Pontifícia Universidade Católica
do Rio de Janeiro

Anderson José Silva de Oliveira

On the Prioritization of Design-Relevant Smells

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informáticada PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
April 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Anderson José Silva de Oliveira

On the Prioritization of Design-Relevant Smells

Dissertation presented to the Programa de Pós–graduação em Informáticada PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Alessandro Fabricio Garcia
Advisor
Departamento de Informática – PUC-Rio

Prof. Marcos Kalinowski
Departamento de Informática – PUC-Rio

Prof.ª Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Rio de Janeiro, April 24th, 2019

Anderson José Silva de Oliveira

The author received his Bachelor degree in Computer Science from the Federal University of Alagoas (UFAL), Brazil, in 2016. During his undergraduate degree, he did an exchange program (2014) at the Eötvös Loránd University (ELTE), Hungary. His main research interests are Design Problems, Software Architecture, Code Smells, Software Maintenance, and Software Evolution.

Bibliographic data

Oliveira, Anderson José Silva de

On the Prioritization of Design-Relevant Smells / Anderson José Silva de Oliveira; advisor: Alessandro Fabricio Garcia. – 2019.
87 f: il. color. ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2019.
Inclui bibliografia

1. Informática – Teses. 2. Problemas de Design. 3. Priorização. 4. Anomalias de Código. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First and foremost, I would like to thank my family. My mother, Ioneide Silva de Oliveira, who gave me all the support and motivation to the decisions that I make. My father, Antonio Oliveira, for all his advice and wisdom that he passed to me. My sister, Leilane Silva Oliveira, for all her advice and motivation.

I would like to thank my advisor, Alessandro Garcia, for all the knowledge that he passed to me during this period. I am deeply grateful for all the professional growth that I had in these two years, which is mainly thanks to Alessandro. Also, I would like to thank him for all the patience with me. It is a great opportunity to work close to him, which increased significantly my skill both as a researcher as well as a person.

I especially thank my friend, Leonardo Sousa, for all his friendship and support during this time. Since the beginning, he was always available to support me, giving me the best advice and always encouraging me. Also, I thank my friend Diego Cedrim, who (with Leonardo) gave me all the support needed upon my arrival in Rio, both in terms of logistics as well as friendship. In addition, I thank my friend Willian Oizumi, for all his support and advice.

I would like to thank my girlfriend Marina Baumgratz, for all the support, encouragement, and caring during this period.

I would like to thank my colleagues (and former colleagues) of OPUS Research group: Ana Carla Bibiano, Anderson Uchôa, Alexander Lopez, Anne Benedicte, Daniel Tenorio, Eduardo Fernandes, João Neves, Luiz Carvalho, Rafael de Mello, and Roberto Oliveira. Also, I would like to thank Baldoino Neto and all the UFAL team, who were always responsive and supportive on our collaborations.

I am also grateful for the members of my dissertation jury: Prof. Marcos Kalinowski and Prof.ª Simone Diniz Junqueira Barbosa, who dedicated time and knowledge to evaluate this dissertation. In addition, I thank all the professors from PUC-Rio for their contributions during this period.

I am grateful for the financial support given by CNPq and PUC-Rio, which made my research possible.

Abstract

Oliveira, Anderson José Silva de; Garcia, Alessandro Fabricio (Advisor). **On the Prioritization of Design-Relevant Smells**. Rio de Janeiro, 2019. 87p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software systems are likely to face what is called design problems. A design problem is the result of bad decisions that can affect some important quality attributes of the software system such as maintainability, performance and the like. Given the typical lack of design documentation, developers have to rely on implementation-level symptoms to identify and remove design problems. An implementation-level symptom usually manifests as a code smell, a micro-structure in the program possibly indicating the presence of (or part of) a design problem. Large programs have hundreds or thousands of program elements (packages, classes, interfaces, and the like) in which a significant proportion is affected by smells. However, many of these smells may bear no relationship with design problems, *i.e.* they are not design-relevant smells. Then, it becomes hard and time-consuming to prioritize smelly program elements being suspects of having a design problem. Unfortunately, the literature fails to provide developers with heuristics to support the prioritization of these suspicious program elements. In this context, this dissertation reports two studies aimed at assisting in the elaboration of such prioritization heuristics. The goal of these heuristics is to locate a short (high priority) list of smelly program elements, which are suspects of having design-relevant smells. Our first study consists of a qualitative analysis on recurring criteria used by developers, in practice, to prioritize elements suspicious of having design problems. Based on these criteria, we derived a preliminary suite of prioritization heuristics. Our second study focused on the evaluation of the proposed heuristics. As a result, we found that two out of nine heuristics reached the best results in precision. The best heuristics are based on two criteria: smell diversity and smell granularity. Our findings suggest that we were able to derive a first promising approach to support developers in prioritizing elements with design-relevant smells.

Keywords

Design Problems; Prioritization; Code Smells.

Resumo

Oliveira, Anderson José Silva de; Garcia, Alessandro Fabricio. **Priorização de Anomalias de Código Relevantes ao Projeto dos Sistemas de Software**. Rio de Janeiro, 2019. 87p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Sistemas de software provavelmente enfrentarão os chamados problemas de projeto. Um problema de projeto é o resultado de más decisões que podem afetar alguns atributos de qualidade importantes do sistema de software, como manutenção, desempenho e afins. Dada a típica falta de documentação do projeto, os desenvolvedores precisam confiar em sintomas que aparecem a nível de implementação para identificar e remover problemas de projeto. Um sintoma a nível de implementação geralmente se manifesta como uma anomalia de código, que se trata de uma microestrutura no programa possivelmente indicando a presença de (ou parte de) um problema de projeto. Grandes programas possuem centenas ou milhares de elementos (pacotes, classes, interfaces e afins) nos quais uma proporção significativa é afetada por anomalias. No entanto, muitas dessas anomalias não possuem relação com problemas de projeto, em outras palavras, elas não são anomalias relevantes ao problema de projeto. Desse modo, torna-se difícil e demorado priorizar os elementos anômalos do programa que são suspeitos de terem problema de projeto. Infelizmente, a literatura não fornece aos desenvolvedores heurísticas que auxiliem a priorização destes elementos de projeto suspeitos. Neste contexto, esta dissertação reporta dois estudos que objetivam auxiliar na elaboração de tais heurísticas, visando auxiliar o desenvolvedor nas decisões de priorização. O objetivo destas heurísticas é localizar uma pequena lista de elementos suspeitos de terem anomalias de código relevantes ao problema de projeto. Nosso primeiro estudo consiste em uma análise qualitativa para determinar os critérios utilizados pelos desenvolvedores para a priorização de elementos suspeitos de terem problemas de projeto. Com base nesses critérios, derivamos um conjunto preliminar de heurísticas de priorização. Nosso segundo estudo centrou-se na avaliação destas heurísticas. Como resultado, descobrimos que duas das nove heurísticas alcançaram os melhores resultados de precisão. As melhores heurísticas são baseadas em dois critérios: diversidade de anomalias e granularidade das anomalias. Nossas descobertas sugerem que fomos capazes de obter uma primeira abordagem promissora para apoiar os desenvolvedores na priorização de elementos com anomalias de código relevantes ao projeto de software.

Palavras-chave

Problemas de Design; Priorização; Anomalias de Código.

Table of contents

List of figures

List of tables

1
Introduction

Software design is the result of a series of decisions made during the software development process (Tang *et al.* 2016). These decisions can come in many ways, from official meetings with developers to daily discussions during the software development (Baker, Hoek and Petre 2012). Even a single design decision may impact a significant part of the system. Therefore, design decisions are fundamental in the software development process (Gamma *et al.* 1995). In particular, they are fundamental to define how the software will be organized into components and how they will communicate with each other. Combined, the design decisions will drive how the system will be developed and maintained in the future. Indeed, modifications in the design are prone to occur frequently since developers cannot fully predict future changes, such as the changes in the system's requirements and changes in the technologies.

These unpredictable modifications also happen due to inappropriate design decisions that are made along with software development. These inappropriate decisions are called *design problems* when they affect negatively quality attributes of the software system such as maintainability and performance (Garcia *et al.* 2009b). As design problems are the result of inappropriate design decisions, they can affect a large part of the software system. The most critical design problems are often those affecting how the system is modularly organized in subsystems and components, and how they interact with each other. Thus, in this dissertation, we focused on design problems that can be identified through the analysis of code elements affected by inappropriate design decisions related to the system modularity. These decisions may be related to, for instance, how the system is decomposed into components, and how they communicate with each other. We are interested in these design problems since they have been associated with major maintenance effort (Schach *et al.* 2002, Garcia *et al.* 2009, Yamashita and Moonen 2012).

An example of design problem occurs when multiple components are responsible for realizing the same high-level functionality, *i.e.,* the realization of the same functionality is scattered over multiple components instead of being modularized in a single one. This type of design problem is named *Scat-*

tered Concern (Garcia *et al.* 2013). Many design problems, like this one, are particularly harmful as they crosscut components of the system, thereby hampering multiple quality requirements, such as maintainability, understandability, and robustness. Other examples of design problems are *Cyclic Dependency* (Parnas 1978), *Fat Interface* (Martin and Martin 2006) and *Concern Overload* (Macia *et al.* 2012).

Once identified, design problems need to be removed from the software system. Software refactoring is the basic means to remove design problems (Paixao *et al.* 2017, Lin *et al.* 2016). Refactoring consists of one or more program transformations used to improve the structural quality (Fowler 1999). Before refactoring, developers have to *prioritize* elements that are likely to contain design problems. For this purpose, they can search for elements that manifest *symptoms* (Sousa *et al.* 2017) of design problems in the system. A symptom is a partial sign of a design problem that manifests in the source code. Given the typical lack of documentation (Kaminski 2007), developers have to rely on implementation-level symptoms to identify and remove design problems (Sousa *et al.* 2017). An implementation-level symptom usually manifests as a code smell (Fowler 1999), a microstructure in the program possibly indicating the presence of (or part of) a design problem.

Unfortunately, many code smells may not help the developer to identify a design problem (Macia *et al.* 2012a, Macia *et al.* 2012b, Macia *et al.* 2012, Oizumi *et al.* 2016). The reason is that these smells may be wrong suspects. Thus, only some code smells can be used by the developers to identify design problems, which we call them *design-relevant smells*. Before the identification of a design problem, the developer has to prioritize the relevant smelly elements. A *relevant smelly element* (or relevant element, for short) is one that is affected by at least one design-relevant smell. Unavoidably, developers will have to prioritize relevant elements in the software system, since there are many elements in the system that are not related to design problems.

The prioritization of program elements to find the relevant ones is a challenging task. Among the challenges, there is the fact that large software systems may have hundreds or even thousands of program elements – packages, classes, interfaces, and the like. Then, it becomes hard and time-consuming to prioritize elements being suspects of having a design problem. In addition to this challenge, not all smelly elements contain design-relevant smells. Thus, developers still have to analyze these elements to decide if the smells indicate a design problem or not. Finally, developers often face time constraints, which may force them to reduce the space of search for design-relevant smells to only a few elements in the system.

1.1 Motivation and Problem Statement

In addition to the large size of the software systems, the design problems can affect elements that are scattered in different components (*i.e.*, classes and packages). This scenario forces the developer to focus on critical design problems of the system. A critical design problem is one that developers end up trying to remove from their systems through refactoring (Godfrey and Lee 2000, Gurp and Bosch 2002, MacCormack, Rusnak and Baldwin 2006, Schach *et al.* 2002). Refactoring is a program operation used for the improvement of the code structure of the system (Fowler 1999). Thus, developers have to prioritize program elements that may contain a design problem. For that matter, they can use the implementation-level symptoms of design problems (Sousa *et al.* 2017), as previously discussed.

Even though a design problem may be associated with more than one type of symptom, we will focus only on code smells in this dissertation. We chose code smells as they can also signal other symptoms. For instance, let us consider the symptom which represents poor structural quality attributes, such as coupling. We can recognize that a class has high coupling using the code smell *Intensive Coupling*. This smell can also signal other symptoms, including the violation of the low coupling and information hiding principles, as well as poor testability and maintainability. Additionally, code smells are one of the most investigated symptoms in the literature (Macia *et al.* 2012a, Macia *et al.* 2012b, Moha *et al.* 2010, Oizumi *et al.* 2016, Oizumi *et al.* 2017, Sousa *et al.* 2018, Palomba *et al.* 2014, Yamashita and Moonen 2013). Thus, from now on, when we cite symptoms, we are referring to code smells.

The prioritization of program elements that may contain a design problem is not trivial. There are many reasons that make this prioritization, based on code smells, difficult. As previously mentioned, not all code smells are related to a design problem. Thus, the developer may find himself analyzing elements with smells that are not relevant to identify a specific design problem. For instance, when a code smell is introduced in the source code due to the use of APIs or frameworks, it does not necessarily indicate the presence of design problems. Given this context, finding the relevant elements to the identification of design problems is cumbersome. Large systems tend to have multiple design problems spread over multiple elements. In addition, as design documentation is often unavailable or outdated (Trifu and Reupke 2007, Kaminski 2007), developers have to analyze the source code of these elements to find the design-relevant smells.

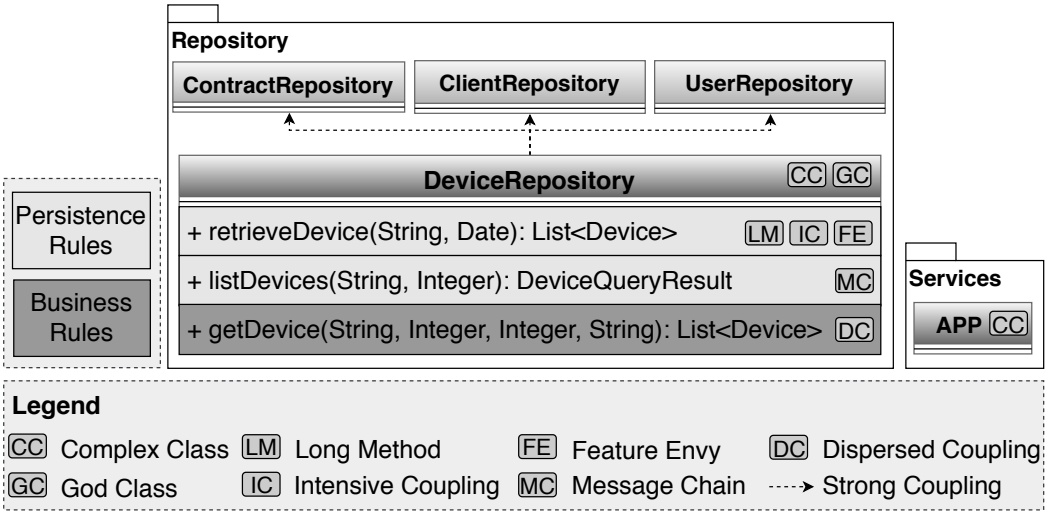

Figure 1.1: Design Problems Affecting the *DeviceRepository* and *APP* classes

To illustrate how complex it is to locate relevant elements, let us consider a system, called S1. This system, shown in Figure 1.1, manages loans and sales of printer devices. After three years of its deployment, developers noticed a high effort to maintain and evolve S1. A reason was the presence of design problems. Even though developers knew about these problems, they mentioned the difficulty to identify them, especially because they did not know which elements and smells they would have to focus their effort. In this example, the developer could be able to identify the design problem if he knew what elements and smells he should prioritize. However, this scenario is far from being the typical one that developers face (Sousa *et al.* 2017).

In most cases, developers do not know in which part of the program they should start the analysis, neither what smells, among the thousands, they should focus on. Given the influence of design problems, code smells appeared in different elements in S1, such as the `APP` and `DeviceRepository` classes. Figure 1.1 details the *DeviceRepository* class and presents the smell affecting the *APP* class. These smelly elements can be the ones that developers can analyze first to find relevant elements to the identification of design problems.

Let us consider the `DeviceRepository` class, which has several different types of smells. Several of them – the design-relevant ones – indicate that there is something wrong with the class implementation. They are presented in Figure 1.1. We discuss some of them here. For instance, the God Class smell indicates that the class may be implementing more than one functionality, which makes the class and its methods to become complex and large – represented by the Complex Class and Long Method smells. Feature Envy is another evidence that the class may be implementing another functionality

since there is a method that is more interested in other classes than its own. Consequently, other methods intensively access other classes, which is a symptom represented by the Dispersed Coupling smell.

As a consequence, the `DeviceRepository` class should be prioritized by the developers. This class is being affected by various design-relevant smells, which indicate the occurrence of a design problem known as *Concern Overload* (Li *et al.* 2014). This problem occurs when an element fulfills many non-cohesive functionalities. Indeed, this is what happens with `DeviceRepository`, which mistakenly implements both persistence and business functionalities. In this example, we can observe that the developer would prioritize this smelly class, for instance, by relying on the fact that it is being affected by many smells.

Now let us consider the `APP` class, which also contains the code smell Complex Class. At first sight, a developer can think that this class is a relevant element, due to its high complexity. However, in a deeper analysis, a developer will notice that the class does not have design problems. First, its complexity is due to the framework used in the implementation. Second, the framework makes the class handle multiple functionalities, which gives the wrong impression of the presence of a design problem. In summary, the element should not have been prioritized by the developer. This scenario illustrated by the `APP` class is one where the developer wasted time analyzing an element that is not relevant to the identification of design problems.

Analyzing smells that are not related to a design problem can be time-consuming and can mislead developers to identify a design problem that does not exist. Unfortunately, `APP` is just one of the many elements on the system that is not related to design problems. Indeed, a system may contain several elements unrelated to any design problem. For example, let us consider the Apache OODT system (Mattmann *et al* 2006). This project deals with the development, management, and storage of scientific data. It has a total of 1473 classes and interfaces, from which 290 are relevant elements (19.68%) according to its original developers. Consequently, due to time constraints and a large number of elements, developers have to prioritize only a few relevant elements that are likely to have design problems. In other words, developers need heuristics that support this prioritization. Therefore, our problem statement is summarized as follows.

> Developers need to prioritize a short list of smelly program elements, which are suspects of having design-relevant smells.

1.2 State of the Art and its Limitations

In this dissertation, we aim at investigating how to support developers on the prioritization of relevant elements (Section 1.3). Some previous studies investigated how to provide, to a certain extent, support of this nature to developers. This section presents the state of the art and limitations of existing studies. With an understanding of these studies and their limitations, we aim at extending the state of the art by better supporting the developers on the prioritization of relevant elements. We classify and characterize our related work in two categories: *identification* and *prioritization* of design problems.

Identification of Design Problems. Diverse studies investigated the use of code smells to identify design problems. For instance, Macia *et al.* (Macia *et al.* 2012b) studied the relevance of code smells to identify design problems. They investigated to what extent smells, detected through automatic detection strategies, were related to some types of design problems. They identified that many of the code smells detected automatically were not strong indicators of design problems.

Moha *et al.* (Moha *et al.* 2010) proposed a method, implemented as a tool called DECOR, that describes the steps on the identification of code smells and design problems. This method, however, depends on a domain-specific language to specify the detection of code smells and design problems (Moha, Gueheneuc and Leduc 2006). Li *et al.* (Li *et al.* 2014), in their study, investigated if some modularity metrics indicate design problems. They conducted a multiple case study with thirteen companies. They were able to identify two metrics that can be used to identify design problems namely index of package changing impact (IPCI) and index of package goal focus (IPGF). However, they did not investigate if developers eventually use these metrics in practice. Moreover, their work does not explicitly cover the prioritization of relevant smelly elements.

Oizumi *et al.* (Oizumi *et al.* 2016) investigated to what extent groups of inter-related code smells, named agglomerations, could help developers to identify design problems. They found that some types of agglomerations are strong indicators of design problems. However, they also did not conduct an investigation with developers. In this sense, Sousa *et al.* investigated how software developers identify design problems in practice (Sousa *et al.* 2018). They discovered that developers tend to combine multiple symptoms to identify design problems. The authors did not focus on providing an approach to help developers to prioritize design problems. In fact, the existing studies

fell short either in investigating how developers prioritize relevant elements or in proposing support for the prioritization of these elements. In addition, they do not propose any heuristics that could somehow support the developer on the prioritization task.

Prioritization of Design Problems. When it comes to prioritization, there are studies focused on the use of smells to prioritize elements that have design problems (Arcoverde *et al.* 2013, Vidal *et al.* 2016, Guimaraes *et al* 2018). For example, Vidal *et al.* (Vidal *et al.* 2016) proposed and evaluated a set of criteria to prioritize classes based on code smells. These proposed criteria based on implementation and architectural information of the system, which includes the version history of the system. However, their heuristics did not achieve a homogeneous result among all projects – they did not perform well in some projects. Arcoverde *et al.* (Arcoverde *et al.* 2013) presented and evaluated four heuristics for prioritizing design-relevant smells. These heuristics explore characteristics of the software system such as the density in which an element changes and the error density of an element. Considering the average precision of all projects evaluated, they reached a maximum of 72.5%, with four projects.

The work of Guimaraes *et al.*(Guimaraes *et al* 2018) is also focused on prioritizing critical code smells for the identification of design problems. In their study, they evaluate how developers use blueprints and source code to identify design-relevant smells. The main limitation of the approach of Guimaraes *et al.* is the dependence on design blueprints since design information is often outdated or unavailable for many systems. Even though they propose heuristics for the prioritization of design-relevant smells, these studies did not follow a practice-based approach to defining their heuristics. Their approach differs from ours as we had followed a well-defined methodology to observe developers o practice along with the task of prioritizing smelly elements. In addition, most of these approaches rely on software information that is usually not available, such as design blueprints.

1.3 Goal and Research Questions

Given the motivational example (Section 1.1), we observe that developers need support on the prioritization of relevant smelly elements. These prioritized smelly elements can be used afterwards for the identification of design problems. Most of these existing approaches (Section 1.2) do not take into account the task of prioritizing program elements, which is a task that developers have to perform before the identification of design problems. Given this

existing limitation, the goal of this dissertation is to support developers along with the prioritization of relevant elements. This prioritization also allows determining design-relevant smells: when a relevant element is prioritized, it has at least one design-relevant smell. Conversely, when an element is discarded during the prioritization, its code smells are also discarded. These discarded smells are, then, not considered relevant to the design.

To achieve this goal, we investigated the criteria that developers tend to use during the prioritization task. We assume that the observation of developers performing this task will provide us with insights on effective prioritization criteria. Based on these criteria, we proposed a suite of prioritization heuristics. These heuristics are intended to support developers on the prioritization of relevant elements. Starting from these elements, the developer will be able to analyze their code smells, which can be relevant to identify one or more design problems. Therefore, we can summarize our research in terms of the following general research question:

> **General Research Question:** How to support the developers on the prioritization of relevant smelly elements?

To better address this research question, we divided it into the following specific research questions:

> ***SRQ1.:*** *What are the criteria that developers tend to use to prioritize relevant smelly elements?*

To gather the data necessary for answering **SRQ1**, we conducted a study on the prioritization of relevant elements (Chapter 3). We divided this study into two parts. In the first part, we asked the developers to prioritize elements relevant to the identification of design problems in their source code. We are interested to find out criteria that they apply during the prioritization. Their prioritization criteria may rely on intrinsic information of the symptoms. For the sake of illustration, let us consider the code smells. The type of each smell is an example of intrinsic information. However, the information that a developer would have to consider when prioritizing an element may go beyond the type of the code smell. He may also consider the size of the source code directly affected by the code smell, which is also intrinsic information. For example, Long Method is a smell that usually comprehends too many lines of code, as compared to a Message Chain smell, which affects only one statement (Fowler 1999).

Observing how developers prioritize elements will allow us to find these criteria, thus answering SRQ1. To answer this question, we highlight that we asked the developers to perform the prioritization until they identify a design problem. By asking them to conduct the prioritization until the identification of a design problem, we could assure that they would eventually perform actions that are useful for effective prioritization of relevant elements.

In the second part, through qualitative analysis, we identified: (i) five criteria that developers use to prioritize program elements as suspects of being affected by design problems; and (ii) two criteria that developers more frequently use to prioritize relevant elements. One criterion considers the number of smells, while the other considers the diversity of smells located in a suspicious element. Based on these criteria, we derived a suite of heuristics to prioritize relevant elements. An example of proposed heuristic is what we called the *Smell Count* heuristic, which prioritizes elements with the highest number of smells.

We expect that the developers can benefit from the proposed heuristics, potentially saving time and effort that is dedicated to such prioritization task. The example in Section 1.1 illustrates a scenario where the developer can benefit from these heuristics. For instance, let us consider the use of the *Smell Count* heuristic. The application of this heuristic on the system *S1*, from the example, would lead to the prioritization of the *DeviceRepository* class over the other classes since it has the highest number of smells in the system – seven code smells in total. In addition, the heuristic would discard the *APP* class, since it has only one code smell. Thus, developers would not spend effort on inspecting *APP* class in the search for design-relevant smells, which would not lead to the identification of a design problem.

Once the heuristics were proposed, the next step consisted of evaluating them. We evaluated our heuristics regarding their effectiveness in finding relevant smelly elements in a top priority list of N elements ($TOPN$). The effectiveness of the heuristics was assessed in terms of their precision. It is important to highlight that it is expected that not necessarily all the heuristics will be effective. In other words, not all the prioritization criteria, which are employed by developers, are necessarily effective to determine relevant smelly elements. Therefore, we defined the following research question:

> ***SRQ2.:*** *What is the precision of the prioritization heuristics?*

To answer **SRQ2**, we conducted two evaluations (Chapter 4). We decided to perform two evaluations in order to observe how the heuristics would

perform in two different scenarios. First, we evaluated our heuristics with software systems found in the GitHub repository. Second, we evaluated the heuristics with software systems provided by our industry partners. The evaluations differ in two points: (i) the set of systems used in the evaluation, and (ii) how we created the list of elements prioritized by developers.

As a result, we found that two out of nine heuristics reached the best precision results. These two heuristics are named *Diversity* and *Smell Granularity*. These heuristics had a better precision probably because they consider the type of each smell affecting an element. In fact, the *Smell Count* heuristic, which is agnostic to the smell type and prioritizes elements with the highest amount of smell instances, fell short in prioritizing elements with design problems. However, the *Smell Count* heuristic was able to prioritize elements that developers should focus their effort on refactoring them. Based on our results, we concluded that heuristics that consider intrinsic information of a smell, such as the type of smell, are likely to help developers to prioritize elements.

By answering our two specific research questions, we were able to provide developers with promising heuristics to prioritize relevant elements. When a developer prioritizes a relevant element, he is automatically prioritizing its design-relevant smells. In this way, when we present the prioritized relevant elements, we also avoid showing code smells that are not relevant to the identification of a specific design problem. The prioritization can save time that would be otherwise spent in (i) manually locating the priority elements, and (ii) manually inspecting irrelevant or worthless information from these elements unrelated to a design problem.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. **Chapter 2** describes the basic concepts that will be used during the dissertation. **Chapter 2** also outlines the related works and the current state of the art on the context of this dissertation. **Chapter 3** present the study where we collected the criteria used by developers on the prioritization of smelly elements. **Chapter 4** presents the evaluation of our proposed prioritization heuristics, created based on the criteria found in the previous study, with two sets of systems. **Chapter 5** presents the final remarks of this dissertation, with our main contributions and the future works.

2
Background and Related Work

This chapter provides the background and related work of this dissertation. **Section 2.1** introduces the concepts associated with software design and design problems. **Section 2.2** presents a discussion on the prioritization of relevant smelly elements, *i.e.*, those containing one or more design-relevant smells. **Section 2.3** discusses the related work in terms of: (i) the relationship between design problems and code smells, (ii) identification of design problems with various types of symptoms, and (iii) the prioritization of relevant smelly elements. **Section 2.4** summarizes this chapter.

2.1
Software Design and Design Problem

Software design can be defined as the result of a series of decisions made during the software development process (Tang *et al.* 2016). A design decision can be defined as a description of the choice and considered alternatives that (partially) realize one or more requirements (van der Ven *et al.* 2006). These decisions will help any given software system in achieving quality requirements, such as maintainability and reusability. The software design can be divided into two main stages: (i) early software design, which focuses on organizing the software system into components, interfaces and how they communicate with each other; and (ii) detailed design, where more specific decisions are taken for each component previously defined (Booch 2004).

Ideally, design decisions should be taken before the implementation stage of the software system. However, this is not what happens in practice since, during the evolution of the software system, some decisions need to be reconsidered or even left aside (Baker, Hoek and Petre 2012). These unexpected changes happen due to what we call design problems. A design problem happens when one or more design decisions negatively impact the quality attributes of a software system (Garcia *et al.* 2009b). The occurrence of design problems can also cause issues such as expensive costs for maintainability in the future, the re-design of the system or even its discontinuity (MacCormack, Rusnak and Baldwin 2006, Godfrey and Lee 2000, Schach *et al.* 2002, Gurp and Bosch 2002). In addi-

tion, design problems can affect critical locations of a system, such as the core elements (Bass, Clements and Kazman 2003) that implement the main software functionalities.

Fat Interface is an example of a design problem. This design problem occurs when a developer aggregate too many responsibilities in a single interface, turning the interface highly coupled with other modules (Martin and Martin 2006). A *Fat Interface* usually emerges from inappropriate decisions related to abstraction and separation of concerns. A concern comprises anything that the stakeholders of a software project may want to consider as a conceptual unit (Robillard and Murphy 2007). Thus, the violation of the *Separation of Concern Principle* occurs when the stakeholders decompose the system concerns into dependents parts that should be independent (Parnas 1972, Dijkstra 1997). When neglected, a *Fat Interface* may cause a high effort on the maintenance of software systems (Macia *et al.* 2012b). This happens since, for each change on the interface, the developer has to change multiple classes that are using or implementing the interface. These classes are affected by the changes because they are coupled with the interface, even though they do not necessarily have a conceptual relation to the changes performed on the interface.

Other examples of design problems are *Scattered Concern* (Garcia *et al.* 2009b), *Ambiguous Interface* (Garcia *et al.* 2009b), and *Cyclic Dependency* (Martin and Martin 2006). As mentioned in Chapter 1, in the context of this dissertation, we decided to focus on design problems related to violations of modularity principles, such as separation of concerns and abstraction. We focused on these design problems because they have been associated with major maintenance effort (Schach *et al.* 2002, Garcia *et al.* 2009, Yamashita and Moonen 2012). They may hamper the introduction of new features, cause major re-engineering or even lead to the discontinuation of a system. In fact, a recent study showed that many changes are either rejected or reverted in software projects due to these design problems (Oliveira, Valente and Terra 2016). Another reason to focus on these design problems is because they occur frequently in the software systems, according to the managers of these systems, studied in the context of this dissertation (Chapter 3 and Chapter 4). In addition, Table 2.1 presents the details of all design problems covered in this dissertation. We focused on this catalog since those design problems cover multiple violations of modularity, such as abstraction and separation of concerns. The first column presents the design problem and the second column presents its description.

The identification and removal of design problems of the software system

Table 2.1: Design Problems Description

Design Problem	Description
Ambiguous Interface	It refers to interfaces representing the abstraction that does not reveal which services it offers.
Cyclic Dependency	Two or more components that directly or indirectly depend on each other
Component Overload	Component responsible for realizing two or more unrelated concerns
Concern Overload	Abstraction that fulfills too many concerns
Fat Interface	Interface with multiple non-cohesive services
Incomplete Abstraction	An element that does not support a responsibility completely in their enclosing component
Misplaced Concern	An element that implements a concern, which is not the predominant one of their enclosing component
Scattered Concern	Multiple components responsible for realizing a crosscutting concern
Unused Abstraction	Abstraction that is never used by other components
Unwanted Dependency	Dependency that violates an intended design rule

present some challenges, mainly due to the time constraints and the effort to perform these tasks. Given the challenges, developers often have to focus on a few critical design problems. We consider a critical design problem as one where the developer has to dedicate some effort in refactoring to remove it.

Some design problems can be scattered through multiple elements of the system (*e.g. Scattered Concern, Fat Interface, Component Overload*). Hence, developers have to prioritize elements that are likely to have critical design problems. In addition, as design documentation is often unavailable or outdated (Trifu and Reupke 2007, Kaminski 2007), developers have to analyze the source code of these elements to find indicators of design problems. These indicators, named symptoms (Sousa *et al.* 2017), are partial signs of the presence of a design problem. Developers can rely on code smells, in which are the most used and investigated symptom (Yamashita and Moonen 2013, Macia 2013, Macia *et al.* 2012b, Moha *et al.* 2010, Oizumi *et al.* 2016, Vidal *et al.* 2016).

2.2 Prioritizing Design Problems with Design-Relevant Smells

In this section, we explain about the code smells (Section 2.2.1) and their role in the prioritization of design problems (Section 2.2.2). Then, we present an example of how they can be used during the prioritization task (Section 2.2.3).

2.2.1 Code Smells

A code smell is a micro-structure on the source code that can serve as an indicator of a design problem (Fowler 1999). The code smells can be divided in terms of its scope on the system. The first group (or level) of code smells, consist of method-level smells (e.g., Long Parameter List, Long Method). This type of code smell affects the scope of every single method. Second, we have the

Table 2.2: Code Smells Description

Level	Code Smell	Description
Method-level	Brain Method	Long and complex method that centralizes the intelligence of a class
	Dispersed Coupling	A method that accesses many elements dispersed among many classes
	Feature Envy	A method that is more interested in another class than its own class
	Intensive Coupling	A method that is tight coupled with other methods, and these coupled methods are defined in the context of few classes
	Long Method	A method that is long in terms of lines of code
	Long Parameter List	A method that have a long list of parameters
	Message Chain	A long chain of methods is called to implement a class functionality
Class-level	Brain Class	Long and complex class that centralizes the intelligence of the system
	Class Data Should Be Private	A class exposing its fields
	Complex Class	A class have at least one method with high cyclomatic complexity
	Data Class	A class that contains only fields and accessors methods
	God Class	A class that centralizes the system functionalities
	Lazy Class	A class with small dimension, few methods and low complexity
	Refused Bequest	A class redefining most of the inherited methods
	Spaghetti Code	A class implement complex methods that interact between them, with no parameters and using global variables
	Speculative Generality	A class declared as abstract that have very few children classes using its methods
Application-level	Shotgun Surgery	When a change performed on it demands many little changes to several different classes

class-level smells (e.g., Complex Class and God Class). As the name suggests, they affect the whole class. Last, we have the application-level smells (e.g., Shotgun Surgery and Divergent Change). They have this classification since they can affect multiple elements, which can be scattered through different components on the application. Since the application-level smells affect several elements, they are the most severe ones in a system. Table 2.2 presents 17 types of code smells considered in this study. They were chosen since they are commonly studied in the literature and are related to design problems.

The relation between code smells and design problems are widely studied (Trifu and Marinescu 2005, MacCormack, Rusnak and Baldwin 2006, Moha *et al.* 2010, Macia *et al.* 2012a, Macia *et al.* 2012b, Palomba *et al.* 2014, Oizumi *et al.* 2016). Multiple studies use the code smells as indicators of design problems, especially because they tend to occur on locations with design problems (Yamashita *et al.* 2015, Macia *et al.* 2012a, Oizumi *et al.* 2015, Oizumi *et al.* 2016). In addition, to identify a design problem using the code smells, the developer can also use them to prioritize the relevant elements likely to have a critical design problem, the so-called *relevant elements*. Next, we provide the basic terminology for the prioritization process.

2.2.2 Prioritizing Task: Meta Model

As previously mentioned (Section 1.1), prioritization of relevant elements can be a challenging task. This section provides the basic concepts associated with the task of prioritizing relevant elements. To better explain the prioritization of relevant elements, consider the meta-model shown in Figure 2.1. The meta-model presents all the concepts related to the prioritization of relevant

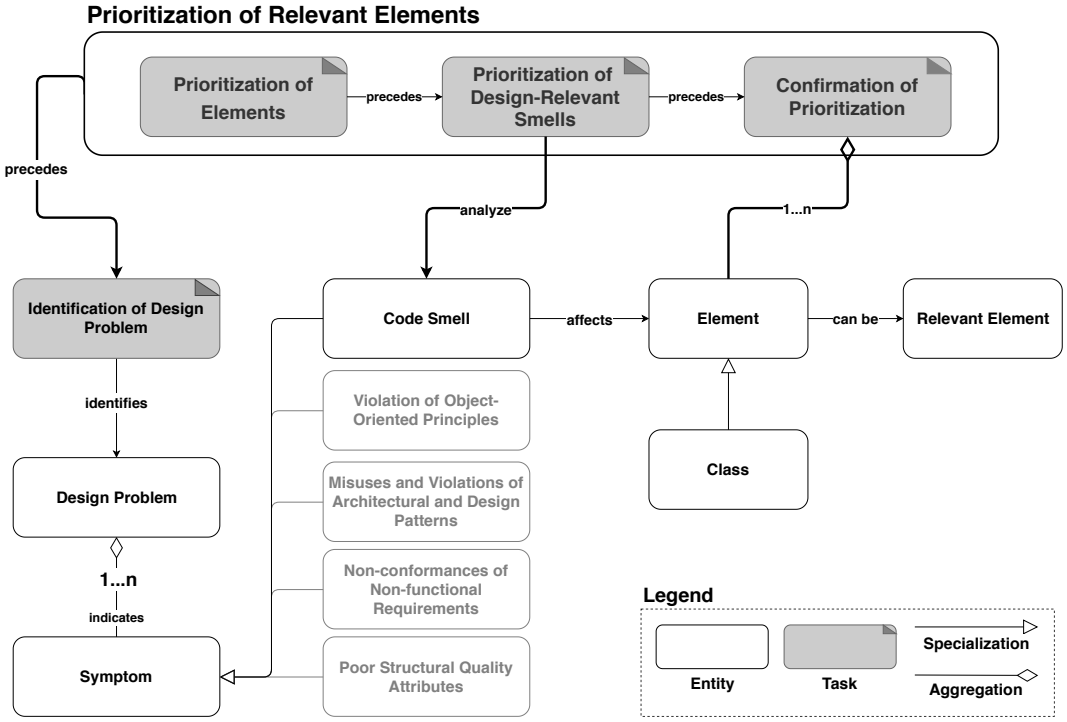

Figure 2.1: The Prioritization Task: A Meta Model with Basic Concepts

elements. In addition, it helps us to highlight the scope of the dissertation. For instance, a design problem can be indicated by one or more types of symptoms, such as violation of object-oriented principles and poor structural quality attributes (Sousa *et al.* 2018). However, in this dissertation, we focus only on the smells affecting code elements (Section 1.1). All the process of prioritizing relevant elements precedes the identification of design problems. Otherwise, the developers would have to analyze many elements not relevant to design problems, spending the time that would be better applied in the identification of other design problems. We describe below this prioritization of relevant elements in detail.

As we can see in the meta-model (Figure 2.1), the prioritization of relevant elements follows three main phases: (i) prioritization of elements, (ii) prioritization of design-relevant smells, and (iii) confirmation of prioritization. The *prioritization of elements* phase occurs when the developer performs a pre-filtering of the elements he thinks that should be analyzed. Among the many elements that he ends up analyzing, many of them may not be related to any design problem. Even if the element contains multiple smells, it may be the case where these smells are not relevant to the identification of design problems.

The *prioritization of design-relevant smells* phase occurs once the developer chooses an element to analyze. Among the smells that this element

contains, he needs to decide whether they are relevant or not in the identification of a design problem. The smells used for the identification of design problems are the so-called design-relevant smells. Once defined that the element contains design-relevant smells, the developer enters the *confirmation of prioritization* phase. In this third phase, he confirms whether the previously selected element is a relevant one to the identification of a design problem or not. Once he finishes this process of prioritization, then the identification of the design problem begins, as presented in the Figure 2.1.

2.2.3 Prioritization Task: A Practical Example

To illustrate how complex the prioritization task can become and how the developers can use the code smells on this task, consider the example in Figure 2.2. This example illustrates a partial view of a system created to manage enrollments in a university. As a design decision, a developer created a component called *Service* to deal with transactions on the database. *AbstractService* is an abstract class that defines the main database operations. All the other service classes within this component extend the *AbstractService*. Thus, this component and the classes related to it are a good starting point to identify critical design problems, especially taking into account the relevance of the component to the system.

Once defined the component, the developer needs to choose an element (in this case a class) to focus his analysis (Phase 1 - Figure 2.1). Among multiple classes in this element, he decided that the *InstitutionalEnrollmentService* class should be analyzed, based on the smells that were affecting the class. Therefore, the developer needs to reflect upon these code smells to decide if they are design-relevant smells (Phase 2 - Figure 2.1). Going through the methods' structure, he can find that they contain common code smells such as Feature Envy and Dispersed Coupling, which were negatively contributing to its high coupling and low cohesion. He can also find that the reason these code smells appear in the class is that the class was implanting two concerns, consequently, violating the single responsibility principle (Martin and Martin 2006). From this analysis, we can see that only considering the code smells, the developer may have hints about other types of symptoms affecting the element. In this example, the smells indicate that the class violates the single responsibility principle.

After combining these smells, he concludes that these classes had these smells (God Class, Feature Envy and Dispersed Coupling) because of the *Concern Overload* design problem (Phase 3 - Figure 2.1). *Concern Overload*

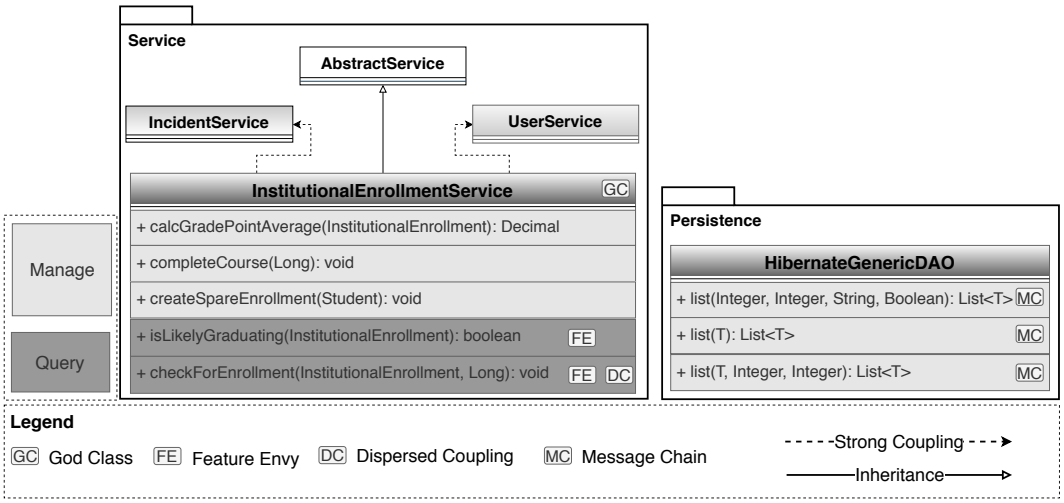

Figure 2.2: Design Problem Occurring in the UniM System

happens when elements are overloaded with multiple unrelated responsibilities, which can hamper the maintainability and understandability of the system (Garcia *et al.* 2009b). Thus, these code smells were affecting these elements due to this design problem.

In this example, the design-relevant smells to identify the design problem were located in the *InstitutionalEnrollmentService* class. Hence, the developer had to analyze these several code smells and reason about them to choose which ones he deems useful or not to identify the design problem. Developers need to reason about multiple code smells since each smell is only a partial indicator of a design problem. Thus, developers need to analyze and combine the information of multiple smells to increase his chance to confirm a design problem. In fact, a study revealed that when code smells appear together in an element, they have a higher chance to indicate a design problem (Abbes *et al.* 2011, Oizumi *et al.* 2016).

To make the analysis of the elements even harder, the developer also had to figure out the reason why these code smells affected the class. This reason is information that would help him to judge how useful a smell is to consider if the element contains a design problem. For instance, the developer could be analyzing a class that contains a *God Class*, and the reason for that is the supposedly high number of lines of code (LOC). However, he may judge this number low, this way he can exclude this smell from his analysis. In summary, the analysis of these several smells is cumbersome for developers. In this example, not only he had to deal with several smells, but he also had to reason about them as well as to consider the smells affecting other elements during the analysis.

A heuristic to prioritize the relevant elements could be used by the developer to avoid the analysis of non-relevant elements. For instance, let us consider the *Persistence*, which is another important component in the UniM system. This component deals with the persistence of the data in the system. At first sight, the *HibernateGenericDao* class appear to be a relevant element, due to its number of smells. This class has more than 10 code smells. However, on a deeper analysis, the developer noticed that these code smells are all of the same type: Message Chain. They appeared in the class due to the framework used in the system. Thus, the developer can discard this smells from his quest to identify a design problem for two reasons. First, he knows that this type of smell is not relevant to identify a design problem in his case. Second, all the smells have the same type. In other words, considering only the number of smells may not be enough. This scenario shows the importance of also considering the diversity of the smells in an element in order to prioritize it.

Since the system may contain even hundreds of smells (Macia *et al.* 2012b), choosing the ones that can be helpful can be challenging to the developer. In addition to the number of smells, they are scattered over multiple code elements. In addition, a code smell is just a partial indicator of a design problem, which makes the developer need to relate two or more smells. Also, analyzing smells that are not related to the design problem can be time-consuming and can mislead developers to a design problem that does not exist. This is a scenario where developers would benefit from heuristics that could prioritize relevant elements. These heuristics would help them to focus on the code elements containing the smells that may be helpful to identify the design problem.

2.3 Related Work

In this dissertation, we have the goal of supporting the developer on the prioritization of a small sub-set of relevant elements in the system. In order to prioritize an element, the developer has to pre-filter the elements he thinks should be analyzed, followed by prioritization of its design-relevant smells (Section 2.2). The output of the prioritization process will help the developer to identify critical design problems on the system. In this context, we searched for studies related to (i) the relation of design problem and code smells (**Section 2.3.1**), (ii) identification of design problems with other symptoms rather than just smells (**Section 2.3.2**), and (iii) prioritization of relevant elements (**Section 2.3.3**).

2.3.1 Relation Between Design Problems and Code Smells

Once identified, a design problem needs to be removed since its prevalence in the program can lead even to discontinuation of the system (MacCormack, Rusnak and Baldwin 2006, Godfrey and Lee 2000, Schach *et al.* 2002, Gurp and Bosch 2002). Prior to this identification, the developer has to prioritize the relevant elements that will be used for the identification. The need to evaluate elements affected by code smells emerges from the lack of design documentation (Kaminski 2007), which ideally would be used on this prioritization. Additionally, some studies had presented code smells as indicators of design problems (Yamashita *et al.* 2015, Macia *et al.* 2012a, Oizumi *et al.* 2016).

Moha *et al.* (Moha *et al.* 2010) proposed a method that embodies and defines the steps necessary to identify and detect code smells and its related design problems. To validate their method, they asked software engineers to analyze classes and detect if they had smells. Additionally, they identified relationships between 15 code smells and four related design problems. However, their technique depends on using a domain-specific language that describes the code smells to every system.

The study of Abbes *et al.* (Abbes *et al.* 2011) analyzed whether systems with *God Class* and *Spaghetti Code* had their understandability hampered. This study was done with 24 subjects and three different systems, where the subjects were asked to perform three different program comprehension tasks. The results revealed that the occurrence of the isolated code smells did not significantly reduce the understandability. However, the combination of both smells had a significant negative impact on the software understandability.

Macia *et al.* (Macia *et al.* 2012) studied the impact of code smells on the degradation of the software design. In their study, they analyzed if a sample of 2056 smells were indicators of design problems. As a result, they identified that 65% of all code smells were related to 78% of all design problems in the system. In addition, they found that certain code smells, such as Long Method and God Class, were consistent indicators of design problems.

Even though these studies demonstrate how the smells can be related to design problems, they do not investigate how developers could use these smells in practice. In addition, they do not propose any heuristics to support the developers on relating smells and design problems. Given this limitation, we investigate in this dissertation how to support the developers on the prioritization of relevant elements (Section 1.3).

2.3.2 Identification of Design Problems with Other Symptoms

The identification of design problems is a widely discussed theme (Kazman *et al.* 2015, Ran *et al.* 2015, Li *et al.* 2014, Oizumi *et al.* 2016, Sousa *et al.* 2017, Sousa *et al.* 2018). Although smells are good indicators of design problems, the developer may use other types of symptoms to identify a design problem. For instance, Li *et al.* (Li *et al.* 2014) investigated if some modularity metrics indicate design problems. They found two modularity metrics that can be used as indicators of design problems. The first metric is the Index of Package Changing Impact (IPCI). This metric quantifies the independence of packages, based on the percentage of the number of non-dependency package pairs versus the total number of all possible package pairs. The second, called Index of Package Goal Focus (IPGF), indicates the average extent that the services of a specific package have for the same goal. However, they do not explore how developers could use these metrics in practice.

Sousa *et al.* investigated how software developers identify design problems in practice (Sousa *et al.* 2018). They conducted a multi-trial industrial experiment with professionals from five software companies, which resulted in a grounded theory. Their theory reveals that, in practice, developers rely on a heterogeneous set of symptoms. Examples of these symptoms include the violation of design patterns and violation of object-oriented principles.

Despite the fact that these studies examine other symptoms to identify design problems, rather than just code smells, they did not provide any heuristic to support developers in this identification. Additionally, they do not focus on the prioritization of design problems. Furthermore, our methodology, both to propose and evaluate the heuristics, was more rigorous. We observed in practice how the developers prioritize relevant elements. This observation led us to the definition of heuristics aligned with how they prioritize. On the evaluation, we used different sets of systems, with both open and closed source.

2.3.3 Studies on the Prioritization of Relevant Elements

When it comes to prioritization, there are studies focused on the use of smells to prioritize elements that have design problems (Arcoverde *et al.* 2013, Vidal *et al.* 2016, Guimaraes *et al* 2018). Vidal *et al.* (Vidal *et al.* 2016) propose and evaluate a suite of criteria to prioritize groups of code smells that may indicate design problems. In their study, in order to rank their group of code smells, they propose heuristics that consider both implementation and

architectural information of the system, which includes the version history of the system. However, they did not follow a systematic approach to propose heuristics. In other words, they did not observe how developers prioritize smelly elements.

Arcoverde *et al.* (Arcoverde *et al.* 2013) presented and evaluated four heuristics for prioritizing design-relevant code smells. These heuristics explore characteristics of the software, such as change-density and error-density, to automatically ranking the code elements that should be refactored according to its design relevance. Considering the average precision of all projects evaluated, they reached a maximum of 72.5%, with four projects. However, they did not follow a systematic and practice-based approach to defining their heuristics. On the previous sub-section, we described how we defined our heuristics, which included the observation of prioritization in practice.

The work of Guimaraes *et al.* (Guimaraes *et al* 2018) is focused on prioritizing critical code smells for the identification of design problems. In their study, they evaluate how developers use both blueprints and source code to reveal some design-relevant smells. They found that the developers took too much time to identify the code smells. Thus, they propose a suite of 3 prioritization criteria, aiming to improve this identification. These criteria used different ways of relating the blueprint elements with the code smells. The main limitation of their approach is the dependence on design blueprints. As we mentioned, design information is often outdated or unavailable for many systems. Thus, such an approach would have little practical application.

In the study of Natthawute *et al.*, the authors tried to determine how developers prioritized code smells on open source projects (Natthawute, Shinpei and Motoshi 2018). They asked developers to select and prioritize code smells on open source projects. Then, they selected the reasons why the developers prioritized these code smells. They identified two reasons for the prioritization of code smells: (i) the task relevance and (ii) the smell severity. However, this study used developers that are not core developers of the systems. This may hamper the quality of the data gathered, since they may not know details about the design of the software system being evaluated. Also, they do not consider how this prioritization could be used for the identification of design problems.

In general, our study overcomes state-of-the-art prioritization approaches in at least two key aspects. First, our heuristics emerged from observing how developers perform the prioritization task in practice. In other words, the heuristics were created based on actions that developers made when properly prioritizing relevant elements. Thus, the heuristics were created in a way

that is aligned with how developers performed the prioritization in practice. This alignment can increase the chance of having developers using these heuristics on their routine. Most of the aforementioned approaches are based on information that is often not available in many software projects (*e.g.* design blueprints, long history of program versions and the like) (Vidal *et al.* 2016, Guimaraes *et al* 2018). Additionally, we performed our studies with original developers of each system, which increases the confidence in the collected information, in particular on the list of design problems affecting the systems. Given the aforementioned limitations of previous approaches, we could not apply their heuristics in our studies.

2.4 Summary

This chapter presented the main terms and concepts used throughout this dissertation. In Section 2.1, we defined the main aspects related to software design and how design problems can appear in a software system. In addition, we presented the design problems that are investigated in this dissertation.

In Section 2.2, we discussed the prioritization of design problems with design-relevant smells. First, we detailed the code smells and how they are related to the identification of design problems. We also presented the code smells used in this dissertation. Second, we discussed the prioritization of smelly relevant elements, which precedes the identification of design problems. To better explain the prioritization task, we provided a meta-model, which presents the phases of the prioritization of relevant elements. These phases are (i) prioritization of elements, (ii) prioritization of design-relevant elements and (iii) confirmation of prioritization. Finally, we presented a practical example of the prioritization of relevant elements.

In Section 2.3, we discussed the studies related to the prioritization of relevant elements as well as their limitations. This discussion helped us to evaluate the state of the art on the prioritization of relevant elements. These studies included (i) the relation between design problems and code smells, (ii) the identification of design problems with other symptoms rather than just code smells and (iii) studies on the prioritization of relevant smells.

Based on the discussion of this chapter, we claim that the use of prioritization heuristics can better assist developers in prioritizing a small set of relevant elements among the relevant elements in the system. Thus, to propose these heuristics, our first step is to conduct a study on how developers prioritize relevant elements. We have the objective of finding criteria that they use during the prioritization task. Based on these criteria and the actions taken by the

developers during the prioritization, we aim to construct our prioritization heuristics. The next chapter presents this study about how developers prioritize relevant elements.

3
On the Prioritization Criteria and Heuristics: A Qualitative Study

Finding design problems in the system is a major step in the process of keeping the software system healthy. When neglected, design problems can even lead to the software discontinuation (Hochstein and Lindvall 2005). The manifestation of a single design problem can be scattered through multiple elements in the program. Among the hundreds of elements that a program may have, the developer is faced with the task of prioritizing the elements that are candidates of hosting design problems. To prioritize an element, the developer has to follow a few steps: (i) he chooses the elements that should be analyzed for some reason, (ii) he analyzes if the element has one or more design-relevant smells and then (iii) he confirms if that element is relevant (Section 2.2.2).

However, many smells are not relevant to the design. For example, a code smell may be merely introduced due to frameworks used in the system and, therefore, is irrelevant to the design. Thus, just recognizing if the element has code smells is not enough. Hence, developers have to prioritize the design-relevant smells, *i.e.* the ones that are actually indicators of design problems. The elements with at least one design-relevant smell are the so-called *relevant smelly elements* (or, simply, *relevant elements*) (Chapter 2).

The effective prioritization of relevant smelly elements requires the use of proper information. In particular, the code smells have intrinsic information (*e.g*, the smell type) that can help developers during the prioritization of relevant elements. How developers rely on this information is part of the criteria they apply to prioritize elements. Therefore, it is necessary to investigate which of this intrinsic information must be considered. For instance, Oizumi *et al.* (Oizumi *et al.* 2017) investigated the co-occurrence of code smells. They found that developers have more chance of finding design problems when explicitly reasoning about such co-occurrences, compared to when they reason about an element with a single code smell. However, they did not investigate if developers actually use this information (*i.e.*, smell co-occurrence) when actually performing the task of prioritizing smelly elements.

In practice, developers can benefit from heuristics based on criteria and

actions they apply to prioritize relevant elements. These heuristics can benefit developers by prioritizing a small sub-set of relevant elements, which includes the prioritization of design-relevant smells as illustrated through the example in Section 2.2. Before proposing heuristics, we still need to know how the developers prioritize relevant elements, which includes (i) the criteria that they take into account during the prioritization and (ii) the actions performed by them during the prioritization. Thus, we need to know which are these criteria and actions in order to propose heuristics aligned with how developers prioritize in practice. To gather this knowledge, we systematically observed how developers prioritize relevant elements.

We performed a qualitative study where we asked the developers to prioritize relevant elements in their systems. In this process, we also observed how they prioritize the design-relevant smells in these elements. We also asked them to perform the prioritization until they were able to identify a design problem. Consequently, we could confirm that the element analyzed was relevant to the design problem identification. This task was recorded on audio and video, which we used on our data analysis, following the Grounded Theory (GT) procedures.

This chapter is organized as follows. In Section 3.1, we define the settings of our study. In Section 3.2, we detail how we collected the data from the tasks performed on the study. On Section 3.3, we discuss the results of this study, presenting the criteria and respective heuristics proposed. In Section 3.4, we present threats to the validity of this study. Section 3.5 presents the final conclusions in this study.

3.1 Study Settings

This section describes our study settings. Section 3.1.1 presents our goal and research question. Section 3.1.2 describes the activities performed by the developers during the experiment. Section 3.1.3 describes the steps taken to choose suitable scenarios and subjects for our study. Finally, Section 3.1.4 describes the instrumentation used during the study.

3.1.1 Goals and Research Question

In this first study, we aimed at analyzing how developers prioritized relevant elements, with a particular focus on understanding how they prioritized design-relevant smells on these elements.

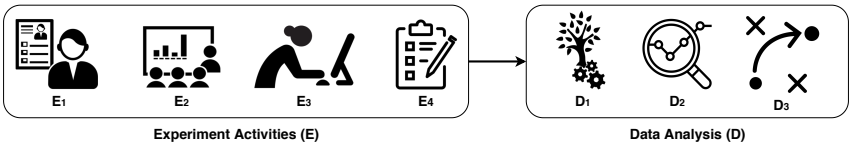

Figure 3.1: Studies settings

To answer our first specific research question (Chapter 1), we divided the study into two parts, as illustrated by Figure 3.1. In the first part, we asked developers to prioritize relevant elements in their source code. For this task, they had to analyze their source code using a summary of smells affecting the elements. Details about these smells are presented in Table 2.2. In the second part of the study, we analyzed how they prioritized the relevant elements. Our goal in this analysis was to observe the criteria and actions that developers applied to prioritize relevant elements prior to the identification of design problems. Finally, we used these criteria and actions to propose prioritization heuristics. In this context, our goal is to find criteria that developers use to prioritize relevant smelly elements answering the following specific research question:

> ***SRQ1.:*** *What are the criteria that developers tend to use to prioritize relevant smelly elements?*

As aforementioned, in order to prioritize the elements, the developers used a summarized list of smells. Each summary shows the smells of an element which is a candidate to have a design problem. The summary also contains the smells' information, such as the smell type, its definition, and the reason why it was presented (*e.g.* threshold and metrics). We took some measures to observe how the developers prioritized the design-relevant smells in those elements. These measures include a questionnaire at the end of the analysis of each element. In this questionnaire, the developer had to answer how helpful each code smell available in the summary was to the identification of a design problem. Thus, we could confirm if the smell was a design-relevant smell, which the developer had to consider to confirm the relevance of a smelly element being prioritized (Figure 2.1).

3.1.2 Experiment Procedures

The study was composed of four experimental activities (E) (Figure 3.1): application of a questionnaire for participants' characterization (E_1), training session (E_2), prioritization of relevant elements with the summarized program

information (E_3), application of a follow-up questionnaire (E_4). We describe in detail each of these activities in the following:

E_1: Application of a questionnaire for participants' characterization. As outlined in the instrumentation (Section 3.1.4), a questionnaire was administered to the participants. The goal was to gather information regarding education, experience with software development, and the terms used in the study. This information is summarized in Table 3.1.

E_2: Training Session. In this activity, two researchers conducted a **training session** with all developers about essential concepts for the study, such as software design and code smells. We also presented examples of design problems: *Ambiguous Interface*, *Unwanted Dependency*, *Component Overload*, *Cyclic Dependency*, *Scattered Concern*, *Fat Interface*, and *Unused Abstraction*. We selected these problems with the project managers, who suspected they were common causes of problems in their projects. However, we made it clear to developers that they could use the prioritization to identify other types of design problems with which they were familiar with. The training session was comprised of a Powerpoint-based presentation (25 minutes long), and a session for discussions and questions (15 minutes long), if necessary.

E_3: Prioritization of relevant elements with the summarized program information. In this task, we asked developers to use the summarized program information in order to prioritize relevant elements. This summary contained all the smells affecting the element investigated. The developers also had to answer if each code smell was relevant or not to identify the design problem. This helped us to discover how they prioritize the design-relevant smells.

They had 1 hour and 30 minutes to perform the prioritization task in pairs. They performed this task in a pair as a way to encourage the discussion of their actions to prioritize a smelly element, consequently allowing more natural discussions about the prioritization being performed. Thus, we asked them to verbalize their thoughts, following the think-aloud method, in which we record it on video and audio during the task. Since the developers had to report the design problems found, we provided a brief description of each design problem used in the study. This way, they could use this information during the task.

E_4: Application of a follow-up questionnaire. A questionnaire was administered after the tasks, where the developers were asked about how the code smells summarized helped in the relevant elements prioritization. We used the collected data about how developers prioritized relevant elements to extract the actions and criteria they applied. We explain this analysis in Section 3.2.

3.1.3 Subjects Selection

For this study, we searched for companies that could provide us with (i) developers with knowledge about design problems and (ii) systems that could have design problems considered critical at some point. One of the motivations for the developers was related to the fact that they would be doing the prioritization on systems which they currently work. To select the companies, some criteria were defined, including the level of experience of their developers, the number of developers working in a project and the development process on that project. We choose two Brazilian companies that meet our criteria (C1 and C2). Following, we describe the companies.

C1 is a company that works with renting and selling of printing devices. Even though they were not an IT company, there is a department specialized in software development. This department act as a software factory, where they produce software for their own company or solicited by external customers. For instance, they produce a management system of their printers, including its contracts and clients. They also have software responsible for electronic document management for the company's clients.

C2 is an incubated company located on the Federal University of Alagoas (UFAL). They provide multiple services for the university, such as a system responsible for all the academic registration and control. In our study, we had access to software developed by the Nucleus of Information Technology (NTI). NTI is responsible for the systems developed for this university.

After choosing the companies to conduct the study, we contacted the companies' managers so they could suggest the systems that met our criteria. First, systems with different stages of design degradation. Second, systems with different domains regarding the size, complexity, and number of developers involved. Last, systems developed in Java. The two selected systems are described as follows.

- **C1: System 1 (S1)** deals with tracking the process of arrival and departure of print devices, contract deals with their customers and replacement and reusability of compatible machinery pieces. This is a software system built using micro-services, in which it uses the micro web framework Jooby. S1 was developed in early 2015, and it is being used until today. S1 has 11,729 LOC (Lines of Code) written in Java.
- **C2: System 2 (S2)** is an academic system that facilitates the processes involving the students, from the enrollment in the institution until the emission of documents. This is a monolithic software, which uses the

Table 3.1: Developers' Characterization

System	ID	Education	Experience in Industry (Years)	Experience with Java (Years)	Perform Code Review?	Familiar with Code Smells?	Familiar with Design Problems?
S1	D1	BSc	7	6	Yes	Yes	Yes
	D2	BSc Student	2	0	Yes	Yes	Yes
	D3	BSc Student	4	5	No	Yes	No
	D4	BSc Student	4	4	No	Yes	Yes
S2	D5	BSc Student	3	3	No	No	No
	D6	BSc	5	10	No	No	Yes
	D7	IT Specialist	13	13	Yes	No	Yes
	D8	MSc	14	12	Yes	Yes	Yes
	D9	BSc	14	12	Yes	No	Yes
	D10	MSc	6	6	No	Yes	Yes

frameworks Spring, JBoss Seam and Hibernate. It was developed in 2010 and it is on production until today. S2 HAS 71,327 LOC written in Java.

After choosing the systems, the companies' managers gave us access to the developers of each system. In our study, we had access to 10 developers, 4 from C1 and 6 from C2. To collect the profile of the developers, we applied a characterization questionnaire. Table 3.1 presents the developers' profile. The first column indicates the system that the developer was working and performing the prioritization task. The second column represents the ID given to each developer. The third column presents the highest degree of education that each developer had. The fourth and fifth columns indicate how many years of experience the developers had in the industry and with Java, respectively. The sixth column informs if the developers are familiar with code reviews. The last two columns present the familiarity of the developers with code smells and design problems. The prioritization task was conducted in pairs by the developers, this way on the table they are grouped by the system that they work and by the pair that they formed.

Observing the Table 3.1, we can notice a heterogeneity on the profile of the participants. We have developers ranging from BSc students to masters. It is interesting to notice that one of the developers had none experience with Java development. However, we took care that he was working with a developer that had previous experience with Java. Also, even though he did not have any experience with Java, he had previous experience with important concepts of our experiment, such as code smells and design problems. We can also see that all developers had familiarity with at least one of the terms. It would be ideal if they had knowledge of both terms. However, if the developer knows about code smells, this might help him to prioritize relevant elements and then identify a design problem.

3.1.4 Instrumentation

In our study to investigate how the prioritization task happens in practice, we used a set of instruments to support the developer on the task as well as help us on monitoring them. We describe the artifacts used for the execution of the study as following.

- **Companies characterization form:** This form was filled by the companies' managers. With this form, we could gather information such as (i) the products provided by the company, (ii) the number of developers and their experience on design problems, code smells. Based on this information, we tried to discover if those concepts were part of the routine of the developers. The form was sent to the managers one week before the study, where we did a presentation with the key terms the would be used on the tasks. With this form, we had the purpose of gathering information about the companies, systems, and developers. Thus, together with the manager of the system, we could choose systems and developers for the study.
- **Participant characterization form:** This form was composed of questions that characterized the participants regarding education, experience with software development, experience with Java language and projects that the developer worked as design reviewer. We collected this information to understand the characteristics of each developer and if somehow, they have some misunderstandings about the concepts used in the study. This allowed us to prepare and adapt to the training session.
- **Task execution records:** We recorded the prioritization task, which included the video and audio captured by Camtasia [1] during the participants' activities. We collected this data to capture the actions that the developers performed to prioritize the relevant elements. Then we analyzed it further using grounded theory.
- **Follow-up questionnaire:** This questionnaire was composed of questions regarding the prioritization of relevant elements. We also asked them if the code smells were useful to identify a design problem. Thus, we could confirm whether these smells were design-relevant smells or not. These answers were used to support the qualitative analysis.
- **Lists of design problems and code smells:** We provided to the developer a list with the descriptions of design problems and code smells. They could use these lists during the experiment if needed.

[1] Available at `https://www.techsmith.com/video-editor.html`

– **Web-framework:** The task was performed on the developers' machines. The prioritization task was performed in a web ambient that we created. We built this ambient using Django[2] . Django is a Python-based web framework. The design of the page is meant to be similar to how SonarQube[3] presents the metrics about the software system. We used the SonarQube as a base for the design since it is a well-known platform to inspect the software system. Based on this strategy, we could reduce the learning curve of how the symptoms were presented. While SonarQube presents information unrelated to the design problem, our web-page focuses on symptoms that can help with the prioritization of relevant elements.
– **System source code:** In parallel with the prioritization task, the developer also had access to the source code of the system analyzed using the Eclipse IDE [4].

3.2 Data Collection and Analysis

In this part of the study, we conducted data collection and analysis (D) (Figure 3.1), which was divided into three steps. In the first step, we **applied Grounded Theory (GT) procedures (D_1)** (Strauss and Corbin 1998) over all collected data (audios, videos and questionnaires). The procedures comprise three phases. *Open coding* (1st phase) involves the breakdown, analysis, comparison, conceptualization, and the categorization of the data. We applied it after the transcription of all audios and videos. The open coding process was subdivided into three steps. In the first step, two researchers were responsible for the transcription over the data collected on the videos and audios. Each researcher was responsible for half of the videos. In the second step, a third researcher was introduced. In this step, the first two researchers reviewed the transcriptions of each other while the third researcher reviewed all the transcriptions. In the third step, the three researchers did the open coding. These codes were associated with quotations of developers' utterance.

The *Axial coding* (2nd phase) examines the relations between the identified categories. In this phase, the codes were merged and grouped into categories that they were related to. The codes were then placed on networks representing the connections between them. The codes, categories, and networks were then revised, and there was a discussion among the researchers until they reach an agreement.

[2]Available at `https://www.djangoproject.com/`
[3]Available at `https://www.sonarqube.org/`
[4]Available at `https://www.eclipse.org/ide/`

The *Selective coding* (3rd phase) performs all the process refinements by finding the core category. Therefore, we created codes for the developers' speeches (1st phase). After, these codes were related to each other through axial coding (2nd phase). We did not apply the selective coding as we were not aiming to reach a theoretical saturation, as expected in the GT method. Thus, we do not claim that we applied the GT method, only some specific procedures. Each transcription, code, and the relationship among categories were reviewed, analyzed and changed upon agreement with the other researchers.

In the analysis step, we **analyzed the GT codes (D_2)** to find actions and criteria used by most developers and that contributed to the prioritization of relevant elements and its design-relevant smells. We analyzed the codes when the developer confirmed a design problem. For instance, to confirm a design problem, they considered the number of code smells on an element, or if this element had different types of code smells. As the developer follow this procedure to confirm the design problem, we labeled this procedure as an *action*.

In addition to the identification of design problems, one of the steps of the study was to confirm if a code smell was useful in the identification of the design problem in the element. Since we were recording this task, we could grasp what information the developers used to confirm that the code smell was useful to the identification. Otherwise, we also observed what made them discard a code smell of the identification in case it was not useful. Observing how developers evaluated the usefulness of each smell allowed us to find the program elements with design-relevant smells. Then, we analyzed these actions during the (i) prioritization of relevant elements and (ii) discussion on how useful or not the code smells were. We used those actions to extract criteria used by developers to prioritize relevant elements (Section 3.3).

Finally, we used the criteria found in the previous activity **to propose the heuristics (D_3)**. To propose them, we combined the criteria found with actions taken by the developers during the prioritization of relevant elements. In the next section, we describe these heuristics.

3.3 Prioritization Criteria and Heuristics

In this section, we provide the answer to our SRQ1: *What are the criteria that developers tend to use to prioritize relevant smelly elements?*. In Section 3.3.1, we present the *general* criteria and heuristics that the developers tend to use on the prioritization of relevant elements. In Section 3.3.2, we present the *specific* criteria and a brief explanation of how their heuristics can

be implemented. In Section 3.3.3, we complement our suite of heuristics with other heuristics derived from actions (Section 3.2) that developers performed when they were prioritizing relevant elements.

After analyzing how developers identify critical design problems, we found the criteria that they use to prioritize relevant elements. We divided these criteria into two groups: general and specific. The general criteria are *smell type*, *number of smells* and *type diversity*. They are defined as general since they use information that does not vary from one system to another, *i.e.*, they are independent of the analyzed systems, which make them general. The specific heuristics are *element role*, and *relation*. We consider them to be specific due to their dependence on information extracted from each system, which makes them specific for each system. To give examples about how the heuristics work, we will be referring to the example presented in Chapter 1 - Figure 1.1. Each heuristic returns a set of $TOPN$ elements, where N is the number of elements prioritized – this value can be defined by the developer who uses the heuristics.

3.3.1 General Criteria and Heuristics

This section presents the three general criteria observed in the study, namely *smell type*, *number of smells* and *type diversity*. For each criterion, we present the proposed heuristic.

Smell Type. This criterion is based on which type a smell belongs. For instance, a code smell can be of the type *Long Method*, *God Class*, *Intensive Coupling*, and so on. The code smell types used in this dissertation are presented in Table 2.2. The type of the symptom indicates, at least partially, what may be wrong with the element. For instance, let us consider the `DeviceRepository` class in the example from Figure 1.1. This class has the `getDevice` with a *Dispersed Coupling* smell. This type of smell indicates that the method is accessing multiple operations across an excessive number of classes, which is due to the *Concern Overload* design problem found in the class (Figure 1.1). Smells can be identified directly in source code, which is often the only available artifact. There is a wide variety of accurate strategies for detecting smells in a program (Aniche *et al* 2016, Vale, Fernandes and Figueiredo 2018, Macia *et al.* 2012a, Macia *et al.* 2012, Macia *et al.* 2012b). They are part of developers' routine (Yamashita and Moonen 2013) and act as hints for refactoring (Fowler 1999), helping developers to partially or fully remove design problems.

Depending on its type, a smell may affect elements located in different

scope levels of the system (Section 2.2.1). For instance, smells such as Shotgun Surgery and Divergent Change affect multiple elements, possibly located in different components of the system. Thus, they are considered to belong to the application level. A smell such as God Class affects an entire class; thus, it belongs to the class level. Finally, a smell such as Long Method affects just a method, belonging to the method level.

During the experiment, the developers did not explicitly declare that they were choosing a smell to analyze due to its scope. However, analyzing their actions during the prioritization task, we observed that they tended to dedicate more time (and effort) in the analysis of class-level smells. In some cases, they even discarded some method-level smells after analyzing the class-level smells. For instance, the pair composed by developers `D7` and `D8` analyzed a class with five code smells, from which three were method-level smells (Feature Envy, Message Chain and Dispersed Coupling) and two were class-level smells (Complex Class and God Class). During their analysis, they considered only the two class-level smells as relevant to identify the design problem *Concern Overload*.

We decided to expand this concept of prioritizing smells according to their scope. Thus, the proposed heuristic prioritizes first the application-level smells, then the class-level smells, and finally method-level smells. This decision was taken since a class that presents an application-level smell is likely to be related to other classes that may also be affected by a design problem. Thus, prioritizing this class would be a good starting point for developers to prioritize other classes that may be part of the design problem.

In this sense, we derived the **Smell Granularity** heuristic. This heuristic prioritizes first elements with application-level smells, followed by the ones with a class-level smell, then the elements with method-level smells. Algorithm 1 presents the pseudo-code for this heuristic. It receives as input the project P that should be analyzed and the number N of elements that the developer wants to prioritize. In `Line 1`, the algorithm uses a tool[5] to collect the code smells. This tool implements detection rules for the 17 code smells used in our study. It also returns a list of all elements of the system, their respective code smells, and the metrics and thresholds used to confirm if the element has a code smell (Phase 1). In `Line 2`, the algorithm iterates over all the elements with code smells collected and then categorizes their respective smells. This categorization is divided into application, class, and method (Phase 2). In `Line 6` those elements are sorted based on the respective numbers of application smells, class smells and method smells (Phase 3). Finally, in `Line`

[5]Available at `https://github.com/opus-research/organic`

7 the algorithm returns the N relevant elements.

Algorithm 1 Smell Granularity Heuristic

Input: a project P, a number N of elements
Output: The $TopN$ relevant elements

```
1: listElementsWithSmell ← extractCodeSmells(P) {Phase 1}
2: for all element ∈ listElementsWithSmell do
3:     categorizedSmells ← categorizeSmells(element) {Phase 2}
4:     listCategorizedSmells.insert(categorizedSmells)
5: end for
6: topN ← sortSmellsByCategories(listCategorizedSmells, N) {Phase 3}
7: return topN
```

Number of Smells. This criterion indicates how many smell instances an element contains. For example, the `DeviceRepository` (Figure 1.1) has 51 smell instances; `ClientRepository`, in its turn, has 21 smell instances. In this criterion, the smell type does not matter, only the number of instances affecting an element. As an example of how developers applied the criterion, let us consider the pair of developers `D3`/`D4`. When they were analyzing the smells in the class, they noticed that the class had several smells. Due to this number of smells, the developers concluded that the class should be prioritized since it was relevant to the identification of a design problem. In fact, they mentioned that due to the number of smells, they could see it potentially had a design problem, which developers were able to identify it later on. To illustrate an example of how developers used the number of smells criterion, we briefly summarize their actions (**A**). These actions happened when developers noticed the relevance of the element due to its number of smells. This moment is represented by the actions and quotation of developers' utterances:

A1: Developers identify the high number of smells on the class;

A2: Developers start analyzing the class;

A3: Developers speculate the presence of a design problem due to the number smells;

A4 Developers use the smells to analyze the class;

A5: Developers declare the class as a relevant element:

> **D4**: *"This (class) is very relevant (for the identification)! We can see that there is a (design) problem, right?"*

A6: Developers confirm the presence of a design problem.

We derived the **Smell Count** heuristic based on how developers used the number of smells criterion. This heuristic finds elements that have the greatest number of smell instances. For example, in the example Figure 1.1, the heuristic would suggest to developers prioritize `DeviceRepository` over `ClientRepository` since the former has more smell instances (51) than the latter (21). Algorithm 2 presents the pseudo-code for this heuristic. In `Line 1` the elements with smells are collected by the Organic tool. In `Line 3`, the algorithm iterates over this list of elements and for each element it computes the respective number of code smells. In `Line 4`, these computed number of smells are saved in a tuple: (element, number of smells). In `Line 6` the tuple is sorted in descending order, based on the number of smells. Finally, in `Line 7` the N elements with the highest number of code smells are returned.

Algorithm 2 Smell Count Heuristic

Input: a project P, a number N of elements
Output: The $TopN$ relevant elements
1: $listElementsWithSmell \leftarrow extractCodeSmells(P)$
2: **for all** $element \in listElementsWithSmell$ **do**
3: $\quad smellsByElement \leftarrow countSmells(element)$
4: $\quad nSmells.put(element, smellsByElement)$
5: **end for**
6: $topN \leftarrow sortDescByNumberOfSmells(nSmells)$
7: **return** $topN$

Type Diversity. This criterion represents the number of different smell types in an element. For instance, on Figure 1.1, `UserRepository` has 12 smell instances, which are from six different types. Thus, its diversity is equal to six. We noticed that this criterion was the least used by developers to prioritize elements – only one team used it. However, developers used it frequently to confirm that an element had a design problem. In this case, they confirmed after using used another criterion to choose (*i.e*, prioritize) a relevant element to analyze.

As an example of how developers used the Type Diversity criterion, let us consider the pair of developers `D7`/`D8`. First, they analyzed a class `A` with a diversity of five smells, in which they identified the design problem *Concern Overload*. Next, they had to find another element to analyze. At this moment, the developers started to search for classes with similar characteristics as the class `A`. Then, they found class `B`. They prioritized this class because it had a diversity of five smells as well. As soon as they saw this similarity between the classes, they mentioned other classes that also were similar regarding the diversity of smells. Consequently, developers used the Diversity criterion to

prioritize other classes. The following actions (A) and quotations present the moment where they use the diversity of smells to choose the other elements to prioritize.

A1: Developers identify that class `A` has diversity equal to five;

A2: Developers start analyzing class `A`;

A3: Developers use the smells to analyze the class;

A4: Developers identify a *Concern Overload* in class `A`;

A5: Developers search for classes with similar characteristics of `A`;

A6: Developers identify that class `B` is similar to `A` in its diversity of smells – both classes have five smells:

> D8: *"Yes, I think that (class `B`) is similar to the other (class `A`)."*

A7: Developers prioritize class `B` due to its diversity of five smells;

A8: Developers decide to use diversity as a criterion to prioritize other elements:

> D8: *"In fact, I think that the next (class) will be quite similar (to both classes `A` and `B`)."*

Based on how developers used the Type Diversity criterion, we derived the **Diversity** heuristic. This heuristic finds elements with the greatest number of different smell types. For example, `ClientRepository` has diversity equal to four, and `UserRepository` has diversity equal to six. According to this criterion, developers would prioritize `UserRepository` over `ClientRepository`, even though the latter has more smell instances (21) than the former (12). For this heuristic, we will only describe how it works, since the algorithm is similar to the *Smell Count* heuristic. For the *Diversity* heuristic, the smells of the projects are collected, then the algorithm computes the number of different types of code smells on each element and saves it in a tuple: *(element, number of different types of smells)*. The algorithm sorts this tuple is in descending order, based on the number of different types of smells. Finally, the algorithm returns the N relevant elements with most diversity of code smells.

These three criteria are related to each symptom affecting an element. Additionally, these criteria are independent of the systems. Consequently, we were able to propose three heuristics that, in theory, can be applied to any software system. Conversely, the remaining two criteria (element role

and relation) depend on subjective information of the system. *Element role* represents the responsibility that an element has in the system. *Relation* indicates how symptoms interact with each other. We will describe these specific heuristics next.

3.3.2 Specific Criteria and Heuristics

In the previous subsection, we discussed three criteria that are independent of the systems. The remaining two criteria (element role and relation) are somehow dependent on subjective information of the system. An example of subjective information is to what extent a program element is related to a certain requirement. As these criteria depend on the analyzed system, we derived specific heuristics. These heuristics are most complex both regarding the underlying criterion and the effort to automate them. We present the specific criteria and their derived heuristics as follows.

Element Role. This criterion represents the responsibility that an element has in the system. This criterion helps developers to focus on specific elements they know what to expect. For instance, let us consider developers `D7`/`D8`. They were analyzing classes responsible for implementing services in the system. In this case, these service classes were responsible for the database transactions in the system. When they analyzed class `C`, they identified a set of smells that led them to identify the design problem *Concern Overload*. When they started to analyze a second class (`D`), which was also responsible for the same services, they noticed that this class had almost the same set of smells. The actions (A) and quotations below present the moment where the pair `D7`/`D8` used the element role as a criterion to prioritize a relevant element.

A1: Developers start analyzing class `C` with a specific role in the system;

A2: Developers use the smells to analyze the class;

A3: Developers identify a *Concern Overload*;

A4: Developers search for classes with a similar role to class `C`;

A5: Developers start analyzing class `D` responsible for the same role as class `C`;

A6: Developers notice that both classes `C` and `D` have a similar set of smells.

At this point, they realized that the classes that were implementing services for database transactions (*i.e.*, elements with the same role) should be prioritized first in case they have a similar set of code smells.

> D8: *"I think that all services (classes) will be... (a relevant element)"*
>
> D8: *"This (class `D`) is very similar to the other (class `C`)"*
>
> D7: *"Yes, I think that (class `D`) is very similar to the other (class `C`). In fact, I believe that the next service also will be similar (to the classes `C` and `D`)"*

Indeed, when they analyzed the third class (`E`) related to a service, they also identified the design problem *Concern Overload*. At this point, they confirmed that all service classes would be relevant elements on their analysis. The following quotation presents this moment.

> D8: *"(The class) is (implementing) the same (service) as the other."*
>
> D8: *"All the services are like that. This mess."*

This criterion derived the **Role** heuristic. This heuristic chooses the elements that have the same role in the system. Thus, developers can analyze them together. Additionally, they may know what to expect from the elements based on their role played in the system. To identify the role of each element for this heuristic, we can use the study of Doséa *et al.* (Dósea, Sant'Anna and Silva 2018), where they extract the design role of the class based on its name. For instance, the class *DeviceRepository* would be automatically assigned as a class responsible for the repository role. Once we extracted the roles of the elements, we can group them and present to the developers.

Relation. This criterion comprises cases where two symptoms are related to each other. Developers use different information to relate symptoms. For instance, if two symptoms co-occur in the same method, developers can use the scope of the method to relate them. Hence, if they co-occur in the same method, there is a change of they be a result of the same design problem. Developers also use the smell type to relate them. For instance, they can relate intensive coupling with the dispersed coupling. Both smells are from different types, but they indicate, in their own way, the same characteristic: coupling. The way developers relate the symptoms may differ from one system to another. For instance, on the relation of symptoms based on their semantic relation, it will depend on the role that the elements affected by the symptoms play in the system.

Developers sometimes explicitly mentioned most criteria when they were discussing, for instance, they mentioned the number of smells and diversity of smells, or even the role played by the elements. However, regarding the relation criterion, we noticed that developers did not explicitly mention the need to relate two or more smells. Instead, they have naturally related the smells.

For instance, when developers `D1` and `D2` were analyzing class `F`, they noticed that method `M1` had Feature Envy and Intensive Coupling. Later on, they used both smells to identify the Concern Overload design problem in the element. Before identifying the design problem, they analyzed a second method (`M2`) that had five code smells (Feature Envy, Message Chain, Dispersed Coupling, Long Method, Brain Method). They dedicated more time in analyzing the method (`M2`) and further using it to confirm the design problem on the class. What is interesting in this example is how developers related the smells. In this case, the developers used two types of relations. First, they used the structural relation in both methods, since the smells were related to each if they were affecting the same method. Second, we noticed that they used the semantic relation between the smells affecting both methods: the Intensive Coupling in the method `M1`, and the Dispersed Coupling in the method (`M2`). Developers used these relations to identify the design problem, and posteriorly to prioritize elements with code smells that had similar relation among them.

This criterion derived the **Relation** heuristic. This heuristic selects the elements that contain more smells related to each other. This heuristic can be divided into three other heuristics according to the type of relation:

> **Structural Relation.** In this heuristic, the smells are related to each other because they affect the scope of the same element or because they affect elements that implement the same functionality. This criterion is based on the study of Abbes *et al.* (Abbes *et al.* 2011), where they identified that elements identified with God Classes and God Methods isolated, *i.e*, elements that had either a God Class or a God Method, but not both, had no effect on maintenance effort. However, when these smells appeared together, they led to a statistically significant increase in maintenance effort. Thus, we could use the structural relation between smells to prioritize the elements containing this relation. To build this heuristic, we could start by adapting the algorithms proposed by Oizumi *et al.* (Oizumi *et al.* 2016), which build groups of smells based on their structural relation. Hence, we could prioritize the elements affected by those groups of smells.

Semantic Relation. In this heuristic, the smells are related to each other based on the same concept, *i.e*, they represent the same conceptual characteristic. For instance, Feature Envy and God Class are related to the cohesion between elements. Thus, they have a conceptual relation according to cohesion. To identify the relation between smells for this heuristic, we will use the semantic relation defined by the study of Oizumi *et al.* (Oizumi *et al.* 2016). Code smells are semantically related if they are addressing the same concern. This concern may be a functionality of the system. To build this heuristic, we could start by using the concepts of agglomerations, defined by Oizumi. Agglomerations are groups of inter-related code smells. This way, we could use the elements affected by the agglomeration as the priority elements.

Role-based Relation. We decided to expand the relation criteria with this heuristic, which is a variation of the *Element Role* heuristic. This heuristic indicates elements that are related to each other because they play the same role in the system and have a similar set of smells. Thus, developers should analyze them together since these elements present the same characteristics. For this heuristic, we could use the smell type as a parameter to group the elements. This way, first we would group the elements based on its roles on the system and then do a new group based on the types of smells that the elements had. In this heuristic, the more common smell types a group has, the highest priority it will have.

3.3.3 Combined Criteria

We found that developers tend to combine multiple criteria when prioritizing elements. Each combination usually consists of a dominant (or primary) criterion, followed by a secondary criterion used as a tiebreaker. For instance, a developer is likely to first find a set of elements with the highest number of smell instances. If two or more elements have the same number of instances, developers put on top the element with the highest number of smell types. We expanded our suite of heuristics based on how they combined the criteria.

We highlight that the actions that developers did to combine multiple criteria were long and complex. Hence, we will not provide quotations of developers' utterances regarding the moments that they combined criteria. Otherwise, the quotations would be too long and containing unnecessary information that would be hard to understand how the combination took place. Instead, we will discuss each combination based on the example presented in Figure 1.1 (Chapter 1).

Number of Smells with Type Diversity. The number of smells was the second most used criterion to prioritize elements, while diversity was a criterion that developers used to confirm if an element has a design problem. Hence, we derived the heuristic **Smell Count & Diversity**, which combines these two criteria. Algorithm 3 presents the pseudo-code for this heuristic. In `Line 3`, this heuristic applies the number of smells criterion to sort the elements in descending order according to the number of smell instances (Phase 1). Then, in `Line 6` it returns the $TOPN$ elements (Phase 2) (If there is no tie between the first N elements – N is defined by the developer). In `Line 7`, the algorithm verifies if the list of top elements is bigger than the number of elements that should be returned. In this case, in `Line 8`, it is applied the type diversity criterion to break a tie between elements (Phase 3). For example, let us suppose that a developer only has time to analyze three classes ($TOP3$). This heuristic selects the elements with the greatest number of smell instances. Thus, it selects classes `A` (10 instances of 3 smells), `B` (9 instances of 3 smells), `C` (7 instances of 2 smells) and `D` (7 instances of 3 smells). Since there is a tie between `C` and `D` based on smell instances, the heuristic uses the diversity criterion to break the tie between `C` and `D`, selecting `D` since it has more types.

Algorithm 3 Smell Count & Diversity Heuristic

Input: a project P, a number N of elements
Output: The $TopN$ priority elements

1: $listElementsWithSmell \leftarrow extractCodeSmells(P)$
2: **for all** $element \in listElementsWithSmell$ **do**
3: $\quad dictNumber(element, nSmells) \leftarrow countSmells(element)$ {Phase 1}
4: $\quad dictType(element, nTypes) \leftarrow countTypes(element)$
5: **end for**
6: $topN \leftarrow sortElements(dict, N)$ {Phase 2}
7: **if** $topN.lenght > N$ **then**
8: $\quad topN \leftarrow runTieBreaker(topNElements, dictTypes)$ {Phase 3}
9: **end if**
10: **return** $topNs$

Type Diversity with Number of Smells. Similar to the previous heuristic, we also derived a heuristic that combines the number of smells and diversity of smell types. However, the primary criterion here is the diversity criterion. The number of smells criterion is applied to break a tie between elements, in case of any. The **Diversity & Smell Count** heuristic first chooses the elements that have the greatest number of different smell types. If the $TOPN$ elements have the same number of smell types, the heuristic analyzes their different types and prioritizes the elements with more smell instances.

Number of Smells with the Smell Type. Since the number of smells and the type of smell were the two most used criteria, we also derived a second heuristic that combines them. In this case, a heuristic that combines the number of smells with how developers used the code smells. The heuristic **Smell Count & Granularity** prioritizes the elements based on the number of smells criterion to return the $TOPN$ elements. In case of a tie between the elements, it uses the level of each smell as the secondary criterion. Elements with the highest number of application-level smells are chosen first. If the tie persists, the number of class-level smells is used to choose the element with the most number of class-level smells. For instance, let us suppose that `A` and `B` classes are in the $TOPN$ borderline with five smells each, but only one should get in. `A` has God Class, Complex Class, and two Long Methods, and `B` has God Class, two Long Methods and two Feature Envies. Since both classes do not have any application-level smell, the heuristic considers the class-level smells to break the tie between `A` and `B`. In this case, `A` class is chosen since it has two class-level smells (God Class, and Complex Class) instead of class `B`, which has only one.

Type Diversity with Smell Type. Finally, we combined the diversity of types with how developers used the code smells. The **Diversity & Granularity**, similar to the previous heuristic, uses the level of each smell in case of a tie. However, it considers the diversity criteria to conduct the first prioritization. For example, consider `D` class as having God Class, Complex Class, and Long Method smell, while `E` class has Duplicate Code, Complex Class, and Feature Envy. Both classes have the same number of smell types (three), and both are in the $TOPN$ borderline. However, `E` has an application-level smell (Duplicate Code). Therefore, `E` is chosen to be part of $TOPN$ instead of `D`.

Smell Type with Number of Smells and Type Diversity. We also created two other heuristics, now using the category of the smells as the first criteria in the combination. This two remaining heuristics were the **Granularity & Smell Count** heuristic and **Granularity & Diversity** heuristic. Both use the premise of the previously described heuristics. First, we consider the category of the code smell to sort the elements, then, if there is a tie on the $TOPN$ elements, we use the second criteria. In this case, the second criteria are respectively the number of smells and the variety of smells types.

3.4 Threats to Validity

This section presents threats to the validity of our study. For each threat, we provide the actions taken to mitigate its impact on the final results.

Construct Validity. We provided a set of symptoms of design problems to the developers. This data could introduce a bias in the study. However, these symptoms were chosen considering the literature (Oizumi *et al.* 2016, Macia *et al.* 2012a, Macia *et al.* 2012b, Macia *et al.* 2012, Gamma *et al.* 1995, Fowler 1999). Also, the companies' managers informed that the developers had previous experience with some of the symptoms and used them on a daily basis. The time allocated for the identification task could be considered another threat to validity. However, we conducted a pilot study to adjust the time required to perform the tasks and thus reduce the threat.

Internal Validity. The difference between the developers' background knowledge can be a threat. To mitigate this threat, we provided a training session to ensure a common knowledge base for all participants. Moreover, all developers had knowledge at least in one of the main themes in our study (design problems and code smells). Nonetheless, we saw the diversity of the developers' experience as an opportunity to generalize our results.

External Validity. The number of subjects represents another threat. All subjects worked for Brazilian companies. To mitigate this threat, we expanded the number of subjects in our second study (Chapter 4).

Conclusion Validity. The participation of the author who followed the GT procedures poses another threat. His beliefs might have caused some distortions when interpreting the data. To mitigate this threat, the GT coding activities were shared with other researchers. This way, at the end of the coding process, the (two) researchers merged their results and discussed until reaching a consensus. The way that developers prioritized the elements poses another threat to our study. We do not know if the actions and criteria that they used are the most suitable to the prioritization of relevant elements. To mitigate this threat, we designed and performed the second study (Chapter 4), where we evaluated the heuristics. We evaluated the heuristics based on their effectiveness in finding a small sub-set of relevant elements. Consequently, we will be able to evaluate whether the criteria were suitable for the prioritization

3.5 Summary

This chapter presented the first study to provide to support for developers along with the prioritization of relevant elements. In the first study, we asked the developers to prioritize elements relevant to the identification of design problems in their own source code. To prioritize the elements, they used a summarized list of smells. Each summary showed the smells of an element which is a candidate to have a design problem. Through qualitative analysis about how developers identify design problems, we found five criteria that they tend to use to prioritize relevant elements. The criteria are *smell type*, *number of smells*, *type diversity*, *element role*, and *relation*.

Based on these criteria, we derived a suite of prioritization heuristics. This suite contains heuristics that can be applied in any system, *i.e.*, they are independent of the analyzed systems. We also derived heuristics that dependent on the system. Additionally, we also noticed that developers tended to combine criteria, which lead us to derived heuristics that combined different criteria. Each combination usually consists of a dominant (or primary) criterion, followed by a secondary criterion used as a tiebreaker.

In the first study, we identified the criteria that developers use to prioritize elements. As a result, we used these criteria to propose a set of heuristics. We also identified some intrinsic information that the developers did not use during the prioritization. However, we are aware that the way developers performed the prioritization may not necessarily be effective in certain settings of different projects. For instance, they could be prioritizing certain elements with a different objective than the identification of design problems. In addition, they could be missing some design-relevant smells and then failing in prioritizing a relevant element. Due to this threat, we need to examine if the prioritization performed in practice was suitable to propose prioritization heuristics. Hence the need for evaluating these proposed heuristics in order to verify if these criteria used by developers were suitable to propose prioritization heuristics. In this context, we need to conduct an evaluation of the effectiveness of the heuristics in terms of finding a small subset of relevant smelly elements. In the next chapter, we describe the study that we conducted to evaluate our heuristics.

4
Evaluations of the Prioritization Heuristics

In our quest to support developers with the prioritization of relevant elements, we investigated how they perform this prioritization. Based on the previous study (Chapter 3), we proposed a suite with nine prioritization heuristics. After proposing these heuristics, our next step is to investigate if they can support the developer on prioritizing relevant elements. The evaluation of these heuristics is required as we do not know if the criteria used by developers in the previous study are suitable for the effective prioritization of relevant elements. For this purpose, we conducted a study to evaluate our nine prioritization heuristics. Even though we have outlined a proposal of specific heuristics (Section 3.3), in this dissertation, we focused on evaluating the general ones.

We decided to concentrate on the evaluation of the general heuristics for a couple of reasons. First, they are independent of the analyzed system. Therefore, they can be applied and replicated in multiple systems without requiring additional information from their developers. Second, they only need the symptoms as input. In our case, they only need the code smells. Therefore, we can automate its evaluation by applying them to several systems. For this purpose, we are able to run a tool that collects the code smells of each system. In our study, we used the Organic[1] tool to collect the smells because it provides all the information required by the prioritization heuristics.

To evaluate the nine heuristics in our study, we selected two sets of projects. For each project, we created a ground truth, which consists of a list of elements prioritized by the developers. In the first set of projects, we chose software projects, from the GitHub, that had a certain degree of structural degradation. To create the ground truth, we identified those elements that developers focused their effort in refactoring, probably due to the presence of a design problem. In the second set of projects, we picked projects from our industry partners, in which the systems' original developers provided us with the list of design problems in their systems. We used the elements affected by those design problem as the relevant elements. Once we collected this list of relevant elements, we ran our selected heuristics on the projects and

[1]Available at `https://github.com/opus-research/organic`

compared our list of prioritized relevant elements with the list of relevant elements prioritized by developers.

Once evaluated, we identified that two of our heuristics reached the best precision results. These heuristics were the *Diversity* and *Smell Granularity*. They had the best precision since they consider the types of smells affecting the elements. The *Smell Count* heuristic failed in prioritizing elements with design problems. However, this heuristic was able to prioritize elements that developers should dedicate effort in their refactoring. From these results, we can observe that the heuristics that consider the intrinsic information of the smells (*e.g.* smell type), are likely to support the developers on the prioritization of relevant elements.

This chapter is divided as follows. Section 4.1 presents the settings of the study proposed to evaluate the heuristics. Section 4.2 comprises the study results and discussions. Section 4.3 presents the threats to the validity of this study, and how we have tried to mitigate each one. Finally, Section 4.4 summarizes the findings of the study.

4.1 Study Settings

This section comprises the settings for this study. Section 4.1.1 defines the goals and research question. Section 4.1.2 presents the experimental activities performed in the study. Section 4.1.3 details how we created our ground truth for further evaluation.

4.1.1 Goals and Research Question

Developers may need to evaluate many elements to identify a design problem. Some of these elements are not relevant to design problems. Analyzing them can be a waste of time, that could be better used analyzing relevant elements. Additionally to the relevant elements, the developer has to analyze the code smells affecting them, in order to confirm if they are design-relevant smells. In fact, prioritizing relevant elements is a task that precedes the identification of design problems. Thus, developers can benefit from heuristics that prioritize these relevant elements.

Our heuristics aim to prioritize relevant elements, more specifically, they aim to prioritize design-relevant smells on elements that have critical design problems. Therefore, our goal is to investigate if they can prioritize elements that have design-relevant smells, and, obviously, elements with design problems. To achieve this goal, we evaluated our suite of heuristics based on

their effectiveness finding relevant smelly elements in a top priority list of elements, aiming to answer the following specific research question:

SRQ2. *What is the precision of the prioritization heuristics?*

To answer *SRQ2*, we conducted two evaluations to compare the list of elements prioritized by developers in their systems (Section 4.1.3) with the list of prioritized elements found by the heuristics. These evaluations differ in two points: (i) the set of systems used in the evaluation, and (ii) how we created the list of elements prioritized by developers. We explain these differences in detail in Section 4.1.2. Regardless of the evaluation, we applied the heuristics to find the list of prioritized elements. Each heuristic returns $TOPN$ relevant elements. The value of N *i.e*, how many relevant elements the heuristic should return, is defined by the developer. For our evaluation, we start by evaluating the top three elements ($TOP3$). After that, we evaluate the list of relevant elements with other sizes. We decided to reduce our set of relevant elements to only three because developers are likely to prioritize only a few elements, especially due to time constraints and effort to identify and remove design problems.

Our heuristics receive the code smells as input. We focused on code smells for some reasons. First, the goal of this dissertation is to investigate design-relevant smells. Hence, smells are the only category of symptom that we are interested in right now. Second, smells are part of developers' routine (Yamashita and Moonen 2013) and one of the most investigated type of symptom (Eick *et al* 2001, MacCormack, Rusnak and Baldwin 2006, Bertran 2011, Macia *et al.* 2012a, Macia *et al.* 2012b, Moha *et al.* 2010, Sousa *et al.* 2018, Palomba *et al.* 2014, Yamashita and Moonen 2013). Third, smells can be identified directly in source code, which is often the only available artifact, and they can be identified with accurate and customizable detection strategies validated in the literature (Aniche *et al* 2016, Vale, Fernandes and Figueiredo 2018). Fourth, restricting our heuristics to smells, we can focus on evaluating them. If we have used other category of symptoms, we could have introduced variables that would make the evaluation harder.

We expected that the *Smell Granularity* heuristic would achieve good results since developers tended to rely on the type of smells to do the prioritization (Chapter 3). However, *Smell Count* outperformed the simple heuristics, including *Smell Granularity*. We also expected that the combined heuristics would achieve good results since the combination of criteria was a common practice among developers. Yet, in many cases, they had the same precision as the simple heuristics.

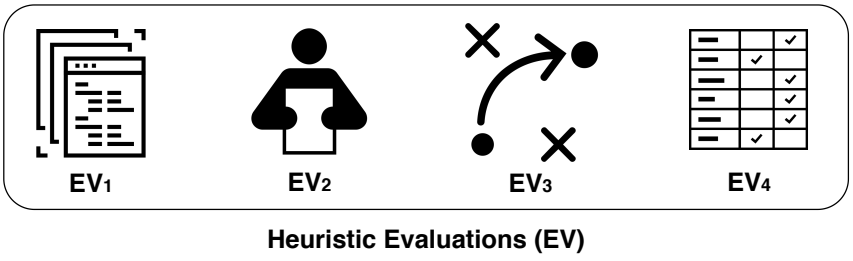

Figure 4.1: Study 2 settings

4.1.2 Experimental Activities

We followed four activities to evaluate our heuristics, as presented in Figure 4.1. First, we selected the projects that were suitable for the evaluation (EV_1). Second, we created a ground truth (EV_2) that would be used on the last step. Third, we ran our heuristics in the selected project (EV_3). On the last activity, we computed the precision of our heuristics (EV_4). Following, we describe in detail each one of these activities.

Projects Selection (EV_1). In the first activity, we selected 17 software projects, which we divided into two groups. Each group was assigned to one of our two evaluations. For the first group, we selected open source projects from GitHub. We chose open source projects aiming at future replications and extensions by other developers. We decided to select the projects from the GitHub since it is the largest open source community, which allowed us to pick heterogeneous projects, from different domains and sizes. To choose the projects, we used the following criteria:

- Projects with different popularity levels. We used the number of GitHub stars as a metric to quantify its popularity.
- Projects with at least 90% of the code repository effectively written in Java.
- Projects with at least 20 elements (classes) refactored during a given span of time of six months. This was the window defined by us to evaluate the commits.

These criteria allowed us to select 12 Java projects with a diversity of structure, size, and popularity, which are active and important to the software community. We focused on Java[2] projects because Java is a very popular programming language. Additionally, the tool that we used to collect the code

[2]Available at `https://www.java.com/en/download/`

Table 4.1: Java Projects' Details

Domain	Project	LOC	Number of Classes	Commits	Stars
Android	Facebook Fresco	50,779	860	744	14,679
	OkHttp	49,379	642	2,645	27,421
	PhilJay MPAndroidChart	23,060	268	1,737	23,036
Application	Achilles	83,123	653	1,188	207
	Containing	4,022,744	136	818	1
	Market-Monitor	3,763	44	125	0
Database	Apache Derby	1,760,766	3,741	8,135	140
	Presto DB	350,976	4,146	8,056	7,740
	Realm Java	50,521	1,018	5,916	19,938
Framework	Apache Dubbo	104,267	1,690	1,836	19,934
	Spring Framework	555,727	12,715	12,974	22,052
Web Application	Apache Tomcat	668,720	2,275	18,068	2,406

smells, Organic, was implemented to collect smells from Java projects. These projects are detailed in Table 4.1. The first column presents the domain of the project. The second column presents the project name. The third and fourth columns show the size of the project, respectively, in terms of lines of code (LOC) and the number of classes. The fifth column presents the number of commits on the project and finally, the sixth column shows the degree of popularity of the project, based on its stars.

We can observe in this table the heterogeneity of the projects: we have 12 projects from five different domains. Considering the size of the project, we have projects that range, for instance, from thousands of LOC to projects with hundreds of thousands of LOC. We also can consider the size of the projects based on the number of classes, where we have from projects with less than 100 classes to projects with more than 10.000 classes. Finally, we can see projects with zero stars to projects with more than 20.000 stars, which indicates the different levels of popularity of these projects. This heterogeneity will help us to evaluate how the heuristics perform in different scenarios.

For the second set of systems, we selected five systems from industry partners: Health Watcher (HW) (Soares, Laureano and Borba 2002), Mobile Media (MM) (Young 2015), Apache OODT (Mattmann *et al* 2006), P1, and P2. P1 and P2 are proprietary and, due to intellectual-property constraints, we omitted their names. We highlight that the systems used in this study are from different companies than the ones used on our qualitative study (Chapter 3). We focused on these systems because: (i) their architecture had degraded, (ii) they present a wide range of design problems, and (iii), different from the first set of projects, their original developers provided us with a reliable list of design problems that were causes of major maintenance effort. Following we describe these systems.

– **Health Watcher (HW)**: This is a web framework system that allows citizens to register complaints about the health issues in public institu-

Table 4.2: Characteristics of the Target Systems

Domain	Project	Design	LOC
Web Framework	Health Watcher	Layers	∼8,000
Software Product Line	Mobile Media	MVC	∼10,000
Desktop Application	P1	Client-Server	∼122,000
Desktop Application	P2	Client-Server	∼118,000
Middleware	OODT	Layers	∼129,000

tions (Soares, Laureano and Borba 2002).

– **Mobile Media (MM)**: This is an academic software product line to derive applications that manipulate photos, videos, and music on mobile devices (Young 2015).

– **P1 and P2**: P1 and P2 manage activities related to production and distribution of oil.

– **Apache OODT**: The goal of this system is to develop and promote the management and storage of scientific data.

Details about these systems are presented in Table 4.2. The first column presents the system domain. The second column presents the system name. The third column presents the design followed in the system implementation. The fourth column shows the size of the software system in terms of its lines of code (LOC). In this table, we observe the heterogeneity of the systems used in the study. This heterogeneity is represented from the different domains that each one belongs to the three different designs applied to them. Similar to the previous project, from GitHub, this set also contains systems both small and large in size.

Ground Truth Creation (EV_2). In the second activity, we created the ground truth to the evaluation, which is the list of elements prioritized by developers. As we have two sets of software projects, we had different procedures to create the ground truth for each set. We discuss these procedures in Section 4.1.3.

Running the Heuristics (EV_3). In the third activity, we ran our heuristics in all 17 software systems with the version used to create the ground truth (Section 4.1.3). Figure 4.2 depicts the phases of running the heuristics. In the first phase, we collected the smells in each project (Phase 1). To collect them, we used the Organic tool, which uses detection strategies based on a set of metrics and thresholds (Marinescu, 2004, Lanza and Marinescu 2006). Once we had the elements and their respective smells, we ran our heuristics in the second phase. In this phase, the heuristics receive as input the code smells.

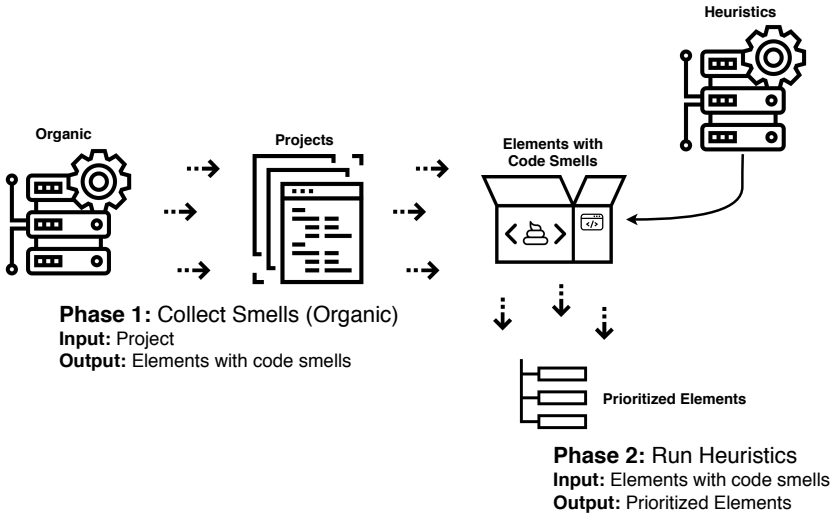

Figure 4.2: Running the Heuristics

Then, it produces the $TOPN$ prioritized elements found by each heuristic (Phase 2). The details about the heuristics were presented in Section 3.3.

Calculating the Precision (EV_4). In the last activity, we compared the list of elements prioritized by the heuristics with the list of elements in our ground truth. We use the comparison to calculate the Precision at K. The notion of precision at k is based on a definable integer K that is set by the developer to match the top-N elements. Since we are most likely interested in prioritizing a short list of smelly elements, we compute precision in the first $TOPN$ elements instead of all the elements. Hence, we define Precision based on the number of elements marked as *true positive* (TP), and *false positive* (FP). A TP occurs when the heuristics put in the $TopN$ an element that is also part of the ground truth list. A FP happens when the heuristics put in the $TopN$ an element that does not match any element in the ground truth list. The precision is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP} \tag{4-1}$$

Regarding the recall, it should measure the fraction of relevant elements that have been prioritized over the total amount of relevant elements in the system. However, in the prioritization context, the recall is a measure that does not play as much important role as precision. As explained before, the developer has to prioritize just a few elements due to time constraints, especially in large systems that may contain several elements. Consequently, to provide the whole set of relevant elements to developers analyze is unfeasible in practice. Therefore, the exactness (precision) of the prioritization heuristics is more important than its completeness (recall). After all, he will analyze only

a reduced set of elements.

4.1.3 Ground Truth Creation

We had two different sets of projects, which led us to two procedures to create our ground truth of elements prioritized by developers. Each procedure corresponds to how developers prioritized (elements with) design problems in each set of projects. As a result, we can evaluate the heuristics in finding those elements that developers had different criteria to prioritize.

Refactoring Prioritization List. For the 12 GitHub projects, we created a ground truth based on refactoring (Fowler 1999). Developers can improve the system structure using refactoring to remove design problems (Murphy Hill, Parnin and Black 2009). Therefore, we can identify the elements that developers prioritized looking at those elements they concentrated effort to refactor. Figure 4.3 presents the steps that we took to create the list of elements prioritized during refactoring. To identify them, our first step was to select a version of each project that could have design problems (Phase 1). For this selection, we analyzed the first 100 commits of each project, and we identified the commit (C_n) that had the highest number of smells. We analyze the first commits because some design problems are born with the systems (Oizumi *et al.* 2016). Thus, we want to verify if our heuristics can prioritize them. We selected the commit with the highest number of smells because they are likely to have design problems (Macia *et al.* 2012b, Bertran 2011, Oizumi *et al.* 2016, Yamashita and Moonen 2013).

After selecting the *initial commit (C_n)* of each project, we used the RefactoringMiner tool (Tsantalis *et al* 2013) to identify all elements refactored from the initial commit (C_n) until the commit six months later (C_{n+m}) (Phase 2). These are the elements that developers concentrate their effort to refactor, and that compose the refactoring prioritization list. We are aware that not all refactored elements contain design problems (Section 4.3); even so, we want to investigate if our heuristics can support developers to prioritize elements that they concentrate effort, for instance, by removing the design problems through refactoring.

Design Problems Prioritization List. For the five industry partners systems, we created a ground truth with the collaboration of the systems' original designers and developers. We performed two steps to incrementally develop the ground truth. First, developers provided us with an initial list of design

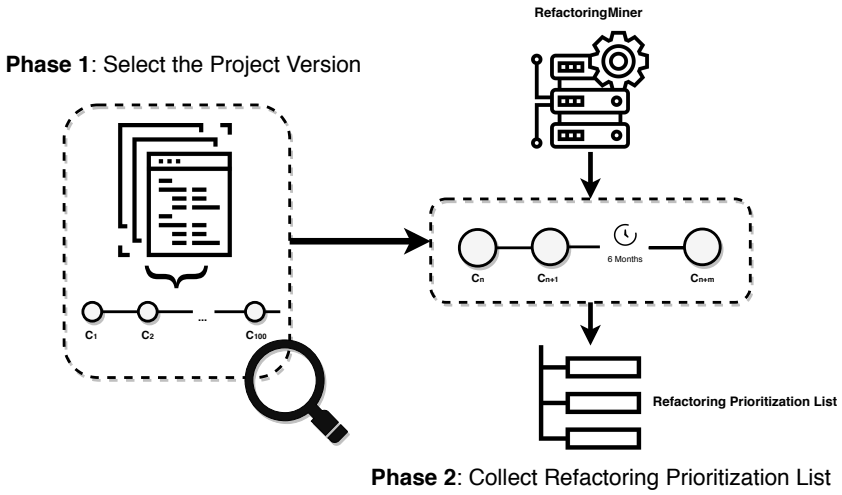

Figure 4.3: Phases to Create the Refactoring Prioritization List

problems. They listed the problems and explained the relevance of each one. They reported the maintenance effort caused by the presence of each design problem. They also described which elements were affected by the design problem. Second, we performed other steps to validate the initial list for correctness and completeness. Additional identification of design problems was performed using the source code and the system design. For systems without design documentation, we relied on a suite of design recovery tools (Garcia *et al.* 2013).

The procedure for deriving the list of design problems with developers was the following:

1. We identified an initial list of design problems using the SCOOP tool (Macia *et al.* 2012a), which detects code smells relevant to the identification of design problems.

2. The developers had to confirm, refute or expand this list.

3. The developers provided a brief explanation about the relevance of the design problem.

4. When we suspected there was still inaccuracies in the list, we asked them for further feedback.

At the end of this procedure, we had the list of design problems validated by the original designers and developers from the 5 software systems.

4.2 Results and Discussion

We discuss here the results for the evaluations of our nine heuristics. As presented in our first study (Chapter 3), we have two sets of heuristics. The first set is composed of *Smell Granularity*, *Smell Count* and *Diversity* heuristics. They are simple heuristics in the sense of using only a single criterion to prioritize elements. Even though we expected that *Smell Granularity* would achieve good results since the developers often rely on smells to identify the design problems, *Diversity* was the heuristic with better results among the simple ones. The second set of heuristics is composed of *Count & Div*, *Div & Count*, *Count & Gran*, *Div & Gran*, *Gran & Count* and *Gran & Div* heuristics. They are combined heuristics since they use a second criterion to make the tiebreaker in the case of a tie. Since developers combine the criteria in practice (Chapter 3), we expected these heuristics to achieve better results than the simple heuristics. However, in multiple cases, they had the same precision as the simple heuristics that served as tiebreakers. At the same time, it was also expected that not necessarily all heuristics would have good effectiveness, since some prioritization criteria used by developers may not be suitable for the prioritization of relevant elements.

The following subsections present the results and discussions for the evaluations. Section 4.2.1 presents the evaluation with the GitHub projects. Section 4.2.2 presents the evaluation with the industry partners projects.

4.2.1 Evaluating the Heuristics with the GitHub Projects

We evaluated the heuristics with 12 GitHub projects. Since we did not have access to the developers of these projects, we used the refactoring operations to identify the elements that developers prioritized in these projects over a period of time, *i.e*, our ground truth. In this sense, we can evaluate to what extent our heuristics is able to find elements that developers would focus their effort to remove design problems, which is the advantage of evaluating the heuristics with these projects. After applying the heuristics to find the $Top3$ elements, we compared the elements with our ground truth of refactoring prioritized elements. Then, we calculated the *precision*, which is shown in Tables 4.3 and 4.4.

Smell Count as the heuristic with the best precision. When we consider only the simple heuristics, *Smell Count* had better precision, 50.55% on average. With this computation of the average of the projects, we can estimate how our heuristics would perform in different projects chosen

Table 4.3: Precision of the heuristics in finding elements prioritized during refactoring operations a)

Simple Heuristics			
Project	**Smell Granularity**	**Smell Count**	**Diversity**
Achilles	1/4 (25.00%)	3/3 (100.00%)	5/5 (100.00%)
Apache Derby	1/3 (33.33%)	2/3 (66.67%)	3/8 (37.50%)
Apache Dubbo	2/3 (66.67%)	2/3 (66.67%)	1/3 (33.33%)
Apache Tomcat	1/3 (33.33%)	0/3 (0.00%)	1/6 (16.67%)
Containing	0/6 (0.00%)	1/3 (33.33%)	2/8 (25.00%)
Facebook Fresco	2/3 (66.67%)	0/3 (0.00%)	1/5 (20.00%)
Market Monitor	0/3 (0.00%)	0/6 (0.00%)	0/5 (0.00%)
Media Magpie	1/9 (11.11%)	2/3 (66.67%)	1/4 (25.00%)
PhilJay MP Android Chart	3/3 (100.00%)	6/6 (100.00%)	5/5 (100.00%)
Presto DB	0/5 (0.00%)	2/3 (66.67%)	1/3 (33.33%)
Realm Java	1/13 (7.69%)	2/3 (66.67%)	1/7 (14.29%)
Spring Boot	0/4 (0.00%)	2/5 (40.00%)	1/6 (16.67%)
Average	**28.64%**	**50.55%**	**32.79%**

Table 4.4: Precision of the heuristics in finding elements prioritized during refactoring operations b)

Combined Heuristics						
Project	**Count & Div**	**Div & Count**	**Count & Gran**	**Div & Gran**	**Gran & Count**	**Gran & Div**
Achilles	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)	5/5 (100.00%)	1/4 (25.00%)	1/4 (25.00%)
Apache Derby	2/3 (66.67%)	2/3 (66.67%)	2/3 (66.67%)	2/4 (50.00%)	1/3 (33.33%)	1/3 (33.33%)
Apache Dubbo	2/3 (66.67%)	1/3 (33.33%)	2/3 (66.67%)	1/3 (33.33%)	2/3 (66.67%)	2/3 (66.67%)
Apache Tomcat	0/3 (0.00%)	0/3 (0.00%)	0/3 (0.00%)	0/4 (0.00%)	1/3 (33.33%)	1/3 (33.33%)
Containing	1/3 (33.33%)	2/8 (25.00%)	1/3 (33.33%)	0/5 (0.00%)	0/6 (0.00%)	0/6 (0.00%)
Facebook Fresco	0/3 (0.00%)	0/3 (0.00%)	0/3 (0.00%)	1/5 (20.00%)	2/3 (66.67%)	1/3 (33.33%)
Market Monitor	0/5 (0.00%)	0/5 (0.00%)	0/6 (0.00%)	0/3 (0.00%)	0/3 (0.00%)	0/3 (0.00%)
Media Magpie	2/3 (66.67%)	1/3 (33.33%)	2/3 (66.67%)	1/4 (25.00%)	1/9 (11.11%)	1/7 (14.29%)
PhilJay MP Android Chart	3/3 (100.00%)	3/3 (100.00%)	6/6 (100.00%)	5/5 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
Presto DB	2/3 (66.67%)	1/3 (33.33%)	2/3 (66.67%)	1/3 (33.33%)	0/5 (0.00%)	0/5 (0.00%)
Realm Java	2/3 (66.67%)	1/5 (20.00%)	2/3 (66.67%)	1/7 (14.29%)	1/13 (7.69%)	1/13 (7.69%)
Spring Boot	1/4 (25.00%)	1/5 (20.00%)	2/5 (40.00%)	0/3 (0.00%)	0/4 (0.00%)	0/4 (0.00%)
Average	**49.30%**	**40.35%**	**50.55%**	**31.30%**	**28.64%**	**26.13%**

randomly. With the exception of three projects, it was able to correctly prioritize at least one element in each analyzed system. Since the smells are used as an indicator of refactoring, it was expected that the element with the most smells would be the one that the developer would do the refactoring. However, we were not expecting that the *Smell Count* would outperform the other heuristics. In fact, we were expecting that the *Smell Granularity* would be the heuristic with the best precision among the simple ones, since it is based on how developers frequently used the smells, considering its type.

The nature of some design problems is the second factor that makes this result about *Smell Count* surprising. When some design problems manifest in the source code, smells from different types will be related to them. Thus, only looking at the number of smell instances may not be a good approach. For instance, let us consider an `A` class that uses a third-party library Z. This class is likely to make several calls in chains to access the functionality of Z (`Z.getX().getY().getZ()`); thus, this class may contain several instances of Message Chain smell. A heuristic such as *Smell Count* would mistakenly prioritize this class.

On the other hand, let us consider our example from Section 1.1 - Figure 1.1, where we have the system S1, that manages loans and sales of

printer devices. The *Concern Overload* manifested in the `DeviceRepository` class as smells from different types: Complex Class, God Class, Long Method, and others. In this scenario, a heuristic such as *Diversity* should supposedly outperform the *Smell Count*, or at least find the same precision. In practice, this result suggests that *Smell Count* heuristic may provide a good start point to developers who do not know to where start the analysis of the elements.

The negative effect of the smell type. Regarding the other two simple heuristics, *Diversity* had better precision than the *Smell Granularity*. Notice that both heuristics consider the type of each smell to prioritize elements instead of blindly considering only the number of smell instances such as the *Smell Count* heuristic does. This result suggests that the smell type did not prove to be appropriated to derive heuristics for these projects. At first sight, this finding seems to be unlikely since contradicts to the first study. As we found in the first study, the smell type was the most used criterion. Thus, one may expect that the smell types would be appropriate to derive the heuristics since developers used them so often.

However, the more the heuristic is based on the smell type, the worse is its precision. For instance, the *Smell Granularity* is the one that most relies on the smell types, which had the worst precision. *Diversity*, in its turn, is a midterm between *Smell Granularity* and *Smell Count*. In other words, it simultaneously relies on the smell type and on smell instances – in this case, relying on the number of different types of smells. Hence, it has better precision than *Smell Granularity*, but lower than *Smell Count*. When we consider the combined heuristics, we can also notice that the smell type was not appropriate to derive heuristics for the projects. Comparing the *Diversity* and *Div & Gran*, there was a slight decrease in the precision.

Combining criteria. As mentioned, we expected that the combined heuristics would achieve good results since developers combine criteria in practice (Section 4.2). However, the results show the opposite. With the exception of *Count & Gran*, the other combined heuristics had lower precision when compared to the *Smell Count*. In fact, our main expectation was that *Count & Gran* would have the best precision since (i) it is based on how developers used smells, and (ii) it is also based on the second most used criterion (number of smells). However, it had the same precision as *Smell Count*. This result suggests that the smell type was not appropriate for breaking a tie between elements in these projects, *i.e*, the smell type had no effect on how the heuristics prioritized elements.

Precision equal to zero. When we look at the data in Tables 4.3 and 4.4, we can notice cases where the heuristics could not prioritize any

element correctly, leading to a precision equal to zero (cells in red color). In this sense, we analyzed the Market-Monitor project to understand why no heuristic was able to prioritize elements correctly. We found that the elements in our ground truth were not related to design problems. Consequently, the heuristics were correct in not prioritizing these elements, leading to the precision of 0%. This result made us wonder if our ground truth was inappropriate for the evaluation. The ground truth list may contain elements that are not related to design problems, which justifies the low precision in other projects as well. In this context, we decided to investigate our ground truth.

To create the ground truth for the GitHub projects, we searched for elements that developers concentrated effort in refactoring during a timespan of six months (Section 4.1.3). These are the elements that may contain design problems, which could have motivated the refactoring. We suspected that these elements could have been refactored due to other reasons unrelated to design problems. Therefore, we conducted an analysis on seven projects that had the best or worst precision with our heuristics: Achilles, PhilJay MPAndroidChart, Presto DB, Realm Java, Apache Tomcat, Facebook Fresco, and Market-Monitor. We manually analyzed them to see if the elements in the ground truth were related to design problems.

Smell Count as the most appropriate heuristic to find elements that should be refactored. After our manual analysis, we found that most elements on these selected projects were not related to design problems. Hence, our ground truth had elements that were prioritized due to other reasons not related to a design problem. This happens because we used the refactoring to identify elements with design problems. However, a refactoring operation can have other goals not related to the removal of design problems (Murphy Hill, Parnin and Black 2009). For instance, the developers may apply refactoring operations to solve a bug or to add a feature. In these cases, the prioritized elements are not the same elements that contain design problems. This result indicates that the *Smell Count* heuristic was able to find the elements that developers refactored. Consequently, it is the most appropriate one to find elements which developers should focus effort during refactoring.

The result of the ground truth also suggests that the ground truth was inappropriate to evaluate if the heuristics prioritize relevant elements. It was inappropriate because it contained elements that did not have design problems. Consequently, we can not evaluate if our set of heuristics find relevant elements. Therefore, we conducted a second evaluation with a new set of projects, in which we had a different procedure to create the ground truth (Section 4.1.3).

Table 4.5: Precision of the heuristics in finding elements prioritized using design problems a)

Simple Heuristics			
Project	**Granularity**	**Smell Count**	**Diversity**
Apache OODT	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
Health Watcher	0/3 (0.00%)	0/3 (0.00%)	0/3 (0.00%)
Mobile Media	3/3 (100.00%)	2/3 (66.67%)	3/3 (100.00%)
P1	3/3 (100.00%)	1/3 (33.33%)	3/3 (100.00%)
P2	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
Average	**80.00%**	**60.00%**	**80.00%**

Table 4.6: Precision of the heuristics in finding elements prioritized using design problems b)

Combined Heuristics						
Project	**Count & Div**	**Div & Count**	**Count & Gran**	**Div & Gran**	**Gran & Count**	**Gran & Div**
Apache OODT	5/6 (83.33%)	3/6 (50.00%)	3/6 (50.00%)	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
Health Watcher	0/7 (0.00%)	0/6 (0.00%)	0/7 (0.00%)	0/3 (0.00%)	0/3 (0.00%)	0/3 (0.00%)
Mobile Media	3/8 (37.50%)	3/6 (50.00%)	3/8 (37.50%)	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
P1	1/6 (16.67%)	3/7 (42.86%)	1/6 (16.67%)	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
P2	3/6 (50.00%)	3/6 (50.00%)	3/6 (50.00%)	3/3 (100.00%)	3/3 (100.00%)	3/3 (100.00%)
Average	**36.36%**	**38.71%**	**30.30%**	**80.00%**	**80.00%**	**80.00%**

4.2.2 Evaluating Design Problems with Prioritized Elements

In the previous section, we found that the *Smell Count* heuristic was the most appropriate to find elements that should be refactored. However, we also found that (i) the smell type was not appropriate to derive heuristics in those projects, and (ii) the combined heuristics did not have the better results, which was contrary to our expectations. However, how we created our ground truth for the first evaluation jeopardizes these two last findings. Hence, we had to conduct another evaluation. For this evaluation, we relied on the list of prioritized elements that the original developers of five systems from our industry partners provided to us. Tables 4.5 and 4.6 presents the precision of the heuristics in finding the $TOP3$ elements for these systems.

The positive effect of the smell type. Analyzing the precision in Tables 4.5 and 4.6, we can notice the inverse result to the one obtained with the GitHub projects. The heuristics with the best precision values were exactly those that somehow use the smell type. In our first study, we noticed that all developers rely on the smell types frequently (Chapter 3). Therefore, we were expecting that the *Smell Granularity* would achieve good results. This second evaluation confirms our expectation, which can justify why developers rely on the smell type so often. In fact, to developers identify a design problem, they have to reason about different types of smells (Section 1.1). In our first study, we noticed that when the developers are concerned about a design problem, they tend to consider also the type of each smell. Consequently, heuristics that consider intrinsic information about the symptoms – in our case, heuristics

Table 4.7: Top N Prioritized Elements (Simple Heuristics)

Simple Heuristics			
Top	**Granularity**	**Smell Count**	**Diversity**
5	68.00%	53.57%	65.38%
7	66.67%	56.76%	70.27%
10	67.31%	58.82%	71.70%
15	77.27%	58.75%	76.29%
20	76.58%	63.85%	76.79%

Table 4.8: Top N Prioritized Elements (Combined Heuristics)

Combined Heuristics						
Top	**Count & Div**	**Div & Count**	**Count & Gran**	**Div & Gran**	**Gran & Count**	**Gran & Div**
5	41.18%	32.08%	35.29%	68.00%	68.00%	68.00%
7	31.08%	30.95%	32.43%	66.67%	66.67%	66.67%
10	26.92%	33.93%	25.38%	67.31%	67.31%	67.31%
15	26.04%	46.45%	27.22%	77.27%	77.27%	77.27%
20	38.25%	42.86%	37.79%	76.58%	76.58%	76.58%

that considered the type of each smell – are likely to reach a good precision and it is aligned to how developers identify design problems in practice.

Unfortunately, the data in Tables 4.5 and 4.6 can not provide us with much more insights: all heuristics have similar results. For instance, none of the heuristics were able to correctly prioritize elements in the Health Watcher system. Additionally, we do not know which heuristic had the best precision only looking at the $TOP3$ elements. We also can not say how the combination of criteria influenced the heuristics. Therefore, we also ran the heuristics with other $TOPN$ elements to gather more data about each heuristic. Tables 4.7 and 4.8 shows the average precision for all the projects in with multiple $TOPN$ prioritized elements.

Using a second criterion as a tiebreaker. Another expectation that we had was about the combination of criteria. Since developers combine criteria in practice, we were expecting that the combined heuristics would achieve good results. Indeed, analyzing the results from Table 4.8, they achieve high precision. However, we can notice the same results between the simple heuristics and the combined heuristics. *Smell Granularity* had the same precision that *Div & Gran* and the other two heuristics where the type was considered first, *Gran & Count* and *Gran & Div*; *Smell Count* had a better precision that *Count & Div* and *Count & Gran*; and *Diversity* had a better precision than *Div & Count*. This result indicates that the way we have combined the criteria is not the best approach. In our cases, we used a second criterion to break a tie between elements. However, this approach is not showing effect. Consequently, this similar result among the heuristics reinforces that we need to find better ways to combine criteria.

Diversity as the most appropriate heuristic for prioritizing

elements with design problems. Since simple and combined heuristics have the same precision, or when combined, they had a lower precision, let us focus on the simple ones to discuss which heuristics are most appropriate to prioritize elements. According to Table 4.7, *Smell Granularity*, and *Diversity* were the two heuristics with the best precision. However, the *Diversity* heuristic seems to be the most appropriate heuristic for the prioritization since it overcame the *Smell Granularity* in three scenarios: $TOP7$, $TOP10$, and $TOP20$. Regarding the Health Watcher system. Both heuristics correctly prioritized elements in the Health Watcher system after $TOP7$, thus no longer having precision equal to 0. These two heuristics had the same precision values in the Health Watcher system. Additionally, they were the only ones that had precision values different from zero after $TOP7$.

Improving the heuristics. An approach to improve the heuristics is finding another way to combine the criteria. This is necessary as the use of a second criterion to generate the combined heuristics did not improve the results of the simple versions of the heuristics. For instance, when the diversity criterion is used as the tiebreaker of the number of smells criterion, the resulting heuristic (*Count & Div*) did not improve the results of its simple version (*Smell Count*), as we can see in Tables 4.7 and 4.8. This result suggests that the way we have used a second criterion as a tiebreaker is not the better way to combine criteria.

This finding on how we combined the criteria can also be noticed by the number of elements that each heuristic returned (Table 4.6). We set the heuristics to return only three most prioritized elements. However, they returned more than 3 elements in some cases. This happens because the heuristics return the three first elements that fit the criterion, however, if a fourth element ties with the third, the heuristics also return it. The heuristic returns more consecutive elements until there is no more tie. This behavior also happens with the combined heuristics, even though they have a mechanism to solve ties. However, their mechanism did not suffice to break the tie, which leads to the same precision results in Tables 4.6 and 4.8. For instance, the precision of the *Smell Granularity* is the same as the combinations with smell count (*Gran & Count*) and with diversity (*Gran & Div*).

Another approach to improve the heuristics is instead of considering the smell levels, a heuristic should consider how the smells are related to each other. In this case, a heuristic that uses the relation criterion may improve the results (Section 3.3.2). The use of these specific heuristics can reach good results since it will consider more intrinsic information of the system. This way, these heuristics will be more related to the system that is being applied.

Another option would be to consider the role of the elements in the system. Nevertheless, to use these criteria, we should consider subjective information about the systems (Section 3.3.2).

4.3 Threats to Validity

This section presents the threats to the validity of this study. For each threat, we present the actions taken to mitigate its impact on the final result.

First, we analyzed just the 100 first commits of the project to select the elements and apply our heuristics. One can argue that this is a low number of commits. However, this decision was based on a study that shows that some design problems born with the system (Oizumi *et al.* 2016), motivating us to consider only the first 100 commits. Also, the use of six months as a time span for the commits analysis can be considered a short period of time. We used this value as an arbitrary value. This way is possible that occurred some refactoring in the same element after those 6 months, which could change the order of the prioritized elements. In this case, we are not able to get these elements. However, this period of time helps us to identify if when a developer identifies a design problem in an element, he promptly does a refactoring in this element.

Even though there are other categories of design problem symptoms, our heuristics have been defined in function of the code smells. We highlight that these heuristics can be applied to any category of symptom. Nevertheless, we have restricted them to smell since our focus is to find design-relevant smells. As explained before, code smells have been used in several studies, and they are one of the most investigated category of symptoms (Oizumi *et al.* 2016, Arcoverde *et al.* 2013). Another reason that we have focused on smells is because the companies' managers mentioned that some of the developers involved in the study not only were familiar with smells but also had the culture of using them. Furthermore, our goal was to explain the prioritization of design problems, and not delve too deeply into the symptoms.

As we found later, our ground truth for the first set of projects (GitHub projects) had elements unrelated to design problems. To mitigate this threat, we used the second set of projects (Industry partners projects), using a new procedure to create the ground truth, where now we consider the elements affected by design problems identified by the SCOOP tool (Macia *et al.* 2012a), confirmed by the developer and then validated by our team.

4.4 Summary

We analyzed the elements prioritized by the heuristics to understand how the smells were related to the design problems. We noticed that the type of each smell was indeed important to prioritize elements with design problems. This was one of the reasons why *Smell Granularity* and *Diversity* had better precision. Indeed, the smell type was also the reason why these both heuristics were able to correctly prioritize elements that were not related to design problems in our first set of projects. As that grounded truth had elements unrelated to design problems, these heuristics did not prioritize them. Another reason why *Smell Granularity* and *Diversity* had better precision is due to the total number of smells. We found some elements that had several smells that were unrelated to design problems. Since these elements had several smells, the *Smell Count* heuristic prioritized them. On the other hand, both *Smell Granularity* and *Diversity* did not prioritize these elements.

In summary, the first evaluation shows us that the *Smell Count* heuristic can be used by developers who are short in time and do not have deep knowledge about the system' design. Conversely, the result found in the second evaluation indicates that a heuristic to prioritize elements with design problems should consider intrinsic information about the symptoms. In our case, our heuristics considered the type of each smell. As a result, the two heuristics based on the smell type had the best precision values. The intrinsic information is important to the developer confirm if the smells are relevant or not to the identification of a design problem. In other words, it helps the developers to confirm if that smell is a design-relevant one, consequently prioritizing the relevant element.

Thus, our evaluations provided us with promising results, which also motivate us to study the specific heuristics (Section 3.3.2) in the future.

5
Conclusions

Design problems are the result of bad design decisions that can affect some quality attributes of the system, such as maintainability and performance (Garcia *et al.* 2009b). When neglected, a design problem can lead the software to high maintenance costs in the future and even its discontinuity (MacCormack, Rusnak and Baldwin 2006, Godfrey and Lee 2000, Schach *et al.* 2002, Gurp and Bosch 2002). Thus, they should be identified and removed as soon as possible. However, identifying a design problem can be a challenging task. For instance, a design problem can be scattered through many elements (*e.g.* classes and packages) of the system. Given the large number of elements in a system, developers have to prioritize the ones that are likely to have design problems. In addition, as design documentation is often unavailable or outdated (Trifu and Reupke 2007, Kaminski 2007), developers have to rely on implementation-level indicators of design problems on the source code. These indicators are the so-called symptoms (Sousa *et al.* 2017), which are partial signs of the presence of a design problem. In this sense, developers can rely on code smells, which is one of the most investigated symptoms in the literature (Yamashita and Moonen 2013, Macia 2013, Moha *et al.* 2010, Oizumi *et al.* 2016, Oizumi *et al* 2018).

Multiple studies have focused on the relation between design problems and code smells. Some of them have even presented the code smells as indicators of design problems (Yamashita and Moonen 2012, Macia *et al.* 2012a, Oizumi *et al.* 2016). However, they do not focus on the task prior to the identification of design problems, which is the prioritization of relevant smelly elements. A *relevant smelly element* (or relevant element) is an element that contains at least one design-relevant smell. These design-relevant smells are the ones used on the identification of design problems. Some studies have covered the prioritization of relevant elements (Arcoverde *et al.* 2013, Vidal *et al.* 2016, Guimaraes *et al* 2018), by proposing some heuristics for this prioritization. However, they do not investigate in practice how the developers prioritized elements. In this context, this dissertation addressed the gaps and limitations of the literature by proposing heuristics for the prioritization of relevant elements; these heuristics were based on criteria used by developers.

To propose and evaluate the prioritization heuristics, we performed two studies. In the first one, we observed how developers prioritized relevant elements in practice. First, we asked them to prioritize relevant elements on their source code. We gave them a summarized list of smells affecting the elements, thus they could prioritize the design-relevant smells and, then, further confirm whether the element was relevant or not to identify a design problem. After the experiment, we analyzed how they prioritize the relevant elements. Based on this analysis, we extracted criteria that they commonly used during the prioritization task. The criteria are *smell type*, *number of smells*, *type diversity*, *element role* and *relation*.

Together with the actions taken by the developers during the prioritization task, we used those criteria to propose a suite of prioritization heuristics. These heuristics used the criteria that we identified in the experiment. For instance, we proposed a suite of nine heuristics that rely on three criteria that are independent of the system (*smell type*, *number of smells* and *type diversity*). Using these three criteria and combining them we derived this extended suite of heuristics. However, we did not know if the criteria applied by the developers on the prioritization were appropriate to derive effective heuristics. Thus, we designed and executed a study to evaluate the heuristics.

In the second study, we evaluated the heuristics in terms of their effectiveness in finding a small sub-set of relevant smelly elements in a top priority list of N elements ($TOPN$). To evaluate these heuristics, we applied them in two sets of projects. The first set of projects was composed of software projects from GitHub, which had a certain degree of structural degradation. To create the ground truth for those projects, we identified the elements that had at least one refactoring, indicating that the developers at some point had to dedicate effort in refactoring this element. The second set of projects were composed of projects of our industry partners. In this case, the original developers provided us with the design problems affecting the elements in the system. We used these affected elements as the relevant elements. For each of this set of projects, we compared the list of relevant elements prioritized by our heuristics against the set of relevant elements defined in the ground truth.

The first evaluation shows that the *Smell Count* heuristic can be used by developers who need to prioritize elements that need to be refactored. Based on the second evaluation, we found that the heuristics for the prioritization of relevant elements with design problems should consider intrinsic information of the smell affecting the elements. For instance, both heuristics with best results used the type of smells to prioritize the elements. This intrinsic information is important to the developers so they can confirm if the smell is a design-relevant

smell, *i.e.*, if the smell can help the developer on the identification of a design problem. Confirming that there is a design-relevant smell in the element, the developer then ends up confirming the prioritization of this relevant element.

5.1 Contributions

In this dissertation, we discussed the need of the developer on receiving support during the prioritization of relevant smelly elements. This prioritization is a challenging and time-consuming task that precedes the identification of design problems. In this context, we identified the criteria that they commonly use during the prioritization task. In addition, we proposed a suite of nine heuristics to support the developer on this prioritization. In a nutshell, the contributions of this dissertation are described as follows.

Criteria used by the developers during the prioritization of design problems (Chapter 3). In order to understand how the developers prioritized design problems, we observed how they do this task in practice. During this observation, we found that they tended to use criteria that can be divided into two categories. First, we have the general criteria, which can be defined as the criteria that can be easily replicated to any system. For instance, the number of code smells in an element can be collected independently of the system. The second scope comprises the specific heuristics, which use more specific information of the system. For instance, for the criterion that considers the role of an element in the system, first, we need to know how what are the roles (for instance a Service) and what are their responsibilities in the system. The responsibilities of a same role can vary from system to system.

Heuristics for the prioritization of relevant elements (Chapter 4). From the observation of the practice, we could propose the prioritization heuristics aligned with how the developers do the prioritization task. Among the heuristics proposed, we reached good results on the precision with heuristics such as *Diversity* and *Smell Granularity*. Also, we proposed a first sketch of how the specific criteria can be used for the creation of new heuristics.

Empirical Findings. We conducted two empirical studies to both propose and evaluate the heuristics. The main findings are described as follows.

1. **Smell Count as the most appropriate heuristic to find elements that should be refactored.** In the evaluation with the first set of

projects, *Smell Count* had the best precision among the others, with 50.55% on average. With the exception of three systems, it correctly prioritized at least one relevant element on each system. Upon manual analysis of the prioritized elements on the evaluated projects, we found that most of them did not have a design problem. Hence, these results indicate that the *Smell Count* heuristic was able to find elements that the developers refactored. Thus, among our evaluated heuristics, *Smell Count* is the most appropriate to identify elements that developers should focus on refactoring. These results are satisfactory since the goal of the heuristics is to reduce the search space, aiming to find the relevant elements. In fact, these results are even more satisfactory when taking into account that only a small percentage of smelly elements are related to design problems, as mentioned in Section 1.1. For example, in the case of the Apache OODT (Section 1.1), less than 20% of the smelly elements are related to design problems, *i.e.*, the heuristic was able to find half of the relevant elements. Additionally, We highlight that the results for this heuristic (and for the others) are exciting considering that this dissertation covered our first attempt to propose heuristics, and, in the future, we want to investigate other heuristics such the specific ones (Section 5.3).

2. **The use of intrinsic information of a smell to prioritize relevant elements.** In the second set of systems, we created a ground truth where the developers of the systems provided us with the design problems and elements affected. With this new ground truth, *Diversity* and *Smell Granularity* had the best results. Both of these heuristics used intrinsic information of the smell: its type. This indicates that to prioritize relevant elements, this intrinsic information should be considered. If we consider the average precision of these heuristics, we have better precision (80%) than the literature approaches that use smells on the prioritization. In addition, this average is applied in more projects than the approaches of the literature. In addition, developers tend to use this information to confirm if a smell is a design-relevant smell and then prioritize the relevant element. In order to identify a design problem, developers tend to evaluate different types of smells (Section 1.1), as we also noticed in our first study (Chapter 3). Thus, the use of this intrinsic information is aligned with how developers identify design problems in practice.

Table 5.1: Papers produced during the MSc

Type	Paper
MSc Research	Willian Oizumi, Leonardo Sousa, Alessandro Garcia, Roberto Oliveira, **Anderson Oliveira**, OI Agbachi, Carlos Lucena. *Revealing design problems in stinky code: a mixed-method study.* In Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '17)*
	Leonardo Sousa, **Anderson Oliveira**, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, Carlos Lucena, Rodrigo Paes. *Identifying design problems in the source code: a grounded theory.* Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*
	Willian Oizumi, Leonardo Sousa, **Anderson Oliveira**, Alessandro Garcia, Anne Benedicte Agbachi, Roberto Oliveira, Carlos Lucena. *On the identification of design problems in stinky code: experiences and tool support.* In Journal of the Brazilian Computer Society (JBCS '18)
	André Eposhi, Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Roberto Oliveira, **Anderson Oliveira**. *Removal of design problems through refactorings: are we looking at the right symptoms?.* In International Conference on Program Comprehension 2019 - Negative Results Track (ICPC' 19)
	Anderson Oliveira, Leonardo Sousa, Alessandro Garcia, Willian Oizumi. *On the Prioritization of Design-Relevant Smells: A Mixed-Method, Multi-Project Study.* In International Symposium on Empirical Software Engineering and Measurement (ESEM '19 - To Submit)
Other Contributions	Leonardo Sousa, Rafael de Mello, Diego Cedrim, Alessandro Garcia, Paolo Missier, Anderson Uchôa, **Anderson Oliveira**, Alexander Romanovsky. *VazaDengue: An information system for preventing and combating mosquito-borne diseases with social networks.* In Information Systems (IS '18)
	Eduardo Fernandes, Anderson Uchôa, Leonardo Sousa, **Anderson Oliveira**, Rafael de Mello, Luiz Paulo Barroca, Diogo Carvalho, Alessandro Garcia, Baldoino Fonseca, Leopoldo Teixeira. *VazaZika: A Software Platform for Surveillance and Control of Mosquito-Borne Diseases.* In 16th International Conference on Information Technology: New Generations (ITNG '19)
*Distinguished paper award	

5.2 Research Publications

During the MSc, some papers were produced in the context of this dissertation. In addition, some other papers were produced in partnership with colleagues. Table 5.1 presents these papers. The first five papers are associated with this dissertation. The fifth paper is not yet published. The remaining papers are the ones produced during the MSc in contexts different than this dissertation.

5.3 Future Work

The prioritization heuristics presented in this dissertation can be seen as a starting point for techniques to support the developer on the prioritization of relevant elements. We also found some specific criteria that can be explored in new heuristics. We summarize below some potential ideas for future work, regarding the prioritization of relevant elements.

Exploring the specific heuristics. As discussed, we identified two specific heuristics: *element role* and *relation*. We aim to explore these heuristics, which we expect will have good results since they consider intrinsic information of the systems. We expect these good results since the heuristics that considered intrinsic information (*diversity* and *smell granularity*) had the best results.

Replication of the study with refactorings. We aim to replicate the study with the GitHub projects, but now with a new ground truth. In this new

approach, we will focus on retrieve refactorings related to design problems. Thus, we will focus on refactorings on an architectural level, such as the ones applied to interfaces or the moving of methods among packages.

Application of the prioritization heuristics with other symptoms. In this dissertation, we focused only on the use of code smells. However, there are other types of symptoms that the developers tend to use to identify design problems (Sousa *et al.* 2017). We aim to explore these symptoms and verify if they reach better results than just considering the smells. Also, we aim to combine more than one type of symptom, which we expect that will lead to heuristics with better precision results (Section 4.2).

Extension of the evaluation to include new projects. We aim to search for projects with higher structural degradation than those used in our empirical studies presented in this dissertation. We will be able to identify more relevant elements in such projects, and further identify which information should be taken into account for the prioritization heuristics. Thus, we may be able to improve the effectiveness of our prioritization heuristics.

Tool support for applying the prioritization heuristics. Once we proposed the heuristics, we aim to provide the developers with a tool that will present them the prioritized elements on the system. A first idea is to implement this tool in a way that allows the developer to analyze the relevant element in terms of the smells affecting the element, the scope of the affected element and the reason for the smell be affecting the element. In addition, we aim to implement suitable visualization that can somehow complement our heuristics. Thus, the developer will be able to better prioritize the design-relevant smells.

Bibliography

[Abbes *et al.* 2011] ABBES, M.; KHOMH, F.; GUEHENEUC, Y. ; ANTONIOL, G.. **An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension**. In: PROCEEDINGS OF THE 15TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE; OLDENBURG, GERMANY, p. 181–190, 2011.

[Aniche *et al* 2016] ANICHE, M.; GEROSA, M. A. ; TREUDE, C.. **Developers' perceptions on object-oriented design and architectural roles**. In: PROCEEDINGS OF THE 30TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES '16, p. 63–72, New York, NY, USA, 2016. ACM.

[Arcoverde *et al.* 2013] ARCOVERDE, R.; GUIMARÃES, E.; MACÍA, I.; GARCIA, A. ; CAI, Y.. **Prioritization of code anomalies based on architecture sensitiveness**. In: 2013 27TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 69–78, Oct 2013.

[Baker, Hoek and Petre 2012] BAKER, A.; VAN DER HOEK, A.; OSSHER, H. ; PETRE, M.. **Guest editors' introduction: Studying professional software design**. IEEE Software, 29(1):28–33, Jan 2012.

[Bass, Clements and Kazman 2003] BASS, L.; CLEMENTS, P. ; KAZMAN, R.. **Software Architecture in Practice**. Addison-Wesley Professional, 2003.

[Bertran 2011] BERTRAN, I. M.. **Detecting architecturally-relevant code smells in evolving software systems**. In: PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '11, p. 1090–1093, New York, NY, USA, 2011. ACM.

[Booch 2004] BOOCH, G.. **Object-Oriented Analysis and Design with Applications (3rd Edition)**. Addison Wesley, Redwood City, CA, USA, 2004.

[Dijkstra 1997] DIJKSTRA, E. W.. **A Discipline of Programming**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[Dósea, Sant'Anna and Silva 2018] DÓSEA, M.; SANT'ANNA, C. ; DA SILVA, B. C.. **How do design decisions affect the distribution of software**

metrics? In: PROCEEDINGS OF THE 26TH CONFERENCE ON PROGRAM COMPREHENSION, ICPC '18, p. 74–85, New York, NY, USA, 2018. ACM.

[Eick *et al* 2001] EICK, S. G.; GRAVES, T. L.; KARR, A. F.; MARRON, J. S. ; MOCKUS, A.. **Does code decay? assessing the evidence from change management data**. IEEE Trans. Softw. Eng., 27(1):1–12, Jan. 2001.

[Fowler 1999] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, Boston, 1999.

[Gamma *et al.* 1995] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-oriented Software**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Garcia *et al.* 2009] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Identifying architectural bad smells**. In: CSMR09; KAISERSLAUTERN, GERMANY. IEEE, 2009.

[Garcia *et al.* 2009b] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Toward a catalogue of architectural bad smells**. In: Mirandola, R.; Gorton, I. ; Hofmeister, C., editors, ARCHITECTURES FOR ADAPTIVE SOFTWARE SYSTEMS, p. 146–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Garcia *et al.* 2013] GARCIA, J.; IVKOVIC, I. ; MEDVIDOVIC, N.. **A comparative analysis of software architecture recovery techniques**. In: PROCEEDINGS OF THE 28TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING; PALO ALTO, USA, 2013.

[Godfrey and Lee 2000] GODFREY, M.; LEE, E.. **Secrets from the monster: Extracting Mozilla's software architecture**. In: COSET-00; LIMERICK, IRELAND, p. 15–23, 2000.

[Guimaraes *et al* 2018] GUIMARÃES, E.; VIDAL, S.; GARCIA, A.; DIAZ PACE, J. A. ; MARCOS, C.. **Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support**. Software: Practice and Experience, 48(5):1077–1106, 2018.

[Gurp and Bosch 2002] VAN GURP, J.; BOSCH, J.. **Design erosion: problems and causes**. Journal of Systems and Software, 61(2):105 – 119, 2002.

[Hochstein and Lindvall 2005] HOCHSTEIN, L.; LINDVALL, M.. **Combating architectural degeneration: A survey**. Information and Software Technology, 47:643–656, 2005.

[Kaminski 2007] KAMINSKI, P.. **Reforming software design documentation**. In: 14TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE 2007), p. 277–280, Oct 2007.

[Kazman *et al.* 2015] KAZMAN, R.; CAI, Y.; MO, R.; FENG, Q.; XIAO, L.; HAZIYEV, S.; FEDAK, V. ; SHAPOCHKA, A.. **A case study in locating the architectural roots of technical debt**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 2, ICSE '15, p. 179–188, Piscataway, NJ, USA, 2015. IEEE Press.

[Lanza and Marinescu 2006] LANZA, M.; MARINESCU, R.. **Object-Oriented Metrics in Practice**. Springer, Heidelberg, 2006.

[Li *et al.* 2014] LI, Z.; LIANG, P.; AVGERIOU, P.; GUELFI, N. ; AMPATZOGLOU, A.. **An empirical investigation of modularity metrics for indicating architectural technical debt**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL ACM SIGSOFT CONFERENCE ON QUALITY OF SOFTWARE ARCHITECTURES, QoSA '14, p. 119–128, New York, NY, USA, 2014. ACM.

[Lin *et al.* 2016] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.

[MacCormack, Rusnak and Baldwin 2006] MACCORMACK, A.; RUSNAK, J. ; BALDWIN, C.. **Exploring the structure of complex software designs: An empirical study of open source and proprietary code**. Manage. Sci., 52(7):1015–1030, 2006.

[Macia 2013] MACIA, I.. **On the Detection of Architecturally-Relevant Code Anomalies in Software Systems**. PhD thesis, Pontifical Catholic University of Rio de Janeiro, Informatics Department, 2013.

[Macia *et al.* 2012] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying**

architecture degradation symptoms. In: CSMR12, p. 277–286, March 2012.

[Macia *et al.* 2012a] MACIA, I.; ARCOVERDE, R.; CIRILO, E.; GARCIA, A. ; VON STAA, A.. **Supporting the identification of architecturally-relevant code anomalies**. In: ICSM12, p. 662–665, Sept 2012.

[Macia *et al.* 2012b] MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N. ; VON STAA, A.. **Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems**. In: AOSD '12, p. 167–178, New York, NY, USA, 2012. ACM.

[Marinescu, 2004] MARINESCU. **Detection strategies: metrics-based rules for detecting design flaws**. In: PROCEEDINGS OF 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM); CHICAGO, USA, p. 350–359, 2004.

[Martin and Martin 2006] MARTIN, R. C.; MARTIN, M.. **Agile Principles, Patterns, and Practices in C# (Robert C. Martin)**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[Mattmann *et al* 2006] MATTMANN, C. A.; CRICHTON, D. J.; MEDVIDOVIC, N. ; HUGHES, S.. **A software architecture-based framework for highly distributed and data intensive scientific applications**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '06, p. 721–730, New York, NY, USA, 2006. ACM.

[Moha, Gueheneuc and Leduc 2006] MOHA, N.; G. GUEHENEUC, Y. ; LEDUC, P.. **Automatic generation of detection algorithms for design defects**. In: 21ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'06), p. 297–300, Sept 2006.

[Moha *et al.* 2010] MOHA, N.; GUEHENEUC, Y.; DUCHIEN, L. ; MEUR, A. L.. **Decor: A method for the specification and detection of code and design smells**. IEEE Transaction on Software Engineering, 36:20–36, 2010.

[Moha *et al.* 2010] MOHA, N.; GUÉHÉNEUC, Y.-G.; MEUR, A.-F.; DUCHIEN, L. ; TIBERGHIEN, A.. **From a domain analysis to the specification and detection of code and design smells**. Form. Asp. Comput., 22(3-4):345–361, May 2010.

[Murphy Hill, Parnin and Black 2009] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A. P.. **How we refactor, and how we know it**. In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '09, p. 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[Natthawute, Shinpei and Motoshi 2018] SAE-LIM, N.; HAYASHI, S. ; SAEKI, M.. **An investigative study on how developers filter and prioritize code smells**. IEICE Transactions on Information and Systems, E101.D(7):1733–1742, 2018.

[Oizumi *et al.* 2015] OIZUMI, W.; GARCIA, A.; COLANZI, T.; STAA, A. ; FERREIRA, M.. **On the relationship of code-anomaly agglomerations and architectural problems**. Journal of Software Engineering Research and Development, 3(1):1–22, 2015.

[Oizumi *et al.* 2016] OIZUMI, W.; GARCIA, A.; SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING; USA, 2016.

[Oizumi *et al.* 2016] OIZUMI, W.; GARCIA, A.; D. S. SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: 2016 IEEE/ACM 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 440–451, May 2016.

[Oizumi *et al.* 2017] OIZUMI, W.; SOUSA, L.; GARCIA, A.; OLIVEIRA, R.; OLIVEIRA, A.; AGBACHI, O. I. A. B. ; LUCENA, C.. **Revealing design problems in stinky code: A mixed-method study**. In: PROCEEDINGS OF THE 11TH BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES, AND REUSE, SBCARS '17, p. 5:1–5:10, New York, NY, USA, 2017. ACM.

[Oizumi *et al* 2018] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; GARCIA, A.; AGBACHI, A. B.; OLIVEIRA, R. ; LUCENA, C.. **On the identification of design problems in stinky code: experiences and tool support**. Journal of the Brazilian Computer Society, 24(1):13, Oct 2018.

[Oliveira, Valente and Terra 2016] SILVA, M. C. O.; VALENTE, M. T. ; TERRA, R.. **Does technical debt lead to the rejection of pull requests?** In: PROCEEDINGS OF THE 12TH BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS, SBSI '16, p. 248–254, 2016.

[Paixao *et al.* 2017] PAIXAO, M.; KRINKE, J.; HAN, D.; RAGKHITWETSAGUL, C. ; HARMAN, M.. **Are developers aware of the architectural impact of their changes?** In: 2017 32ND IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 95–105, Oct 2017.

[Palomba *et al.* 2014] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R. ; LUCIA, A. D.. **Do they really smell bad? a study on developers' perception of bad code smells**. In: 2014 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 101–110, Sept 2014.

[Parnas 1972] PARNAS, D. L.. **On the criteria to be used in decomposing systems into modules**. Commun. ACM, 15(12):1053–1058, Dec. 1972.

[Parnas 1978] PARNAS, D. L.. **Designing software for ease of extension and contraction**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '78, p. 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[Ran *et al.* 2015] MO, R.; CAI, Y.; KAZMAN, R. ; XIAO, L.. **Hotspot patterns: The formal definition and automatic detection of architecture smells**. In: SOFTWARE ARCHITECTURE (WICSA), 2015 12TH WORKING IEEE/IFIP CONFERENCE ON, p. 51–60, May 2015.

[Robillard and Murphy 2007] ROBILLARD, M. P.; MURPHY, G. C.. **Representing concerns in source code**. ACM Trans. Softw. Eng. Methodol., 16(1), Feb. 2007.

[Schach *et al.* 2002] SCHACH, S.; JIN, B.; WRIGHT, D.; HELLER, G. ; OFFUTT, A.. **Maintainability of the linux kernel**. Software, IEE Proceedings -, 149(1):18–23, 2002.

[Soares, Laureano and Borba 2002] SOARES, S.; LAUREANO, E. ; BORBA, P.. **Implementing distribution and persistence aspects with aspectj**. In: PROCEEDINGS OF THE 17TH ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA '02, p. 174–190, New York, NY, USA, 2002. ACM.

[Sousa *et al.* 2017] SOUSA, L.; OLIVEIRA, R.; GARCIA, A.; LEE, J.; CONTE, T.; OIZUMI, W.; DE MELLO, R.; LOPES, A.; VALENTIM, N.; OLIVEIRA, E. ; LUCENA, C.. **How do software developers identify design**

problems?: A qualitative analysis. In: PROCEEDINGS OF 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES'17, 2017.

[Sousa *et al.* 2018] SOUSA, L.; OLIVEIRA, A.; OIZUMI, W.; BARBOSA, S.; GARCIA, A.; LEE, J.; KALINOWSKI, M.; DE MELLO, R.; FONSECA, B.; OLIVEIRA, R.; LUCENA, C. ; PAES, R.. **Identifying design problems in the source code: A grounded theory**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 921–931, New York, NY, USA, 2018. ACM.

[Strauss and Corbin 1998] STRAUSS, A.; CORBIN, J.. **Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory**. SAGE Publications, 1998.

[Tang *et al.* 2016] TANG, A.; ALETI, A.; BURGE, J. ; VAN VLIET, H.. **What makes software design effective?** Design Studies, 31(6):614 – 640, 2010. Special Issue Studying Professional Software Design.

[Trifu and Marinescu 2005] TRIFU, A.; MARINESCU, R.. **Diagnosing design problems in object oriented systems**. In: WCRE'05, p. 10 pp., Nov 2005.

[Trifu and Reupke 2007] TRIFU, A.; REUPKE, U.. **Towards automated restructuring of object oriented systems**. In: CSMR '07, p. 39–48, Washington, DC, USA, 2007. IEEE.

[Tsantalis *et al* 2013] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A multidimensional empirical study on refactoring activity**. In: PROCEEDINGS OF THE 2013 CONFERENCE OF THE CENTER FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH, p. 132–146. IBM Corp., 2013.

[Vale, Fernandes and Figueiredo 2018] VALE, G.; FERNANDES, E. ; FIGUEIREDO, E.. **On the proposal and evaluation of a benchmark-based threshold derivation method**. Software Quality Journal, p. 1–32, 2018.

[Vidal *et al.* 2016] VIDAL, S.; GUIMARAES, E.; OIZUMI, W.; GARCIA, A.; PACE, A. D. ; MARCOS, C.. **Identifying architectural problems through prioritization of code smells**. In: SBCARS16, p. 41–50, Sept 2016.

[Yamashita and Moonen 2012] YAMASHITA, A.; MOONEN, L.. **Do code smells reflect important maintainability aspects?** In: ICSM12, p. 306–315, 2012.

[Yamashita and Moonen 2013] YAMASHITA, A.; MOONEN, L.. **Do developers care about code smells? an exploratory survey**. In: 2013 20TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 242–251, Oct 2013.

[Yamashita and Moonen 2013] YAMASHITA, A.; MOONEN, L.. **Exploring the impact of inter-smell relations on software maintainability: an empirical study**. In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING; SAN FRANCISCO, USA, p. 682–691, 2013.

[Yamashita *et al.* 2015] YAMASHITA, A.; ZANONI, M.; FONTANA, F. A. ; WALTER, B.. **Inter-smell relations in industrial and open source systems: A replication and comparative analysis**. In: SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2015 IEEE INTERNATIONAL CONFERENCE ON, p. 121–130, Sept 2015.

[Young 2015] YOUNG, T. J.. **Using aspectj to build a software product line for mobile devices**. Master's thesis, University of British Columbia, 2015.

[van der Ven *et al.* 2006] VAN DER VEN, J. S.; JANSEN, A. G. J.; NIJHUIS, J. A. G. ; BOSCH, J.. **Design Decisions: The Bridge between Rationale and Architecture**, p. 329–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.