallis test rejects the null hypothesis, we additionally apply a multiple comparison procedure to identify pairs of samples with significant differences using the Mann-Whitney U test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947). The result of this procedure is a table ordered according to the distance between samples. In this way, the first and the last rows of the table contain the most distant samples.

Finally we apply Cliff's δ (CLIFF, 1993) to quantify the importance of the difference between distribution values of pairs of samples. To avoid excessive comparisons, we only compare the most distant samples. This is enough to verify whether there are at least two groups of methods that has distinct distribution of metric values. We use Romano *et al.* (ROMANO et al., 2006) approach to interpret the effect size based on Cliff's δ. Supposing δ as effect size, ranging from -1 to 1, $|\ \delta\ | < 0.147$ means negligible effect, $|\ \delta\ | < 0.33$ means small effect, $|\ \delta\ | < 0.474$ means medium effect, and $|\ \delta\ | >= 0.474$ means large effect. Cohen (COHEN, 1992) states that a small effect size is noticeably smaller than medium but not so small as to be trivial, a medium effect size represents an effect likely to be visible to the naked eye of a careful observer, while large effect is noticeably larger than medium.

Therefore, we decided to use small, medium and large effect sizes to consider that two samples of methods should be manually analyzed. Effect sizes must be judged according to the context and even small effects might be of practical importance (KAMPENES et al., 2007).

5.2 RESULTS AND DISCUSSION

In this section, we report and discuss the main findings of our study guided by each research question.

RQ1*: Is there a significant difference in the distribution of metric values of different design roles in the same system?*

Motivation: If design roles affect the distribution of metric values, we should consider taking them into account when building metric-based benchmarks.

Method: To examine the overall impact of design roles on each metric and system, we test the following null hypothesis.

$H0_1$*: there is no difference in the distributions of metric values among all design roles in the same system.*

For each system, we execute the steps described in Section 5.1.3 four times, one for each metric. If, using Cliff's δ, we find a large, medium or small difference between the most distant design roles, this means that, for the analyzed system and metric, there are at least two design roles with significantly different metric distributions. So, we can answer "yes" to RQ1. Our website provides R scripts for replication purposes.

Findings: All 60 executions of the Kruskal-Wallis test (four metrics times fifteen systems) reject the null hypothesis ($H0_1$). This means that at least one design role has distribution of values distinct from the others. In addition, when comparing the two most distant pairs of design roles, we obtained Cliff's δ corresponding to large effect size for 57 of the 60 combinations of metrics and systems (95%). For the other 3 combinations (5%), we obtained Cliff's δ corresponding to medium effect size. These results mean that for all systems and metrics there are significant differences between the distributions of metric values of at least two design roles.

Most of the design roles compared with Cliff's δ comprise more than 15 methods. Therefore, we consider them as representative. For the Android application domain, 85% of the design roles taken into account have from 15 to 958 methods. For the Web application domain, 84.6% have between 25 to 4088 methods. Finally, for the Eclipse Plugin domain, 79.1% have from 15 and 223 methods.

Figure 5.1 illustrates some differences between the distributions of values of design roles from the K-9 Mail Android application. It shows four graphs with three box plots each. Each graph is about one of the four metrics. The box plots on the ends of each graph correspond to design roles with distribution largely different from each other (large effect size). The box plot on the middle shows the distribution of values of a design role with medium or small difference to the other two (medium or small effect size.)

Regarding the LOC metric, Figure 5.1 shows *BodyPart*, *Persistence* and *AsyncTask* design roles. *BodyPart* involves methods ranging from 3 to 6 lines of code. *Entity* and *Exception* are other design roles (not shown in Figure 2) with similar distributions. In

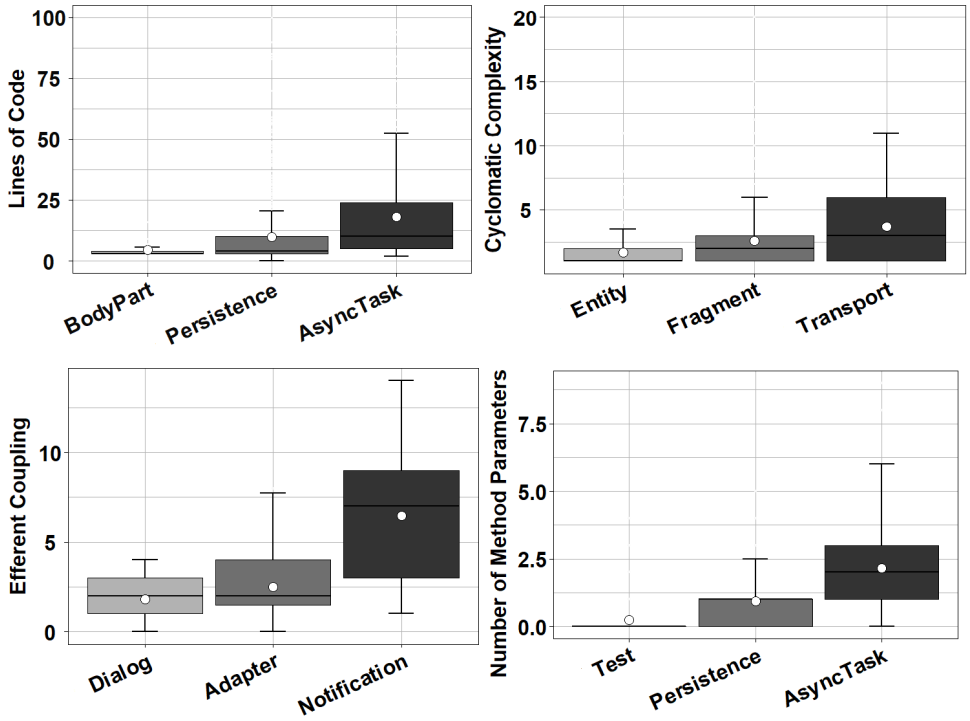

Figure 5.1: Distributions of Metrics of K-9 Mail System

fact, methods from these design roles have a low number of lines of code because they are usually only responsible for encapsulating data. In the *Persistence* and *AsynkTask* design roles, methods have maximum value of 21 and 53 lines of code, respectively. In fact, classes assigned to *AsynkTask* are responsible for more complex tasks, such as automatically updating mail folders.

Figure 5.1 shows the *Entity*, *Fragment* and *Transport* design roles to illustrate differences regarding the CC metric. These three design roles present, respectively, 3, 6 and 11 as CC maximum value. In fact, implementing domain entities (*Entity* design role) is quite simpler than implementing message transport following security protocols (*Transport* design role). Using the same threshold for assessing methods of both design roles might lead to false negatives or false positives.

Regarding the efferent coupling metric, the maximum values obtained for the three design roles shown in Figure 5.1 are 4, 7, 14, respectively. Methods associated to the *Notification* design role have higher efferent coupling because they call other parts of the mobile device to notify changes in the email box state. Finally, regarding number of parameters, Figure 5.1 shows design roles with 0, 2 and 6 as maximum values. The methods associated to the *Test* design role, for instance, have a very low number of parameters because each method usually is responsible to test only one method. Methods associated with *AsyncTask* design role have more parameters because they implement application business logic, which requires more information as input parameters.

Another interesting point, is that we noticed a large number of outliers in the *Undefined* design role. An outlier is an observation that appears to deviate from other

observations in a sample. This finding is expected because, in fact, the *Undefined* design role ends up accommodating classes for which our heuristic was not able to identify a design role using syntactic structures. Classes associated with *Undefined* design role are more likely to be developed following distinct design decisions.

In summary, the significant differences we observed on metric distributions of distinct design roles, allow us to answer RQ1 as follows:

> *Design roles affect the distribution of metric values. Therefore, we should consider taking design roles into account when using benchmarks for metric-based source code analysis.*

***RQ2**: Is there a significant difference in the distribution of metric values of the same design role across different systems of the same domain?*

Motivation: In RQ1, we found that design roles affect metric distributions. However, can we group together classes of the same design role but from different systems when building benchmarks? Or do different systems have other design decisions for the same design role that make metric distributions distinctive for different systems?

Method: To address RQ2, we test the following null hypothesis.

$H0_2$: there is no difference among distributions of metric values of the same design role across systems of the same domain.

For that, we only took into account design roles present in at least two systems of the same domain. We restrict the comparison among systems of the same domain, because systems of different domains barely have design roles in common developed with similar design decisions. We compared the distributions of the same design role in different systems. For each design role, we execute the steps described in Section 5.1.3 four times, one for each metric. If, using Cliff's δ, we find a large, medium or small difference between the most distant systems, this means that, for that design role and metric, there are at least two systems with significantly different metric distributions. When this occurs, we manually investigate the source code, identifying if any design decision, in particular, is responsible for that difference. Initially, we create two sets of classes associated with the same design role, one for each distinct system evaluated. We then compare the methods of each set with similar goals. For example, we observe methods that aim to insert data in the database, assigned to the Service design role in both systems. However, the exception handle mechanisms were used in only one of the evaluated sets of methods. As exception handling is a design decision that impacts evaluated metrics, we consider it on our evaluation.

Findings: For Android applications, we evaluated 10 predefined design roles (eg. *Activity* and *Service*) and one non-predefined (*BroadcastReceiver*). For the Eclipse plugins domain, we considered nine predefined design roles (eg. *Dialog* and *View*) and nine non-predefined (eg. *Plugin* and *AbstractHandler*). Finally, for the Web application domain, we analyzed 11 predefined design roles (eg. *View* and *Persistence*) and five non-predefined (eg. *Validator* and *DispatcherServlet*). Also, we examined the *Undefined* design roles for the three domains. In total, we analyzed 46 design roles (the *Undefined* design role counts three times, one for each domain). Some design roles are present on all five systems of the domain. For instance, this is the case of the *Activity* and *Persistence*

Table 5.1: Cliff's δ Interpretation

Effect	LOC	CC	EC	NMP
Large	**02**	**01**	**09**	**09**
Medium	**12**	**05**	**11**	**05**
Small	**16**	**14**	**09**	**10**
Negligible	**02**	**04**	**03**	**03**

design roles in the Android domain. Other design roles are present in some of the systems. For example, *BroadCastReceiver* is present in only three systems of the Android domain.

For each design role, we executed the statistical tests four times, one for each metric, totaling 192 tests (48 times 4). The null hypothesis ($H0_2$) was rejected in 115 of the 192 tests. For these cases, we applied the multiple comparison procedure and the Cliff's δ to quantify the size of the difference between the two systems with highest difference among the samples. Table 5.1 summarizes the Cliff's δ results. It shows, for each metric, the number of design roles per effect size found. For instance, for the EC metric, we found large effect size for nine design roles and medium effect size for eleven design roles. The full table with individual results for each design role and metric is available on our website. Then, we manually analyzed the source code of all design roles for which we found small, medium or large effect sizes. The goal was to find out which design decisions contributed to the difference between the systems. In the following subsection, we discuss the design decisions we identified. In some cases, more than one of them contribute to the difference regarding the same design role.

A. Used Libraries: We found the use of distinct libraries as one design decision that makes metric distributions of the same design role significantly different when comparing different systems. The *Persistence* design role is a clear example of this. We identified the use of distinct persistence mechanisms or libraries across systems of the three domains. This affected the distributions of the metrics in the three domains. In the Android application domain, for instance, we found medium effect size for LOC and EC and large effect size for NMP when comparing the *Persistence* design role of Bitcoin Wallet and SMS Backup+. The Bitcoin Wallet application uses both the *ContentProvider* Android native library and *SQLiteDatabase* to share and persist data, respectively. We observed that 90% of the *Persistence* methods in Bitcoin Wallet range from 3 to 29 lines of code. On the other hand, the SMS Backup+ only uses native Android libraries to open, read, and store SMS and MMS messages. This mechanism is simpler because it does not use database libraries. Then, we observed that 90% of the *Persistence* methods in SMS Backup+ range from 2 to 11 lines of code. Another example of the use of different libraries occurs with the *Test* design role. For instance, we found medium effect size for LOC and EC metrics when comparing the LibrePlan and BigBlueButton systems. The former uses libraries for implementing integration test while the latter uses libraries for unit testing. Implementing unit tests is usually simpler than implementing integration tests.

In Web domain we also calculated MEDIUM effect size to the LOC, EC and NMP

metrics in the Persistence design role between Qalingo e Heritrix applications. Qalingo uses Hibernate libraries to perform persistence and Heritrix uses libraries to perform persistence in Apache Hbase, a distributed database to big data store. In Qalingo application 90% of the methods ranging from 3 to 18 lines of code. In Heritrix we have 90% of the methods ranging from 2 to 12 lines of code. The small difference is explained by the simple way to manipulate big data in Heritrix application. It could be complex in a large Web application, making more evident differences in the distribution of metric values.

B. Coding Style: We also identified coding style as a design decision that affected the distribution of metric values. For instance, we found medium effect size for the EC and LOC metrics when comparing the *Persistence* design role of Qalingo and OpenMRS Web applications. Both systems use the Hibernate framework to implement persistence. However, developers of Qalingo decided to use the *Criteria* mechanism, a type-safe way to express queries in Hibernate. Although some methods in OpenMRS also use the *Criteria* mechanism, most methods use Hibernate Query Language (HQL), a non-type-safe way to express queries. Both mechanisms are common in systems using Hibernate, but source code using HQL usually needs fewer lines of code and uses fewer external classes. These results complement the study of Higo *et al.* (HIGO; KUSUMOTO, 2017) that reports the effect of coding style on the LOC metric.

C. Exception Handling, Logging and Debugging Code: We also observed cases in which decisions related to Exception Handling, Logging or Debugging affected metric distributions. For instance, we found medium effect size for the EC metric when comparing the *Service* design role of Bitcoin Wallet and SMS Backup+ Android applications. Methods in Bitcoin Wallet contain try-catch blocks to handle exceptions, while most of SMS Backup+ methods throw exceptions rather than handle them. The efferent coupling is higher in Bitcoin Wallet methods due to references to other classes within catch blocks.

Regarding Logging and Debugging, we found significant differences of EC and LOC when comparing the *Service* design role of K-9 Mail and SMS Backup+. This occurs due to code snippets used to log actions when the application is executed in debug mode. This is a common mechanism developers use to debug Android applications. However, in K-9 Mail developers placed this logging code in *Service* methods, while SMS Backup+ developers placed it in methods related to the *Activity* design role, causing differences on metric distributions. Listing 5.1 illustrates part of a method from K-9 Mail that logs debugging data.

Listing 5.1: Debug Code in K-9 Mail system.

```
1 if (K9.DEBUG) {
2   Log.i(K9.LOG_TAG, "Notification dismissed");
3 }
```

In summary, the significant differences we observed on metric distributions when comparing design roles common to different systems, allow us to answer RQ2 as follows:

Design decisions related to a design role may make metric distributions for this design role different in distinct systems. We found that used libraries, coding style, exception handling, logging, and debugging code are design decisions that may impact the distribution of method-level metrics. Therefore, we should be aware of these design decisions, not only design roles, when building tools and benchmarks for metric-based source code analysis.

***RQ3**: Is there a significant difference among metric distributions of the same design role across different releases of the same system?*

Motivation: In RQ2, we found that different systems may have different metric distributions for the same design role due to different design decisions. This means that only considering design roles to group classes when building or using benchmarks with different systems may not be enough to have accurate metric-based source code analysis.

A possible alternative is to use previous system releases as a benchmark. Evidently, the idea is to use well designed releases or releases that underwent source code quality review. Following this idea, it is important to investigate if there are design decisions along system releases that change the design in a way that significantly affect metric values.

Method: To address RQ3, we test the following null hypothesis.

$H0_3$*: there is no difference on the distribution of metric values of the same design role across different releases of the same system.*

For that, we decided to compare the release considered in the investigation of RQ1 and RQ2 with the three most recent preceding releases. We did that for each system. We did not make an exhaustive analysis of all releases because very old releases are likely to be very different from recent ones. All releases we used in our study are available on our website. We also decided to only consider design roles with variation on the number of lines of code higher than 1% between at least two of the analyzed releases. Variations smaller than 1% of LOC hardly imply differences on the distribution of metric values. Finally, for each design role, we executed the steps described in Section 5.1.3 four times, one for each metric. If, using Cliff's δ, we find a large, medium or small difference between the most distant releases, this means that, for that design role and metric, there are at least two releases with significantly different metric distributions. In this case, we manually investigate the source code to identify the reasons behind the difference.

Findings: For the Android application domain, we evaluated 36 design roles. Thus, considering the four studied metrics, we executed 144 Kruskal-Wallis tests (36 times 4). The null hypothesis ($H0_3$) was only rejected in 11 tests. For these 11 pairs of design role and metric, we performed the multiple comparison procedure and calculated the effect size between the most distant releases. The effect size was negligible in 10 tests. Only EC metric for the *Persistence* design role presented small effect size. The reason was that the persistence mechanism underwent some refactoring changes along the releases, such as the extract class refactoring.

For the Eclipse Plugins, we evaluated 51 design roles. We performed 204 Kruskal-Wallis tests (51 times 4), one for each metric. The null hypothesis $H0_3$ was only rejected in 7 tests. Then, we obtained negligible effect size for five of these seven pairs of design role

and metric. Only two of them presented small effect size. This occurred for the *Trackable* design role in Sonarlint and the LOC metric due to some simpler classes created from one release to the other within this design role. Similar reason affected the NOP metric for the *Undefined* design role. Some new simpler methods were also created and associated to this design role.

Finally, for Web Applications, we evaluated 59 design roles and executed the Kruskal-Wallis test for each metric, totaling 236 tests (59 times 4). The null hypothesis $H0_3$ was only rejected in 31 tests. We calculated the effect size using Cliff's δ and found six cases with small effect size and four cases with medium effect size. Three cases of small effect size involved the LOC, CC and EC metrics and the *Entity* design role in the OpenMRS system. The differences occurred because the developers decided to exchange the library for generating reports. The other three cases of small effect size involved LOC, CC and EC and the *View* design role in the OpenMRS system. In this case, the developers also decided to use a simpler library for implementing entity searches. Cases of medium effect size also occurred due to changes of libraries. For instance, developers of BigBlueButton modified the *MessageHandler* design role to change the message handling mechanism, which affects the distribution of EC from one release to the other. Also, developers of the Libreplan system decided to use a different library to deal with time, which affects the distribution of NOP of the *TimeTrackerState* design role.

In summary, we only observed very few cases of significant differences on metric distributions when comparing design roles across different releases. This allow us to answer RQ3 as follows:

> *Differences on the distribution of metrics across different stable releases may not be frequent. Therefore, we should consider taking previous releases into account when building benchmarks for metric-based source code analysis.*

5.3 THREATS TO VALIDITY

This section discusses the threats to validity of our study following common guidelines (KITCHENHAM et al., 2006).

Internal validity. There might be a threat associated with the correctness of the tool we used to calculate metrics and identify design roles. DesignRoleMiner extends the MetricMiner tool (SOKOL et al., 2013), which was already used in other studies (ANICHE, 2015; ROZENBERG et al., 2016). Also, we manually checked many metric values and identified design roles. Moreover, we evaluated our tool and heuristic for design role identification by means of a study with developers and Web-based governmental systems, as described in Section 4.2. Other possible threat is that some classes may accommodate more than one design role, but we assign only the most prominent design role according to our proposed heuristic. Although this overlapping of responsibilities is not considered adequate in object-oriented systems (BOOCH, 1986), future studies could assess the impact of having classes with multiple design roles and their effect on metric distribution.

Construction validity. There is a possible threat related to metrics we selected for our

study. The selected method-level metrics cover important aspects of source code quality and are widely used for software fault prediction (CATAL; DIRI, 2009; GIGER et al., 2012). Errors in calculating metrics may also occur (ALVES; YPMA; VISSER, 2010). However, these errors are usually small and to minimize these interferences we use the Kruskal-Wallis and Cliff's δ statistical tests.

External validity. Some of the findings might be specific to the selected software systems and domains assessed. To minimize this bias, we discussed in Section 5.1.1 some well-defined and replicable criteria for selecting representative systems in each application domain. Although other domains use similar mechanisms to implement the architecture, we still intend to extend this investigation other systems and domains. We also do not claim that the design decisions considered in this study are the only design decisions that impact metric distributions. However, they were the most evident in the systems involved in our study. Future studies with other systems may evidence new design decisions impacting on the distribution of metric values. So, although we are restricted to the systems and domains analyzed, this is an important step toward improving the accuracy of metric-based assessment of source code.

5.4 RELATED WORK

Some studies have assessed the effect of coarse-grained design decisions on the distribution of software metrics. Zhang *et al.* (ZHANG et al., 2013) discuss that distribution of metric values depends greatly on the context of the project. They found six context factors that affect the distribution of at least 20 metrics. Programming language, application domain, and lifespan are three most important factors impacting over distribution values of 80% of the metrics. Therefore, our work complements it as we found fine-grained design decisions that also influence the distribution of metric values.

Aniche *et al.* (ANICHE et al., 2016) show that architectural roles affect the distribution of metric values. For example, a class playing the architectural role Controller, in an MVC-based system, has a different distribution of metric values from other architectural roles. However, they were able to identify and associate architectural roles to only 17.5% of the classes in MVC-based systems and 10.5% of the classes in Android applications. Consequently, metric-based assessments following such approach would disregard the design roles played by the rest of the classes. We deepen this discussion by showing that other design decisions also impact on metrics. We propose a heuristic that could correctly identify the class design role of 86.2% of the 1039 analyzed classes from five governmental systems. Considering the 15 selected systems used in our study, our heuristic was able to propose design role to 62.1% from Android classes, 73.5% for Eclipse classes and 77.2% for Web Classes. Therefore, the *Undefined* design role was assigned to 37.9% from Android classes, 26.5% for Eclipse classes and 22.8% for Web ones. As future work, we plan to investigate whether integrating other techniques, such as concern mining (WANG et al., 2011), to our heuristic improves the coverage and accuracy of design role identification. Additionally, we identified that others design decisions also affect the distribution of metric values.

Budi *et al.* (BUDI et al., 2011) propose a classification framework using a machine

learning technique that predicts a stereotype for each class. The heuristic identifies three class stereotypes (Entity, Control, and Boundary) introduced as an extension to the standard UML (RUMBAUGH; JACOBSON; BOOCH, 2004). Dragan *et al.* (DRAGAN; COLLARD; MALETIC, 2010) extends this set of class stereotypes to C++ systems. The approach uses patterns of the method stereotype distributions at the class level. Both approaches propose predefined stereotypes used in the analysis phase. Our approach to identify the design role played by classes could be used to define stereotypes focused on the design and implementation phases. We propose to use the class hierarchy and a customizable token-based method that is not limited to a predefined set of design roles. In the future, we plan to assess whether the stereotypes benefits, related to program comprehension, design recovery, and identification of code smells could be obtained with the design role information.

Oliveira et al. (2017) reports an industrial case study aimed at observing how 13 developers individually and collaboratively performed smell identification in five software projects from two software development organizations. The project manager of each system created the reference list of code smells derived of the initial list of code smells obtained running a smell detection tool. After creating the initial list, two researchers and system project manager performed a two steps manual validation. They propose using collaboration among developers to improve effectiveness on smell identification. However, strategies to support collaborative identification of code smells should provide contextual information.

5.5 SUMMARY

We conducted an empirical study to assess whether fine-grained design decisions affect the distribution of four method-level metrics. Our analysis was driven by the concept of design role. We consider design role itself as a design decision in the sense that the developer decide to assign a responsibility to a class in the context of a reference architecture. To support our study, we defined a heuristic to automatically identify the design role played by the classes of a system. The study involved fifteen real-world systems, from three different domains. The results and their implications for research and practice can be summarized as follow.

Design roles impact the distribution of metrics. Initially, our results showed that design roles affected the distribution of metrics (RQ1). A potential implication of this is that future researches should propose and evaluate metric-based analysis methods that take design roles into account. In fact, the major reason for the occurrence of false positive and negatives on smell detection methods is the lack of context for metric thresholds(SHARMA; SPINELLIS, 2018). Design roles might be considered to define this context. For instance, methods that use system benchmarks to calculate metric thresholds could derive thresholds according to design roles.

Fine-grained design decisions impact the distribution of metrics. Our results also showed that, due to different design decisions (for instance, coding style or used libraries) the same design role might have different metric distributions on different systems (RQ2). A potential implication of this is that we should select systems with similar design de-

cisions when building benchmarks for metric-based source code analysis. In addition, our results showed that differences on metric distributions across different releases were not frequent (RQ3). A practical implication of this is that companies should consider building their benchmarks from system releases that underwent design quality reviews.

In this context, as future work, we suggest extending this study with more systems, programming languages, metrics, and domains. We use these findings to propose two new techniques, detailed in Chapter 6, to derive design-sensitive metric thresholds.

Chapter

6

DESIGN-SENSITIVE TECHNIQUES TO DERIVE METRIC THRESHOLDS

This chapter describes the two proposed design-sensitive techniques to derive metric thresholds. Section 2.1 shows state-of-the-art automated static analysis tools (ASATs) used in real-world environment developments rely on metrics to identify code smells. However, as discussed in Section 3.2, selecting proper metric thresholds is still a challenge faced by many development teams. On the one hand, a low metric threshold could lead to many false code smell alarms; on the other hand, a high metric threshold could hide potential code smells. We hypothesize that using the class' design role as context to derive metric thresholds could improve the accuracy of code smells detection strategies. For example, does it make sense to evaluate a Web system's quality using metric thresholds derived from a benchmark composed only of Android applications? Does it make sense to evaluate business classes with thresholds derived from classes playing the persistence design role?

The proposed techniques take a step forward by proposing solutions for (i) selecting similar systems to compose the benchmark used to derive metric thresholds based on the similarity of design roles and (ii) deriving multiples metric thresholds, one for each design role. Both solutions use the concept of design role proposed in Section 4. We conducted some preliminary studies to evaluate the proposed approaches (DÓSEA; SANT'ANNA; SANTOS, 2016; DÓSEA; SANT'ANNA, 2016; LIMA; DÓSEA; SANT'ANNA, 2016). We use the lessons learned obtained in these studies to propose improvements in the heuristic to assign the design role played by each system class.

The proposed techniques rely on Alves et al. technique (ALVES; YPMA; VISSER, 2010), discussed in Section 2.3. We use it because they propose a repeatable, transparent, and straightforward methodology for deriving software metric thresholds. This methodology respects metric statistical properties and proposes to derive thresholds based on data analysis from a representative set of systems (benchmark). Also, Boucher and Badri (BOUCHER; BADRI, 2018) compare techniques to derive metric thresholds for

fault-proneness prediction and show that Alves' technique outperformed machine learning and clustering techniques. Although Alves' technique under-performed ROC curves techniques, it is wholly unsupervised and can give pertinent threshold values when fault data is not available to ROC curves technique (BOUCHER; BADRI, 2018).

Finally, both techniques respect the following requirements proposed and discussed by Alves et al. (ALVES; YPMA; VISSER, 2010) and Vale and Figueiredo (VALE; FIGUEIREDO, 2015): (i) it should respect the statistical properties of the metric, such as scale and distribution; (ii) it should be based on data analysis from a representative set of systems (benchmark); (iii) it should be repeatable, transparent and straightforward to execute; (iv) it should calculate upper and lower thresholds; (v) it should derive thresholds in a step-wise format.

6.1 DERIVING GENERIC METRIC THRESHOLDS FROM BENCHMARK OF SYSTEMS DEVELOPED WITH SIMILAR DESIGN DECISIONS

The first proposed technique derives generic metric thresholds from a benchmark of systems developed with similar design decisions. According to Vale et al. (2015), metric thresholds are sensitive to benchmarks. Therefore, we need to make the benchmark buiding methodology clear because it influences derived metric thresholds. Both proposed techniques start building a benchmark of systems developed with similar design decisions. Our preliminary empirical studies (DÓSEA; SANT'ANNA; SILVA, 2018; LIMA; DÓSEA; SANT'ANNA, 2016) discussed that developers could quickly point out high-quality systems developed with similar design decisions. We also propose an automated alternative, discussed in Section 4.4, when such identification performed manually by expert developers is not possible. Figure 6.1 illustrates the seven steps of the first design-sensitive technique:

1) *Benchmark creation.* We propose to extract metric thresholds from a benchmark of systems developed with similar design decisions. We considered that two systems have similar design decisions when they have similar (i) set of design roles, (ii) set of used libraries, (iii) coding style, (iv) exception handling code and, (v) logging and debugging code. Expert developers can carry out this step manually, selecting high-quality systems to compose the benchmark based on previous knowledge of these design decisions. Another alternative is using our proposed approach to identify similar systems detailed in Section 4.4. A third alternative is taking a previous stable release of the system itself to compose the benchmark because some design decisions can be specific to the evaluated system. Besides, we have not found significant differences in the distribution of metrics across different stable releases as discussed in Section 5.2.

2) *Metrics extraction.* We extract code metrics for all classes in the benchmark. For each system *System*, and for each entity *Entity* belonging to *System* (e.g. method), we record a metric value *Metric*, and a weight metric *Weight*. As weight we will consider the source lines of code (LOC) of the entity.

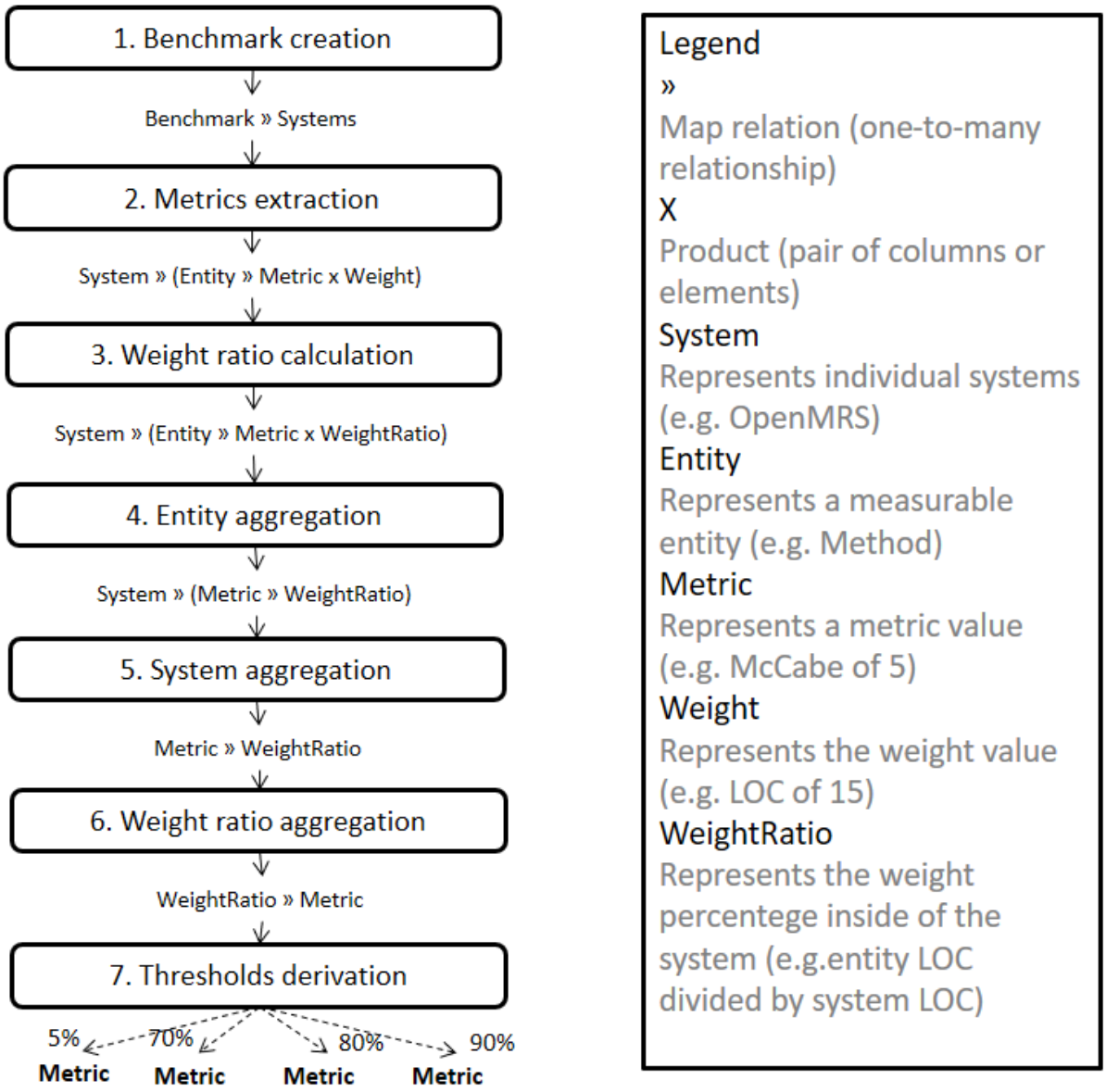

Figure 6.1: Technique to Derive Metric Thresholds from Systems Developed with Similar Design Decisions

3) *Weight ratio calculation.* Following the approach proposed by Alves et al. (ALVES; YPMA; VISSER, 2010), for each entity, our technique computes the weight percentage within its system, i.e., we divide the entity weight by the sum of all weights of the same system. For each *System*, the sum of all *Entities* must be 100%.

4) *Entity aggregation.* We aggregate the weights of all entities per metric value, which is equivalent to computing a weighted histogram (the sum of all bins must be 100%). Hence, for each system, we have a histogram describing the distribution of weight per metric value.

5) *System aggregation.* We normalize the weights for the number of systems and aggregate the weight for all systems. Normalization ensures that the sum of all bins remains 100%, and then the aggregation is just a sum of the weight ratio per metric value.

6) *Weight ratio aggregation.* We order the metric values in ascending way and take the maximal metric value that represents 1%, 2%, ..., 100% of the weight. This is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale.

7) *Thresholds derivation.* For each metric, we derive a generic threshold by choosing the percentage of the overall code we want to represent. We use the following percentiles to characterize metrics value according to four categories: low values (0-5%); low risk (0-70%); high risk (70-80%); very high risk (>90%) to metrics correlated with LOC (e.g., McCabe complexity). To metrics not correlated with LOC (e.g., number of parameters by method), we use the percentiles low values (0-7%), low risk (0-80%), high risk (90-95%), very high risk (>95%). Vale e Figueiredo (2015) discussed that not considering LOC correlation may damage the results of Alves' technique. Alves et al. (ALVES; YPMA; VISSER, 2010) also suggest higher percentiles when metric variability is relatively small. Finally, we also define a percentile to low values because it may identify lower bound outliers to identify some code smells. For instance, Lazy class (FOWLER; BECK, 1999) is a class that knows or does too little in the software system. We could consider the lower LOC of classes to indicate this code smell.

6.2 DERIVING METRIC THRESHOLDS PER DESIGN ROLE

The second technique, additionally to the technique detailed in Section 6.1, proposes to derive distinct metric thresholds for each class design role. This technique has been evolved based on our findings, and lesson learned obtained through a set of empirical studies.

We use the class design role as context in two steps: (i) on creating a benchmark to derive metric thresholds. This benchmark must be composed of high-quality systems with similarity on design decisions; and (ii) on the derivation of multiple metric thresholds for each design role rather than a single generic threshold to evaluate all system classes.

Figure 6.2 illustrates the eight steps of the second proposed design-sensitive technique. Steps one, three, four, and five are the same as the technique detailed in Section 6.1.

1) *Benchmark creation.* We propose to extract metric thresholds from a benchmark of systems developed with similar design decisions. We considered that two systems have similar design decisions when they have similar (i) set of design roles, (ii) set of used libraries, (iii) coding style, (iv) exception handling code and, (v) logging and debugging code. Expert developers can carry out this step by selecting the high-quality systems to compose the benchmark based on previous knowledge of these design decisions. Another alternative is using our proposed approach to identify similar systems detailed in Section 4.4. A third alternative is taking a previous stable release of the system itself to compose the benchmark because some design decisions can be specific to the evaluated system. Besides, we have not found significant differences in the distribution of metrics across different stable releases as discussed in Section 5.2.

2) *Metrics and design role extraction.* We extract code metrics and design roles for all classes in the benchmark. For each system class, we record the design role proposed by the heuristic discussed in Section 4.2. Also, for each system *System*, and for each entity *Entity* belonging to *System* (e.g. method), we record a metric value *Metric*, and weight metric *Weight*. As weight, we will consider the source lines of code (LOC) of the entity.

3) *Weight ratio calculation.* Following the approach proposed by Alves et al. (ALVES; YPMA; VISSER, 2010), for each entity, our technique computes the weight percentage within its system, i.e., we divide the entity weight by the sum of all weights of the same system. For each *System*, the sum of all *Entities* must be 100%.

4) *Entity aggregation.* We aggregate the weights of all entities per metric value, which is equivalent to computing a weighted histogram (the sum of all bins must be 100%). Hence, for each system, we have a histogram describing the distribution of weight per metric value.

5) *System aggregation.* We normalize the weights for the number of systems and aggregate the weight for all systems. Normalization ensures that the sum of all bins remains 100%, and then the aggregation is just a sum of the weight ratio per metric value.

6) *Design role aggregation.* We also normalize the weights for the number of design roles and then aggregate the weight for all design roles. Normalization ensures that the sum of all bins remains 100%, and then the aggregation is just a sum of the weight ratio per design role and metric value.

7) *Weight ratio aggregation.* We order the metric values in ascending way and take the maximal metric value that represents 1%, 2%, ..., 100% of the weight. This is equivalent to computing a density function, in which the x-axis represents the

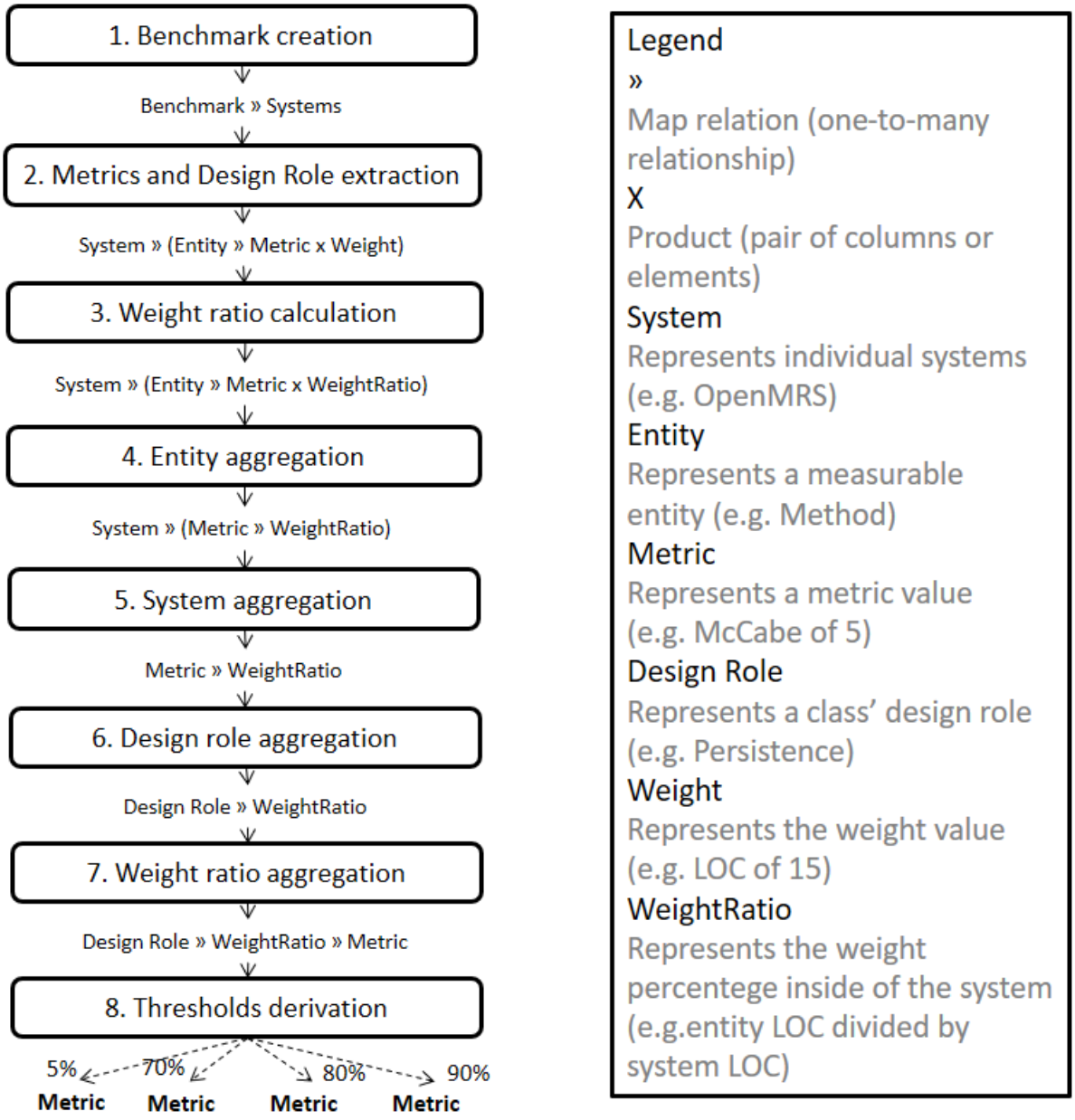

Figure 6.2: Technique to Derive Multiple Metric Thresholds for each Metric from Systems Developed with Similar Design Decisions

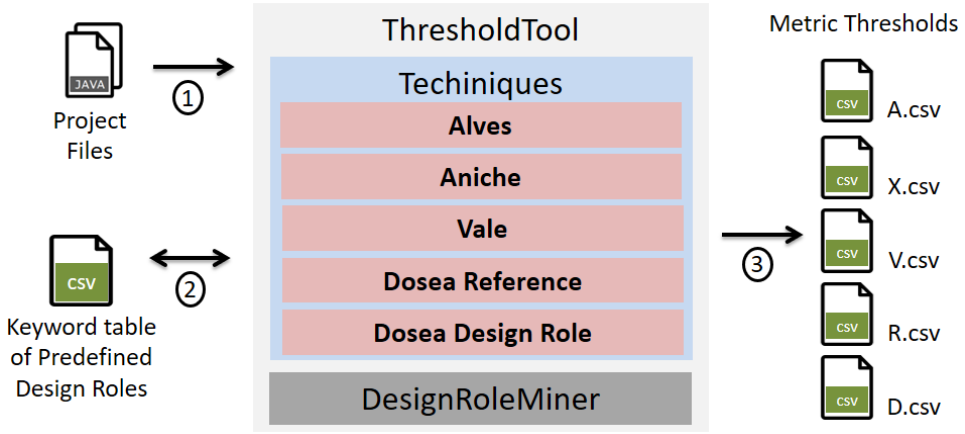

Figure 6.3: ThresholdTool Deriving Metric Thresholds

weight ratio (0-100%), and the y-axis the metric scale. We aggregate the weight ratio per metric and also per design role and metric.

8) *Thresholds derivation.* For each metric and design role, we derive a threshold by choosing the percentage of the overall code we want to represent. Each metric and design role has a distinct threshold value. The metric threshold is the highest value obtained between the threshold calculated for the design role and the threshold calculated for overall systems' classes. For instance, if the LOC metric threshold calculated to represent 90% of the overall code is 70 and the threshold value calculated to represent 90% of the design role *Business* code is 95, then the threshold value for the LOC metric for classes playing *Business* design role is 95. However, if the threshold calculated to represent 90% of this design role code is less than 70, then the value 70 is considered. On the one hand, in well-defined architectures, the source code assigned to each design role is usually representative to define specific metric thresholds. On the other hand, it does not make sense to define metric thresholds lower than metric thresholds that represent the benchmark's source code. We use the following percentiles to characterize metrics value according to four categories: low values (0-5%); low risk (0-70%); high risk (70-80%); very high risk (>90%) to metrics correlated with LOC (e.g., McCabe complexity). To metrics not correlated with LOC (e.g., number of parameters by method), we use the percentiles low values (0-7%), low risk (0-80%), high risk (90-95%), very high risk (>95%). We discussed in Section 6.1 our motivations to consider these decisions.

6.3 TOOL SUPPORT TO DERIVE METRIC THRESHOLDS

We provide a tool, called ThresholdTool, to derive metric thresholds using our two proposed techniques. Besides, the tool also derives metric thresholds by means Alves' technique (ALVES; YPMA; VISSER, 2010), Aniche's technique (ANICHE et al., 2016), and Vale's technique (VALE; FIGUEIREDO, 2015)

Figure 6.3 illustrates the three steps performed to extract metric thresholds using the

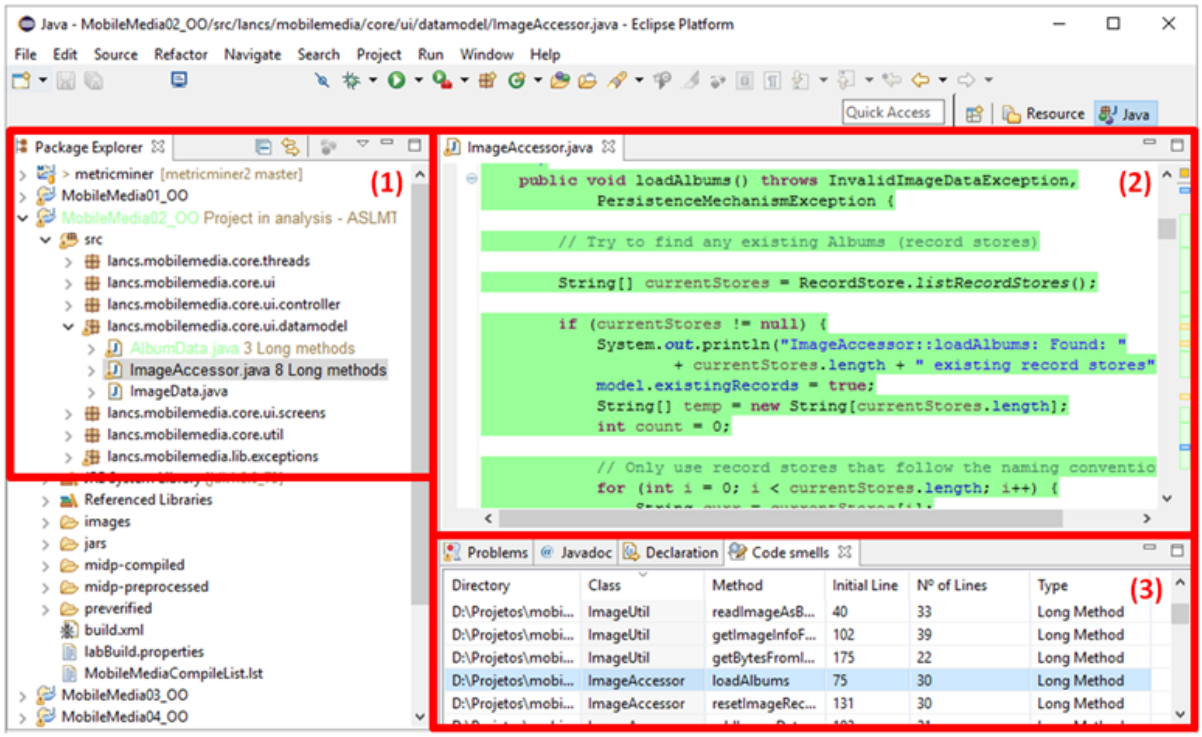

Figure 6.4: ContextSmell Plug-in for Eclipse.

five techniques. Firstly, we select the benchmark of projects to derive metric thresholds. In the second step, we can adjust the table of predefined design roles to evaluate systems' design decisions. In the third step, the tool generates five files, one for each technique, with metric thresholds.

We also developed the ContextSmell tool, which extends ContextLongMethod (SANTOS; DÓSEA; SANT'ANNA, 2016). The tool can perform with a command-line interface or as an Eclipse plugin. It uses as input the thresholds' files generated by ThresholdTool and generates a list of code smells identified by the thresholds derived from each technique. Each line shows the class, method, code smell, and the list of techniques that derived metric thresholds that pointed out the code smell.

Figure 6.4 shows the three views used by the ContextSmell plugin to recommend code smells for software developers. View (1) shows in the package explorer the analyzed systems by the tool. View (2) shows details about the identified code smells in the Eclipse editor. Finally, view (3) shows a list of all identified smelly methods in the evaluated system.

Chapters 7 and 8 describe studies we carried out to evaluate the proposed techniques.

Chapter

7

COMPARING TECHNIQUES TO DERIVE METRIC THRESHOLDS BASED ON DEVELOPERS' PERCEPTION OF CODE SMELLS

In Chapter 3, we showed a large-scale survey conducted with Brazilian practitioners to answer our first general research question (RQ1) *How do practitioners perceive automated static analysis for code smell identification?* Among other findings, we found that practitioners' perception is that metric-based strategies to detect code smells should consider some context information to select the most appropriate metric thresholds. In Chapter 5, we answer our second general research question (RQ2) *Are there statistically significant differences between measures obtained from classes developed with different design decisions?* We discussed class design role as a design decision that impacts the distribution of metrics, and, therefore, could be considered as context to derive metric thresholds.

This chapter shows our first empirical study to evaluate our two proposed techniques, described in Chapter 6, that considers class design role as context to derive metric thresholds. We compare our two techniques with other three state-of-the-art techniques to derive metric thresholds. As comparison criteria, we use developers' perception about if they would refactor code smells pointed out by thresholds from the five techniques. Developers evaluated the existence of four code smells in a sample of methods systematically selected from Web systems they maintain.

Defining accurate metric thresholds to be used by code smells detection strategies faces some challenges. One of those challenges is the subjectivity involved in developers' evaluation of code smells (Kamei; Shihab, 2016). Initial studies show that distinct developers can disagree about code smell in a particular method, and inter-rater agreement was low for simple code smells and low for refactoring decision (MANTYLA; LASSENIUS, 2006; MANTYLA, 2005). Hozano et al. (2018) perform a broader study to investigate how similar developers detect code smells. They found that the developers presented a low agreement on detecting all 15 smell types analyzed. We hypothesize that many disagreements in previous studies occur because developers have no experience and clarity about design decisions of the systems they evaluated. Refactoring some code smells,

detectable by tools, may not make sense to a team of developers. Tools disregard many design decisions agreed upon by the team.

For this reason, we carry out a study with developers who maintained the evaluated systems. They have experience and clarity about design decisions on evaluated methods. We aim to answer our third general research question (RQ3):

RQ3: Are design-sensitive metric thresholds more accurate to detect code smells prone to be refactored?

To answer this research question, we conducted an industrial multi-project study analyzing the developers' perception of the source code maintained by them. Developers evaluated Web systems that use the Struts framework, or the Java Server Faces framework. They analyzed four types of code smells pointed out by metric thresholds derived from five distinct techniques. Evaluating all smelly methods is very difficult due to the developers' time required and fatigue in the process. Therefore, we propose a systematic process to select a representative set of smelly methods to be analyzed. Next, we show these methods to the developers, asking if they would refactor these methods. We aim to identify which technique derived the most precise metric threshold to detect smelly methods prone to be refactored according to developers' perception.

We summarize our findings as follows:

- We found a high degree of agreement among developers. We suppose that the familiarity with the evaluated system code and design decisions determined this high agreement degree.

- We obtain that our proposed design-sensitive techniques derived metric thresholds that reduce false positives according to the developer's perception.

- We also investigated qualitatively if the design roles influenced the developer's perception of smelly methods. The results we obtained show that the class design role influenced many developers' perceptions about smelly methods prone to refactoring.

We carried out this study in partnership with a master's degree student from PGCOMP/UFBA. We helped him in the planning, execution, and analysis of the results. The student performed the study within his work environment. The tools used to derive metric thresholds and point out code smells are also contributions of this thesis. This chapter includes our analysis of the obtained data. The student conducted other analyzes, and more in-depth discussions focused on interviews in his master's work (LIMA, 2021).

The remainder of this chapter is organized as follows. Section 7.1 describes the settings of our empirical study. Section 7.2 presents the results of the study. Section 7.3 discusses threats to validity and Section 7.4 discusses related work. Finally, Section 7.5 summarizes the results and discusses implications of our research.

7.1 STUDY SETTINGS

This study compares metric thresholds derived from five distinct techniques to point out smelly methods prone to refactoring. We compare the proposed code smells with developers' perceptions responsible for maintaining the evaluated source code. We hypothesize that the class design role could influence developer's perception of smelly methods.

7.1.1 Research Questions

We formulated the following research questions:

RQ1: *What is the level of agreement between developers' perception of code smells identified in software systems maintained by them?* This research question evaluates the developers' agreement about pointed out code smells in systems maintained by them, that is, systems that they are familiar with design decisions. In our study, two developers agree when both of them either do not consider that a method has a code smell or consider that method has a code smell and say that could refactor it. Previous works show low agreement between developers who assess code smells in the same code snippet (HOZANO et al., 2018). However, most of the developers in earlier studies are not familiar with the design decisions of the evaluated system.

RQ2: *Which technique proposes metric thresholds that best reflect the individual developers' perception about code smells in software systems maintained by them?* The second research question quantitatively identifies the individual developers' perception of code smells pointed out by metric thresholds derived from the five studied techniques. We aim to point out the one that derived metric thresholds more similar to most individual perceptions of developers. We consider a method as smelly when the developer claims that there is a need to refactor that method. In this research question, we consider each developer's opinion with the same weight because previous studies show wildly divergent perceptions about code smells among developers who analyze the same source code snippet (SCHUMACHER et al., 2010; SANTOS et al., 2018; HOZANO et al., 2018). To address RQ2, we test the following null hypothesis.

$H0_1$: no technique proposes metric thresholds that improve Matthews Correlation Coefficient (MCC) to detect code smells compared to individual developers' perception of code smells in software systems maintained by them.

$H0_2$: no technique proposes metric thresholds that improve precision to detect code smells compared to individual developers' perception of code smells in software systems maintained by them.

RQ3: *Which technique proposes threshold values that best reflect two developers' joint perception of code smells in software systems maintained by them?* The third research question aims to analyze the perception of pairs of developers. We consider valid only the answers where there was an agreement between the two developers who examined the same familiar source code. We consider that developers agree when both do not detect any code smell or propose refactoring the same code smell in the same evaluated method. This research question aims to reinforce the individual developer's evaluation results due to the divergences of perception between them pointed out by previous studies.

Table 7.1: Developers' Background

Study	**Experience in Evaluated Projects**	**Developers** ($n = 10$)
Pilot	1-5 years	D1, D2
Final	1 year	D8, D9
	1-5 years	D3, D4, D5, D6
	5-10 years	D7, D10

To address RQ3, we test the following null hypothesis.

$H0_3$: no technique proposes metric thresholds that improve Matthews Correlation Coefficient (MCC) to detect code smells compared to two developers' joint perception of code smells in software systems maintained by them.

$H0_4$: no technique proposes metric thresholds that improve precision to detect code smells compared to two developers' joint perception of code smells in software systems maintained by them.

RQ4: *Do developers familiar with the source code consider the class design role information to identify refactoring-prone code smells?* The fourth research question aims to assess qualitatively developers' perception of how the class design role influenced them to point out refactoring-prone smelly methods.

7.1.2 Context Selection

We carried out the study with ten developers who work in the software development industry. Initially, we conducted a pilot study with two developers to evaluate the study settings. Then, we performed some adjustments in the study settings, and we executed the final study with the other eight developers. They are the most experienced in their development teams. Each developer evaluated the quality of methods of only one system, the one he maintains. We aim to determine whether design decisions influenced code smell developers' perception in source code whose design decisions are familiar.

We conducted the study at the Information Technology Superintendence of the Federal University of Bahia. The team has more than 80 developers maintaining about 100 software systems that automate academic and administrative processes. Most systems are developed in Java and have some architectural differences due to technological evolution. We selected the participants by convenience since it is not easy to have available practitioners usually concerned with project deadlines in progress. Table 7.1 summarizes the experience of the ten developers (D1 to D10) in the evaluated projects. It is worth mentioning that all developers have at least five years of experience in software development. D4, D6, and D7 have more than ten years of experience.

Regarding the participants' role, six of them are system analysts, and four are system developers. Both developers and system analysts are roles concerned with source code maintenance. All respondents are familiar with the evaluated source code. Eight respondents have 1 to 5 years of experience, and only D8 and D9 developers have less than one year of experience in the evaluated projects.

Table 7.2: Target Systems

View Layer	System	#Classes	#Methods	#LOC	#Design Roles	#Smelly Methods	#Evaluated Methods
JSF	S1	53	525	5277	4	156	53
	S3	158	1898	22944	11	460	83
	S4	349	3127	41081	13	915	117
Struts	S2	76	844	8545	11	106	85
	S5	457	5138	60240	7	664	63
	S6	949	12759	231183	9	2287	91

Target Systems: We use six real-world Web systems, developed in Java, by the Information Technology Superintendence of the Federal University of Bahia. Empirical studies aimed to evaluate developers' perception of code smells usually do not use web-based domain (PALOMBA et al., 2014; VALE; FERNANDES; FIGUEIREDO, 2018). However, some data indicate this application domain appears to have the largest number of assigned software developers. For instance, Borges, Hora e Valente (2016) discussed that Web libraries and frameworks are the top-domain (33%) between the top-2,500 public repositories with more stars in GitHub. According to the 2020 StackOverflow Developer's Survey, the 3-top most popular technologies with professional developers are Javascript (69.7 %), HTML/CSS (62.4 %), and SQL (56.9 %), which are the basis for Web-based development. Stack Overflow's annual Developer Survey is the largest survey of people who code around the world. It presents detailed and anonymized results and made them available for download. In 2020, the survey was taken by nearly 65,000 people.

The selection of these systems was made by convenience, prioritizing systems where the most experienced developers were available for interview. We use two systems for to pilot study and four systems for the final study.

Table 7.2 summarizes the main characteristics of the six selected Web systems. The main architectural difference of the systems is on the View layer. The three older systems use the Struts framework, and the three newer ones use the JSF framework. We aim to compare systems developed with different design decisions. The View layer column shows the framework used in each system. The #classes, #methods, and #LOC columns show the number of classes, methods, and lines of code in each system. The #design roles column shows the number of design roles identified by our heuristic (Section 4.2). Finally, the #smelly methods column shows the number of methods pointed out for at least one code smell, and the #evaluated methods columns show the number of methods evaluated by each system developer. For instance, S1 is a system to manage Teachers and Student Assessments. The S1 system has 53 classes, 525 methods, 5277 lines of code, four design roles, and 53 methods evaluated by a system developer out of 156 methods pointed out as smelly. S2 is a system to manage Integrated Services and Users developed with the Struts framework. The S2 system has 76 classes, 844 methods, 8545 lines of code, 11 design roles, and have 85 methods evaluated by a system developer out of 106 pointed out as smelly.

Code Metrics: Our study considered the four method-level metrics and code smells

Table 7.3: Method-level Metrics and Code Smells

Metric	**Description**	**Code Smell**
McCabe's Cyclomatic Complexity (CC) (MCCABE, 1976)	It counts the number of branching points of each method.	High Complexity
Lines of Code (LOC) (LANZA; MARINESCU, 2006)	It counts the number of executable statements of each method, excluding comments and blank lines.	Long Method
Efferent Coupling (EC) (MARTIN, 1995)	It counts the number of classes from which each method calls methods or accesses attributes.	High Efferent Coupling
Number of Method Parameters (NMP) (FOWLER; BECK, 1999)	It counts the number of parameters of each method.	Long List of Parameters

discussed in Table 7.3.

We selected these method-level metrics, and associated code smells because we can manually compute them without tool support. This criterion is essential for conducting the manual analysis planned for our study. Also, these metrics are available in many tools (PAIVA et al., 2017) and have been successfully used for fault-proneness prediction (FONTANA et al., 2013; GIL; LALOUCHE, 2017; BOUCHER; BADRI, 2018), for instance. We used only one metric for each code smell detection strategy because we aim to evaluate the precision of metric thresholds derived by techniques according to developers' perception. Besides, many automated static analysis tools also use only a single metric to detect these code smells.

Techniques to Derive Metric Thresholds: We selected five techniques that do not consider the normal distribution of metric values since this is very difficult to occur (GIL; LALOUCHE, 2016, 2017). We also consider techniques that can be fully automated. Based on these criteria, we have selected Alves et al. (ALVES; YPMA; VISSER, 2010) and Vale et al. (VALE; FIGUEIREDO, 2015) techniques, detailed in Section 2.3, which generate a generic threshold value for each metric. We propose a variation of Alves et al. technique, detailed in Section 6.1, that considers as context the class design roles in selecting systems to benchmark. This benchmark must be composed of high-quality systems with similarities in design decisions. We call this proposed technique as [T1]. We also evaluated Aniche et al. technique (ANICHE, 2015), detailed in Section 2.3, which is also based on Alves et al. technique but defines specific metric thresholds, each one related to each architectural role. For example, to evaluate a class associated with the architectural role *Controller* it generates a specific metric threshold assigned to this role. Finally, we propose a second technique, discussed in Section 6.2 that also considers the class' design role in the benchmark creation process. This technique also

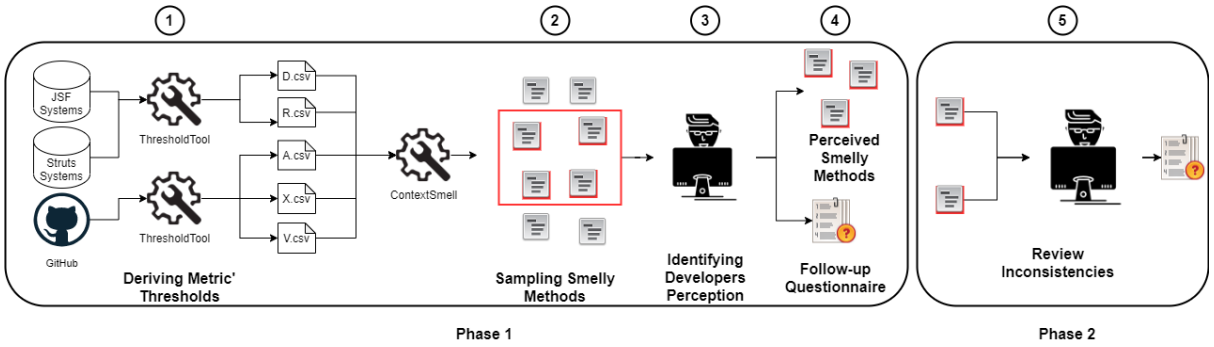

Figure 7.1: Study Procedures conducted with Software Developers

derives multiple metric thresholds ,one for each design role, rather than a single generic threshold to evaluate all system classes. The class design role, discussed in Chapter 4.2, is an extension of the architectural role concept allowing to assign a role to classes not bound to a predefined reference architecture. We call our second proposed technique of [T2].

7.1.3 Study Procedure

To answer the research questions, we conducted interviews with software developers. Figure 7.1 outlines the research method divided into five activities and two phases. Each phase represents an interview with the developer. Initially, we derived metric thresholds using the five evaluated techniques (activity 1). Then, we performed a sampling of smelly methods (activity 2). Then, the software developers evaluated the selected methods (activity 3), and we performed a final interview (activity 4). Finally, we reviewed detected inconsistencies with the software developers (activity 5). Below we detail these five activities.

Activity 1: Building benchmarks, deriving metric thresholds, and selecting smelly methods. For Alves et al., Vale et al., and Aniche et al. techniques, we derived metric thresholds from the same benchmark composed of 17 real-world Java Web systems. We selected systems with at least 50 stars and one update since 01/03/2019 from the Github repository on 17/07/2019. These criteria returned a list of 268 projects. We excluded libraries and frameworks, systems used as implementation examples, systems with no release, and with less than 100 classes. These criteria aimed to select real-world Web systems and exclude systems too different from the evaluated systems.

We also built two other distinct benchmarks to execute [T1] and [T2] techniques. We used high-quality reference systems, selected by the two most experienced developers according to architectural similarity. Therefore, the second benchmark is composed of eight Web systems developed with Java Server Faces (JSF) framework, and the third benchmark is composed of six Web systems developed with the Struts framework. Despite these frameworks on View layer being the main architectural difference, some minor differences in some design decisions exist. For example, some JSF benchmark' classes also use the Java Persistence API to manage objects' persistence. The Java Persistence

API (YANG, 2010) simplifies Java EE and Java SE applications' development, providing a single, standard persistence API.

We used the ThresholdTool, discussed in Section 6.3, to derive metric thresholds using the five techniques. Then, we used the line-command interface of ContextSmell tool, discussed in Section 6.3, to generate a worksheet with a list of methods with code smells, detailed in Table identified by means of the metrics and corresponding thresholds derived from the five evaluated techniques.

Activity 2: Sampling of smelly methods. The pilot study showed that evaluating all smelly methods would be tedious and error-prone. In fact, in previous similar studies involving the evaluation of code snippets, the participants did not evaluate the entire source code of systems (YAMASHITA, 2013; PALOMBA et al., 2014; OLIVEIRA et al., 2017; HOZANO et al., 2018). For example, by using the metric thresholds derived from the five techniques, we detected 664 smelly methods in the S5 system. Initially, we tried to use a classic sample calculation formula, which suggested that the participants should evaluate 244 methods for the study to have a statistically significant result with 95% confidence. In the pilot study, we realized that the evaluation time could not exceed 2 hours to mitigate participants' fatigue. This time would be insufficient to evaluate 224 methods. Then, we decided to divide the set of smelly methods into equivalent partitions (MYERS et al., 2004). We defined design role as criteria to group methods in equivalent partitions. For each partition, we select a representative subset to be evaluated by the developers.

In the pilot study, the developers evaluated 10% of the methods in each partition. For instance, if *Business* design role contains 120 methods, we select 12 methods for evaluation. In order to select the 10% of methods of each partition, we proceeded as follow. We sorted the methods in a partition in ascending order by the metric value. Then, we selected the first (lowest metric value), and last method (highest metric value) from this ordered list. Then we selected the method located in the median of the metric values. When we did not reach 10% of methods, we repeated this procedure to select one more method in the median of the interval between the methods in previous first and median positions. If necessary, we selected another method in the median of the interval between the methods in previous median and last positions. We continued this procedure recursively until the 10% percentage was reached.

We carried out the pilot study with systems from which the resulting code smell list was not long. Even though, we observed developer did not evaluate the last methods as carefully as the first ones. This might have occurred due to fatigue. Also, we identified that methods of classes assigned to the same design role and with similar metric values had a similar evaluation by the software developers. For example, if a class playing the *Persistence* design role had a method with 50 lines of code evaluated as long, another method with 53 lines of code and playing the same design role was also considered long. Similarly, a previous large-scale study (HOZANO et al., 2018) reported that developers dedicated, on average, 2.5 hours to evaluate 80 code snippets. Participants also said that such activity became slightly dull, and such issue negatively influenced their last evaluations. To mitigate this problem, the authors adjusted the main study limiting the number of code snippets evaluated by each developer to 15. Our pilot study showed better

participants' performance since they were familiar with the evaluated code snippets and design decisions.

Given the experience with the pilot study, we decided to select fewer methods for the final methods. This would avoid fatigue and avoid participants to evaluate many methods with the same design role and similar metric values. Therefore, we decided to have a maximum of five methods per design role and metric to compose our sample. We used procedure we used in the pilot study to select the methods, but stopping with five methods, instead of 10%. For instance, the *Persistence* design role had seven methods pointed out as high complexity. The values of the McCabe complexity metric in ascending order were 10, 11, 15, 16, 20, 25, 35. Following the procedure, we selected the method with the lowest value (10), then the highest (35), then the median (16), then the median between the lowest and the first median (15), and, then, the median between the first median and the highest (25), stopping with five selected methods, with the following cyclomatic complexity values: 10, 35, 16, 15 and 25.

Activity 3: Collecting developers' perception about smelly methods. Initially, we conducted a brief training to align the terminology of the four code smells that the developer should look for in each evaluated method. For each code smell, we showed the name and description contained in Table 7.3. We focused on evaluating only four method-level code smells to avoid fatigue and keep developers focused on the systems evaluation process. The selected code smells are popular and easily recognized by system developers. An extensive list of code smells could divert developers' attention and increase the time needed for evaluation.

Next, we conducted semi-structured interviews with participants. This research approach is used in exploratory studies to understand phenomena and seek new insights. We carried out interviews with developers from each system one at a time. We structured the interview into three parts.

In the first part, each developer received a worksheet containing the design roles identified by the heuristic proposed in Section 4.2. Each row of the worksheet included a class name, the class' design role assigned by the heuristic, and a field for the developer to answer if he agreed or not with the proposed design role. Each developer did that for all system classes, except the classes assigned to the *Undefined* design role. The heuristic assigns *Undefined* when it cannot identify the design role play by the system class. We started by validating the design roles, aiming that the developers become familiar with the design roles' names. The results are discussed in Section 4.3.2.

In the second part, developers evaluated the sample of methods selected on Activity 2. Each developer received a worksheet in which each row contains a class name, the class design role, a method name, and two questions: "Would you refactor this method before starting a maintenance activity?" and "What code smells influenced your refactoring' decision?". They did not know that the evaluated methods were pointed out as having code smells. We guided developers to focus on the four code smells we trained them on. This way, for each method, each developer could: (i) say he did not consider it is affected by any code smell, or (ii) indicate one or more of the four code smells he think it affected. Finally, to allow a qualitative analysis, we asked about each method: "What implementing features of the method influenced you to point out the code smell(s)?". With

this question, we intended to identify whether the class design role influenced developers' perceptions in code smells identification.

Aiming to avoid bias in the developers' evaluation, we presented the methods alternating metric low and high metric values. For example, considering M_1, M_2, M_3, M_4, and M_5 methods with metric values ordered in ascending order, we present them to system developers in the following order: M_4, M_1,M_5, M_2, and M_3.

Activity 4: Answering the follow-up questionnaire. After evaluating the source code of the methods, developers answered a follow-up questionnaire.

We started by asking the following question: "Do you think that methods associated with the same design role and similar metric values will have the same code smell? Please justify your answer and cite examples.". We intend to understand the propensity for a code smell identified in a design role to occur in other methods associated with the same design role with similar metric values since evaluating all system methods would be very difficult and error-prone.

Finally, we asked some background questions intended at characterizing the sample of the involved developers. We intended that the interview time should not exceed 2 hours. All interviews were recorded and then transcribed for analysis, preserving the anonymity of the interviewees.

Activity 5: Reviewing inconsistencies in code smell identification. The pilot study showed some inconsistencies in developers evaluation. Similar methods of classes playing the same design role and with similar metric values were not always associated with the same code smell. We then conducted a second interview with the same developer to evaluate again methods he did not consider smelly, whenever there was at least one smelly method with a lower metric value. For example, considering that M_1 method playing the design role *Business*, classified by the developer with high cyclomatic complexity. The developer evaluated two other methods M_2 and M_3 with higher metric values of complexity than the M_1 method as not smelly. In this case, we carried out a new evaluation of M_1, M_2 and M_3 methods asking the developer: " Why does M_1 method has high complexity and M_2 and M_3 methods have not high complexity?". We aimed to remove inconsistencies, reduce the subjectivity of the evaluation process, and identify other design decisions that could influence developers to evaluate a method as smelly.

7.1.4 Data Analysis

To answer the RQ1 research question, we computed the inter-rater agreement among developers' evaluation using *Fleiss' Kappa* measure (FLEISS; COHEN; EVERITT, 1969). This measure reports a number between 0 and 1. When Kappa measure is closer to zero, we have a smaller agreement between developers. We have a more significant agreement between developers if Kappa measure closer to one. We also used the categories proposed by Landis and Koch (LANDIS; KOCH, 1977) listed in Table 7.4 and used by previous works related to code smell detection (MANTYLA, 2005; HOZANO et al., 2018).

We compared the agreement of evaluations done by developers that evaluated the same system methods (eight developers and four systems). Each developer evaluated the code smells of one system, which he is responsible for maintaining. Each developer

Table 7.4: Strength of Agreement of Kappa Statistics

Kappa Statistic	**Strength of Agreement**
<0.00	Poor
0.00-0.20	Slight
0.21-0.40	Fair
0.41-0.60	Moderate
0.61-0.80	Substantial
0.81-1.00	Almost Perfect

assessed between 61 and 117 methods. After calculating the Kappa measure, we used the strength of agreement to answer RQ1. We considered a good agreement when the strength of agreement is *Substantial* or *Almost perfect*.

To answer RQ2 and RQ3 research questions, we performed quantitative data analysis. In RQ2, we compared the effectiveness of metric thresholds based on the developers' individual perception about the code smell in the sample of methods. In RQ3, we carried out a similar analysis considering the inter-agreement of pairs of developers who evaluated the same set of methods.

To assess metric thresholds' effectiveness, we computed two well-known metrics: precision, and Matthews Correlation Coefficient (MCC). Precision represents the fraction of instances of methods predicted as smelly by metric thresholds that are actually smelly according to developers' perception. MCC is a correlation coefficient based on all four quadrants of the confusion matrix. It has values in the range [-1,+1] where a coefficient of +1 represents a perfect prediction and -1 indicates total disagreement between prediction and observation.

Since we considered several systems, we aggregated the results achieved for each metric to have a more straightforward overview of the quality of the results. Aggregate metrics are more robust than the mean, which is biased by the fact that datasets are unbalanced for different smell types in terms of smelly and non smelly instances (in some cases the datasets do not contain any smelly instance) (PECORELLI et al., 2019, 2020). Therefore, we aggregated the obtained confusion matrices before computing precision, and MCC (ANTONIOL et al., 2002; PECORELLI et al., 2019). We used these metrics to compare metric thresholds effectiveness with developers' perception in both scenarios (individually and inter-agreement). The research question RQ2 compared smelly methods pointed out by metric thresholds with reference lists of code smells proposed by each expert system developer. RQ3 compared this smelly method list with a single reference list of code smells, which contains the same classification proposed by both the evaluated system's maintainers.

A true positive (TP) occurs when the metric thresholds proposed by a technique point out a smelly method identified by the system developer. A false positive (FP) happens when the metric threshold proposed by a technique identifies a code smell that does not match developer's perception of the smelly method. A true negative (TN) occurs when the metric threshold does not point out a code smell, and developers do not perceive the

method as smelly. Finally, a false negative (FN) happens when the metric thresholds do not identify a code smell perceived by the system developer. The index i ranges over the entire dataset of values of each metric. Based on these assumptions, we computed:

$$Precision = \frac{\sum_i \mathrm{TP}_i}{\sum_i (\mathrm{TP}_i + \mathrm{FP}_i)} \tag{7.1}$$

$$MCC = \frac{\sum_i (\mathrm{TP}_i * \mathrm{TN}_i - \mathrm{FP}_i * \mathrm{FN}_i)}{\sum_i \sqrt{(\mathrm{TP}_i + \mathrm{FP}_i)(\mathrm{TP}_i + \mathrm{FN}_i) + (\mathrm{TN}_i + \mathrm{FP}_i) + (\mathrm{TN}_i + \mathrm{FN}_i)}} \tag{7.2}$$

We intend to indicate the technique with high precision that implies a low number of false positives. Previous studies have reported many false positives as factor that harming to apply automated static analysis tools (ASATs) based on metrics (OLBRICH; CRUZES; SJØBERG, 2010; KHOMH et al., 2011; SJOBERG et al., 2013; YAMASHITA, 2013; PALOMBA et al., 2014; HOZANO et al., 2018). We did not measure recall because of the high number of pointed out code anomalies in the evaluated systems. We proposed a systematic method to select the methods that developers would evaluate due to the impossibility of evaluating all the methods pointed out by the thresholds derived from techniques. Therefore, we focused on the precision. Each developer evaluated between 63 and 117 methods in the final study.

To answer RQ4, we carried out a qualitative data analysis over the records of video and audio based on the procedures of Grounded Theory (GT) suggested by Corbin and Strauss(CORBIN; STRAUSS, 1990). We used open coding (1st phase) and axial coding (2nd phase) from the GT method. In open coding, we compare events, actions, interactions, and so forth against others for similarities and differences. They are also conceptually labeled, and we group similar ones to form categories and their subcategories. In axial coding, we related categories to their subcategories, and these relationships are tested against data. Also, further development of categories takes place, and one continues to look for indications of them. We do not carry out the selective coding (3rd phase) in which all categories are unified around a central "core" category, and categories that need further explication are filled-in with descriptive detail. We would reach this core category with the circularity between the collection and analysis stages until theoretical saturation. We postpone this selective coding phase, and therefore, we do not claim that we applied the GT method, only some specific procedures.

7.1.5 Pilot Study

We conducted a pilot study of our proposed study protocol with two developers that evaluated code smells in different systems (LIMA; DÓSEA; SANT'ANNA, 2016). The first system uses the JSF framework, and the other one uses the Struts framework. Each developer evaluated only one system. The differences between the pilot study and the final study was the number of assessed methods by each developer and the number of developers that evaluated each system. In the pilot study, developers evaluated 10% of the smelly methods in each design role. However, we observed developers' fatigue in the

Table 7.5: Metric Thresholds Derived by Techniques to Struts and JSF based Systems

	S3/S4					S5/S6				
Techniques	**Design Role**	**LOC**	**CC**	**EC**	**NPM**	**Design Role**	**LOC**	**CC**	**EC**	**NPM**
Alves (2010)	*Undefined*	61	11	12	3	*Undefined*	61	11	12	3
Vale (2015)	*Undefined*	17	3	6	2	*Undefined*	17	3	6	2
Aniche (2016)	*Undefined*	66	12	12	3	*Undefined*	66	12	12	3
	Controller	27	4	8	0	*Controller*	27	4	8	0
	Persistence	47	12	16	3	*Persistence*	47	12	16	3
	Service	36	7	15	4	*Service*	36	7	15	4
	Entity	8	1	2	0	*Entity*	8	1	2	0
Dósea (2016)	*Undefined*	55	15	16	5	*Undefined*	167	27	22	4
Dósea (2018)	*Undefined*	55	15	16	5	*Undefined*	167	27	22	4
	Authorizer	22	9	8	3	*DefaultValidator*	53	7	15	4
	Persistence	79	15	8	6	*Persistence*	271	28	15	6
	Validator	49	21	7	3	*Exception*	53	7	15	3
	View	19	4	7	3	*View*	53	10	15	4
	Entity	50	16	13	3	*Entity*	53	7	15	4
	HttpServlet	79	14	21	6	*HttpServlet*	99	7	29	4
	DelegateCrud	55	10	15	4	*Action*	151	29	30	4
	Controller	135	25	25	12	*AbstractBO*	61	19	15	4
	Service	32	9	13	7	*ValidatorForm*	76	11	15	4
	JpaCrud	19	4	7	5	*Authenticator*	53	7	15	4

pilot study that could negatively influence the last evaluations. We limited the assessment to five methods for each design role in the final study to mitigate this problem. The size of the systems considered in the final study also influenced this limitation. If we consider evaluating 10% of the smelly methods in each design role, we will have many methods for each developers' evaluation. Due to this small difference in the protocol, we decided not to consider the pilot study results when analyzing the results of research questions.

7.2 RESULTS AND DISCUSSION

In this section, we report and discuss our main findings guided by each research question. The interviews lasted from 90 to 120 minutes, and we conducted them between August 2019 and September 2019 through either face-to-face meetings.

Table 7.5 shows the metric threshold values derived from five benchmark-based techniques detailed in Section 7.1. Metric thresholds derived from Alves et al. (ALVES; YPMA; VISSER, 2010), Vale & Figueiredo (VALE; FIGUEIREDO, 2015), and Aniche et al. (ANICHE et al., 2016) techniques use the same benchmark since these techniques do not explicitly recommend considering design decisions to composing the benchmark. We create two distinct benchmarks, one to each evaluated system, to derive metric thresholds proposed by [T1] and [T2] techniques discussed in Chapter 6. Additionally, Aniche et al. and [T2] techniques propose specific metric threshold values according to the architectural roles and design roles played by each system class, respectively.

In the following, we discussed the main results that support the answers for the research questions defined in Section 7.1.

RQ1: *What is the level of agreement between developers' perception of code smells*

Table 7.6: Level Agreement between developers of the same System

Systems	LOC	CC	EC	NOP
S3	0,49	0,72	0,21	0,28
S4	0,87	0,61	0,39	0,78
S5	1,00	0,83	1,00	0,76
S6	0,89	0,76	0,78	0,28

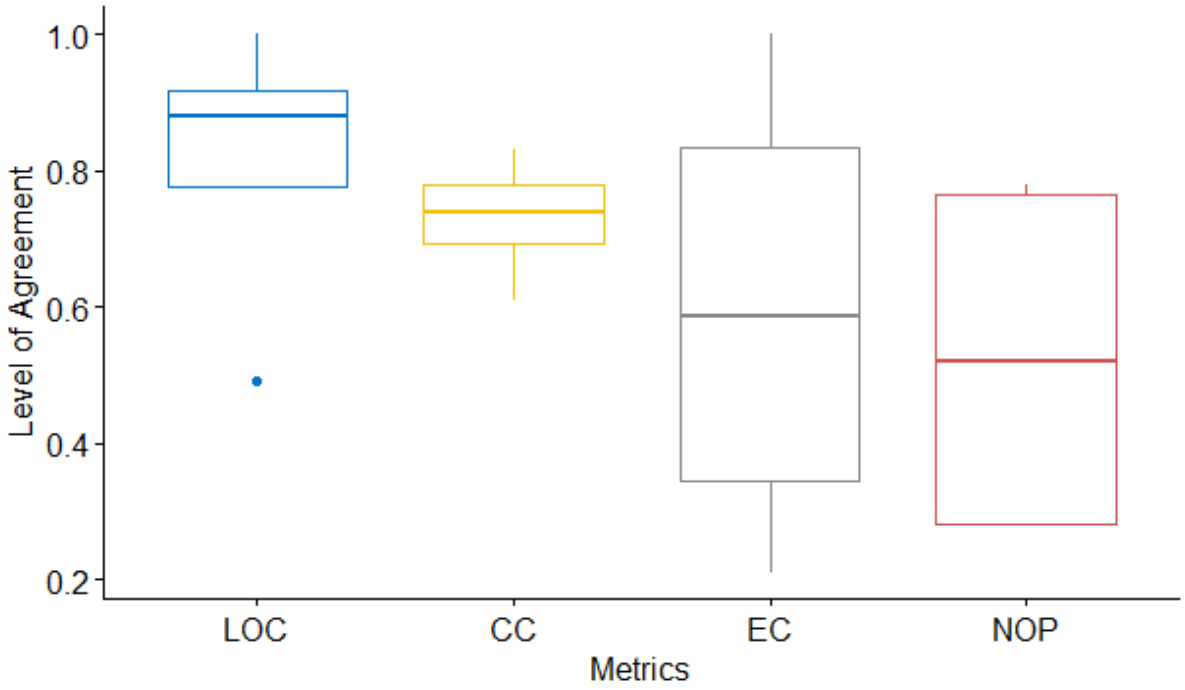

Figure 7.2: Level Agreement by Code Smell

identified in software systems maintained by them?

We aim to evaluate developers' degree of agreement about pointed out code smells in methods of software systems maintained by both developers. To address RQ1, we computed the inter-rater agreement among system developers' evaluation using *Fleiss' Kappa* measure (FLEISS; COHEN; EVERITT, 1969). We also use the categories proposed by Landis and Koch (LANDIS; KOCH, 1977) listed in Table 7.4.

Each pair of developers assessed 63, 83, 91, and 117 methods, respectively. Table 7.6 shows the strength of agreement using Kappa statistics about code smells identified by pair of developers in the same system (eight developers and four systems). Figure 7.2 illustrates the distribution of values of level of agreement between developers for each metric that point out a single code smell.

Regarding long methods pointed out using LOC metric thresholds, the strength of agreement using Kappa statistics shows moderate (one system) to substantial (three systems). Considering high complexity methods, the strength of agreement was substantial

(three systems) to almost perfect (one system). Regarding high efferent coupling, we found fair agreement in two systems, substantial in one system and almost perfect in one system. Finally, we found agreement fair (two systems) and substantial (two systems) evaluating developers agreement in methods with long parameter list.

These results shows substantial and almost perfect strength of agreement between developers in many evaluations of (11 out of 16). The proposed interview process showed that the four evaluated code smells are simple and easily detectable by participants. However, our results contradict another large-scale evaluation (HOZANO et al., 2018). They found a low agreement among the developers' evaluations in all investigated code smells, including Long Method and Long Parameter List, and suggest that experience and background factors cannot make developers agree on detecting code smells. They also indicate that the developers' heuristic to detect the smells is the most predominant factor to some improvement on the agreement. We believe that the predominant factor in obtaining a high level of agreement between developers is familiarity with the evaluated code snippets and corresponding design decisions. Understanding the design decisions that influenced the implementation of the code is essential to classify it as smelly.

In summary, the results allow us to answer RQ1 as follows:

> *Consciousness of the design decisions in the evaluated source code increases developers' agreement to recognize smelly methods in software systems maintained by them. Therefore, we suggest that assessing the accuracy of techniques that automatically point out code smells needs to be carried out by developers familiar with design decisions of the evaluated source code.*

RQ2: *Which technique proposes metric thresholds that best reflect the individual developers' perception about code smells in software systems maintained by them?*

This research question aims to point out the technique that derived metric thresholds more similar to the individual perceptions of most developers. We considered the same weight to each developer's perception about a smelly method that he will refactor. We aggregated the results achieved for each metric, as discussed in Section 7.1, because aggregate metrics are more robust than the mean.

Regarding the LOC metric, Table 7.7 shows the aggregate results of true positives (TP), false negatives (FN), false positives (FP), true negatives (TN), precision, and MCC for individual developer's perception of long methods. We observe that [T1] had higher precision and MCC metrics, followed by [T2] and Alves techniques. Although other techniques achieve a more significant number of true positives, they also significantly increase false negatives reducing precision. For instance, Vale's technique has higher true positives; however, low precision occurs because it derives very low metric thresholds that increase false positives.

Figure 7.7 illustrates differences between the distribution of MCC and precision metrics according to individual software developers' perception. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We also observe a small advantage of precision and MCC metrics of the [T1] technique. MCC to the [T1] technique ranges from 0.24 to 0.73.

To test the hypothesis $H0_1$ about MCC, we use the Shapiro-Wilk test of normal-

Table 7.7: Aggregate Results for Individual Developer's Perception of Long Methods

	TP	FN	FP	TN	Precision	MCC
Alves et al.	31	14	25	177	0,55	0,52
Vale and Figueiredo	45	0	198	4	0,19	0,06
Aniche et al.	31	14	41	161	0,43	0,41
[T1]	28	17	16	186	**0,64**	**0,55**
[T2]	29	16	20	182	**0,59**	**0,53**

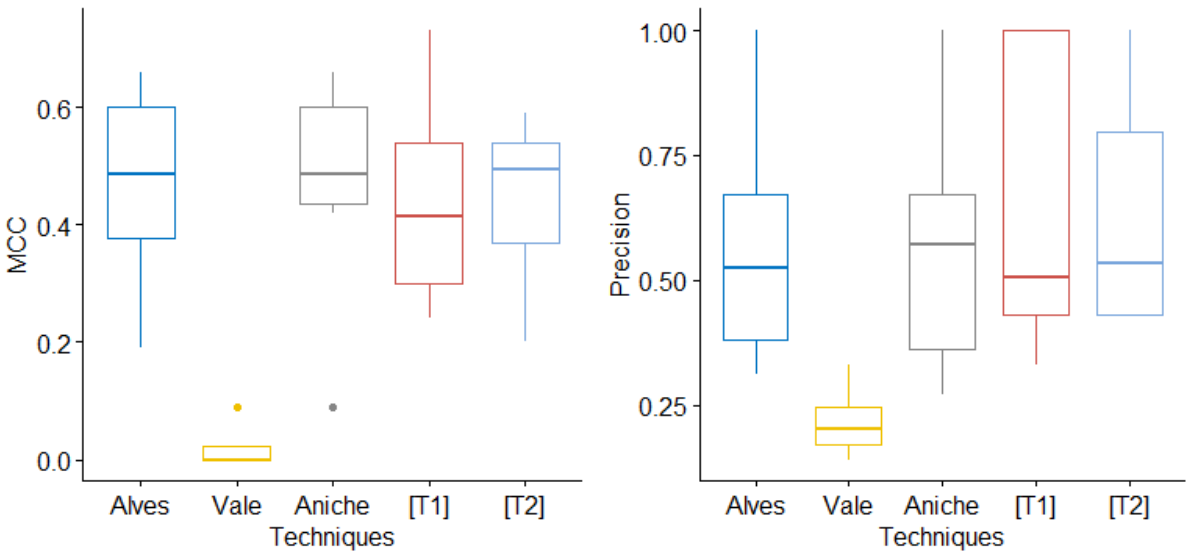

Figure 7.3: Distributions of MCC and Precision Metrics according to Individual' Developers Perception of Long Methods

Table 7.8: Aggregate Results for Individual Developer's Perception of Complex Methods

	TP	**FN**	**FP**	**TN**	**Precision**	**MCC**
Alves et al.	45	32	43	148	**0,51**	**0,35**
Vale and Figueiredo	78	0	188	2	0,29	0,06
Aniche et al.	50	28	48	142	**0,51**	**0,37**
[T1]	30	47	28	162	**0,52**	0,27
[T2]	33	45	37	153	0,47	0,24

ity, and we can not assume the normality for distribution of the MCC values of Vale's technique. So, we perform the Kruskal-Wallis test, and we identify there are significant differences between the techniques ($p - value = 0.0007$) rejecting the null hypothesis ($H0_1$). This means that at least one technique derived metric thresholds that improve MCC on the detection of code smells compared to the others. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the MCC of Vale's technique is significantly lower than other techniques. We obtained a large effect size applying Cliff's δ. These results mean that Vale's technique derived metric thresholds for LOC metric that do not improve MCC metric to detect long methods according to individual developers' perception of code smells. We do not identify statically significant differences between other techniques. To evaluate if some technique derived metric thresholds that improve the precision to detect long methods, we test the hypothesis $H0_2$. We can not assume the normality of the precision values distribution of [T1] and [T2] techniques. The Kruskal-Wallis test rejected ($p-value = 0.0008$) the null hypothesis ($H0_2$). Pairwise comparisons using the Mann-Whitney-Wilcoxon showed that the precision of Vale's technique also is significantly lower than other techniques. We also obtained a large effect size applying Cliff's δ. These results mean that Vale's technique derived metric thresholds for LOC metric that do not improve the precision to detect long methods. These results also confirm our visual analysis of Figure 7.3, which shows the distribution of MCC and precision values of Vale's technique much lower than the other techniques.

Regarding the CC metric, Table 7.8 shows the aggregate results for individual developer's perception of high complex methods. We observe that [T1] had higher precision, but Alves and Aniche's techniques obtained higher MCC metric values than [T1] technique. Again, Vale's technique obtained a higher number of true positives and false positives, implying the worst precision.

Figure 7.4 illustrates differences between the distribution of MCC and precision metrics according to individual software developers' perception of complex methods. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We also observe a small advantage of precision and MCC metrics to the [T1] technique. MCC of the [T1] technique ranges from 0.15 to 0.53. The distribution of precision values of Aniche's technique shows a small visual advantage over other techniques.

To test the hypothesis $H0_1$ to MCC, we use the Shapiro-Wilk test of normality, and

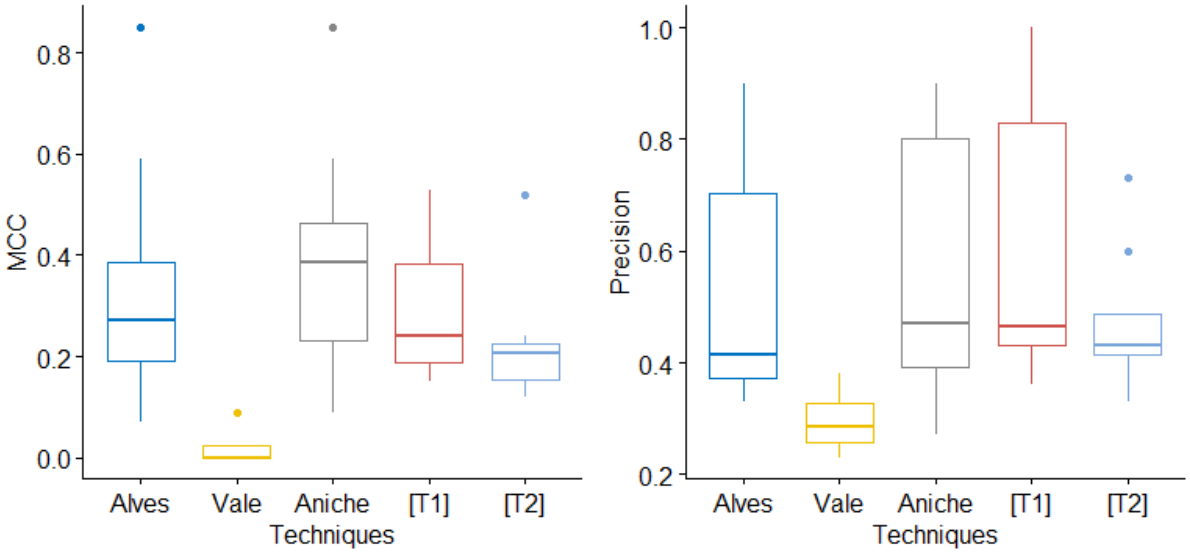

Figure 7.4: Distributions of MCC and Precision Metrics according to Individual' Developers Perception of Complex Methods

we can not assume the normality for distribution of the MCC values of Vale and [T2] techniques. So, we perform the Kruskal-Wallis test, and we identify there are significant differences of MCC between the techniques ($p - value = 0.0057$) rejecting the null hypothesis ($H0_1$). This means that at least one technique proposes metric thresholds that improve the MCC to detect code smells compared to individual developers' perception of code smells. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the MCC of Vale's technique is significantly lower than other techniques. We obtained a large effect size applying Cliff's δ. These results mean that the Vale's technique proposes metric thresholds for CC metric that do not improve MCC metric to detect high complex methods. We do not identify statically significant differences between other techniques.

To evaluate if some technique proposes metric thresholds that improve the precision to detect complex methods, we test the hypothesis $H0_2$. We can not assume the normality of the precision values distribution of [T1] technique. The Kruskal-Wallis test rejected ($p - value = 0.003$) the null hypothesis ($H0_2$). Pairwise comparisons using the Mann-Whitney-Wilcoxon showed that the precision of Vale's technique also is significantly lower than only [T1] and [T2] techniques. We also obtained a large effect size applying Cliff's δ. These results mean that Vale's technique proposes metric thresholds for CC metrics that do not improve the precision metric to detect complex methods than [T1] and [T2] techniques. These results also confirm the visual analysis of Figure 7.4, which shows the distribution of MCC values of Vale's technique much lower than the other techniques. However, we only found the distribution of precision values of Vale's technique statistically significantly lower than [T1] and [T2] techniques. We do not found differences between other evaluated techniques.

Regarding the EC metric, Table 7.9 shows the aggregate results of true positives (TP),

Table 7.9: Aggregate Results for Individual Developer's Perception of High Efferent Coupling

	TP	FN	FP	TN	Precision	MCC
Alves et al.	22	1	53	154	0,29	**0,44**
Vale and Figueiredo	23	0	200	7	0,10	0,06
Aniche et al.	22	1	62	145	0,26	0,41
[T1]	19	4	14	193	**0,58**	**0,65**
[T2]	16	7	26	181	**0,38**	**0,44**

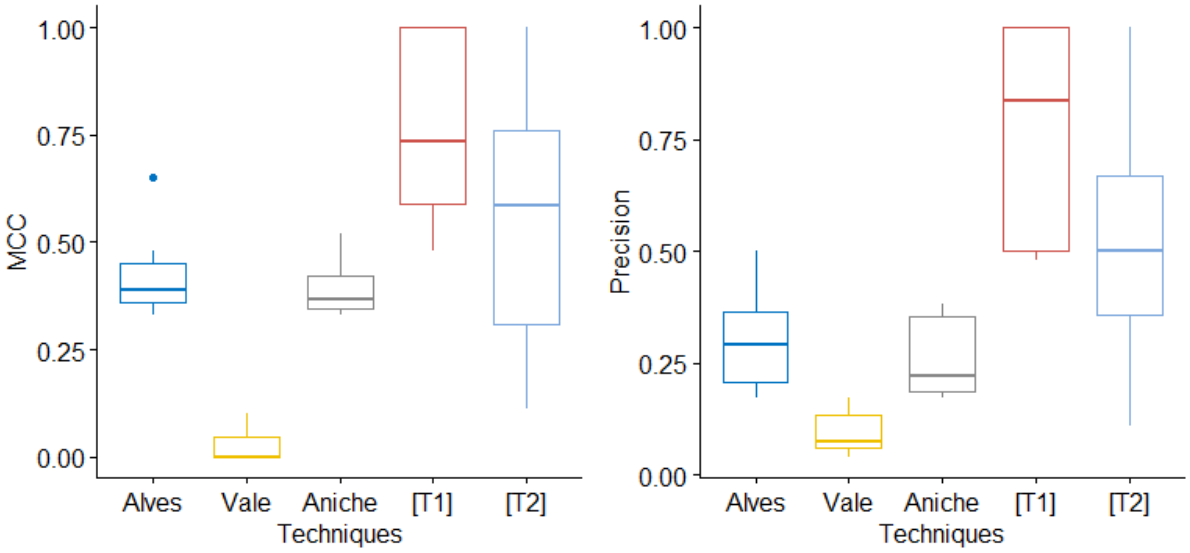

Figure 7.5: Distributions of MCC and Precision Metrics according to Individual' Developers Perception of High Efferent Coupling

false negatives (FN), false positives (FP), true negatives (TN), precision, and MCC for individual developer's perception of high efferent coupling of methods. We observe that [T1] and [T2] had higher precision and MCC metrics. Vale's technique obtained only one true positive more than the other techniques, however it obtained a considerable increase in the number of false positives. For this reason Vale's technique obtained low values of MCC and precision.

Figure 7.5 illustrates differences between the distribution of MCC and precision metrics according to individual software developers' perception of high efferent coupling. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We also observe the advantage of precision and MCC metrics to [T1] and [T2] techniques. For instance, MCC and precision to the [T1] technique range from 0.48 to 1.00.

To test the hypothesis $H0_1$ to MCC, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distribution of the MCC values of Vale's technique. So, we perform the Kruskal-Wallis test, and we identify there are significant differences

of MCC between the techniques ($p-value = 0.0002$) rejecting the null hypothesis ($H0_1$). This means that at least one technique proposes metric thresholds that improve the MCC to detect code smells compared to individual developers' perception of code smells. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the MCC of Vale's technique is significantly lower than other techniques. We also identified that the [T1] technique obtained better MCC values in a statistically significant way than Vale, Aniche, and Alves' techniques. We did not find any statistically significant differences from [T1] compared to the [T2] technique. We obtained a large effect size by applying Cliff's δ in all these comparisons. These results mean that the [T1] technique proposes metric thresholds for EC metric that improve MCC metric to detect high efferent coupling in methods according to individual developers' perception of code smells.

To evaluate if some technique proposes metric thresholds that improve the precision to detect complex methods, we test the hypothesis $H0_2$. We can not assume the normality of the precision values distribution of [T1] and Aniche techniques. The Kruskal-Wallis test rejected ($p - value = 0.0002$) the null hypothesis ($H0_2$). Pairwise comparisons using the Mann-Whitney-Wilcoxon showed that the precision of Vale's technique also is significantly lower than other techniques. We also obtained a statistically significant improvement in the precision values of the technique [T1] compared to the techniques of Vale, Aniche, and Alves. Only the [T2] technique did not obtain significant differences in precision compared to [T1]. We also obtained a large effect size applying Cliff's δ for these comparisons. These results mean that Vale's technique proposes metric thresholds for EC metrics that did not improve the precision metric to detect methods with high efferent coupling compared to other techniques. According to individual developers' perception of code smells, the [T1] technique also proposes metric thresholds for EC that improve precision metric to detect methods with high efferent coupling. We do not found differences between precision obtained by [T1] and [T2] techniques. These results also confirm the visual perception of Figure 7.5, which shows the distribution of MCC and precision values of the [T1] and [T2] techniques with values higher than other techniques.

Regarding the NOP metric, Table 7.10 shows the aggregate results of true positives (TP), false negatives (FN), false positives (FP), true negatives (TN), precision, and MCC for individual developer's perception of the long parameter list. We observe that [T1] and [T2] had higher precision and MCC metrics. Despite other techniques obtained a higher number of true positives, they also obtained a considerable increase in false positives and lower precision than [T1] and [T2] techniques.

Figure 7.6 illustrates differences between the distribution of MCC and precision metrics according to individual software developers' perception of long list of parameters. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We also observe the advantage of precision and MCC metrics to [T1] and [T2] techniques. For instance, MCC and precision to the [T1] technique range from 0.35 to 0.88.

To test the hypothesis $H0_1$ to MCC, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distribution of the MCC values of Vale's technique. So, we perform the Kruskal-Wallis test, and we identify there are significant differences

Table 7.10: Aggregate Results for Individual Developer's Perception of Long Parameter List

	TP	FN	FP	TN	Precision	MCC
Alves et al.	31	0	59	103	0,34	**0,47**
Vale and Figueiredo	31	0	137	24	0,18	0,17
Aniche et al.	31	0	81	81	0,28	0,37
[T1]	25	6	21	141	**0,54**	**0,58**
[T2]	20	11	20	142	**0,50**	**0,47**

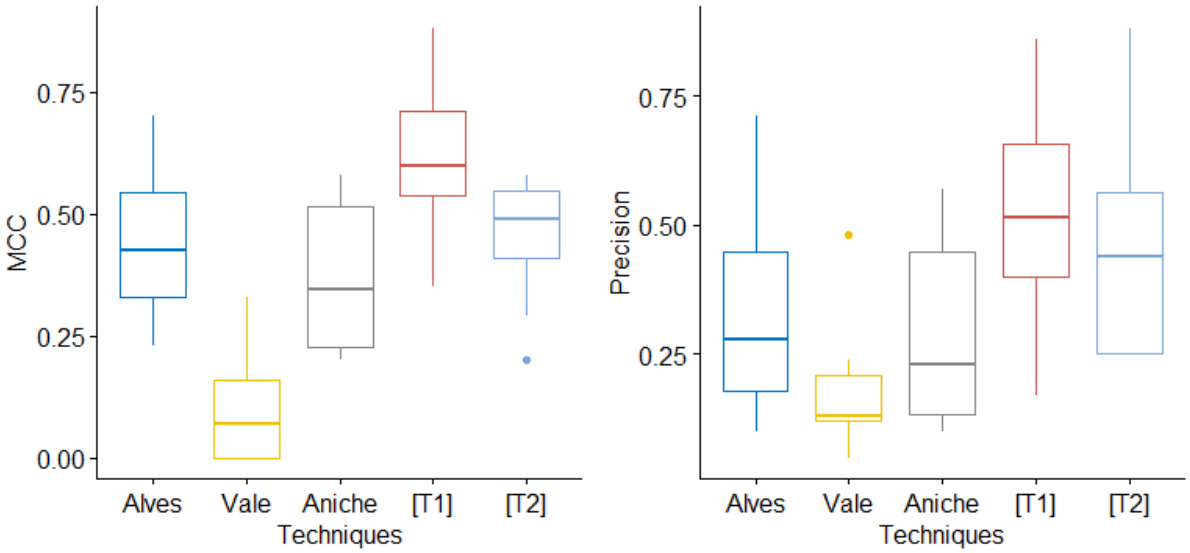

Figure 7.6: Distributions of MCC and Precision Metrics according to Individual' Developers Perception of Long Parameter List

of MCC between the techniques ($p-value = 0.0001$) rejecting the null hypothesis ($H0_1$). This means that at least one technique proposes metric thresholds that improve the MCC to detect long parameter list compared to individual developers' perception of code smells. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the MCC of Vale's technique is significantly lower than other techniques. We also identified that the [T1] technique obtained better MCC values than Aniche's technique. We did not find any statistically significant differences among [T1], [T2] and Alves techniques. We obtained a large effect size by applying Cliff's δ in all these comparisons. These results mean that Vale's technique proposes metric thresholds for NOP metric that do not improve the precision to detect long parameter list than other techniques. Also, [T1] technique proposes metric thresholds that improve MCC metric compared to Aniche's technique.

To evaluate if some technique proposes metric thresholds that improve the precision to detect long parameter list, we test the hypothesis $H0_2$. We can not assume the normality of the precision values distribution of Vale's technique. The Kruskal-Wallis test rejected ($p - value = 0.01$) the null hypothesis ($H0_2$). Pairwise comparisons using the Mann-Whitney-Wilcoxon showed that only [T1] and [T2] techniques obtained a statistically significant improvement in the precision values compared to Vale's technique. We obtained a large effect size applying Cliff's δ for these comparisons. According to individual developers' perception of code smells, the [T1] and [T2] technique proposes metric thresholds for NOP metric that improve precision to detect methods with long parameter list compared to Vale's technique. We do not found differences between precision obtained by [T1] and [T2] with other techniques. These results also confirm the visual perception of Figure 7.6, which shows the distribution of MCC and precision values of the [T1] and [T2] techniques with values higher than other techniques.

In summary, the differences we observed on distributions of precision and MCC values from five evaluated techniques, allow us to answer RQ2 as follows:

No technique proposed metric thresholds that demonstrated statistically significant improvements in precision and MCC values for the four evaluated metrics concerning individual developers' perception. However, the technique [T1] improved MCC compared to Vale, Aniche, and Alves's techniques in at least one evaluated metric. [T1] also improved precision compared to the Vale, Aniche, and Alves' techniques in at least one of the evaluated metrics. [T1] did not show statistically significant differences in MCC and precision compared to [T2] technique.

RQ3: *Which technique proposes threshold values that best reflect two developers' joint perception of code smells in software systems maintained by them?*

We consider two developers have similar perceptions of a code smell when they have an equal opinion about code smells evaluating the same method.

Regarding the LOC metric, Table 7.11 shows the aggregate results of true positives (TP), false negatives (FN), false positives (FP), true negatives (TN), precision, and MCC for individual developer's perception of long methods. We observe that [T1] had higher precision and MCC metrics, followed by [T2] and Alves techniques. Although other techniques achieve a more significant number of true positives, they also significantly

Table 7.11: Aggregate Results for Two Developer's Joint Perception of Long Methods

	TP	FN	FP	TN	Precision	MCC
Alves et al.	14	4	11	85	0,56	**0,58**
Vale and Figueiredo	18	0	94	2	0,16	0,06
Aniche et al.	14	4	19	77	0,42	0,47
[T1]	12	6	6	90	**0,67**	**0,60**
[T2]	12	6	7	89	**0,63**	**0,58**

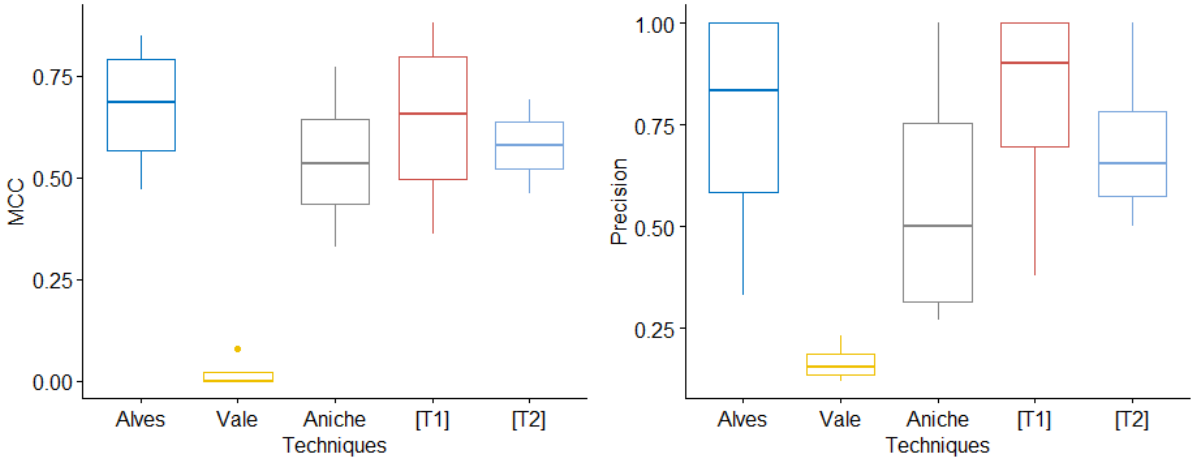

Figure 7.7: Distributions of MCC and Precision Metrics according to Joint Developers Perception of Long Methods

increase false negatives reducing precision. For instance, Vale's technique has higher true positives; however, low precision occurs because it derives very low metric thresholds that increase false positives. Refactoring also occur in methods not point out as smelly, but methods perceived as smelly seem more prone to refactoring.

Figure 7.7 illustrates differences between the distribution of MCC and precision metrics according to joint software developers' perception. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We observe a small advantage of precision and MCC metrics to the [T1] and Alves techniques. MCC to the [T1] technique ranges from 0.36 to 0.88.

To test the hypothesis $H0_3$ to MCC, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distribution of the MCC values of Vale's technique. So, we perform the Kruskal-Wallis test, and we identify there are significant differences between the techniques ($p - value = 0.038$) rejecting the null hypothesis ($H0_3$). This means that at least one technique proposes metric thresholds that improve the MCC to detect code smells compared to joint developers' perception of code smells. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the MCC of Vale's technique is significantly lower than other techniques. We obtained a large effect size applying Cliff's

Table 7.12: Aggregate Results for Two Developer's Joint Perception of Complex Methods

	TP	**FN**	**FP**	**TN**	**Precision**	**MCC**
Alves et al.	19	12	18	69	**0,51**	**0,39**
Vale and Figueiredo	31	0	86	1	0,26	0,06
Aniche et al.	20	11	19	68	**0,51**	**0,40**
[T1]	13	18	12	75	**0,52**	0,30
[T2]	13	18	15	72	0,46	0,26

δ. These results mean that Vale's technique proposes metric thresholds for LOC metric that do not improve MCC metric to detect long methods according to joint developers' perception of code smells. We do not identify statically significant differences between other techniques. To evaluate if some technique proposes metric thresholds that improve the precision to detect long methods, we test the hypothesis $H0_4$. We can assume the normality of the distribution of precision values and homogeneity of variance using Levene's Test ($p - value = 0.255$). As the ANOVA was significant, we compute Tukey HSD for performing multiple pairwise-comparison between the means of groups. The pairwise comparisons showed that the precision of Vale's technique also is significantly lower than Alves e [T1] techiniques. These results also confirm our visual analysis of Figure 7.7, which shows the distribution of MCC and precision values of Vale's technique much lower than the other techniques.

Regarding the CC metric, Table 7.12 shows the aggregate results for the joint developer's perception of high complex methods. We observe that [T1] had higher precision, but Alves and Aniche's techniques obtained higher MCC metric values than [T1] technique. Again, Vale's technique obtained a higher number of true positives and false positives, implying the worst precision.

Figure 7.8 illustrates differences between the distribution of MCC and precision metrics according to joint software developers' perception of complex methods. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We observe a small advantage of precision and MCC metrics to the Aniche and [T1] techniques. For instance, MCC to the [T1] technique ranges from 0.17 to 0.51. The distribution of precision values of [T1] technique shows a small advantage over other techniques.

To test the hypothesis $H0_3$ to MCC, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distribution of the MCC values of Aniche and Vale' techniques. So, we perform the Kruskal-Wallis test, and we identify there are significant differences of MCC between the techniques ($p - value = 0.03$) rejecting the null hypothesis ($H0_3$). This means that at least one technique proposes metric thresholds that improve the MCC to detect code smells compared to joint developers' perception of complex methods. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) do not identify statically significant differences between distribution of MCC values of five techniques. To evaluate if some technique proposes metric thresholds that improve the precision to

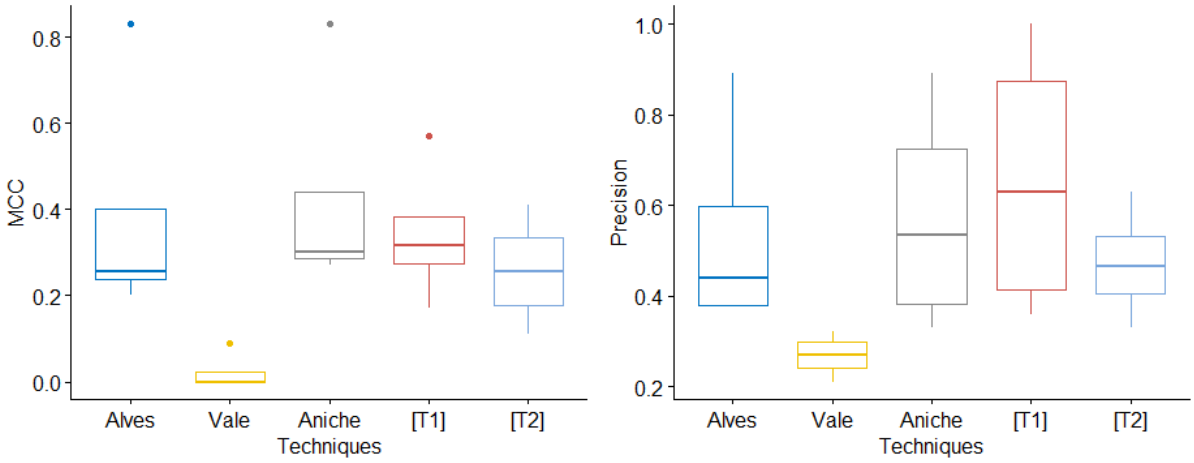

Figure 7.8: Distributions of MCC and Precision Metrics according to Joint' Developers Perception of Complex Methods

Table 7.13: Aggregate Results for Two Developer's Joint Perception of High Efferent Coupling

	TP	**FN**	**FP**	**TN**	**Precision**	**MCC**
Alves et al.	6	0	22	76	0,21	0,41
Vale and Figueiredo	6	0	95	3	0,06	0,04
Aniche et al.	6	0	26	72	0,19	0,37
[T1]	6	0	4	94	**0,60**	**0,76**
[T2]	5	1	10	88	**0,33**	**0,49**

detect complex methods, we test the hypothesis $H0_4$. We can assume the normality of the distributions of precision values and homogeneity of variance using Levene's Test ($p-value = 0.077$). As the ANOVA was significant, we compute Tukey HSD for performing multiple pairwise-comparison between the means of groups. The pairwise comparisons do not identify statically significant differences between distribution of precision values of five techniques.

Regarding the EC metric, Table 7.13 shows the aggregate results of true positives (TP), false negatives (FN), false positives (FP), true negatives (TN), precision, and MCC for the joint developer's perception of high efferent coupling of methods. We observe that [T1] and [T2] had higher precision and MCC metrics. Vale's technique obtained only one true positive more than the other techniques, however it obtained a considerable increase in the number of false positives. For this reason Vale's technique obtained lower values of MCC and precision.

Figure 7.9 illustrates differences between the distribution of MCC and precision metrics according to the joint software developers' perception of high efferent coupling. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We also observe the advantage of precision and MCC metrics

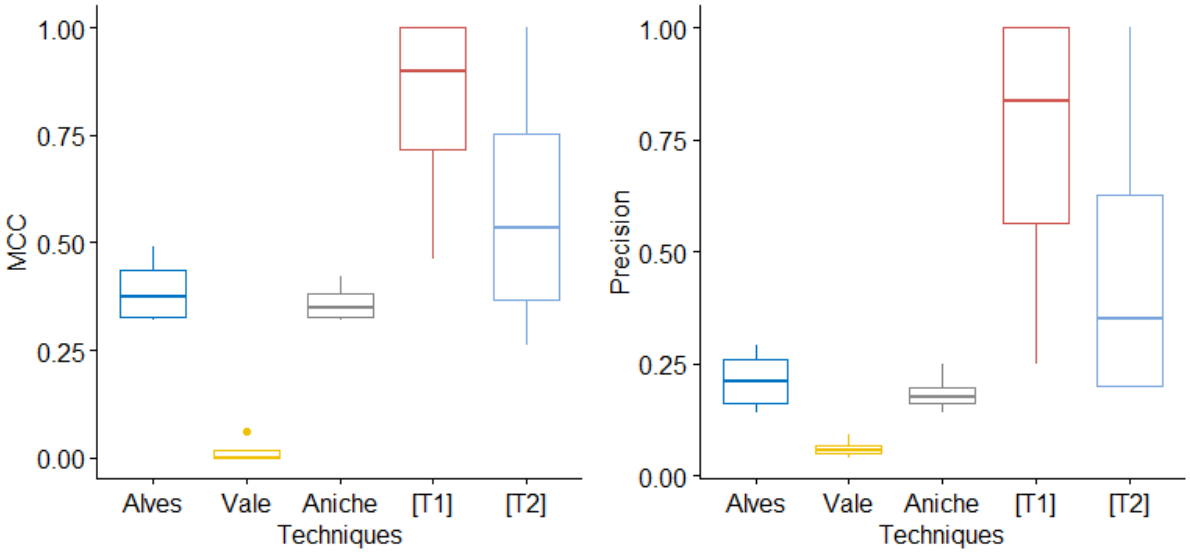

Figure 7.9: Distributions of MCC and Precision Metrics according to Joint' Developers Perception of High Efferent Coupling

to [T1] and [T2] techniques. For instance, MCC and precision to the [T1] technique range from 0.46 to 1.00.

To test the hypothesis $H0_3$ to MCC, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distribution of the MCC values of Vale's technique. So, we perform the Kruskal-Wallis test, and we identify there are significant differences of MCC between the techniques ($p - value = 0.009$) rejecting the null hypothesis ($H0_3$). This means that at least one technique proposes metric thresholds that improve the MCC to detect code smells compared to joint developers' perception of code smells. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) did not find any statistically significant differences from five technique. To evaluate if some technique proposes metric thresholds that improve the precision to detect methods with high efferent coupling, we test the hypothesis $H0_4$. We can assume the normality of the distributions of precision values and homogeneity of variance using Levene's Test ($p - value = 0.05$). As the ANOVA was significant, we compute Tukey HSD for performing multiple pairwise-comparison between the means of groups. The pairwise comparisons showed that the precision of [T1] technique was significantly higher than Alves, Aniche and Vale' techniques. These results also confirm our visual analysis of Figure 7.9, which shows the distribution of MCC and precision values of [T1] technique higher than the other techniques. We do not identify statically significant differences between distribution of precision values between [T1] and [T2] techniques.

Regarding the NOP metric, Table 7.14 shows the aggregate results of true positives (TP), false negatives (FN), false positives (FP), true negatives (TN), precision, and MCC for joint developer's perception of the long parameter list. We observe that [T1] and [T2] had higher precision and MCC metrics. Despite other techniques obtained a higher

Table 7.14: Aggregate Results for Two Developer's Joint Perception of Long Parameter List

	TP	FN	FP	TN	Precision	MCC
Alves et al.	8	0	22	51	0,27	0,43
Vale and Figueiredo	8	0	61	12	0,12	0,14
Aniche et al.	8	0	33	40	0,20	0,33
[T1]	8	0	6	67	**0,57**	**0,72**
[T2]	6	2	6	67	**0,50**	**0,56**

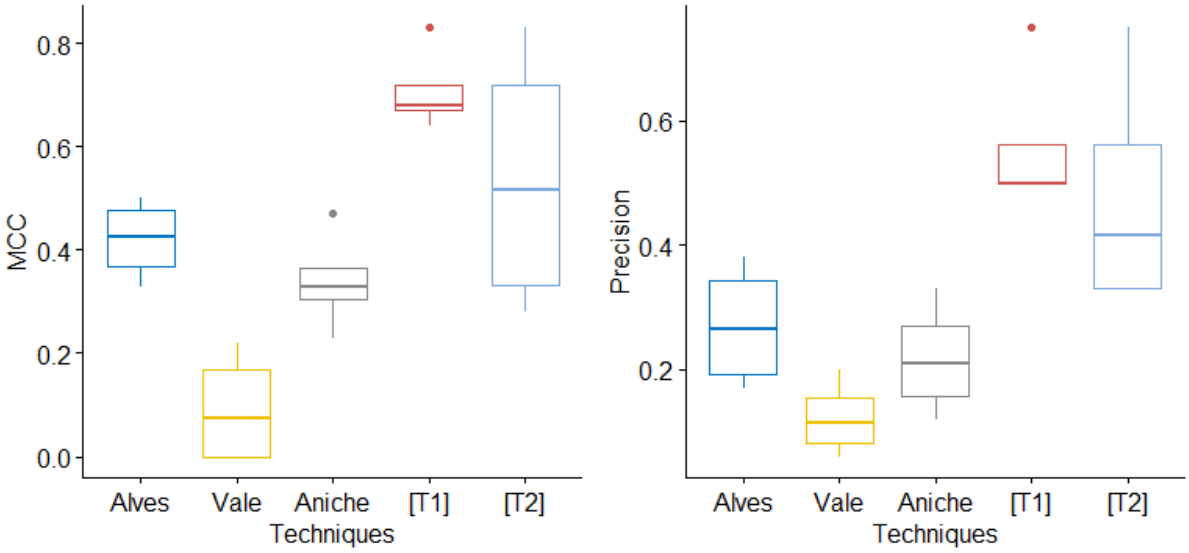

Figure 7.10: Distributions of MCC and Precision Metrics according to Joint' Developers Perception of Long Parameter List

number of true positives, they also obtained a considerable increase in false positives and lower precision than [T1] and [T2] techniques.

Figure 7.10 illustrates differences between the distribution of MCC and precision metrics according to the joint software developers' perception of long parameter list. It shows two graphs with five box plots each. Each graph is about one of the five techniques to derive metric thresholds. We also observe the advantage of precision and MCC metrics to [T1] and [T2] techniques. For instance, MCC and precision to the [T1] technique range from 0.64 to 0.83.

To test the hypothesis $H0_3$ to MCC, we use the Shapiro-Wilk test of normality, and we can assume the normality for distribution of the MCC values of Vale's technique and homogeneity of variance. As the ANOVA was significant, we compute Tukey HSD for performing multiple pairwise-comparison between the means of groups. The pairwise comparisons showed that the MCC of [T1] technique was significantly higher than Aniche and Vale techniques. We also found that [T2] and Alves techniques were significantly higher than Vale's technique. To evaluate if some technique proposes metric thresholds

that improve the precision to detect long list of parameters, we test the hypothesis $H0_4$. We can not assume the normality of the precision values distribution [T1] technique. The Kruskal-Wallis test rejected ($p - value = 0.0061$) the null hypothesis ($H0_4$). Pairwise comparisons using the Mann-Whitney-Wilcoxon did not show that statistically significant improvement in the precision values between five techniques.

In summary, the differences we observed on distributions of precision and MCC values from five evaluated techniques, allow us to answer RQ3 as follows:

No technique proposed metric thresholds that demonstrated statistically significant improvements in precision and MCC values for the four evaluated metrics concerning joint developers' perception. However, the technique [T1] improved MCC compared to Vale and Aniche's techniques in at least one evaluated metric. [T1] also improved precision compared to Vale, Aniche, and Alves' techniques in at least one evaluated metric. Similar to individual developers' perception, [T1] also did not show statistically significant differences in MCC and precision compared to the [T2] technique.

RQ4: *Do developers familiar with the source code consider the class design role information to identify refactoring-prone code smells?*

This research question aimed to identify if developers use the class design role to explain their perception of a smelly method prone to refactoring. According to the procedures detailed in Section 7.1, we carry out a qualitative study. For each evaluated method, we ask, "What implementing features of the method influenced you to consider it as smelly?". We analyzed 700 responses from developers. Next, we ask developers' perceptions about the possibility that a code smell occurs in other methods playing the same design role. Finally, we carry out the second phase of interviews to review inconsistencies in responses.

We create codes for the developer's speeches (1st phase). After, these codes were related to each other through axial coding (2nd phase). For each transcript, the codes and identified memos showing the relationships in the categories were reviewed by the author of this thesis and a master student, and if necessary, changed upon agreement with both researchers. This section presents our main results, including direct quotes from respondents related to proposed categories.

We found 187 ($26,7\%$) responses that associated at least one code smell with the evaluated method and 513 ($73,3\%$) responses that did not identify any of the code smells considered. We identified the design role category in developers' responses to justify their assessments in $27,2\%$ (51 out of 187) of positive responses and $21,4\%$ (110 out of 513) of negative responses.

According to the developer and evaluated system, we also noticed a variation in the number of citations of the design role. The S5 system developers were the ones that most used the design roles in positive answers: 70.0% (14 out of 20) and 52.2% (12 out of 23). We also noticed a variation between the developers of the same system. For example, in the S6 system, the first developer mentioned at least one design role in 18.2% (4 out of 22) of his positive answers, already the second developer cited at least one design role in 36.4% (8 out of 22) of the positive answers. We identified the category design role in

responses of all developers. Everyone used the design role to justify at least one positive and negative response. These data suggest that the design role seems to influence the developers' perception of code smells regardless of the evaluated system and developer. We discuss below other findings obtained.

Did class design role influence developers' perception about long methods? Developers pointed out 84 out of 700 evaluated methods as long. They usually justified their motivation to classify methods as long with short answers. The most common reasons perceived by software developers match with the categories **high number of lines of code** (32 out of 84) and **the need to split the method into smaller ones to improve understanding** (37 out of 84). Many respondents mentioned *"The method does a lot. We should divide it into more methods."*

Respondents used the category **class design role** (21 out of 85) to explain their decision for classifying methods as long. A developer mentioned that *"I would split the method to handle specific rules and extract business rules from Actions"*. Some developers used the class design role as the main reason to justify a method as longer. For instance, a developer mentioned that *"Even though the method is in a Utility class, the method is very long and complex "*. Similarly, another developer states that *"for a method in an Entity class it is too long and complex"*. A developer used design role to explain his justification in detail *" It is expected long methods in Actions classes because Action receives input data, call Service classes or Business methods, and then return a result to the system. "*. These reports illustrate situations where design role (e.g., Action, Services, and Utility) influenced developers' perception of long methods.

Did class design role influence developers' perception about high complexity methods? Developers pointed out 129 out of 700 evaluated methods with high complexity. The most common reasons perceived by software developers match with the category **many conditional statements or loop** (64/129 occurrences) and **the need to split the method into smaller ones to improve understanding** (40/129 occurrences). Many respondents mentioned *"many conditionals and loop statements for a single method."*

We also identify developers using the category **class design role** (43 out of 129) to explain their perception about high complexity methods. A developer explain that the design role can influence the number of complex methods *"Although an Action class, which makes many accesses to other classes, this method overstates the number of deviations and dependencies."*. Some developers used the design role as the main reason to classify the method as high complexity. A developer claims that *"A TO (i.e., transfer object design pattern) cannot have many conditionals statements, these statements need to be placed in another class."*. Another developer mentions that *"It is a complex method, even more, because it is a method responsible for carrying out persistence"*. These reports also suggest that design role decisively influenced developers' perception to classify high complexity methods.

Did class design role influence the developers' perception about high efferent coupling methods? Developers pointed out 46 out of 700 evaluated methods with high efferent coupling. The most common reasons perceived by software developers match with the category **method assumes many responsibilities** (18/46 occurrences) and

method makes many accesses to other classes (15/46 occurrences). For instance, some developers mention that *"method has many dependencies."* other developers claim that *"method is very coupled to other classes."* Due to a large number of short responses, we found few developer's answers (4 out of 46) that matched with the category **class design role** to justify their perception of high efferent coupling. A developer justified that *"the method mixes responsibilities of architectural components and validates at the same time."*. Another developer mentioned that *"even though it is an Action, which makes many accesses to other classes, this method exaggerates both the conditional statements and dependent classes"*.

Did class design role influence the developers' perception about methods with a long list of parameters? Developers point out 51 out of 700 evaluated methods with a long parameter list. Most developers' answers are short, and the most common reasons perceived by software developers match with the category **many parameters** (42/51 occurrences).

Similar to the efferent coupling metric, due to the high number of short answers, we found few developer's answers (5 out of 51) matching with the category **class design role** to justify their perception of a high number of parameters. A developer mentioned that *"even though it is a class of service, it would be possible to reduce the number of parameters"*. Another developer mentions design role as a possible solution *"could create a TO to pass the parameters"*.

Did class design role influence the developers' perception to not assign a smell to a method? Developers did not point out any code smell to 515 out of 700 evaluated methods. Most responses (512/515 occurrences) match with the category **No problems**. Many developers mentioned *"everything is normal"*.

Interestingly, we also found several developers' responses (110/515 occurrences) using the class design role to justify their perception. Developers argue that some design roles require higher metric thresholds. Some developers state that *"Everything normal for a validator"* or *"Everything normal for a Service"* or *"Everything normal for a Repository class"*. Other developer says *"classes playing the Controller design role are responsible to handle exceptions"*. Developers also explain that some methods in classes playing the Entity design role require higher metric thresholds. Some developers say that *"Equal methods (from Entity classes) need to have all of these deviations"* or *"A hashcode (from Entity classes) method needs to be like this"*. Thus, these reports illustrate that developers also used design roles to justify their perception of non-smelly methods.

Did class design role influence the developers' perception to assign the same code smell to classes playing the same design role?

Developers answered a follow-up questionnaire in the third part of the interview. We ask "Do you think that methods associated with the same design role and similar metric values will have the same code smell? Please justify your answer and cite examples.". We applied this question after the developers evaluated all methods. We proposed this question because evaluating all system methods would be very difficult and error-prone. Thus, we wanted to have an idea in which degree the identification of code smells in a method could be extended to other methods of classes playing the same design role.

All developers confirmed that they would repeat the assessment on classes playing the

same design role. One developer said that *"Yes. Usually, the methods of a particular type of class (design role) maintain a relatively high degree of similarity within a system. So if we identified a problem for one of these methods, if it is not something particular, then this problem will probably also be in other similar methods."*. Another developer states that *"Yes. If a problem occurs for a type of class (design role) because the classes have more or less the same content, other classes with the same parameters will also have the same problems. For instance, the method save from AlunoBO and MatriculaBO classes have the same problems, that is, they are long, coupled, and complex"*.

Did class design role influence the developers to reconsider inconsistent evaluations?

Finally, we asked some questions intended to find out inconsistencies in developers' answers. We considered an inconsistency when a developer evaluated differently methods playing the same design role and with similar metric values. We found few inconsistencies (17 out of 700), and the developers also took design role into account to reconsider their evaluations.

Developers reconsidered their initial evaluation in 10 out of 17 inconsistencies. Developers used the characteristics of other classes playing the same design role to reconsider their opinion. A developer claims that *"I evaluated the methods (...) as long. However, now, looking at another BO methods, I perceive that they are not long."*. Another developer replies *"The method (...) was mistaken evaluated when I considered it as high complexity since it is a validator method. Especially if we compare it with the validation methods of another validator classes, which also have more deviations, but they need to be that way because they aim to validate a Form."*.

In summary, the qualitative analysis allow us to answer RQ4 as follows:

> *Regarding the four studied code smells, developers' perception considered class design role when classifying a method as smelly or non-smelly. Therefore, techniques and tools should consider taking design role into account to improve accuracy to detect code smells.*

7.3 THREATS TO VALIDITY

This section discuss threats to the validity of the results and the actions we take to minimize them.

Construct validity: A possible threat is that the benchmarks used to derive metric threshold values may bias the results. We applied well-defined criteria to select real-world Web systems from the Github repository to build the benchmark we used to execute Alves, Aniche and Vale's techniques to reduce this bias. To reduce the bias to select similar systems to compose the benchmark for our proposed techniques, we asked a team of expert developers to suggest systems they consider as reference of good quality. Another threat that could have influenced the developers' perception was if we let they know that the methods they would evaluate were pointed out by metrics as smelly. To avoid that, we stated in our interview protocol that the developers should be told that the methods might or might not have some of the four code smells. The order the methods were

presented to the developers could also have influenced their evaluation. For instance, if the methods were presented in increasing order, the developers could wait to the last ones to indicate them as smelly. Thus, we defined a presentation protocol to alternate methods with high and low metric values to avoid this bias. Another threat was selecting methods for evaluation since it would be impossible for developers to evaluate all methods. Our protocol also defines a systematic approach for selecting methods to have methods in all design roles of the systems.

Internal Validity: A possible threat was the developers' fatigue during the interviews, which could influence the evaluations of the methods, especially the last ones to be presented. To minimize this threat, we conducted a pilot study to define the average time for developers to concentrate and left all the methods to be evaluated open in the IDE to reduce the total interview time. Another threat was the different levels of experience of the developers. To reduce this problem, we also carry out joint developers' evaluations of code smells.

External Validity: The main threat is the number of developers interviewed. However, it is also not common in other similar studies a large number of respondents. We select the two most experienced developers from each team or the most experienced in the project to minimize this threat, and we evaluate distinct software projects. The results are valid for the four evaluated systems and the four studied metrics. We do not suggest that they should be generalized.

7.4 RELATED WORKS

We found some studies assessing the perception of developers in code systems such as IDEs (e.g., Eclipse), text editors (e.g., JEdit), frameworks, and modeling tools (e.g., ArgoUML). However, these systems do not represent a Web-based system that is the application domain of most software developers.

Oliveira et al. (OLIVEIRA; VALENTE; LIMA, 2014) propose a technique to derive metric thresholds from the benchmark of systems. They validate obtained threshold values with software developers. The results showed that high-quality systems respect derived metric thresholds. Our study also evaluates code anomalies with expert developers, but the evaluation was not limited to code anomalies suggested by metric thresholds obtained from a single technique. We compare the results applying metric thresholds derived from five distinct benchmark-based techniques.

Vale et al. (VALE; FERNANDES; FIGUEIREDO, 2018) evaluated metric thresholds obtained from three distinct techniques. They used threshold values to detect God Class and Lazy Class in a software product line. Expert developers previously suggested these code anomalies. As a result, the authors' technique's metric thresholds have a slight advantage over the other evaluated techniques. Our study also assessed the same techniques, except the Lanza's technique (LANZA; MARINESCU, 2006) because it considers that the metrics have a normal distribution, which rarely occurs. Also, our study includes new techniques that take into account the class design role played by system class to derive metric thresholds, and we use real-world web systems and expert developers to evaluate identified code anomalies. Vale and Figueiredo's technique usually had the worst

performance due to the very low metric thresholds it derives. Our techniques have shown improvements considering the individual and joint perceptions of expert developers.

Hozano et al. (2018) conducted an empirical study with 75 developers who evaluated instances of 15 different code smell types. They aim to investigate how similar the developers detect code smells. They analyzed factors related to the developers' profiles and the heuristics adopted by developers on detecting code smells that may influence such (dis)agreement. The results indicated that the developers presented a low agreement on detecting all 15 smell types analyzed. They suggested that background and experience factors did not consistently influence the agreement among the developers, and specific heuristics employed by developers consistently influenced the agreement. Our results contradict this study showing a high degree of agreement between developers in most evaluations. We claim that familiarity with design decisions and class design roles impacted developers' assessments. Palomba et al. (2014) evaluated the developers' perception of metric threshold values for 12 bad smells in three open-source projects. A master student developed the bad smells dataset relying on the definition of the code smells reported in the literature, and metric-based purposes of the DECOR tool (MOHA et al., 2010). The resulting list was validated manually by two masters' students. The authors claim that the process aimed to find reliable bad smells on the object systems and did not find instances of all considered smells. Some original developers, industrial developers, and master's students evaluated the dataset of bad smells. The results show the divergent perception of each group of developers and that the developer's experience and system knowledge play an essential role in identifying some smells. Also, some smells are generally not perceived by developers as design problems because they are simply the result of conscious choices (design decisions) made by developers. They claim that approaches to (semi)automatically improve source code quality should continuously consider the developer's point-of-view. Taibi, Janes e Lenarduzzi (2017) strengthen these results conducted a study analyzing developers' perceived harmfulness of code smells. They compared the developers' perception based on the description of 23 smells, their ability to identify and name them, and the smelly source code's perceived harmfulness. The results confirm that still a lot of misunderstanding surrounding code smells. They recommend that researchers regularly investigate developers' current perceptions since developers will act based on their perceptions and not based on the real consequences of injecting or removing code smells. Our study conducted the assessment from the developers' point of view. However, we argue that only developers familiar with code design decisions should participate in the assessments since the design decisions need to be known and considered on carrying out evaluations.

7.5 SUMMARY

We conducted an empirical study to evaluate the developers' perception of code smells pointed by metric thresholds derived from five benchmark-based techniques. All system developers maintain the source code of evaluated systems, and they are conscious of related design decisions. We compare metric thresholds derived from two proposed techniques, discussed in Chapter 6 with the other three state-of-the-art techniques. We

compare developers' perceptions about code smells refactoring pointed out by metric thresholds from these five techniques. We summarize the results and their implications for research and practice as follow:

Class Design' roles influence developer's perception of code smells. The quantitative results indicated that metric thresholds derived using the class design role as context improved precision and MCC metrics according to developers' perceptions. We reinforce these conclusions using qualitative analysis of developers' answers. A potential implication of this is that techniques and tools also must consider the class design role to improve accuracy to detect smelly methods prone to refactoring.

Design decision awareness on source code increases developers' agreement to recognize smelly methods. The results showed substantial agreement between developers in many evaluations of four evaluated method-level code smells. Our results contradict another large-scale evaluation (HOZANO et al., 2018) that found a low agreement among the developers' evaluations in all investigated code smells, including Long Method and Long Parameter List. A potential implication of this for research is that future works assessing the accuracy of techniques that automatically point out code smells need to consider developers' viewpoint familiar with design decisions of evaluated source code.

Finding systems that follow the same design decisions may not be a trivial task when the available repository of systems is unknown by developers. In the empirical study conducted at the Federal University of Bahia, only the most experienced developers were able to indicate other systems to compose the benchmark that followed the same design decisions. Selecting systems to build a benchmark without these experts could be challenging. Additionally, if we need to build a benchmark from a public repository (e.g., GitHub), the challenge to find out systems developed with similar design decisions could be even higher. In Section 4.4, we propose an automatic approach to select systems developed with similar design decisions. An implication for research of this result is that future works may assess the accuracy of techniques using benchmarks built automatically by the proposed approach since it cannot even be that simple to find systems developed with similar design decisions.

Chapter

8

COMPARING TECHNIQUES TO DERIVE METRIC THRESHOLDS BASED ON CODE REFACTORED ALONG SOFTWARE EVOLUTION

8.1 INTRODUCTION

Refactoring is a well-known technique used by developers to remove code smells and improve software readability and maintainability. Refactoring opportunities are also argued as the main reason for detecting code smells (FOWLER; BECK, 1999). However, very few studies evaluate the impact of code smells detected by state-of-the-art strategies on refactorings effectively performed during the software evolution.

Tufano et al. (2017) discuss that code smells tend to survive for a long time, and 80 percent of smell instances are never removed from the system after their introduction. One possible reason is that developers do not perceive some code smells as design problems (PALOMBA et al., 2014). Previous studies show wildly divergent perceptions about code smells among developers who analyze the same source code snippet (MANTYLA; LASSENIUS, 2006; MANTYLA, 2005; SCHUMACHER et al., 2010; SANTOS et al., 2018; HOZANO et al., 2018). Therefore, despite the widely available tools support for detecting smelly methods (MARINESCU, 2004; KHOMH et al., 2009; MOHA et al., 2010), developers usually need to confirm each pointed out code smell.

Despite the reduced effectiveness of tools and the low agreement of the developers' perception to point out smelly methods, many studies are still building oracles to evaluate the effectiveness of code smell detection techniques based on these tools and developers' perception. In Chapter 7, we discussed that take into account developers' perceptions is essential to assess the precision of techniques that point out smelly methods. Moreover, we argue that developers' awareness of design decisions on the evaluated source code impacted smelly methods detection. In fact, we obtained a high degree of agreement between developers familiar with design decisions evaluating the same source code snippets.

However, developers' fatigue makes the evaluation process challenging to obtain comprehensive coverage, especially in large systems. For this reason, we decided to study the accuracy of metric thresholds using as oracle methods effectively refactored during software evolution. We also compared our two proposed techniques, described in Chapter 6, that consider class design role as context to derive metric thresholds, with the other three state-of-the-art techniques.

We aim to answer our third general research question (RQ3):

RQ3: Are design-sensitive metric thresholds more accurate to detect code smells prone to be refactored?

To investigate which technique derived metric thresholds that pointed out more refactored methods, we conducted a large-scale retrospective study over the commit history of 20 Web-based and 26 Android-based software projects. Our findings are based on the analysis of 23,057 refactorings distributed in 9382 commits of evaluated software projects. We used as oracle the refactorings detected by means of the RefactoringMiner tool on the commit history of these systems. We only considered refactorings that are able to solve each evaluated code smell. For example, we consider extract method refactorings in oracle because it is able to solve long methods. We analyzed how the metrics thresholds derived by the distinct techniques were able to point as smelly the methods effectively refactored during the software evolution. We call smelly method the method whose metric value is above at least one of the metric thresholds derived from the five evaluated techniques. We summarize our findings as follows:

- We observed that smelly methods are always more prone to be refactored than non-smelly methods. We found this result using metric thresholds derived from the five evaluated techniques. This result suggests that metric-based strategies are helpful to evaluate source code quality.

- Techniques that derived low threshold values had high recall values but lower precision values. We observed that techniques that consider design roles as context achieved a better balance between recall and precision.

- Finally, even though most refactorings touch in smelly methods, we found many refactorings applied in non-smelly methods. For example, regarding long methods, 39% of the refactorings in Web-based systems and 52.3% of the refactorings in Android-based systems were applied in non-smelly methods.

We organize the remainder of this chapter as follows. Section 8.2 describes our empirical study settings to mine software repositories evolution aiming to identify if the smelly methods, pointed by thresholds derived from the five evaluated techniques, were refactored. Section 8.3 presents the results of the study. Section 8.4 discusses threats to validity and Section 8.5 discusses related work. Finally, Section 8.6 summarizes the results and discusses our research implications.

8.2 STUDY SETTINGS

The main *goal* of our empirical study is to identify the technique that derived the most accurate metric thresholds to point out methods which were refactored during software evolution. In this study, we only considered the types of refactorings which are able to remove the four code smells we have been studying: long methods, complex complexity methods, high efferent coupling methods and long list of parameters. We used systems from two distinct architectural domains: Web-based systems and Android-based systems.

8.2.1 Research Questions

We formulated the following research questions:

RQ1: *Are smelly methods more prone to be refactored during software evolution?* This research question quantitatively assesses occurrences of refactorings in smelly methods and non-smelly methods during software evolution. We aim to verify if derived metric thresholds pointed out smelly methods most likely to be refactored.

RQ2: *Which technique proposes metric thresholds that best pointed out refactored smelly methods during software evolution?*

This research question quantitatively evaluates the accuracy of metric thresholds derived from the five techniques to point out methods prone to refactoring. We compared our techniques, detailed in Chapter 6, with other three state-of-the-art techniques to derive metric thresholds. We used as oracle the methods effectively refactored during the software evolution. To address RQ1, we test the following null hypothesis.

H0: no technique proposes metric thresholds that improve Matthews Correlation Coefficient (MCC) to detect refactored code smells during software evolution.

8.2.2 Techniques to Derive Metric Thresholds

We compared our two proposed techniques with the other three techniques also involved in the previous study. Thus, in this study, we involved Alves et al. (ALVES; YPMA; VISSER, 2010) and Vale and Figueiredo (VALE; FIGUEIREDO, 2015) techniques, detailed in Section 2.3, which generate a generic threshold value for each metric. We also evaluated Aniche et al. technique (ANICHE, 2015), detailed in Section 2.3, which is based on Alves et al. technique but defines specific metric thresholds according to the class architectural role. We evaluated our two proposed techniques, detailed in Section 6, which considers class design role as context to derive metric thresholds. The class design role, discussed in Chapter 4.2, is an extension of the architectural role concept allowing to assign a role to classes not bound to a predefined reference architecture. Our first technique, called [T1], proposes building a benchmark composed of high-quality systems with similarities in class design roles. Our second technique, called [T2], also considers the class design role in the benchmark creation process. In addition, this technique derives multiple metric thresholds for each class design role rather than a single generic threshold to evaluate all system classes.

8.2.3 Target Systems

We carried out this study with 46 real-world systems developed in Java from two distinct architectural domains: (i) Web-based Systems (20 systems) and (ii) Mobile-based Systems based on the Android platform (26 systems). We chose systems implemented in the same language (Java) because cation domain and programming language are recognized factors that impact the distribution of metrics (ZHANG et al., 2013). The selected architectural domains are popular in the software development industry[1] and follow well-defined reference architectures (MEDVIDOVIC; TAYLOR, 2010), which is an essential requirement to compare our technique with Aniche et al. (2016) technique.

We selected the web-based systems from GitHub using the search string "web language:java stars:>50 pushed:>2020-01-01 size:>200" to select systems developed in Java, with more than 50 stars, with at least 200 files and that have received some contribution from 2020-01-01. These criteria returned a list of 321 projects in 2020-06-07. We excluded libraries, frameworks, web servers, APIs, non-English projects, systems used as implementation examples, and systems with less than two releases. These criteria aimed to select real-world web systems and exclude systems developed with design decisions different from the Web-based architectural domain. We also included systems with at least two available releases because we used the first and the last stable releases as initial and final milestones, respectively, to carry out the proposed study. The resulting list had 22 Web-based systems, which we used as the benchmark to derive metric thresholds. However, we did not find refactorings in two systems to carry out the proposed study, and therefore they were not evaluated.

Table 8.1 summarizes the main characteristics of the 22 selected Web systems. The #classes, #methods, and #LOC columns show the number of classes, methods, and lines of code in each system. The selected systems have between 3 to 137 thousand lines of code (#LOC column). The #design roles column shows the number of design roles identified by our heuristic (Section 4.2). The #releases column shows the number of releases of each system. The $\#Evaluated \backslash \#Total$ Refactorings column shows the number of refactorings evaluated in each system and the number of refactorings reported by RefactoringMiner (TSANTALIS et al., 2018) tool. We only considered types of refactorings which are able to reduce the values of the four metrics we are studying: Lines of Code, McCabe's Cyclomatic Complexity, Efferent Coupling and Number of Method Parameters. Thus, we took into account 8581 from 93749 reported refactorings (9.15%). The #Evaluated Revisions with Related Refactorings column shows the number of revisions evaluated. We evaluated 2387 revisions. Finally, the Initial and Final releases column shows the interval of stable releases considered in each system.

To select widely used Android-based systems from GitHub, we used the search string "android stars:>1000 pushed:>2020-01-01 size:>200 language:Java" to select systems developed in Java, with 1000 stars, at least 200 files and that received contribution from 2020-01-01. We used a higher number of stars in the search string for Android projects to reduce the initial list of systems since Android projects seem more starred than Web-based projects on GitHub. These criteria returned a list of 545 projects in 2020-07-18.

[1] https://octoverse.github.com/

Table 8.1: Web-based Systems

System	Description	#Classes	#Methods	#LOC	#Design Roles	#Releases	#Evaluated / #Total Refactorings	#Evaluated Revisions with Related Refactorings	Initial and Final Releases
Bigbluebutton	Web conferencing system	975	8880	81116	102	77	4/16	2	2.0 - 2018-07-07 2.2.19 - 2020-06-24
OpenMRS-Core	patient-based medical record system	684	7367	68666	52	145	3310 / 30060	688	1.1.0 - 2012-07-25 2.3.1 - 2020-06-04
Heritrix3	Internet Archive's open-source	583	5495	46810	107	08	135 / 689	57	3.2.0 - 2018-07-04 3.4.0 - 2020-05-18
Kafdrop	web UI for viewing Kafka topics	49	329	2178	10	44	21 / 373	15	1.0.0 - 2018-01-06 2.5.1 - 2020-05-19
Karma	integrate data from a variety of data sources	788	5403	61606	66	26	350 / 4064	137	1.197 - 2013-12-27 2.3 - 2020-05-29
WebAnno	annotation tool for a wide range of linguistic annotations	426	3197	34717	73	227	843 / 8642	305	2.0.0 - 2014-07-30 4.0.0 - 2020-05-13
ProjectForge	web-based solution for project management	1785	15315	137776	127	44	287 / 2490	130	6.1.1 - 2016-07-27 6.25.0 - 2019-03-08
Kafka WebView	reading data out of kafka topics	175	1515	11380	15	20	40 / 217	10	1.0.0 - 2018-01-06 2.5.1 - 2020-05-19
Web Budge	manage personal budget	129	713	5480	17	16	176 / 2799	90	1.1.0 - 2015-02-23 3.0.2 - 2019-05-26
Metl	web-based integration platform	390	3953	39192	50	73	243 / 2666	117	1.0.0 - 2016-01-22 3.6.1 - 219-12-06
Hawtio	web console helps you manage your JVM	183	1380	12711	17	139	74 / 434	33	2.0.0 - 2017-04-28 2.10.0 - 2020-04-13
Bastillion	web-based SSH console	63	602	5576	8	82	43 / 647	16	1.08.20 - 2013-06-29 3.10.0 - 2020-05.23
Webofneeds	Finding people to cooperate with	829	5401	42839	108	09	807 / 4693	75	0.3 - 2018-09-05 0.9 - 2020-02-17
Vehicle Routing	Vehicle Routing Problem using OptaPlanner	114	497	3467	17	15	89 / 616	30	7.25.0 - 2019-09-07 7.38.0 - 2020-05-25
VIVO	pen source semantic web tool for research discovery	140	1048	11000	15	95	284 / 2288	68	0.9.0 - 2010-01-29 1.11.1 - 2020-03-07
AET	detects changes on web sites and performs basic page health check	461	2504	17354	36	31	237 / 2163	108	1.3.2 - 2016-06-09 3.3.0 - 2019-08-19
PhenoTips	record clinical findings	742	6480	61499	120	86	570 / 4999	166	1.0.0 - 2014-10-07 1.4.9 - 2019-07-30
NGB	New Genome Browser	459	4731	34303	38	08	38 / 330	6	2.1.0 - 2017-01-27 2.5.1 - 2017-09-27
Asqatasun	web site analyser	3750	14999	136736	141	60	133 / 11257	52	3.0.1 - 2014-04-24 4.1.0 - 2020-04-03
WebProtégé	ontology development environment	3150	20878	105655	330	08	897 / 14306	282	2.5.0 - 2014.07.08 4.0.0 - 2019.08.12
Drools Workbench	web system and repository to govern Drools assets	253	1651	14166	37	94	none	none	none
bambooBSC	Balanced Scorecard (BSC) Business Intelligence (BI) Web platform	932	11225	74214	38	02	none	none	none

We used the same inclusion and exclusion criteria of Web-based systems selection. The resulting list had 30 Android-based systems, which we also used in the benchmark to derive metric thresholds. However, we excluded four projects from the final evaluation because the used tool did not find refactorings.

Table 8.2 summarizes the main characteristics of the 30 selected real-world Android systems. The #classes, #methods, and #LOC columns show the number of classes, methods, and lines of code in each system. The selected systems have between 5 to 111 thousand lines of code (#LOC column). The #Design Roles column shows the number of design roles identified by our heuristic (Section 4.2). The #Releases column shows the number of releases of each evaluated system. The $\#Evaluated \setminus \#Total$ Refactorings column shows the number of refactorings considered in each system and the number of refactoring reported by RefactoringMiner (TSANTALIS et al., 2018) tool. We evaluated 14476 from 114614 reported refactorings (12.63%). The #Evaluated Revisions with Related Refactorings column shows the number of revisions evaluated. We evaluated 6995 revisions. Finally, the Initial and Final Releases column shows the interval of stable releases evaluated in each system.

8.2.4 Code Metrics, Code Smells and Refactorings

Our study considered the four method-level metrics discussed in Table 8.3. Each metric point out the following code smells: long methods, complex complexity methods, high efferent coupling methods and long list of parameters.

Table 8.2: Android-based Systems

System	Description	#Classes	#Methods	#LOC	#Design Roles	#Releases	#Evaluated / #Total Refactorings	#Evaluated Revisions with Related Refactorings	Initial and Final Releases
AntennaPod	open-source podcast manager for Android	401	3692	40730	49	88	400 / 2937	196	1.0.0 - 2015-08-30 1.8.1 - 2020-02-05
Keepass2Android	password manager app	655	6213	63238	84	13	14 / 115	8	1.0.0 - 2016-08-20 1.0.7 - 2019-10-21
QKSMS	open source replacement to the stock messaging app.	349	3916	45455	53	101	52 / 465	32	2.1.0 - 2015-09-01 3.8.1 - 2020-01-27
Slide	free Reddit browser for Android	339	3495	62368	140	163	147 / 841	78	5.3.6 - 2016-05-09 6.5.0 - 2020-07-12
Conversations	the very last word in instant messaging	307	4408	51222	37	221	686 / 3122	402	1.0.0 - 2015-06-25 2.8.8 - 2020-06-25
FairEmail	works with virtually all email providers	250	2973	51615	39	1205	631 / 5157	356	1.74 - 2018-09-30 1.1252 - 2020-07-11
K-9 Mail	open-source email client for Android	256	7000	66294	84	385	114 / 871	59	2.102 - 2009-11-24 5.717 - 2020-06-19
AnExplorer	Another Android Explorer (File Manager)	283	3809	36669	40	8	102 / 518	46	3.4 - 2017-01-07 4.1.1 - 2019-07-11
RedReader	An unofficial open source Reddit client for Android	300	2402	26491	25	71	129 / 1527	48	1.3.5 - 2013-04-06 1.11 -2020-07-01
Nextcloud	allows you to access all your files on your Nextcloud	408	4091	49073	43	632	669 / 5170	633	1.0.0 - 2012-06-16 3.12.1 - 2020-07-07
Signal	messaging app for simple private communication with friends	1369	13686	109622	107	635	1535 / 15111	556	1.0 - 2013-08-06 4.66.5 - 2020-07-14
PocketHub	GitHub Android app	157	707	5984	23	20	101 / 534	39	1.0 - 2012-07-09 1.9 - 2014-02-20
Phonograph	A material designed music player for Android	196	2140	16913	27	24	19 / 357	10	1.0 - 2018-05-01 1.3.4 - 2020-06-30
OpenHub	GitHub Android client	251	3089	20168	23	29	78 / 369	39	1.0.0 - 2017-09-05 3.1.0 - 2020-03-01
Bitcoin Wallet	Standalone Bitcoin node	148	1220	12927	20	377	88 / 117	107	2.46 - 2013-03-19 8.03 - 2020-06-09
AmazeFileManager	Material design file manager for Android	321	2498	31046	43	44	363 / 4536	176	1.1 - 2014-12-03 3.4.3 - 2020-02-14
Wikipedia	The official Wikipedia Android app	607	6229	42513	54	195	1084 / 10604	485	2.0 - 2015-03-23 2.7.5 - 2020-06-29
FastHub	GitHub Android client	493	4717	35389	39	65	314 / 3345	126	1.0.0 - 2017-02-24 4.7.3 - 2019-12-29
PSLab	Repository for the PSLab Android	165	1897	28391	16	20	28 / 314	14	2.0.0 - 2018-08-05 2.0.20 - 2019-10-18
SimpleNote	All notes, synced on all your devices.	91	923	10345	21	81	85 / 601	57	1.0.0 - 2013-09-06 2.7.1 - 2020-06-30
Telegram	Messaging app with a focus on speed and security.	1251	27282	517444	141	27	71 / 309	6	5.13.0 - 2019-12-31 6.2.0 - 2020-06-06
Wordpress	Manage blog from Android	585	8360	84899	55	604	3333 / 19041	1949	2.8.1 - 2015-12-01 15.2 - 2020-07-13
Omni-Notes	Open source note-taking application for Android	173	1182	11892	28	127	354 / 2140	127	3.0.0 - 2013-11-13 6.0.5 - 2019-11-17
ExoPlayer	media player for Android	825	11463	111228	118	170	3314 / 30417	1239	1.0.10 - 2014-07-06 2.11.7 - 2020-06-29
Shuttle	open source, local music player for Android	349	2980	24402	54	118	216 / 1864	59	1.6.5 - 2017-04-12 2.0.17 - 2020-07-12
Loop Habit Tracker	Helps to create and maintain good health habits.	297	2308	17803	29	32	549 / 4232	148	1.0.0 - 2016-02-19 1.8.8 - 2020-06-21
Haven	protect their personal spaces and possessions.	40	386	4155	12	44	none	none	none
Open Event	Mobile App for Organizers and Entry Managers	388	2835	19808	32	10	none	none	none
OwnCloud	Securely access and share data from everywhere and any device	156	1843	20155	22	106	none	none	none
Hijacker	GUI for the penetration testing tools	43	403	7090	61	6	none	none	none

Table 8.3: Method-level Metrics and Code Smells

Metric	Description	Code Smell
McCabe's Cyclomatic Complexity (CC) (MCCABE, 1976)	It counts number of branching points of each method.	High Complexity
Lines of Code (LOC) (LANZA; MARINESCU, 2006)	It counts the number of executable statements of each method, excluding comments and blank lines.	Long Method
Efferent Coupling (EC) (MARTIN, 1995)	It counts the number of classes from which each method calls methods or accesses attributes.	High Efferent Coupling
Number of Method Parameters (NMP) (FOWLER; BECK, 1999)	It counts the number of parameters of each method.	High Number of Parameters

We selected these method-level metrics, and associated code smells because these metrics are available in many tools (PAIVA et al., 2017) and have been successfully used for fault-proneness prediction (FONTANA et al., 2013; GIL; LALOUCHE, 2017; BOUCHER; BADRI, 2018), for instance. We used only one metric for each code smell detection strategy because we aim to evaluate the accuracy of metric thresholds derived by five techniques to point out smelly methods prone to refactoring.

8.2.5 Study Procedure

To answer the research questions, we mined data about the evolution of the target systems according the procedures detailed in this section. Initially, we derived metric thresholds using the five evaluated techniques (activity 1). Then, we identified refactorings along the commits between two stable versions of each system (activity 2). Then we identified all smelly methods in each the commit just before each commit with refactoring (activity 3). Finally, we created a resulting list with smelly and non-smelly methods which were refactored during software evolution (activity 4) to answer our proposed research questions. Below we detail these four activities.

Activity 1: Building benchmarks and deriving metric thresholds. We built and derived metric thresholds from two distinct benchmarks of two architectural domains: Web-based and Android-based systems. We used 22 systems for the web-based benchmark and 30 systems for the Android benchmark. Alves et al., Vale et al., and Aniche et al. techniques derived metric thresholds from these two original benchmarks. Then, we built new benchmarks to derive metric thresholds to [T1] and [T2] techniques. Both techniques propose to derive metric thresholds from a benchmark of systems developed with similar design decisions. Therefore, each new benchmark contains systems, from the original benchmark of each domain, developed with similar design decisions to the evaluated system. To do that, we used the SystemSimilarity tool, discussed in Section 4.4, to compose the new benchmarks. Table 8.1 contains the original benchmark for Web-based systems and Table 8.2 contains the original benchmark to evaluate Android-based systems. The tool calculates the similarity in design decisions between two systems, ranging from 0 to 1. Two systems have high similarity (closer to 1) when they have similar percentages of lines of code associated with each identified class design role. We composed each new benchmark by systems with similarity to the system to be evaluated above 0.1. This value aimed to ensure that the benchmarks used to derive the threshold values for each evaluated system were composed of a minimum subset of four different systems from the original benchmarks. We consider this threshold of four systems to guarantee some diversity in the benchmarks. For example, following this criterion, to evaluate the AET system, we end up with a benchmark composed of seven similar systems: Hawtio, Asqatsun, Metl, PojectForge, Phenotips, BigBlueButton and the AET itself. These seven systems are the ones from the 22 Web systems (Table 8.1) which our SystemSimilarity tool found a similarity with the AET system higher than 0.1. We built 20 Web-based benchmarks, each one to derive thresholds by means of the [T1] and [T2] techniques for evaluating one of the 20 Web systems that contains refactorings (Table 8.1). The number of systems in these benchmarks ranged from four to eighteen systems. We built

26 Android-based benchmarks, each one to derive thresholds by means of the [T1] and [T2] techniques for evaluating each of the 26 Android systems that contains refactorings (Table 8.2). The number of systems in these benchmarks ranged from 29 to 30 systems. Android-based systems have stricter design decisions and, therefore, Android-based systems usually have higher similarity between each other than Web-based systems. Having the benchmarks, we used the ThresholdTool, discussed in Section 6.3, to derive metric thresholds to the five evaluated techniques.

Activity 2: Extracting refactorings between two stable versions. Initially, we selected the oldest and newest stable release available for each evaluated project from the GitHub repository. Usually, a tag is marked as a release when the source code is stable enough to be made available to end-users of the software. Next, we extracted refactoring operations between the first stable release $Release_1$ and the last stable release $Release_n$. We used the RefactoringMiner tool (TSANTALIS; KETKAR; DIG, 2020) which achieved very high precision (98%) with a recall that is competitive to the previous state-of-the-art (87%) with a minimal computation cost (TSANTALIS et al., 2018). To carry out the proposed evaluation, we only considered types of refactorings which are able to reduce values of the four metrics involved in our study and, as consequence, are able to remove the four code smells also involved in this study. Regarding the long method, high complexity method, and high efferent coupling code smells, we considered the `EXTRACT OPERATION` and `EXTRACT AND MOVE OPERATION` refactorings detectable by the RefactoringMiner tool. Related to the high number of parameters smell, we considered the `RENAME METHOD` refactoring also detectable by the RefactoringMiner. But, we only considered this refactoring when it reduced the number of parameters of the refactored method. Therefore, the resulting list contains only the refactorings able to reduce metrics and to solve the four studied code smells. Besides the name of the refactoring type, each line of the list contains the commit identification and the method signatures before and after the application of the refactoring.

Activity 3: Identifying smelly methods in commits just before commits with refactoring. During Atitivity 2, we identified commits with at least one method that suffered refactoring. Here, in this activity, for each of those commits, we took the commit the occurred just before it and verified if there were code smells in its corresponding source code. To detected the code smells, we used the ContextSmell tool (Section 6.3) which analyzed the source code according to the four metrics and thresholds derived by means of the five techniques. In another words, given a refactored method m_i in a commit c_i, between stable releases $Release_1$ and $Release_n$, we identify all smelly methods pointed out by metric thresholds derived from the studied techniques on the commit c_{i-1}.

Activity 4: Creating a final resulting list with smelly and non-smelly methods refactored along software evolution. Finally, we merged the resulting lists from Activities 2 and 3 to generate a final list containing refactored smelly methods, refactored non-smelly methods, and non-refactored smelly methods during the evolution of each evaluated system. We added a method to the refactored smelly methods list when a developer refactored the smelly method between the initial and final releases. Similarly, we added to refactored non-smelly methods list when a developer refactored the method between evaluated releases, but it was never smelly. Finally, we added a method

to the non-refactored smelly methods list when the developer never refactored a smelly method during software evolution. We automated the last three activity in a tool called SmellRefactored[2].

8.2.6 Data Analysis

To answer the RQ1 research question, we computed the probability of occurrence of the target refactorings in smelly methods and non-smelly method. Probability is the likelihood of an event occurring divided by the number of expected outcomes of the event.

Since we considered several systems, we aggregate the results of all systems per metric to have a more straightforward overview of the result quality. Aggregate metrics are more robust than the mean, which is biased by the fact that datasets are unbalanced for different smell types in terms of smelly and non-smelly instances (in some cases, the datasets do not contain any smelly instance) (PECORELLI et al., 2019, 2020). Therefore, we compute the probability of refactoring in methods pointed out as smelly in the datasets of Web-based systems and Android-based systems. Later we calculate the probability of refactoring occurs in non-smelly methods for each architectural domain.

To answer RQ2 research question, we performed quantitative data analysis. In RQ2, we compared the effectiveness of metric thresholds based on with a dataset of smelly methods effectively refactored during software evolution. Our dataset included code smells pointed out by metric thresholds derived from the five techniques, and we only considered type of refactorings that are able to reduce values of the four metrics of the study.

To assess the effectiveness of metric thresholds, we compute four well-known metrics: *precision*, *recall*, *F-measure* and *Matthews Correlation Coefficient (MCC)*. *Precision* represents the fraction of instances of methods predicted as smelly by a metric threshold that were effectively refactored during software evolution. *Recall* represents the fraction of refactored methods which were predicted as smelly by a metric threshold. *F-measure* is the weighted harmonic mean of the precision and recall. Finally, MCC is a correlation coefficient between the observed and predicted binary classifications. It has values in the range [-1,+1] where a coefficient of +1 represents a perfect prediction and -1 indicates total disagreement between prediction and observation. MCC is based on all four quadrants of the confusion matrix.

Similarly to RQ1, we aggregated the results of all evaluated systems for each metric to have a more straightforward overview of the result quality (ANTONIOL et al., 2002; PECORELLI et al., 2019).The research question RQ2 compared smelly methods pointed out by metric thresholds derived from each of the five techniques with the reference list of refactored methods.

A true positive (TP) occurs when a metric threshold proposed by a technique point out as smelly a methods that suffered a refactoring which is able to decrease the value of that metric. A false positive (FP) happens when a metric threshold proposed by a technique identifies a code smell that does not match with refactored methods. A true

[2]https://github.com/marcosdosea/SmellRafactored

negative (TN) occurs when the metric threshold does not point out a method as smelly, and the method was not refactored during software evolution. Finally, a false negative (FN) happens when a metric threshold do not identify a method as smelly but it is refactored during software evolution. The index i ranges over the entire dataset of values of each metric. Based on these assumptions, we computed:

$$Precision = \frac{\sum_i \mathrm{TP}_i}{\sum_i (\mathrm{TP}_i + \mathrm{FP}_i)} \tag{8.1}$$

$$Recall = \frac{\sum_i \mathrm{TP}_i}{\sum_i (\mathrm{TP}_i \mathrm{FN}_i)} \tag{8.2}$$

$$F - measure = 2 * \frac{precision * recall}{precision + recall} \tag{8.3}$$

$$MCC = \frac{\sum_i (\mathrm{TP}_i * \mathrm{TN}_i - \mathrm{FP}_i * \mathrm{FN}_i)}{\sum_i \sqrt{(\mathrm{TP}_i + \mathrm{FP}_i)(\mathrm{TP}_i + \mathrm{FN}_i) + (\mathrm{TN}_i + \mathrm{FP}_i) + (\mathrm{TN}_i + \mathrm{FN}_i)}} \tag{8.4}$$

8.3 RESULTS AND DISCUSSION

In this section, we report and discuss the main findings of our study guided by each research question.

RQ1*: Are smelly methods more prone to be refactored during software evolution?*

Findings: We analyzed 2118 refactorings applied in Web-based systems and 5434 applied in Android-based systems related to the long method smell. We found that 1292 (61%) refactorings in Web-based systems and 2596 (47.7%) refactorings from Android-based systems were applied in smelly methods. We consider a method as long when it has LOC value higher than at least one of the thresholds for LOC derived from the five evaluated techniques. These results show that many refactorings were also applied in non-smelly methods.

We also observed the refactoring probability in non-smelly methods and long methods considering each evaluated technique. Table 8.4 summarizes the results using metric thresholds derived from five techniques. For instance, considering the metric threshold derived by Alves technique, we observed in Web-based systems that the refactoring probability was 20.69% in methods pointed out as smelly versus 0.65% in methods pointed out as non-smelly. We observe similar numbers evaluating Android-based systems. For instance, developers refactored 30.3% of the methods pointed out as long and 1.34% of the non-smelly methods. These results show that the refactoring probability of long methods was always higher than non-smelly methods. In Web-based systems, this probability was at least twenty-five times higher than they occur in non-smelly methods. Whereas in Android-based systems, the refactoring probability of long methods was at least twenty-two times greater than non-smelly methods.

We evaluated 1959 refactorings applied in Web-based systems and 4608 applied in Android-based systems regarding the high complexity smell. We observed that 1268

Table 8.4: Refactoring probability in non-smelly and long methods pointed out by metric thresholds derived from distinct techniques

	Web-based		**Android-based**	
Technique	**Smelly**	**Non-Smelly**	**Smelly**	**Non-Smelly**
Alves *et al.*	20.69%	0.65%	30.9%	1.34%
Vale and Figueiredo	11.43%	0.23%	20.6%	0.75%
Aniche *et al.*	15.24%	0.59%	32.3%	1.34%
[T1]	16.72%	0.64%	31.0%	1.34%
[T2]	14.50%	0.55%	29.8%	1.32%

Table 8.5: Refactoring probability in non-smelly methods and methods with high complexity pointed out by metric thresholds derived from distinct techniques

	Web-based		**Android-based**	
Techniques	**Smelly**	**Non-Smelly**	**Smelly**	**Non-Smelly**
Alves *et al.*	15.4%	0.61%	16.7%	1.15%
Vale and Figueiredo	10.0%	0.23%	15.6%	0.75%
Aniche *et al.*	14.6%	0.57%	20.4%	1.13%
[T1]	15.6%	0.60%	16.7%	1.15%
[T2]	11.7%	0.54%	19.7%	1.10%

(64.7%) refactorings in Web-based systems and 1760 (38.1%) refactorings in Android-based systems were applied in smelly methods. We consider a method with high complexity when it has cyclomatic complexity higher than at least one of the thresholds derived from the five evaluated techniques. These results mean that many refactorings able to reduce ciclomatic complexity were also applied in non-smelly methods.

We also observed the refactoring probability in non-smelly methods and complex methods per each technique. Table 8.5 summarizes the results. For instance, considering the metric threshold derived by [T1] technique, we observed that refactoring occurred in 15.6% of the methods pointed out as smelly versus 0.60% of the methods pointed out as non-smelly in Web-based systems. In Android systems, developers refactored 16.7% of smelly methods with high complexity versus 1.15% of non-smelly methods. These results show that the refactoring probability of complex methods code smell was always higher than non-smelly methods. In Web-based systems, this probability was at least twenty-five times higher than they occur in non-smelly methods. Whereas in Android-based systems, the refactoring probability of complex methods was at least fourteen times greater than non-smelly methods.

We analyzed 1914 refactorings applied in Web-based systems and 4531 applied in Android-based systems related to the high efferent coupling smell. We observed that 1191 (62.2%) refactorings in Web-based systems and 1734 (38.2%) refactorings in Android-based systems were applied in smelly methods. We consider a method with high efferent coupling when it is has efferent coupling metric higher than at least one metric threshold derived from the five evaluated techniques. These results mean that many refactorings

Table 8.6: Refactoring probability in non-smelly methods and methods with high efferent coupling pointed out by metric thresholds derived from distinct techniques

	Web-based		**Android-based**	
Techiniques	**Smelly**	**Non-Smelly**	**Smelly**	**Non-Smelly**
Alves *et al.*	15.0%	0.53%	17.3%	1.07%
Vale and Figueiredo	10.3%	0.25%	14.5%	0.71%
Aniche *et al.*	11.3%	0.51%	16.7%	1.05%
[T1]	14.7%	0.52%	16.7%	1.07%
[T2]	10.3%	0.49%	15.2%	0.99%

able to reduce efferent coupling were also applied in non-smelly methods.

We also observe the refactoring probability of non-smelly methods and smelly methods with high efferent coupling per technique. Table 8.6 summarizes the results. For instance, considering the metric threshold derived by Alves technique, we observed in Web-based systems that the refactoring probability was 15.0% in methods pointed out as smelly versus 0.53% in methods pointed out as non-smelly. In Android-based systems, developers refactored 17.3% of the methods pointed with high efferent coupling versus 1.07% of the non-smelly methods. These results show that the refactoring probability of methods with high efferent coupling was always higher than non-smelly methods. In Web-based systems, this probability was at least twenty times higher than they occur in non-smelly methods. Whereas in Android-based systems, the refactoring probability of methods assigned to this code smell was at least fifteen times greater than non-smelly methods.

Finally, we analyzed 239 refactorings applied in Web-based systems and 535 applied in Android-based systems related to the long list of parameters code smell. We observed that 62 (25.94%) refactorings in Web-based systems and 115 (21.4%) refactorings in Android-based systems were applied in smelly methods. We consider a method with a long list of parameters when it is pointed out as smell by at least one metric threshold derived from five evaluated techniques. These results mean that most refactorings able to reduce the number of parameters were applied in non-smelly methods.

We also observed the refactoring probability of non-smelly methods and smelly methods with long list of parameters per technique. Table 8.7 summarizes the results using metric thresholds derived from the five techniques. For instance, considering the metric threshold derived by Alves technique, we observed in Web-based systems that the refactoring probability was 0.69% in methods pointed out as smelly versus 0.08% in methods pointed out as non-smelly. In Android-based systems, developers refactored 0.6% of the methods pointed with high efferent coupling versus 0.16% of the non-smelly methods. Despite the low refactoring probability in both cases, we observe that smelly methods with a long list of parameters was more refactored than non-smelly methods. In Web-based systems, this probability was at least two times higher than they occur in non-smelly methods. Whereas in Android-based systems, the refactoring probability of methods assigned to this code smell was at least three times greater than non-smelly methods.

In summary, the results allow us to answer RQ1 as follows:

Table 8.7: Refactoring probability in non-smelly methods and methods with high number of parameters pointed out by metric thresholds derived from distinct techniques

	Web-based		**Android-based**	
Techiniques	**Smelly**	**Non-Smelly**	**Smelly**	**Non-Smelly**
Alves *et al.*	0.69%	0.08%	0.60%	0.16%
Vale and Figueiredo	0.55%	0.06%	0.62%	0.13%
Aniche *et al.*	0.21%	0.08%	0.56%	0.17%
[T1]	0.65%	0.08%	0.60%	0.16%
[T2]	0.50%	0.07%	0.57%	0.16%

Table 8.8: Aggregate Results for Long Methods Refactored during Software Evolution

	Web-based					**Android-based**				
	TP	**Precision**	**Recall**	**F-measure**	**MCC**	**TP**	**Precision**	**Recall**	**F-measure**	**MCC**
Alves et al.	216	0.20	0.20	0.20	0.20	230	0.31	0.09	0.14	0.16
Vale and Figueiredo	**1214**	0.11	**0.81**	0.20	**0.29**	**2596**	0.21	**0.68**	**0.32**	**0.35**
Aniche et al.	346	0.15	0.31	**0.21**	0.21	285	0.32	0.11	0.17	0.18
[T1]	200	0.17	0.19	0.18	0,17	230	0.31	0.09	0.14	0.16
[T2]	**408**	0.15	**0.36**	**0.22**	**0.23**	**367**	0.30	**0.14**	**0.19**	**0.19**

Regarding the four studied code smells in Web-based and Android-based systems, the refactoring probability of smelly methods was always higher than non-smelly methods. Therefore, metric-based strategies to detect code smells are helpful to point out methods more prone to refactoring. However, there were still a considerable number of refactorings in non-smelly methods.

RQ2: *Which technique proposes metric thresholds that best pointed out refactored smelly methods during software evolution?*

Findings: Regarding the LOC metric, Table 8.8 shows the aggregate results of true positives (TP), Precision, Recall, F-measure, and MCC for long methods refactored during Web-based and Android-based systems evolution. We observe that Vale and Figueiredo's technique had higher True positives, Recall, and MCC metrics for both architectural domains. However, the technique always presents the lowest precision among the evaluated techniques because it derives the lowest metric thresholds, significantly increasing false positives. The [T2] technique presented precision similar to the other techniques, but it obtained better recall. The result is that the technique [T2] obtained the second-best performance concerning the aggregated F-measure and MCC metrics. It is worth mentioning that the precision metric here is about the methods that were effectively refactored and not the methods that should be refactored. Developers may not refactor some methods due to organizational pressure (LAVALLÉE; ROBILLARD, 2015). Additionally, thresholds too low can also point out methods whose refactoring aimed at not exclusively improving the maintainability and comprehensibility of the source (PALOMBA et al., 2017).

Figure 8.1 illustrates differences between the distribution of Recall metric for refactored long methods. It shows the two graphs for two evaluated architectural domains, one for Web-based and the other one for Android-based systems, with five box plots each.

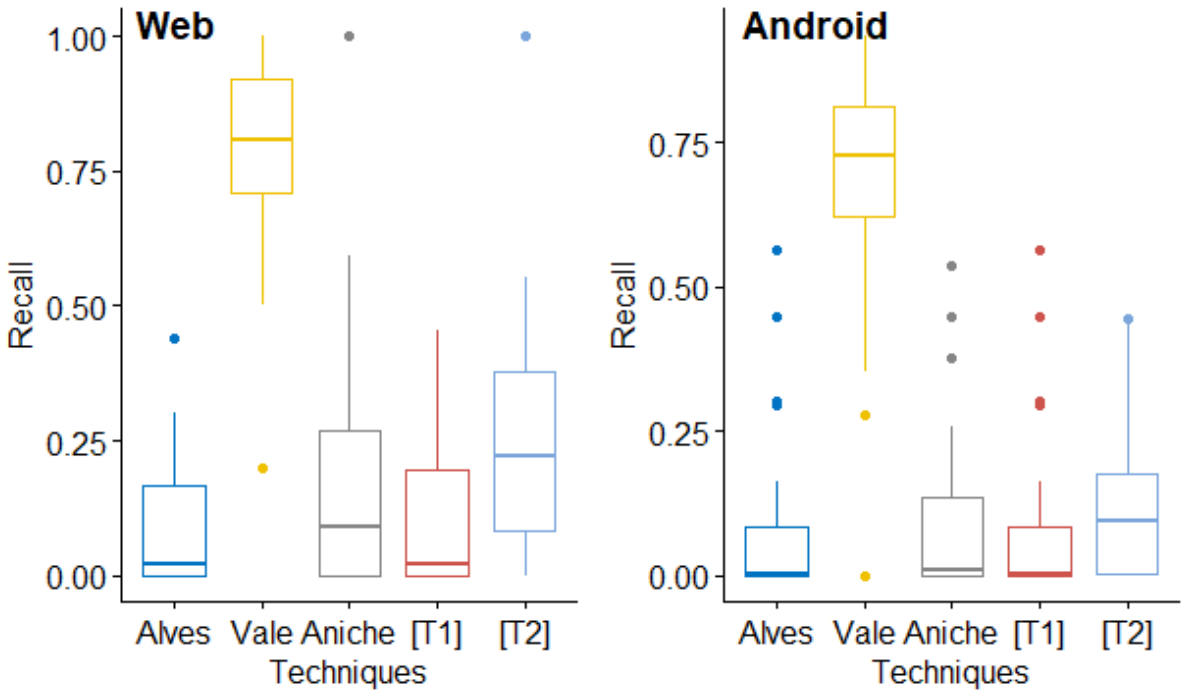

Figure 8.1: Distribution of Recall Metrics for Long methods Refactored during Software Evolution

Each graph is about Recall of refactored long methods pointed out by LOC threshold derived from one of the five evaluated techniques. The graphs also illustrated the advantage of Recall metrics to Vale and Figueiredo's technique, followed by the [T2] technique for both architectural domains evaluated.

We obtained similar results using statistical tests to Web-based and Android-based systems. To test the hypothesis H_0 to Recall, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distributions of the Recall values from evaluated techniques. So, we perform the Kruskal-Wallis test, and we identify there are significant differences between the techniques ($p - value$ = 1.722e-09 and $p - value$ = 1.353e-11, respectively) rejecting the null hypothesis (H_0). This means that at least one technique proposes metric thresholds that improve the Recall to detect code smells refactored. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the Recall of Vale and Figueiredo' technique is significantly greater than other techniques. We obtained a large effect size applying Cliff's δ. In Web-based systems, we also obtained a statistically significant improvement in the recall values of the technique [T2] compared to the technique of Alves. We obtained a medium effect size applying Cliff's δ. These results mean that the Vale technique proposes metric thresholds for LOC metric that improved Recall metric to detect long methods refactored. In Web-based systems, [T2] technique also proposes metric thresholds for LOC that improved recall metric to detect long methods. We do not found differences between recall obtained by [T1] and Aniche techniques.

Table 8.9: Aggregate Results for Methods with High Complexity Refactored during Software Evolution

	Web-based					Android-based				
	TP	**Precision**	**Recall**	**F-measure**	**MCC**	**TP**	**Precision**	**Recall**	**F-measure**	**MCC**
Alves et al.	482	0.15	0.39	**0.22**	0.23	216	0.17	0.10	0.12	0.12
Vale and Figueiredo	**1268**	0.10	**0.83**	0.18	**0.27**	**1760**	0.16	**0.59**	**0.25**	**0.28**
Aniche et al.	577	0.15	0.46	**0.22**	**0.24**	322	0.20	0.14	0.17	0.16
[T1]	493	0.16	0.40	**0.23**	**0.24**	216	0.17	0.10	0.12	0.12
[T2]	**630**	0.12	**0.49**	0.19	0.23	**462**	0.20	**0.20**	**0.20**	**0.19**

Regarding the CC metric, Table 8.9 shows the aggregate results of true positives (TP), Precision, Recall, F-measure, and MCC of methods with high complexity refactored during Web-based and Android-based systems evolution. Similar to LOC metric evaluation, we observe that Vale and Figueiredo's technique obtained higher True positives, Recall, and MCC metrics for both architectural domains. However, the technique also presented the lowest precision among the evaluated techniques because it derives the lowest metric thresholds, significantly increasing false positives. In Web-based systems, we observed a tiny difference between the techniques evaluated in the MCC metric and a slightly more significant difference in Android-based systems. After Vale and Figuiredo's technique, the [T2] technique obtained the best-aggregated results to the MCC metric.

Figure 8.2 illustrates differences between the distribution of Recall metric for refactored methods with high complexity during software evolution. It shows two graphs for two evaluated architectural domains, one for Web-based and the other one for Android-based systems, with five box plots each. Each graph is about the distribution of Recall of refactored methods with high complexity pointed out by CC metric thresholds derived from five evaluated techniques. The graphs also illustrated the advantage of Recall to Vale and Figueiredo's technique, followed by the [T2] technique for both architectural domains evaluated.

We obtained similar results using statistical tests to Web-based and Android-based systems. To test the hypothesis H_0 to Recall, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distributions of the Recall values from evaluated techniques. So, we perform the Kruskal-Wallis test, and we identify there are significant differences between the techniques ($p - value$ = 1.114e-06 and $p - value$ = 2.552e-10, respectively), rejecting the null hypothesis (H_0). This means that at least one technique proposes metric thresholds that improve the Recall to detect smelly methods refactored during software evolution. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the Recall of Vale technique is significantly greater than other techniques. We obtained a large effect size applying Cliff's δ. In Android-based systems, we also obtained a statistically significant improvement in the Recall values of the technique [T2] compared to the technique of Alves and [T1]. We obtained a medium effect size applying Cliff's δ. These results mean that the Vale technique proposes metric thresholds for CC metric that improved Recall metric to detect methods with high complexity refactored during software evolution. In Android-based systems, [T2] technique also proposes metric thresholds for CC that improved Recall metric to detect methods with high complexity.

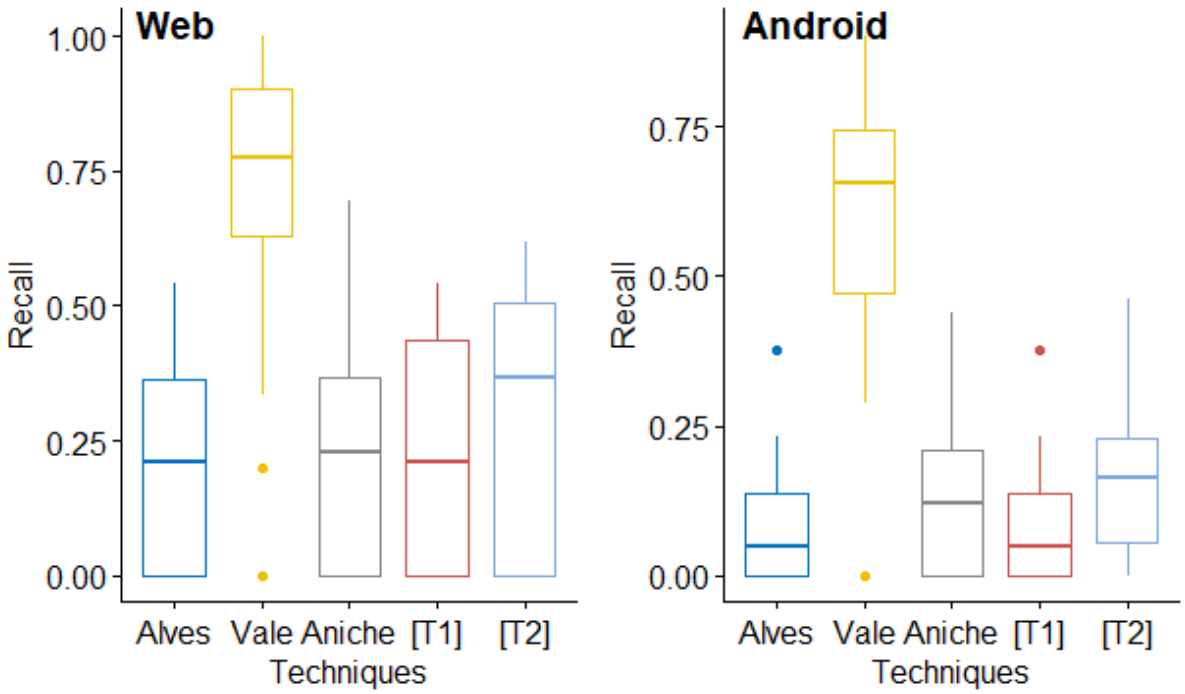

Figure 8.2: Distribution of Recall Metrics for High Complexity Methods Refactored during the Software Evolution

Table 8.10: Aggregate Results for Methods with High Efferent Coupling Refactored during Software Evolution

	Web-based					Android-based				
	TP	**Precision**	**Recall**	**F-measure**	**MCC**	**TP**	**Precision**	**Recall**	**F-measure**	**MCC**
Alves et al.	664	0.15	0.50	**0.23**	**0.26**	507	0.17	0.22	0.19	0.18
Vale and Figueiredo	**1191**	0.10	**0.80**	0.18	**0.27**	**1734**	0.14	**0.60**	**0.23**	**0.27**
Aniche et al.	705	0.11	0.53	0.19	0.23	582	0.17	0.25	0.20	0.19
[T1]	675	0.15	0.51	**0.23**	**0.26**	507	0.17	0.22	0.19	0.18
[T2]	**728**	0.10	**0.55**	0.17	0.22	**753**	0.15	**0.31**	**0.20**	**0.20**

We do not found differences between recall obtained by [T1] and Aniche techniques.

Regarding the EC metric, Table 8.10 shows the aggregate results of true positives (TP), Precision, Recall, F-measure, and MCC of methods with high efferent coupling refactored during Web-based and Android-based systems evolution. Similar to LOC and CC metric evaluations, we observe that Vale and Figueiredo's technique obtained higher True positives, Recall, and MCC metrics for both architectural domains. However, the technique also presented the lowest precision among the evaluated techniques because it derives the lowest metric thresholds, significantly increasing false positives. In Web-based systems, we observed a tiny difference in the MCC metric between the techniques evaluated and a slightly more significant difference in Android-based systems. After Vale and Figuiredo's technique, the [T1] and [T2] techniques obtained the best-aggregated results to the MCC metric for Web-based and Android-based systems, respectively.

Figure 8.3 illustrates differences between the distribution of Recall for refactored meth-

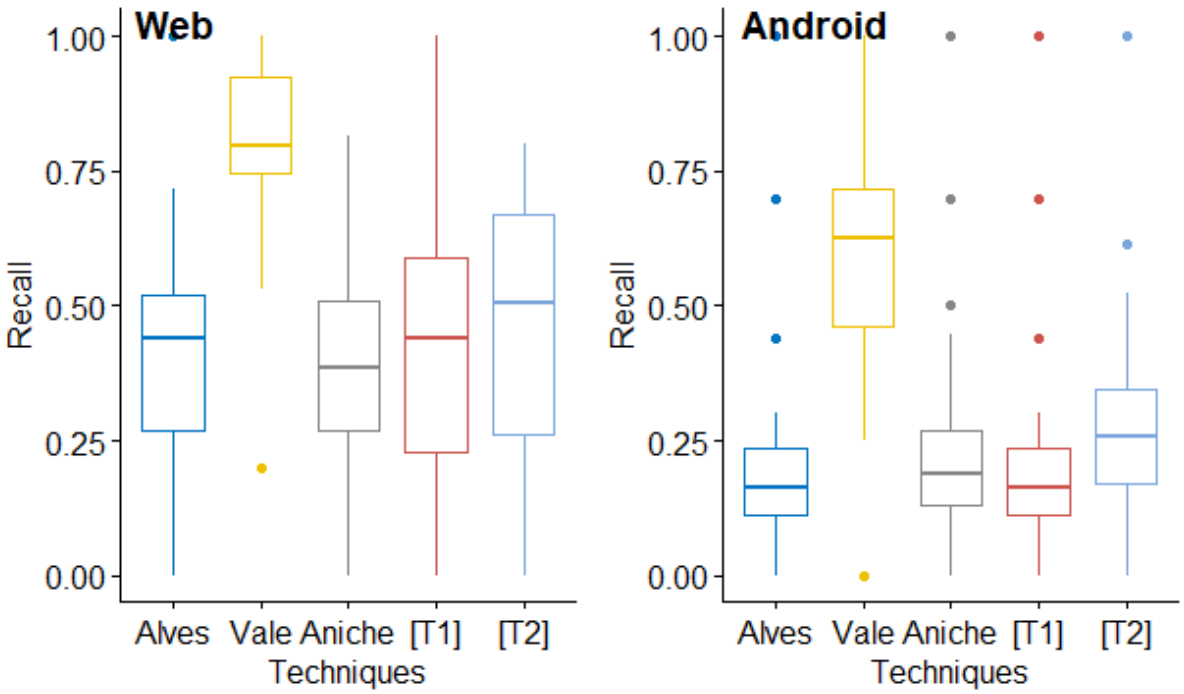

Figure 8.3: Distribution of Recall Metrics for High Efferent Coupling Methods Refactored during the Software Evolution

ods with high efferent coupling during the software evolution. It shows two graphs for two evaluated architectural domains, one for Web-based and the other one for Android-based systems, with five box plots each. Each graph is about the distribution of Recall of refactored methods with efferent coupling pointed out by EC metric thresholds derived from five evaluated techniques. The graphs also illustrated the advantage of Recall to Vale and Figueiredo's technique, followed by the [T2] technique for both architectural domains evaluated.

We obtained similar results using statistical tests to Web-based and Android-based systems. To test the hypothesis H_0 to Recall, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distributions of the Recall values from evaluated techniques. So, we perform the Kruskal-Wallis test, and we identify there are significant differences between the techniques ($p - value$ = 2.782e-06 and $p - value$ = 3.835e-08, respectively), rejecting the null hypothesis (H_0). This means that at least one technique proposes metric thresholds that improve Recall to detect smelly methods refactored during the software evolution. Pairwise comparisons using the Mann-Whitney-Wilcoxon test with 5% confident level and Bonferroni correction (MANN; WHITNEY, 1947) showed that the Recall of Vale technique is significantly greater than other techniques. We obtained a large effect size applying Cliff's δ. In Android-based systems, we also obtained a statistically significant improvement in the Recall values of the technique [T2] compared to Alves and [T1] techniques. We obtained a medium effect size applying Cliff's δ. These results mean that the Vale technique proposes metric thresholds for EC metric that

Table 8.11: Aggregate Results for Methods with Long List of Parameters Refactored during Software Evolution

	Web-based					Android-based				
	TP	**Precision**	**Recall**	**F-measure**	**MCC**	**TP**	**Precision**	**Recall**	**F-measure**	**MCC**
Alves et al.	34	0.007	0.25	**0.014**	0.036	45	0.006	0.146	**0.012**	0.021
Vale and Figueiredo	**62**	0.006	**0.43**	0.011	**0.041**	**115**	0.006	**0.361**	**0.012**	**0.036**
Aniche et al.	32	0.002	0.23	0.004	0.013	38	0.006	0.125	0.011	0.018
[T1]	**36**	0.007	0.26	**0.013**	**0.035**	45	0.006	0.146	**0.012**	0.021
[T2]	35	0.005	**0.26**	0.010	0.029	**61**	0.006	**0.196**	0.011	**0.023**

improved Recall metric to detect methods with high efferent coupling refactored during software evolution. In Android-based systems, [T2] technique also proposes metric thresholds for EC that improved Recall metric to detect methods with high efferent coupling. We do not found differences between Recall obtained by [T1] and Aniche techniques.

Finally, the NOP metric, Table 8.11 shows the aggregate results of true positives (TP), Precision, Recall, F-measure, and MCC of methods with long list of parameters refactored during Web-based and Android-based systems evolution. We observe that Vale and Figueiredo's technique obtained a tiny difference in True positives, Recall, and MCC metrics for both architectural domains. However, all techniques presented low MCC and F-measure values due to the low precision. Despite the low accuracy of all techniques, Vale and Figueiredo's technique obtained a slightly better performance followed by [T1] and Alves techniques in Web-based systems. In Android-based systems, [T2] had the second-best MCC result.

Figure 8.4 illustrates differences between the distribution of Recall for refactored methods with long list of parameters during software evolution. It shows two graphs for two evaluated architectural domains, one for Web-based and the other one for Android-based systems, with five box plots each. Each graph is about the distribution of Recall of refactored methods with long list of parameters pointed out by NOP metric thresholds derived from five evaluated techniques. The graphs illustrate some advantage of Recall to Vale and Figueiredo's technique, followed by the [T2] technique for both architectural domains evaluated.

We carry out procedures using statistical tests to Web-based and Android-based systems. To test the hypothesis H_0 to Recall, we use the Shapiro-Wilk test of normality, and we can not assume the normality for distributions of the Recall values from evaluated techniques. So, we perform the Kruskal-Wallis test, and we identify there are not significant differences between the techniques ($p - value = 0.5736$ and $p - value = 0.2052$, respectively), not rejecting the null hypothesis (H_0). This means that no technique proposes metric thresholds for NOP metric that improved Recall metric to detect methods with long list of parameters refactored during software evolution.

In summary, the results allow us to answer RQ2 as follows:

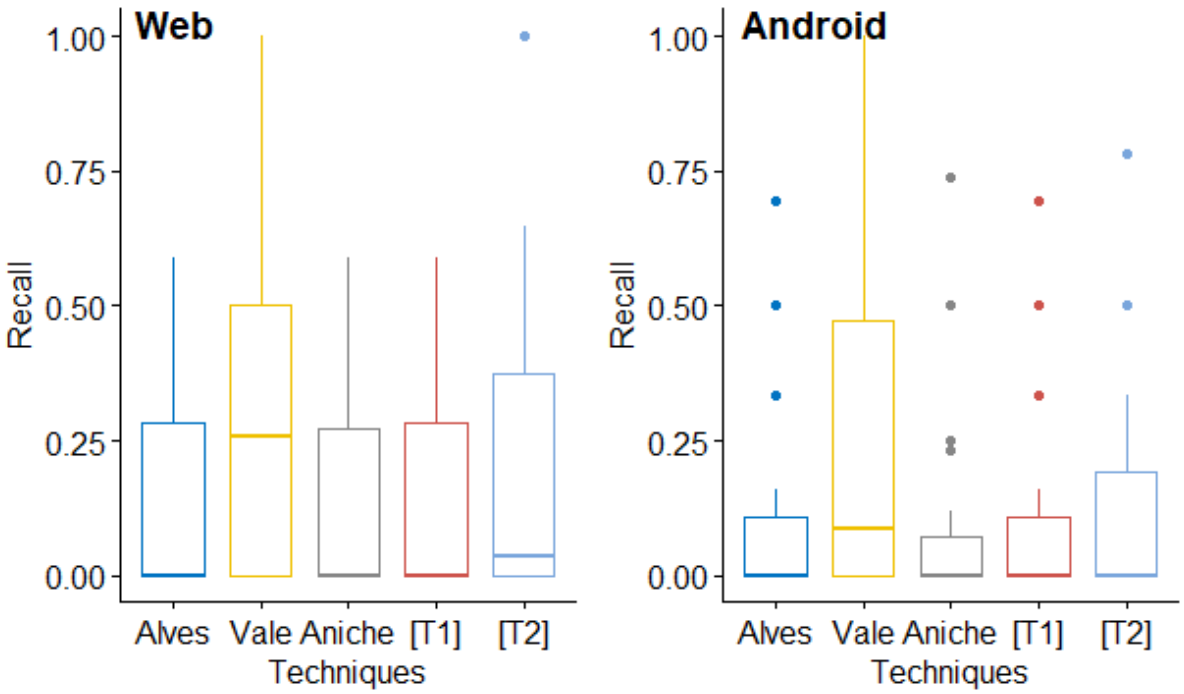

Figure 8.4: Distribution of Recall Metrics for Methods with a Long List of Parameters Refactored during the Software Evolution

Vale and Figueiredo's technique proposed metric thresholds that improved the Recall metric to detect the four evaluated code smells during software evolution. The highest MCC values are due to the high recall generally achieved by the technique due to the very low derived threshold metric. However, the precision of this technique was always very low, consequently increasing the number of false positives. Our two proposed techniques, which relied on design roles, improved results, better balancing the recall and precision metrics. We suggest considering them for deriving thresholds that improve recall without damaging precision.

8.4 THREATS TO VALIDITY

This section discusses the threats to validity of our study following common guidelines (KITCHENHAM et al., 2006).

Internal validity. There might be a threat associated with the correctness of the tools we used to identify refactored methods and calculate software metrics. We used the RefactoringMiner tool (TSANTALIS; KETKAR; DIG, 2020) , which was already used in other studies. Also, we manually checked many metric values and identified refactorings. Moreover, we evaluated our tool and heuristic for design role identification by means of a study with developers and Web-based governmental systems, as described in Section 4.2.

Construction validity. There is a possible threat related to metrics we selected for our study. The selected method-level metrics cover important aspects of source code quality

and are widely used for software fault prediction (CATAL; DIRI, 2009; GIGER et al., 2012). Errors in calculating metrics may also occur (ALVES; YPMA; VISSER, 2010). However, these errors are usually small and to minimize these interferences we use the Kruskal-Wallis and Cliff's δ statistical tests.

External validity. Some of the findings might be specific to the selected metrics, software systems, and domains assessed. To minimize this bias, we discussed in Section 8.2.3 some well-defined and replicable criteria for selecting representative systems in each architectural domain. Although other domains use similar mechanisms to implement the architecture, we still intend to extend this investigation to other systems and domains. So, although we restricted to the systems and domains analyzed, this is an important step toward improving the accuracy of derived metric thresholds.

8.5 RELATED WORKS

Bavota et al. (2015) mined the evolution history of three Java open source projects to investigate whether refactoring activities occur on smelly code detected by tools. Results indicate that refactoring operations focused on code components for which quality metrics do not suggest there might be a need for refactoring operations. Finally, 42% of refactoring operations were performed on code entities affected by code smells. Moreover, only 7% of the performed operations remove the code smells from the affected class. We performed a large-scale study using Web-based and Android-based systems focusing on refactoring able to solve related code smell. For example, we consider the extract method refactoring to reduce the LOC metric, but it is not a guarantee of code smell resolution. We evaluated distinct techniques to derive metric thresholds that impact the selection of smelly methods. We found that between 10% to 20% of refactoring was applied in smelly methods. We argue that, despite some questioning the usefulness of metrics as a filter mechanism for evaluating source code quality, this result becomes representative when we observe that refactored non-smelly methods were below 1%. It is noteworthy that we evaluate metrics individually because our goal was to evaluate derived metric thresholds, but metric-based detection strategies of code smells can combine these metrics aiming to improve the accuracy. Cedrim et al. (2017) analyze how often commonly-used refactoring types affect the density of 13 types of code smells along with the version histories of 23 projects. The results show that even though 79.4% of the refactorings touched smelly elements, 57% did not reduce their occurrences, and only 9.7% of refactorings removed smells, while 33.3% introduced new ones. More than 95% of such refactorings induced smells not removed in successive commits, which suggest refactorings tend to introduce long-living smells instead of eliminating existing ones more frequently. We argue that there are significant variations according to the used metric thresholds derived by evaluated techniques. Additionally, we discuss that many refactorings able to solve related code smells applied in non-smelly methods. We suggest that future studies assess the context of the application of these refactorings.

Palomba et al. (2017) argue that software repository studies could corroborate the achieved findings surveying developers. They perform a quantitative investigation on the relationship between different types of code changes and 28 different refactoring types

coming from 3 open-source projects. Results showed that developers tend to apply a higher number of refactoring operations to improve the maintainability and comprehensibility of the source code when fixing bugs. They also argued that developers perform more complex refactoring operations to improve code cohesion when new features are implemented. Our study using real-world software repositories from popular architectural domains aim to complement the results discussed in Chapter 7. We reinforce the results showing that the deriving of metric thresholds considering class design role can improve code smell detection.

Sousa et al. (2020) investigate if smells can serve as indicators of architectural refactoring opportunities performing a retrospective study over the commit history of 50 software projects. They investigated if refactored elements had architectural problems that automatically-detected smells could have indicated. They found that the proportion of refactored elements without smells is much lower than those refactored with smells. They concluded that smells are indicators of architectural refactoring opportunities, and smells that often co-occurred with other smells (67.53%) are indicators of architectural refactoring opportunities in most cases (88.53% of refactored elements). However, the study use strategies to detect cod smells based on predefined metric thresholds. Our study evaluates distinct metric thresholds carrying out a large-scale study with software projects from popular architectural domains. We suggest that future studies evaluate using detection strategies combining metric thresholds derived from techniques that consider the class design roles.

8.6 SUMMARY

We conducted a large-scale retrospective study to investigate which technique derived metric thresholds that pointed out more methods refactored during the software evolution. We investigated software projects from two popular architectural domains: Web-based and Android-based projects. The study analyzed the software evolution of 20 Web-based and 26 Android-based real-world projects. We summarize the results and their implications for research and practice as follow:

Smelly methods pointed by metric thresholds derived using five evaluated techniques are more refactored than non-smelly methods. Initially, our results showed that five evaluated techniques derived metric thresholds that point out methods more prone to refactoring. A practical implication of these results is that, regardless of the technique used to derive metric thresholds, metric-based strategies can point out a relevant subset of methods for quality assessment.

Many refactorings are performed in non-smelly methods. We observe a large number of refactorings performed in methods that which metric thresholds derived from five evaluated techniques not pointed out as smelly. As discussed in Chapter 7, Vale and Figueiredo's technique proposed metric thresholds so low that developers end up evaluating most of them as false positives. Interestingly, we found several refactorings effectively applied to non-smelly methods during the software evolution. A potential implication of this for research is that future works could investigate these refactoring in non-smelly methods. Refactorings in non-smelly methods that aim to improve the source code's

quality can help improve code smell detection strategies.

Metric thresholds derived from techniques using Design role as context improving recall. Vale and Figueiredo's technique proposed metric thresholds that improved the MCC metric to detect four evaluated code smells during the software evolution. The highest MCC values are due to the high recall generally achieved by the technique due to the very low derived threshold metric. However, the precision of the technique was always very low, consequently increasing the number of false positives. Our two proposed techniques that considered the design role as context achieved the second-best results, better balancing the recall and precision metrics. A practical implication of this is that derive metric thresholds using our proposed techniques improve recall without damaging precision.

Chapter 9

FINAL REMARKS

State-of-the-art techniques implemented by current automated static analysis tools (ASATs) rely on metric-based detection strategies. However, the accuracy of a detection strategy is heavily influenced by the calibration of thresholds for the used metrics (SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018). Metric thresholds not tailored can generate an overload of alarms. Many of these alarms are false positives since manual inspection reveals no effect on the software quality and maintenance effort (OLBRICH; CRUZES; SJØBERG, 2010; KHOMH et al., 2011; SJOBERG et al., 2013; YAMASHITA, 2013; PALOMBA et al., 2014; HOZANO et al., 2018).

This research aimed to propose and evaluate two novel techniques to derive design-sensitive metric thresholds. Our central hypothesis is that deriving a metric threshold that relies on fine-grained design decisions could point out more relevant code smells. We carry out a series of empirical studies to understand the problem, propose a design context to consider by novel techniques, and evaluate the accuracy of metric thresholds derived by them.

Firstly, we conducted a web-based survey with 350 Brazilian practitioners engaged in the software development industry. We found that code analysis practices are widespread among Brazilian practitioners who recognize its importance. However, there is no routine for applying these practices. In addition, they report difficulties in fitting static analysis tools in the software development process. One possible reason, recognized by practitioners, is that most of these tools use a single metric threshold, which might not be adequate to evaluate all system classes. As a result, we propose improving guidelines to use and fit code analysis practices into the software development process to make them widely used. We also propose that ASATs offer different ways to be used in distinct phases of the software development process, making it easier to fit them into any process. Finally, the survey proposes investigating whether multiple metric thresholds that take source code context into account reduce static analysis tool false alarms.

Secondly, we proposed investigating design decisions that could be considered context to derive design-sensitive metric thresholds based on previous insights. We investigated

whether other fine-grained design decisions also influenced the distribution of software metrics. Our findings showed that distribution of metrics are sensitive to the following design decisions: (i) design role of the class (ii) used libraries, (iii) coding style, (iv) exception handling, and (v) logging and debugging code mechanisms. In this way, the distribution of software metrics is sensitive to fine-grained design decisions, and we should consider taking them into account when building benchmarks for metric-based source code analysis. We used these findings to propose new techniques to derive design-sensitive metric thresholds.

Finally, we carry out two empirical studies aiming to evaluated proposed techniques. The first study evaluated developers' perception of code smells pointed by metric thresholds derived from two proposed techniques with the other three state-of-the-art techniques. Among other findings, we found evidence that class design roles influenced developers' perception of code smells. The second study investigated which technique derived metric thresholds that pointed out more methods refactored during the software evolution from two popular architectural domains. Our two proposed techniques that considered the design role as context improved recall without damaging precision.

In summary, based on a series of empirical studies, our work proposed automated approaches to identify the class design role, the similarity between systems, and two novel design-sensitive techniques to derive metric thresholds. Software engineers practitioners can directly use our results to improve the detection of code smells using automated techniques. Thereby, in the next section, we describe possible directions for future research.

9.1 FUTURE RESEARCH DIRECTIONS

We suggest the following directions for future work that arise from our research:

Carry out studies to show and spread code analysis benefits to software developers. In Chapter 3, we showed that Brazilian practitioners know code analysis practices and recognize their importance. However, development teams do not apply these practices regularly. We suggest to conduct studies to clarify the impact and benefits of applying code analysis practices.

Improve static analysis tools to fitting in software development processes. In Chapter 3, we reported that many practitioners stated unaware of automated static analysis tools. They also reported many issues and challenges to adopt these tools regularly. We suggest to investigate how to fit these tools to be adopted in software development processes regularly.

Investigate whether the class design role concept helps in software comprehension. In Chapter 4, we observed developers regularly using the class design role to comprehend or make design decisions over classes. We suggest evaluating whether the design role assigned by our heuristic improves the comprehension of software design. Therefore, software development tools could also create class design role views to help source code comprehension.

Investigate whether the similarity between systems relied on class design role helps to monitor design violations. In Chapter 4, we proposed an approach to calculate the similarity between systems using the class design role. We suggest to carry out studies to

assess whether the proposed similarity measure could detect the distancing of the planned design during the software evolution.

Investigate whether class design roles impact the distribution of class-level metrics. In Chapter 5, we showed that design roles affected the distribution of method-level metrics. We suggest to carry out studies to evaluate the impact of class design role on the distribution of class-level metrics. A potential implication of this is that metric-based analysis methods can take design roles into account when using class-level metrics.

Investigate whether ContextSmell tool prevents the introduction of code smells by software developers. In Section 6.3, we showed ContextSmell, an Eclipse plugin that enables the just-in-time detection of the four studied code smells. The tool allows identifying code smells using metric thresholds derived from distinct techniques. We made the tool publicly available and open-source to encourage the research community to improve further the tool with additional code smells detectors or evaluate other techniques to derive metric thresholds.

Carry out a deep investigation about factors of agreement on developers' perception of code smells. In Chapter 7, we compared developers' perceptions about code smells refactoring. Our results contradict previous studies because we found substantial agreement between developers. We argue that familiarity with design decisions impacted obtained results. We suggest carrying out an in-depth study looking for factors that lead to a high or low agreement regarding the developer's perception of code smell.

Investigate categories of code smells performed in non-smelly methods. In Chapter 8 we showed many refactorings performed in non-smelly methods. We suggest applying a taxonomy, like proposed by Hassan et al. (HASSAN, 2009), based on commit message analysis. They suggest (i) Fault Repairing Modification (FR), (ii) Feature Introduction Modification (FI), and (iii) General Maintenance Modification (GM).

Evaluate the accuracy of more complex detection strategies to detect code smells using metric thresholds derived from our proposed techniques. In Chapters 7 and 8 we used strategies based on a single metric because our goal is to individually assess the impact of the metric threshold proposed by different techniques. We suggest evaluating other metric-based strategies to detect code smells that combine metric thresholds derived from our proposed techniques.

Evaluate the accuracy of our proposed techniques according to the number of classes assigned to design roles. We observed a significant difference in the number of classes playing the *Undefined* design role among the systems. Also, we identify some design roles assigned to a few classes in a system. Although it is possible to adjust the predefined design roles table, we suggest evaluating the impact of considering all class design roles to calculate system similarity and derive metric thresholds. One possibility would be to evaluate to derive different metric thresholds only to class design roles assigned to a significant number of classes.

BIBLIOGRAPHY

AGNER, L. T. W. et al. A brazilian survey on uml and model-driven practices for embedded software development. *Journal of Systems and Software*, v. 86, n. 4, p. 997 – 1005, 2013. ISSN 0164-1212.

ALUR, D.; CRUPI, J.; MALKS, D. *Core J2EE Patterns*. [S.l.: s.n.], 2003. 650 p. ISSN 1098-6596. ISBN 0-13-142246-4.

ALVES, T. L.; YPMA, C.; VISSER, J. Deriving metric thresholds from benchmark data. In: *International Conference on Software Maintenance (ICSM)*. Washington, DC, USA: IEEE, 2010. p. 1–10.

ANICHE, M. et al. SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In: *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Raleigh, North Carolina: IEEE, 2016. p. 41–50.

ANICHE, M. F. Detection strategies of smells in web software development. In: *International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.]: IEEE, 2015. p. 598–601.

ANTONIOL, G. et al. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, v. 28, n. 10, p. 970–983, 2002.

ARCOVERDE, R. et al. Automatically detecting architecturally-relevant code anomalies. In: *International Workshop on Recommendation Systems for Software Engineering (RSSE)*. Piscataway, NJ, USA: IEEE, 2012. p. 90–91.

AYEWAH, N. et al. Using static analysis to find bugs. *IEEE Software*, v. 25, n. 5, p. 22–29, Sept 2008.

AYEWAH, N.; PUGH, W. The google findbugs fixit. In: *International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: ACM, 2010. p. 241–252. ISBN 978-1-60558-823-0.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: *International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, 2013. p. 712–721. ISBN 978-1-4673-3076-3. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=2486788.2486882⟩.

BALACHANDRAN, V. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In: *International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, 2013. p. 931–940. ISBN 978-1-4673-3076-3.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 3rd. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 0321815734, 9780321815736.

BAUER, C.; KING, G.; GREGORY, G. *Java Persistence with Hibernate, Second Edition*. [S.l.]: Manning Publications, 2015. 608 p. ISBN 9781617290459.

BAVOTA, G. et al. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, Elsevier, v. 107, p. 1–14, 2015.

BELLER, M. et al. Analyzing the state of static analysis: A large-scale evaluation in open source software. In: *Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.]: IEEE, 2016. v. 1, p. 470–481.

BENLARBI, S. et al. Thresholds for object-oriented measures. In: *International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.]: IEEE, 2000. p. 24–38. ISSN 1071-9458.

BESSEY, A. et al. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, ACM, New York, NY, USA, v. 53, n. 2, p. 66–75, fev. 2010. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/1646353.1646374⟩.

BOOCH, G. Object-oriented development. *Transactions on Software Engineering (TSE)*, IEEE, n. 2, p. 211–221, 1986.

BORGES, H.; HORA, A.; VALENTE, M. T. Understanding the factors that impact the popularity of github repositories. In: *International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2016. p. 334–344.

BOSU, A.; CARVER, J. C. Impact of peer code review on peer impression formation: A survey. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.]: ACM/IEEE, 2013. p. 133–142. ISSN 1949-3770.

BOSU, A. et al. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *Transactions on Software Engineering (TSE)*, v. 43, n. 1, p. 56–75, Jan 2017. ISSN 0098-5589.

BOUCHER, A.; BADRI, M. Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology (IST)*, v. 96, p. 38 – 67, 2018.

BUDI, A. et al. Automated Detection of Likely Design Flaws in Layered Architectures. In: *International Conference on Software Engineering and Knowledge Engineering (SEKE)*. [S.l.]: Research Collection School Of Information Systems, 2011.

CATAL, C.; DIRI, B. A systematic review of software fault prediction studies. *Expert Systems with Applications*, Elsevier, v. 36, n. 4, p. 7346–7354, 2009.

CEDRIM, D. et al. Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In: *Joint Meeting on Foundations of Software Engineering (FSE)*. [S.l.]: ACM, 2017. p. 465–475.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *Transactions on Software Engineering (TSE)*, IEEE, v. 20, n. 6, p. 476–493, June 1994. ISSN 0098-5589.

CHRISTAKIS, M.; BIRD, C. What developers want and need from program analysis: An empirical study. In: *International Conference on Automated Software Engineering (ASE)*. [S.l.]: IEEE/ACM, 2016. p. 332–343.

CLIFF, N. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, American Psychological Association, v. 114, n. 3, p. 494, 1993.

COHEN, J. A power primer. *Psychological bulletin*, American Psychological Association, v. 112, n. 1, p. 155, 1992.

COLEMAN, D.; LOWTHER, B.; OMAN, P. The application of software maintainability models in industrial software systems. *Journal of Systems and Software (JSS)*, v. 29, n. 1, p. 3 – 16, 1995. ISSN 0164-1212. Oregon Metric Workshop. Disponível em: ⟨http://www.sciencedirect.com/science/article/pii/0164121294001257⟩.

CORBIN, J. M.; STRAUSS, A. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, Springer, v. 13, n. 1, p. 3–21, 1990.

DÓSEA, M.; SANT'ANNA, C. Um Método para Detectar Similaridade entre Sistemas baseado em Decisões de Design: um Estudo Preliminar. In: *In VI Workshop on Software Visualization, Evolution and Maintenance (VEM). Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*. São Carlos: [s.n.], 2016.

DÓSEA, M. et al. A survey of software code review practices in brazil. *CoRR*, abs/2007.14276, 2020. Disponível em: ⟨https://arxiv.org/abs/2007.14276⟩.

DÓSEA, M.; SANT'ANNA, C.; SANTOS, C. Towards an Approach to Prevent Long Methods Based on Architecture-Sensitive Recommendations. In: *In IV Workshop on Software Visualization, Evolution and Maintenance (VEM). Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*. Maringá: [s.n.], 2016.

DÓSEA, M.; SANT'ANNA, C.; SILVA, B. C. How do design decisions affect the distribution of software metrics? In: *International Conference on Program Comprehension (ICPC)*. New York, NY, USA: ACM, 2018. p. 74–85. ISBN 978-1-4503-5714-2.

DRAGAN, N.; COLLARD, M. L.; MALETIC, J. I. Automatic identification of class stereotypes. In: *International Conference on Software Maintenance (ICSM)*. [S.l.]: IEEE, 2010. p. 1–10.

EADDY, M.; AHO, A.; MURPHY, G. C. Identifying, assigning, and quantifying crosscutting concerns. In: *International Workshop on Assessment of Contemporary Modularization Techniques (ACoM)*. Washington, DC, USA: IEEE, 2007.

ERNI, K.; LEWERENTZ, C. Applying design-metrics to object-oriented frameworks. In: *International Software Metrics Symposium (METRIC)*. [S.l.]: IEEE, 1996. p. 64–74.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, IBM, Riverton, NJ, USA, v. 15, n. 3, p. 182–211, set. 1976. ISSN 0018-8670. Disponível em: ⟨http://dx.doi.org/10.1147/sj.153.0182⟩.

FELDT, R. et al. Links between the personalities, views and attitudes of software engineers. *Information and Software Technology (IST)*, v. 52, n. 6, p. 611 – 624, 2010. ISSN 0950-5849.

FERREIRA, K. A. et al. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software (JSS)*, Elsevier, v. 85, n. 2, p. 244–257, feb 2012. ISSN 01641212.

FLEISS, J. L.; COHEN, J.; EVERITT, B. S. Large sample standard errors of kappa and weighted kappa. *Psychological bulletin*, American Psychological Association, v. 72, n. 5, p. 323, 1969.

FONTANA, F. A. et al. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In: *International Conference on Software Mainteinance (ICSM)*. [S.l.]: IEEE, 2013. p. 260–269.

FONTANA, F. A. et al. Automatic Metric Thresholds Derivation for Code Smell Detection. In: *International Workshop on Emerging Trends in Software Metrics (WETSOM)*. [S.l.]: IEEE, 2015. p. 44–53. ISBN 978-1-4673-7103-2. ISSN 23270969.

FOWLER, M.; BECK, K. *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Professional, 1999.

GAMMA, E. et al. Design patterns: Abstraction and reuse of object-oriented design. In: NIERSTRASZ, O. M. (Ed.). *ECOOP' 93 - Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 406–431. ISBN 978-3-540-47910-9.

GHANNEM, A.; BOUSSAIDI, G. E.; KESSENTINI, M. On the use of design defect examples to detect model refactoring opportunities. *Software Quality Journal (SQJ)*, v. 24, n. 4, p. 947–965, Dec 2016. ISSN 1573-1367. Disponível em: ⟨https://doi.org/10.1007/s11219-015-9271-9⟩.

GIGER, E. et al. Method-level bug prediction. In: *International symposium on Empirical software engineering and measurement (ESEM)*. [S.l.]: ACM, 2012. p. 171–180.

GIL, J. Y.; LALOUCHE, G. When do software complexity metrics mean nothing?-when examined out of context. *Journal of Object Technology (JOT)*, v. 15, n. 1, p. 1–25, 2016.

GIL, Y.; LALOUCHE, G. On the correlation between size and metric validity. *Empirical Software Engineering (ESE)*, v. 22, n. 5, p. 2585–2611, Oct 2017.

HASSAN, A. E. Predicting faults using the complexity of code changes. In: *International Conference on Software Engineering (ICSE)*. [S.l.]: IEEE, 2009. p. 78–88. ISSN 0270-5257.

HEITLAGER, I.; KUIPERS, T.; VISSER, J. A practical model for measuring maintainability. In: *International Conference on the Quality of Information and Communications Technology (QUATIC)*. [S.l.]: IEEE, 2007. p. 30–39.

HERMANS, F.; AIVALOGLOU, E. Do code smells hamper novice programming? a controlled experiment on scratch programs. In: *International Conference on Program Comprehension (ICPC)*. [S.l.]: IEEE, 2016. p. 1–10.

HERMANS, F.; PINZGER, M.; DEURSEN, A. van. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering (ESE)*, Springer, v. 20, n. 2, p. 549–575, 2015.

HIGO, Y.; KUSUMOTO, S. Flattening code for metrics measurement and analysis. In: *International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.]: IEEE, 2017. p. 494–498.

HOLMES, R.; MURPHY, G. C. Using structural context to recommend source code examples. In: *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2005. p. 117–125. ISBN 1-58113-963-2.

HOZANO, M. et al. Using developers' feedback to improve code smell detection. In: *Symposium on Applied Computing (SAC)*. New York, NY, USA: ACM, 2015. p. 1661–1663. ISBN 978-1-4503-3196-8.

HOZANO, M. et al. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology (IST)*, v. 93, p. 130 – 146, 2018. ISSN 0950-5849.

JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: *International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, 2013. p. 672–681. ISBN 978-1-4673-3076-3.

Kamei, Y.; Shihab, E. Defect prediction: Accomplishments and future challenges. In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.]: IEEE, 2016. v. 5, p. 33–45.

KAMPENES, V. B. et al. A systematic review of effect size in software engineering experiments. *Information and Software Technology (IST)*, Elsevier, v. 49, n. 11, p. 1073–1086, 2007.

KASUNIC, M. *Designing an Effective Survey*. [S.l.], 2005.

KESSENTINI, W. et al. A cooperative parallel search-based software engineering approach for code-smells detection. *Transactions on Software Engineering (TSE)*, IEEE, v. 40, n. 9, p. 841–861, Sep. 2014. ISSN 0098-5589.

KHOMH, F. et al. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering (ESE)*, Springer, v. 17, n. 3, p. 243–275, 2012.

KHOMH, F. et al. A bayesian approach for the detection of code and design smells. In: *International Conference on Quality Software (ICQS)*. [S.l.]: IEEE, 2009. p. 305–314. ISSN 1550-6002.

KHOMH, F. et al. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software (JSS)*, v. 84, n. 4, p. 559 – 572, 2011. ISSN 0164-1212. The Ninth International Conference on Quality Software.

KIM, S.; ERNST, M. D. Which warnings should i fix first? In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2007. p. 45–54. ISBN 978-1-59593-811-4. Disponível em: ⟨http://doi.acm.org/10.1145/1287624.1287633⟩.

KITCHENHAM, B. et al. Evaluating guidelines for empirical software engineering studies. In: *International Symposium on Empirical Software Engineering (ISESE)*. New York, NY, USA: ACM, 2006. p. 38–47. ISBN 1-59593-218-6.

KITCHENHAM, B. A.; PFLEEGER, S. L. Principles of survey research part 2: Designing a survey. *SIGSOFT Software Engineering Notes*, ACM, New York, NY, USA, v. 27, n. 1, p. 18–20, jan. 2002. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/566493.566495⟩.

KONONENKO, O.; BAYSAL, O.; GODFREY, M. W. Code review quality: How developers see it. In: *nternational Conference on Software Engineering (ICSE)*. [S.l.]: IEEE, 2016. p. 1028–1038. ISSN 1558-1225.

KONONENKO, O. et al. Investigating code review quality: Do people and participation matter? In: *International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.]: IEEE, 2015. p. 111–120.

KUMAR, R.; NORI, A. V. The economics of static analysis tools. In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2013. p. 707–710. ISBN 978-1-4503-2237-9.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. *biometrics*, JSTOR, p. 159–174, 1977.

LANZA, M.; MARINESCU, R. *Object-Oriented Metrics in Practice*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. 205 p. ISBN 978-3-540-24429-5.

LAVALLÉE, M.; ROBILLARD, P. N. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: *International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, 2015. p. 677–687. ISBN 978-1-4799-1934-5.

LAVAZZA, L.; MORASCA, S. An empirical evaluation of distribution-based thresholds for internal software measures. In: *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. New York, NY, USA: ACM, 2016. p. 6:1–6:10. ISBN 978-1-4503-4772-3.

LAYMAN, L.; WILLIAMS, L.; AMANT, R. S. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.]: IEEE, 2007. p. 176–185. ISSN 1949-3770.

LIMA, R.; DÓSEA, M.; SANT'ANNA, C. Comparando Técnicas de Extração de Valores Limiares para Métricas: Um Estudo Preliminar com Desenvolvedores Web. In: *In VI Workshop on Software Visualization, Evolution and Maintenance (VEM). Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*. São Carlos: [s.n.], 2016.

LIMA, R. A. d. J. *Comparando técnicas de extração de valores limiares para métricas de código fonte: um estudo com desenvolvedores Web*. 2021. Disponível em: ⟨http://repositorio.ufba.br/ri/handle/ri/33641⟩.

LINAKER, J. et al. *Guidelines for Conducting Surveys in Software Engineering*. [S.l.], 2015. Disponível em: ⟨http://lup.lub.lu.se/record/5366801⟩.

MACLEOD, L. et al. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, v. 35, n. 4, p. 34–42, July/August 2018. ISSN 0740-7459. Disponível em: ⟨doi.ieeecomputersociety.org/10.1109/MS.2017.265100500⟩.

MANN, H. B.; WHITNEY, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, JSTOR, p. 50–60, 1947.

MANSOOR, U. et al. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal (SQJ)*, v. 25, n. 2, p. 529–552, Jun 2017. ISSN 1573-1367. Disponível em: ⟨https://doi.org/10.1007/s11219-016-9309-7⟩.

MANTYLA, M. V. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: *International Symposium on Empirical Software Engineering*. [S.l.: s.n.], 2005. p. 10 pp.–.

MANTYLA, M. V.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, Springer, v. 11, n. 3, p. 395–431, 2006.

MARINESCU, C. Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications. In: *International Conference on Program Comprehension (ICPC)*. [S.l.]: IEEE, 2006. p. 169–180. ISBN 0-7695-2601-2. ISSN 1092-8138.

MARINESCU, R. Detection strategies: metrics-based rules for detecting design flaws. In: *International Conference on Software Maintenance (ICSM)*. [S.l.]: IEEE, 2004. p. 350–359. ISSN 1063-6773.

MARTIN, R. C. *Designing object-oriented C++ applications*. [S.l.]: Prentice Hall, 1995.

MCCABE, T. J. A complexity measure. In: *International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE, 1976.

MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: Foundations, theory, and practice. In: *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2010. p. 471–472. ISBN 978-1-60558-719-6.

MOHA, N. et al. Decor: A method for the specification and detection of code and design smells. *Transactions on Software Engineering (TSE)*, IEEE, v. 36, n. 1, p. 20–36, Jan 2010. ISSN 0098-5589.

MORI, A. et al. Evaluating domain-specific metric thresholds: An empirical study. In: *International Conference on Technical Debt (TechDebt)*. New York, NY, USA: ACM, 2018. p. 41–50. ISBN 978-1-4503-5713-5. Disponível em: ⟨http://doi.acm.org/10.1145/3194164.3194173⟩.

MYERS, G. J. et al. *The art of software testing*. [S.l.]: Wiley Online Library, 2004.

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2013. p. 466–476. ISBN 978-1-4503-2237-9. Disponível em: ⟨http://doi.acm.org/10.1145/2491411.2491415⟩.

NEJMEH, B. A. Npath: A measure of execution path complexity and its applications. *Communications of the ACM*, ACM, New York, NY, USA, v. 31, n. 2, p. 188–200, fev. 1988. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/42372.42379⟩.

OIZUMI, W. et al. Code anomalies flock together. *International Conference on Software Engineering (ICSE)*, IEEE, p. 440–451, 2016. ISSN 02705257.

OLBRICH, S. M.; CRUZES, D. S.; SJØBERG, D. I. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: *International Conference on Software Maintenance (ICSM)*. [S.l.]: IEEE, 2010. p. 1–10.

OLIVEIRA, P.; VALENTE, M. T.; LIMA, F. P. Extracting relative thresholds for source code metrics. In: *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. [S.l.]: IEEE, 2014. p. 254–263.

OLIVEIRA, R. et al. Collaborative identification of code smells: A multi-case study. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. [S.l.: s.n.], 2017. p. 33–42.

PAIVA, T. et al. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, v. 5, n. 1, Oct 2017.

PALOMBA, F. et al. Detecting bad smells in source code using change history information. In: *International Conference on Automated Software Engineering (ASE)*. Piscataway, NJ, USA: IEEE, 2013. p. 268–278. ISBN 978-1-4799-0215-6. Disponível em: ⟨https://doi.org/10.1109/ASE.2013.6693086⟩.

PALOMBA, F. et al. Do they really smell bad? a study on developers' perception of bad code smells. In: *International Conference on Software maintenance and evolution (ICSME)*. [S.l.]: IEEE, 2014. p. 101–110.

PALOMBA, F. et al. An exploratory study on the relationship between changes and refactoring. In: *International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2017. p. 176–185.

PANICHELLA, S. et al. Would static analysis tools help developers with code reviews? In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2015. p. 161–170. Disponível em: ⟨doi.ieeecomputersociety.org/10.1109/SANER.2015.7081826⟩.

PECORELLI, F. et al. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *Journal of Systems and Software*, v. 169, p. 110693, 2020. ISSN 0164-1212. Disponível em: ⟨http://www.sciencedirect.com/science/article/pii/S0164121220301448⟩.

PECORELLI, F. et al. Comparing heuristic and machine learning approaches for metric-based code smell detection. In: *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019. (ICPC '19), p. 93–104. Disponível em: ⟨https://doi.org/10.1109/ICPC.2019.00023⟩.

PUNTER, T. et al. Conducting on-line surveys in software engineering. In: *International Symposium on Empirical Software Engineering (ISESE)*. [S.l.]: IEEE, 2003. p. 80–88.

RADJENOVIć, D. et al. Software fault prediction metrics: A systematic literature review. *Information and Software Technology (IST)*, v. 55, n. 8, p. 1397 – 1418, 2013. ISSN 0950-5849.

R.AL-MSIE'DEEN et al. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In: *2013 IEEE 14th International Conference on Information Reuse Integration (IRI)*. [S.l.]: IEEE, 2013. p. 586–593.

RIEHLE, D.; GROSS, T. Role model based framework design and integration. ACM, New York, NY, USA, p. 117–133, 1998.

RIGBY, P. C.; BIRD, C. Convergent contemporary software peer review practices. In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2013. p. 202–212. ISBN 978-1-4503-2237-9. Disponível em: ⟨http://doi.acm.org/10.1145/2491411.2491444⟩.

ROMANO, J. et al. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In: *Annual meeting of the Southern Association for Institutional Research*. [S.l.]: Citeseer, 2006. p. 1–51.

ROY, C. K.; CORDY, J. R.; KOSCHKE, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Elsevier, Amsterdam, The Netherlands, The Netherlands, v. 74, n. 7, p. 470–495, maio 2009. ISSN 0167-6423. Disponível em: ⟨http://dx.doi.org/10.1016/j.scico.2009.02.007⟩.

ROZENBERG, D. et al. Comparing repositories visually with repograms. In: *Working Conference on Mining Software Repositories (MSR)*. [S.l.]: IEEE, 2016. p. 109–120.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified modeling language reference manual*. [S.l.]: Pearson Higher Education, 2004.

RUTHRUFF, J. R. et al. Predicting accurate and actionable static analysis warnings: An experimental approach. In: *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2008. p. 341–350. ISBN 978-1-60558-079-1. Disponível em: ⟨http://doi.acm.org/10.1145/1368088.1368135⟩.

SANTOS, C.; DÓSEA, M.; SANT'ANNA, C. ContextLongMethod: Uma Ferramenta Sensível à Arquitetura para Detecção de Métodos Longos. In: *In Sessão de Ferramentas (Tools). Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*. Maringá: [s.n.], 2016.

SANTOS, J. A. M. et al. A systematic review on the code smell effect. *Journal of Systems and Software*, v. 144, p. 450–477, 2018. ISSN 0164-1212.

SCHUMACHER, J. et al. Building empirical support for automated code smell detection. In: *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2010.

SEAMAN, C. B. Qualitative methods in empirical studies of software engineering. *Transactions on Software Engineering (TSE)*, v. 25, p. 557–572, 07 1999. ISSN 0098-5589. Disponível em: ⟨doi.ieeecomputersociety.org/10.1109/32.799955⟩.

SHARMA, T.; FRAGKOULIS, M.; SPINELLIS, D. House of cards: Code smells in open-source c# repositories. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.]: IEEE, 2017. p. 424–429.

SHARMA, T.; SPINELLIS, D. A survey on software smells. *Journal of Systems and Software (JSS)*, v. 138, p. 158 – 173, 2018. ISSN 0164-1212.

SHATNAWI, R. et al. Finding software metrics threshold values using roc curves. *Journal of Software Maintenance and Evolution: Research and Practice*, v. 22, n. 1, p. 1–16, 2009. Disponível em: ⟨https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.404⟩.

SHESKIN, D. J. *Handbook of Parametric and Nonparametric Statistical Procedures*. 4. ed. [S.l.]: Chapman & Hall/CRC, 2007. ISBN 1584888148, 9781584888147.

SINGER, J. et al. An examination of software engineering work practices. In: *First Decade High Impact Papers (CASCON)*. [S.l.]: ACM, 2010. p. 174–188.

SJOBERG, D. I. et al. Quantifying the Effect of Code Smells on Maintenance Effort. *Transactions on Software Engineering (TSE)*, v. 39, n. 8, p. 1144–1156, aug 2013.

SMITH, E. et al. Improving developer participation rates in surveys. In: *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. [S.l.]: IEEE, 2013. p. 89–92.

SOBRINHO, E. V. d. P.; LUCIA, A. D.; MAIA, M. d. A. A systematic literature review on bad smells — 5 w's: which, when, what, who, where. *Transactions on Software Engineering (TSE)*, IEEE, p. 1–1, 2018. ISSN 0098-5589.

SOH, Z. et al. Do code smells impact the effort of different maintenance programming activities? In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.]: IEEE, 2016. v. 1, p. 393–402.

SOKOL, F. Z. et al. Metricminer: Supporting researchers in mining software repositories. In: *Source Code Analysis and Manipulation (SCAM)*. [S.l.]: IEEE, 2013. p. 142–146.

SOUSA, L. et al. When are smells indicators of architectural refactoring opportunities: A study of 50 software projects. In: *International Conference on Program Comprehension (ICPC)*. New York, NY, USA: [s.n.], 2020. p. 354–365. ISBN 9781450379588. Disponível em: ⟨https://doi.org/10.1145/3387904.3389276⟩.

SOUZA, L. B. L. d.; MAIA, M. d. A. Do software categories impact coupling metrics? In: *Working Conference on Mining Software Repositories (MSR)*. [S.l.]: IEEE, 2013. p. 217–220. ISBN 978-1-4673-2936-1. ISSN 21601852.

TAIBI, D.; JANES, A.; LENARDUZZI, V. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, v. 92, p. 223 – 235, 2017. ISSN 0950-5849.

TANAKA, T. et al. Improvement of software process by process description and benefit estimation. In: *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 1995. p. 123–132. ISBN 0-89791-708-1. Disponível em: ⟨http://doi.acm.org/10.1145/225014.225026⟩.

TIBERMACINE, O.; TIBERMACINE, C.; CHERIF, F. A Practical Approach to the Measurement of Similarity between WSDL-based Web Services. *Revue des Nouvelles Technologies de l'Information*, p. 3–18, 2014.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

TSANTALIS, N. et al. Accurate and efficient refactoring detection in commit history. In: *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2018. p. 483–494. ISBN 978-1-4503-5638-1.

TUFANO, M. et al. When and why your code starts to smell bad (and whether the smells go away). *Transactions on Software Engineering (TSE)*, v. 43, n. 11, p. 1063–1088, Nov 2017. ISSN 0098-5589.

VALE, G. et al. Defining metric thresholds for software product lines: A comparative study. In: *International Conference on Software Product Line (SPLC)*. New York, NY, USA: ACM, 2015. p. 176–185. ISBN 978-1-4503-3613-0.

VALE, G.; FERNANDES, E.; FIGUEIREDO, E. On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal*, Springer, p. 1–32, 2018.

VALE, G. A. D.; FIGUEIREDO, E. M. L. A Method to Derive Metric Thresholds for Software Product Lines. In: *Brazilian Symposium on Software Engineering (SBES)*. [S.l.: s.n.], 2015. p. 110–119. ISBN 978-1-4673-9272-3.

VASSALLO, C. et al. Context is king: The developer perspective on the usage of static analysis tools. In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.]: IEEE, 2018. p. 38–49.

WANG, S. et al. Concern localization using information retrieval: An empirical study on linux kernel. In: *Working Conference on Reverse Engineering (WCRE)*. [S.l.]: IEEE, 2011. p. 92–96. ISSN 2375-5369.

WILKINSON, R.; HINGSTON, P. Using the cosine measure in a neural network for document retrieval. In: *International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. New York, NY, USA: ACM, 1991. p. 202–210. ISBN 0-89791-448-1.

WIRFS-BROCK, R.; MCKEAN, A. *Object design: roles, responsibilities, and collaborations*. [S.l.]: Addison-Wesley Professional, 2003.

WOHLIN, C. et al. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, 2012. ISBN 3642290434, 9783642290435.

WU, Y.; MAR, L. W.; JIAU, H. C. Codocent: Support api usage with code example and api documentation. In: *International Conference on Software Engineering Advances (ICSEA)*. [S.l.]: IEEE, 2010. p. 135–140.

YAMASHITA, A. How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study. In: *International Conference on Software Maintenance (ICSM)*. [S.l.]: IEEE, 2013. p. 566–571. ISBN 978-0-7695-4981-1. ISSN 1063-6773.

YANG, D. *Java Persistence with JPA*. [S.l.]: Outskirts Press, 2010.

YOON, K.; KWON, O.; BAE, D. An approach to outlier detection of software measurement data using the k-means clustering method. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.]: IEEE, 2007. p. 443–445. ISSN 1949-3770.

ZHANG, F. et al. How does context affect the distribution of software maintainability metrics? In: *International Conference on Software Maintenance (ICSE)*. [S.l.]: IEEE, 2013. p. 350–359. ISSN 1063-6773.

ZHENG, J. et al. On the value of static analysis for fault detection in software. *Transactions on Software Engineering (TSE)*, v. 32, n. 4, p. 240–253, April 2006. ISSN 0098-5589.

ZHU, H.; ZHOU, M. Roles in information systems: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, v. 38, n. 3, p. 377–396, May 2008. ISSN 1094-6977.