PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Leonardo da Silva Sousa

Understanding How Developers Identify Design Problems in Practice

Tese de Doutorado

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor : Prof. Alessandro Fabricio Garcia
Co-advisor: Prof. Carlos José Pereira de Lucena

Rio de Janeiro
August 2018

Leonardo da Silva Sousa

Understanding How Developers Identify Design Problems in Practice

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the undersigned Examination Committee.

Rio de Janeiro, August 30th, 2018

Leonardo da Silva Sousa

The author received his Bachelor degree in Computer Science from the Instituto de Informática (INF) of Universidade Federal de Mato Grosso (UFMT) in 2011. He also received his Master degree in Computer Science from Universidade Federal de Goiás (UFG) in 2014. During his academic career, he participated in several research projects. His main research interests are: Code Smells, Design Problems, Refactoring, Software Architecture, Software Testing and Empirical Software Engineering.

Bibliographic data

Sousa, Leonardo da Silva

Understanding how developers identify design problems in practice / Leonardo da Silva Sousa ; advisor: Alessandro Fabricio Garcia ; co-advisor: Carlos José Pereira de Lucena. – 2018.
210 f. : il. color. ; 30 cm

Tese (Doutorado em Informática)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2018.
Inclui bibliografia

1. Informática – Teses. 2. Problemas de projeto. 3. Anomalias de código. 4. Refatoração. 5. Sintomas. 6. Teoria. I. Garcia, Alessandro Fabricio. II. Lucena, Carlos José Pereira de. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

To my parents, for their support
and encouragement.

Acknowledgments

I would like to start by expressing my gratitude to my family, who have supported me since the beginning of this journey. My father, Francisco Costa de Sousa, who tough me to always be the best version of myself. My mom, Elizabete da Silva Sousa, who is forever in my corner. My sister, Francisca Elaine Kroth, the person that I love the most in this word and I cannot live without. My brother-in-law, Marcio Kroth, someone with whom I learn to admire. Last but not least, my nieces, Sabrina and Valentina, my two newest favorite people in the word.

My sincerest gratitude to my advisor Alessandro Garcia. I have no words to describe my admiration and gratitude. Without his guidance, I would have accomplished nothing. I am thankful for his patience and for believing in me. I know how much trouble I gave him, but he never lost his faith in me. For that, I am eternally indebted. Thank you so much. A very special thanks to my co-advisor, Carlos Lucena. His energy and wisdom are incredible. I hope others have the same fortune to work with you. I would also like to thank Jaejoon Lee. His was a most essential contribution: always asking the most pertinent questions to get me thinking in different ways and pushing me to my highest standard.

My deepest gratitude to Diego Cedrim. His friendship is easily one of the best things to come from my Ph.D. experience. Indeed, I'll carry this friendship with me the rest of my life. There were so many points where his words – often times harsh ones – made the difference between giving up or carrying on. He truly is my brother now. A special thanks goes out to his fiancé, Juliana Leal, for both accepting me as a friend and making my friend so happy. Roberto Oliveira, or Mr. Robert to those closest to him, is another amazing friend that I'm so happy to have made. He is definitely, the kindest person I've ever met. His lovely wife Givanilde Oliveira and newborn Sophia Oliveira must also be showered in thanks for making Robert so happy. Last but not least, I would like to thank Alexandre Navarro, Anderson Oliveira, Isabella Vieira and Willian Oizumi, who I admire so much, personally and professionally, as well as my reliable, if sometimes reluctant proofreader, Leonard Wilkes.

I would like to thank all of my colleagues from the OPUS Research Group: Ana Carla Bibiano, Anderson Uchôa, Alexander Lopez, Anne Benedicte, Bruno Cafeo, Daniel Tenorio, Eduardo Fernandes, João Neves, Luiz Carvalho, Marcelo Garnier, and Rafael de Mello. I would also like to thank Tayana Conte, who I have had the distinct pleasure to work with. Indeed, I want to thank her, along with Adriana Lopes, Edson Oliveira and Natasha Valentim, who (patiently) taught me Grounded Theory. I also extend my gratitude to my colleagues from the USES group. Last but not least, I would like to thank Nenad Medvidovick: what an honor to have the opportunity to work with you. Many thanks to his Softarch group members as well: Arman Shahbazian, Daniel Link, Duc Le, Jae Young, YounKyu Lee, and my dearest friend Yixue Zhao.

I thank all the professors from PUC-Rio for their invaluable contribution to my education. In particular, I would like to thank Simone Barbosa, for the incredible brownies... I mean for her[1] amazing insights and inspiring words during our work together. Thank you and I'm sorry for the extra trouble. I also want to thank the members of my thesis defense team: Arndt von Staa, Claudia Werner, Leonardo Murta, Marcos Kalinowski, Simone Barbosa, and Tayana Uchôa Conte.

I am so grateful to Capes, CNPq, FAPERJ and PUC-Rio for the financial support that made my research possible in the first place. Finally, my sincere thanks to the administrative staff of the DI at PUC-Rio.

[1]Pronouns are your friends!

Abstract

Sousa, Leonardo da Silva; Garcia, Alessandro Fabricio (Advisor); Lucena, Carlos José Pereira (Co-Advisor). **Understanding How Developers Identify Design Problems in Practice**. Rio de Janeiro, 2018. 210p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A design problem is the manifestation of one or more inappropriate design decisions that negatively impact non-functional requirements. For example, the Fat Interface, a problem that indicates when an interface exposes non-cohesive services, hampers the extensibility and maintainability of a software system. Despite its harmfulness, identifying a design problem in a system is difficult, especially when the source code is the only available artifact. Although researchers have been investigating techniques to help developers in identifying design problems, there is little or no knowledge about the process of identifying design problems. For instance, code smells, microstructures that are a surface indication of design problems, have been used in several techniques to support developers during the design problem identification. However, there is no knowledge if code smells suffice to help developers to identify design problems. In particular, no study has tried to understand how developers identify design problems in practice. Thus, in this thesis, we have conducted a series of studies to understand design problem identification. In our two first studies, we investigated the role that code smells play in supporting developers during the design problem identification. Our results indicate that code smells are relevant for developers in practice; for instance, they are relevant to indicate elements that need to be refactored. However, we found that code smells, despite their relevance, do not suffice in helping developers to identify design problems. In this vein, we conducted another study to investigate what indicators developers use in practice, and how they use them. This study resulted in a theory about how developers identify design problems in practice. For instance, the theory reveals the indicators that developers use, how they use these indicators, and the characteristics of such indicators that are perceived as helpful by developers. The results found by our studies provided us with a better understanding of the process of identifying design problems thitherto nonexistent. Moreover, our findings pave the way for the elaboration of more effective techniques to identify design problems in the source code.

Keywords

Design Problems; Code Smells; Refactoring; Symptoms; Theory.

Resumo

Sousa, Leonardo da Silva; Garcia, Alessandro Fabricio; Lucena, Carlos José Pereira. **Entendendo Como os Desenvolvedores Identificam Problemas de Projeto na Prática**. Rio de Janeiro, 2018. 210p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Um problema de projeto é a manifestação de uma ou mais decisões de projeto inadequadas que afetam negativamente requisitos não funcionais. Por exemplo, Fat Interface, um problema que indica quando uma interface expõe serviços não coesos, no qual dificulta a extensibilidade e a manutenibilidade de um sistema de software. Apesar de problemas de projeto serem prejudiciais aos sistemas, identificá-los é uma tarefa difícil, especialmente quando o código-fonte é o único artefato disponível. Embora pesquisadores venham investigando técnicas para ajudar os desenvolvedores a identificar problemas de projeto, há pouco conhecimento sobre o processo de identificar problemas de projeto. Por exemplo, anomalias de códigos, um indicador de problemas de projeto, têm sido usadas para ajudar desenvolvedores a identificar problemas de projeto. No entanto, ainda não sabemos se elas são suficientes para ajudá-los ou não. Em particular, nenhum estudo tentou entender como os desenvolvedores identificam problemas de projeto. Nesse contexto, nós realizamos alguns estudos para entender a identificação de problemas de projeto. Em nossos dois primeiros estudos, nós investigamos o papel que as anomalias de código desempenham durante a identificação de problemas de design. Nossos resultados indicam que as anomalias de código são relevantes para os desenvolvedores na prática, por exemplo, eles são relevantes para indicar elementos a serem refatorados. Apesar da relavância, descobrimos que as anomalias de código não são suficientes para ajudar os desenvolvedores a identificar problemas de projeto. Nesse sentido, conduzimos outro estudo para investigar quais outros indicadores os desenvolvedores usam na prática e como eles são usados. Este estudo resultou em uma teoria sobre como os desenvolvedores identificam problemas de projeto na prática. A teoria revela quais são os indicadores que os desenvolvedores usam, como eles usam esses indicadores e as características de tais indicadores que os desenvolvedores consideram úteis. Os resultados encontrados nos forneceram uma melhor compreensão do processo de identificação de problemas de projeto, abrindo caminho para a elaboração de técnicas mais eficazes em ajudar os desenvolvedores a identificar problemas de projeto.

Palavras-chave

Problemas de Projeto; Anomalias de Código; Refatoração; Sintomas; Teoria.

Table of contents

List of figures

List of tables

List of Abbreviations

AMI	Ambiguous Interface
C	Construct
CCD	Cyclic Dependency
CCO	Concern Overload
CPO	Component Overload
DLA	Delegating Abstraction
DP	Design Problem
FP	False Positive
FTI	Fat Interface
GT	Grounded Theory
ICA	Incomplete Abstraction
IDE	Integrated Development Environment
LOC	Lines of Code
MPC	Misplaced Concern
NA-smells	Non-agglomerated code smells
P	Proposition
RQ	Research Question
SE	Software Engineering
SoC	Separation of concern
SRQ	Specific Research Question
STC	Scattered Concern
TP	True Positive
UA	Unused Abstraction
UWD	Unwanted Dependency

Never forget what you are, for surely the world will not. Make it your strength. Then it can never be your weakness. Armour yourself in it, and it will never be used to hurt you.

George R.R. Martin, *A Song of Ice and Fire.*

1 Introduction

Software design results from a series of decisions made during the software development (1, 2). A previous study indicated that 25% of discussions in commits, issues and pull requests are about design (3). That happens because software design is a fundamental matter during the software development process (4, 5). Thus, several design decisions are made during this process, all of which will drive how the system will be developed. Consequently, design decisions are expected to influence the software design positively. Unfortunately, that is not always the case. Some design decisions may not contribute to properly address non-functional requirements such as maintainability, understandability, testability, and robustness.

Each non-functional requirement may be impacted, either positively or negatively, by design decisions (6). When design decisions impact non-functional requirements negatively, we state that a design problem exists. In practice, design decisions include (but are not limited to) how the system is organized into subsystems and components, how and which code elements encapsulate process and data to address each functionality, and how they interact with each other and their execution environment (7, 8, 9). Therefore, a design problem usually affects relevant code elements for the system design, such as interfaces, components, hierarchies and other elements that encapsulate process and data in the system design (10).

In summary, a design problem is the manifestation of one or more inappropriate design decisions that affect code elements relevant to the system design. As a consequence, these design decisions negatively impact non-functional requirements when affecting these elements. Although not always made intentionally, an inappropriate design decision can be an improper application of a design solution, use of a design abstraction at the wrong context, violation of a modularity principle, a misprioritization of an objected-oriented principle over another, misusing a certain design pattern (4), and any other decision that has an undesirable impact on non-functional requirements.

An example of design problem is *Cyclic Dependency* (11). This design problem happens when two or more elements depend on each other directly or indirectly. When there are cyclic dependencies among elements, the system might end up at a stage where these dependencies compromise the maintainability, understandability, reusability and testability of software systems (11). Additionally, *Cyclic Dependency* may eventually cause deadlock (12), which negatively impacts the system performance and availability. Other well-known examples of design problems include *Scattered Concern* (7), *Ambiguous Interface* (7) and *Fat Interface* (13).

Design problems are the results of inappropriate design decisions with regard to how the system is organized and how the elements encapsulate process and data. Thus, due to their adverse effect on non-functional requirements, design problems are often harmful in several software systems (14, 15, 16, 17). An industrial study with 745 software systems, from 160 different organizations, showed that technical debts – primarily associated with design problems – were directly related with a significant increase in software project costs (18). Another study showed that design problems are one of the most common categories of technical debt that lead to the rejection of code contributions (19). Given the harmfulness of certain design problems, developers should identify and remove them as early as possible (7, 17, 20).

Design problems can be removed through refactoring operations. A refactoring operation is a program transformation used for improving the design structure of a system (21). Thus, there is a close relation between a design problem and refactoring: while the former has a negative effect on non-functional requirements, the latter can have a positive effect on them (22, 23, 24). Indeed, refactoring is commonly applied to repair a program with deteriorated design (25). A design problem is one of the reasons why the source code can reach a deteriorated state (15, 16); thus, developers can repair the deteriorated code using refactoring to remove design problems (25). Therefore, once design problems are identified, developers can use refactoring to remove them.

1.1 Design Problem Identification

We call design problem identification the task of finding a design problem in a software system. Unfortunately, design problem identification is not trivial (26, 27). To begin with, systems tend to be large in size and complexity, increasing the search space for design problems. Second, each design problem may pervade the implementation of several code elements (7, 28). Hence, developers may need to analyze several elements to identify a single design problem (27). Third, design documentation is often unavailable or outdated. Thus, the source code is frequently the only artifact available for supporting developers on the identification of design problems.

For the identification of design problems, developers need to locate indicators of the presence of a design problem directly on the source code. In this context, we define a design problem **symptom** as a partial sign or indication of the presence of a design problem that developers use in practice. In other words, a symptom is an indicator of the presence of a design problem which we observed that developers use in practice. Code smell is an example of a symptom usually associated with design problem identification (21) and that is part of the developers' routine (29). A code smell is a microstructure in the system that represents a surface, sometimes only partial, indication of a design problem (21). `Feature Envy` is an example of a code smell, which refers to a method that is more interested in other classes than in the one to which it belongs (21). Other examples include `Long Method`, `Long Parameter List`, `God Class`, `Shotgun Surgery`, and `Divergent Change` (21, 30).

Some design problems can be identified by a single code smell. For instance, let us consider the design problem *Incomplete Abstraction*, which happens when a code element, usually a class, does not support a responsibility completely in its enclosing component (31, 32, 33). This design problem can be identified by the `Lazy Class` smell, which is a class that does not do enough (21). This smell may happen when a class that used to implement a functionality gets downsized or when a class is added in anticipation of a future need that never eventuates, which lead to *Incomplete Abstraction*. Other design problems that can be identified by single code smells are *Concern Overload* (52) and *Unused Abstraction* (34, 9). They can be identified by `God Class` and `Speculative Generality` smells (21), respectively.

On the other hand, some design problems may not be identified by only a single smell. In these cases, design problem identification may itself quickly turn into a very complex task (26, 27). Thus, developers may need to rely on multiple smells instead of just one. For instance, *Scattered Concern*, which happens when a functionality is implemented by several elements instead of few ones (7, 8). To identify this design problem, a developer would have to find the elements that are implementing the scattered functionality. Thus, he would have to identify at least one smell in each code element to realize that it is implementing the scattered functionality.

Although some design problems appear to be suitable for being identified by multiple smells, in practice this may not be true. Whether the developer will use only one smell or several ones is more likely to depend on the instance of the problem than on its type itself. Unfortunately, as this knowledge about single smell and multiple smells is still unknown, we can only speculate when

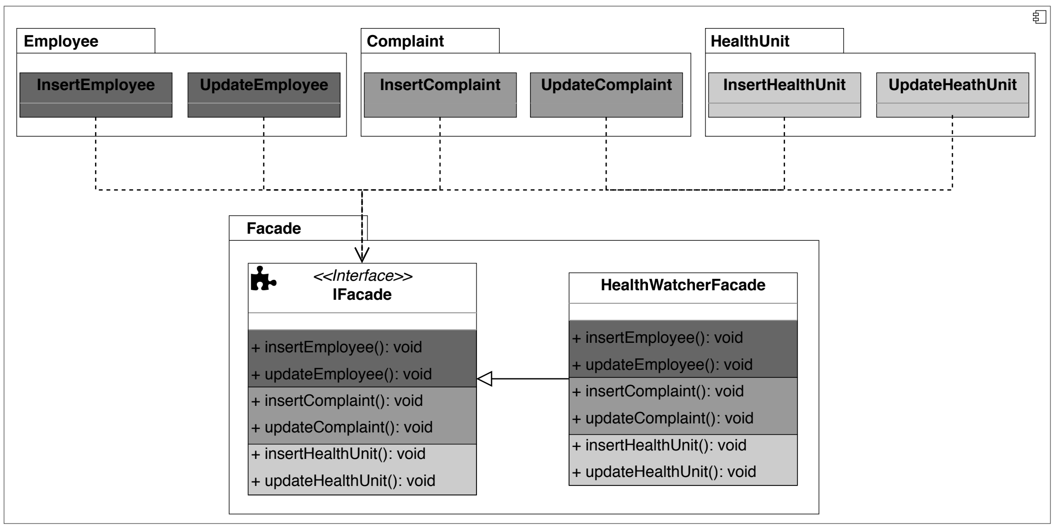


Figure 1.1: Fat Interface Example

developers use only one or various smells to identify a design problem. For example, let us consider *Fat Interface*, a design problem that occurs when a developer decides to aggregate multiple non-cohesive services in a single system interface (13). A single smell in the interface could be enough to spot the design problem, such as the `Shotgun Surgery` smell (21). This smell indicates that a change in the interface triggers changes in other code elements. The reason for triggering various changes is that the interface aggregates multiple non-cohesive services. Even though a single smell seems to be enough to spot this design problem, developers may need to rely on multiple smells to identify some instances of *Fat Interface*, as illustrated by the following example based on the Health Watcher system (35, 36).

Health Watcher is a web-based system for improving the quality of services that health vigilance institutions provide. The system allows users to register several kinds of health vigilance complaints, such as complaints against restaurants and food shops. Thus, health vigilance institutions can investigate and resolve complaint reports appropriately. Figure 1.1 relies on a UML-like notation (37) to show a partial view of some Health Watcher components. In this example, the IFacade has an instance of *Fat Interface*, which is represented in the figure by the puzzle symbol. This interface declares methods to access three non-cohesive services. Each service is represented in the figure by a shade of gray.

In this particular example, the IFacade does not contain any code smell; consequently, a developer would have to analyze smells in elements related to the interface in order to identify the design problem. For instance, InsertEmployee, UpdateEmployee, InsertComplaint, UpdateComplaint, InsertHealthUnit, and UpdateHealthUnit are clients which contain code smells such as `Divergent Change` and `Dispersed Coupling`. `Divergent Change` appears in these classes since they are affected whenever a change is made in one of the three services. Furthermore, the classes are all connected through the interface, which leads to the `Dispersed Coupling`. The interface clients are not the only affected elements. Classes that implement IFacade are also affected. As the interface declares more than one responsibility (*i.e.*, one service), it forces other classes to implement more than one responsability as well. Consequently, classes such as HealthWatcherFacade likely contain `God Class` and `Feature Envy` smells. `God Class` emerges because the class implements more than one responsibility. Furthermore, some of the methods may contain `Feature Envy` since they are more interested in other classes that are related to its service.

In summary, a developer would have to reason about these multiple aforementioned smells to realize that the interface has an instance of *Fat Interface*. On the one hand, the analysis of multiple smells can help developers to identify some design problems, as illustrated by the example. On the other hand, the analysis of multiple smells may increase the complexity related to the identification of design problems. As developers may need to analyze multiple smells, they can be overloaded with information. To make matters worse, some instances of code smells are not related to any design problem (38), which can make the analysis of multiple smells even harder since they have to find out which instances are related to a design problem. Unfortunately, there is no knowledge whether in practice developers benefit from the analysis of multiple code smells.

This example also serves to illustrate how developers can introduce a design problem due to an inappropriate design decision. In this case, the developer decided to implement the Facade pattern. This pattern intends to provide a unified interface to a set of interfaces in a subsystem (4). Thus, all clients that need to use these services could access them through the interface. However, the developer inappropriately decided to implement only a single interface (IFacade) to provide all services. As a result, this interface became highly coupled to all the other modules, which negatively impacted the maintainability and extensibility of Health Watcher. In this example, there was a case of a design pattern misuse. The Facade pattern was intended to make the subsystem easier to use, but its implementation is complicating the logic of its clients, since each client class has to decide in which services it is interested. According to the developers of Health Watcher, the most appropriate use of this pattern would be the creation of an unified interface for each service, as illustrated by Figure 1.2. Thus, each client would access only the interface that

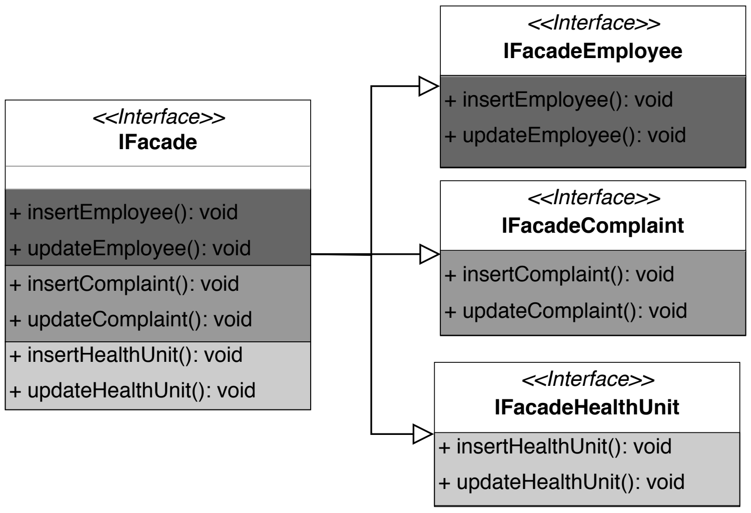

<<Interface>>
IFacade
+ insertEmployee(): void
+ updateEmployee(): void
+ insertComplaint(): void
+ updateComplaint(): void
+ insertHealthUnit(): void
+ updateHealthUnit(): void
<<Interface>>
IFacadeEmployee
+ insertEmployee(): void
+ updateEmployee(): void
<<Interface>>
IFacadeComplaint
+ insertComplaint(): void
+ updateComplaint(): void
<<Interface>>
IFacadeHealthUnit
+ insertHealthUnit(): void
+ updateHealthUnit(): void

Figure 1.2: Fat Interface Solved

provides the service to which it is interested.

1.2 Problem Statement

This section presents our specific and general research problems. Moreover, we discuss why related work does not address such research problems. In the previous section, we discussed how one or more code smells can be used to identify a design problem. Additionally, we presented an example that illustrates how developers may need to analyze multiple smells to identify a design problem. In the example, the code smells appeared in the code elements due to the relation between the elements and the interface that has a design problem. Hence, those code smells could have been used to indicate the design problem. Indeed, code smells have been used as indicators of design problems, especially because they can be identified directly in source code, which is often the only tangible artifact available for developers.

Several studies have presented techniques based on code smells to help developers to identify design problems (28, 38, 39, 41, 47, 46). Code smells have been explored by these studies due to their co-occurence with design problems (47, 53, 54). Some studies found that elements affected by design problems tend to have code smells, *i.e.*, code smells and design problems co-occur in these elements (38, 41, 46, 47). Thus, they used such co-occurrence to state that code smells are consistent indicators of design problems. Unfortunately, previous studies did not investigate if code smells are key symptoms for developers to identify design problems in practice. A key symptom is one that helps developers to identify a design problem that is relevant in the software system. They also did not investigate if developers are able of identifying design problems in practice through the use of code smells.

In fact, we have little or no knowledge about how developers identify design problems in practice. We do not know if developers will actually consider one or multiple code smells to identify design problems, let alone if they can identify design problems using code smells. Even though some studies have showed how code smells are consistent indicators of design problems (38, 41, 46, 47), we still have to investigate if: (i) code smells are key symptoms to identify design problems in practice, and (ii) developers are able to use them. These investigations aim to understand whether code smells suffice in helping developers to identify design problems.

Let us suppose that code smells do not suffice in helping developers to identify design problems, either because code smells are not key symptoms or because developers cannot use them. Either way, developers may not be willing to use them despite their potential benefit for identifying design problems as reported by previous studies (38, 41, 46, 47). In other words, code smells can, in fact, be consistent indicators of design problems, but in the end, the consistency has little or no importance if in practice they are not key symptoms and if developers cannot use them. Therefore, we still need to find out if code smells suffice in helping developers. For this purpose, we need to investigate first whether code smells are key symptoms for developers to identify design problems (Research Problem 1).

Research Problem 1. Whether code smells are key design problem symptoms for developers in practice is unknown.

A way to investigate this phenomenon is studying whether and how code smells indicate design problems that are indeed relevant for the developers. In the context of our research, a relevant design problem is one that developers focus their effort to remove from the source code. As some design problems can be so damaging to the point of discontinuing software projects (14, 15, 16, 17), developers focus their effort to identify and remove them through refactoring. Refactoring is a transformation used for repairing a deteriorated code. Thus, to address Research Problem 1, we need to find the code elements that were refactored, *i.e.*, elements that were likely to have at least one design problem, which led developers to refactor them. Then, we investigate if these code elements contain at least one code smell closely related to the design problem. If they do, then smells have the potential to serve as key symptoms for helping developers to identify relevant design problems in practice.

Let us assume that code smells have strong potential for assisting developers on identifying design problems. Then, a follow-up investigation is confirming if developers are able to explicitly use smells to identify design problems (Research Problem 2). This follow-up investigation is important because, while code smells can be key symptoms of relevant design problems, developers still can struggle to use them to identify a design problem in practice.

Research Problem 2. Whether developers are able to reason about code smells to actually identify design problems is unknown.

Code smells have been a well-researched topic over the last decade (53, 28, 47, 51, 27, 55, 29). However, there is still little knowledge about the role that code smells play on how developers identify design problems in practice. We do not know if code smells are key symptoms for developers (Research Problem 1), and even if they are, we do not know if developers are able to actually use them to identify design problems (Research Problem 2).

However, addressing these research problems is not sufficient. For instance, we can find in these investigations that either code smells are not key design problem symptoms or developers cannot use them to identify design problems. Thus, since our goal is to understand how developers identify design problems in practice, we still need to investigate if they use other symptoms in addition to code smells. Unfortunately, we do not know if developers use other symptoms in practice (Research Problem 3). Then, we should understand the relative importance of various types of symptoms.

Research Problem 3. What design problem symptoms that developers use in practice are unknown.

By studying these three specific research problems, we expect to understand how developers identify design problems in practice. In summary, the investigation of these specific problems discussed heretofore will contribute to better understand how developers identify design problems in practice, in particular when the source code is the only artifact available in a project (General Problem).

General Problem. How developers identify design problems in practice is unknown.

Lack of sufficient knowledge of how developers identify design problems in practice may be preventing us, researchers, from providing the required support to assist developers during design problem identification. We are not able to conceive efficient assistant techniques if we do not know: (i) what other design problem symptoms that they use, (ii) what steps they follow to identify a design problem, and (iii) what criteria they use to confirm the presence of one. In fact, we do not even know how to assess existing solutions to help developers on identifying design problems, since we do not know how the identification task takes place in practice.

1.3 Goal and Research Questions

Design problem identification is not trivial (Section 1.1), and we do not have much information about it (Section 1.2). The complexity of this task and the lack of knowledge lead to various unaddressed research problems, as discussed in the previous section. Without addressing these research problems, we will continue unfamiliar with how design problem identification happens in practice. Consequently, we may be doomed to not providing the necessary support for developers. In fact, these research problems are essential to provide the knowledge required to anyone understand (i) how design problem identification happens in practice, (ii) the role that code smells play in the identification task, (iii) what symptoms of design problems developers use, and, finally, (iv) how to provide support that is aligned with how developers identify design problems in practice. Given this context, the goal of this thesis is stated as follows:

Goal. Understand how developers identify design problems in practice.

To achieve this goal, we mapped the aforementioned research problems into one or more research questions. For instance, Research Problems 1 and 2 were mapped onto two research questions, which address the role that code smells play in helping developers to identify design problems. As previously discussed, code smells have been the center of different techniques to support the identification of design problems (41, 38, 39, 28, 47, 46). However, if smells are not key symptoms or developers cannot use them, then techniques that rely on code smells may fall short of expectations, or worse, they may not be adopted in practice (41, 38, 39, 28, 47, 46). Therefore, to address the first research problem, our first research question is stated as follows:

RQ1. Are code smells key symptoms to indicate relevant design problems for developers?

To answer RQ1, we need to investigate if, in practice, code smells indicate design problems that are somehow relevant for developers. Another alternative is to ask developers if they consider code smells key symptoms to indicate design problems, similar to what Yamashita and Moonen did (29). In that study, the authors asked 73 developers to answer a survey about code smells. The majority of respondents (50 developers – 68%) indicated that they apply code smell concepts in their daily activities or, at least, they know about them.

Unfortunately, the results of this survey are not enough to answer RQ1. First, the authors were more interested in knowing if developers care about code smells rather in knowing if code smells are key symptoms to indicate design problems. Second, RQ1 requires observing in practice if code smells are key symptoms to indicate design problems. In other words, we cannot simply rely on a survey without actually observing the practice. Responses in a survey often do not represent what the developers actually do in their projects (57). In fact, this study reported that what people say they do in response to surveys bears no relationship to what they do in their work practices (57). Therefore, just asking developers if they apply smell concepts is not enough. Thus, we need to investigate the phenomenon in terms of actions indeed realized by developers in their projects.

We can answer RQ1 by investigating if code smells are key symptoms to indicate relevant design problems. As aforementioned, a relevant design problem is one that developers focused their effort to identify and remove from the source code. We can find these relevant design problems by searching for elements that developers refactored due to their deterioration. Since, the presence of design problems is one of the reasons why elements can reach a deteriorated state (15, 16), we can investigate elements that developers focused on repairing them through refactoring. If these refactored elements contain code smells, then we can assume that code smells are key symptoms to indicate relevant design problems. Nevertheless, we highlight that we can only make this assumption if we have a sign indicating that these elements had design problems.

The answer for RQ1 can provide us a better understanding of to what extent code smells are key design problem symptoms. For instance, we expect to find out if relevant design problems can be spotted by multiple code smells or just one is enough to locate them. We can find out, on the other hand, that code smells are not key symptoms for developers identify design problems after all. Even if code smells are key symptoms for developers identify relevant design problems, in practice, they may not be able of using smells (Research Problem 2). Unfortunately, our knowledge about how developers identify design problems with code smells is limited. Therefore, we need to find an answer to the following research question.

RQ2. Are developers able to use code smells to identify design problems?

To answer RQ2, we should conduct a study in which we provide code smells for developers, and we investigate if they can use them to identify design problems. As some studies have proposed smell-based techniques to help developers to identify design problems (27, 28, 29, 47, 51, 53, 55), we cannot neglect these techniques if we also want to find out if code smells suffice in helping developers to identify design problems. Therefore, in the context of RQ2, we also want to investigate if developers using one of these techniques can find better precision in identifying design problems. Code smell agglomeration, proposed by Oizumi *et al.* (47), is an example of one of these techniques.

An agglomeration is a group of interrelated code smells. In their studies, Oizumi *et al.* found that most code smells associated with a design problem were part of one or more agglomerations (82). Due to their results, there is a high chance that developers can identify design problems using code smell agglomerations. Consequently, it seems reasonable to choose agglomeration as a technique to investigate if developers can obtain better precision in identifying design problems. Furthermore, agglomeration is a technique that exclusively relies on code smells. Different from other smell-based techniques (27, 28, 39, 58), agglomeration only groups code smells, *i.e.*, there is no other symptom used together with smells.

We can find out in the investigation of RQ2 that either developers are not able to use smells in practice, or smells do not suffice to help them identify design problems. Either way, these results can be an issue since we do not know what other symptoms developers use in practice (Research Problem 3). Since our goal is to understand how developers identify design problems in practice, we cannot focus only on code smells. We also have to investigate if developers use other design problem symptoms. Hence, our third research question is stated as follows:

RQ3. What are the design problem symptoms that developers use in practice?

In order to answer RQ3, we should conduct a study in which developers have to identify design problems in their source code. Our goal is to observe them working in practice, allowing us to find other symptoms that they may use. Indeed, there is a high chance that developers use other symptoms, which explains why other studies focused on proposing techniques that rely on other indicators (44, 40, 59, 42, 45). Unfortunately, this is knowledge that we do not have.

Our lack of knowledge does not restrict to not knowing only which symptoms developers use. In fact, we do not know how they use these symptoms. We do not know what steps developers follow to identify a design problem. We do not even know what criteria developers consider before confirming or refuting the presence of a design problem. This limited knowledge has prevented researchers and tool engineers from providing most appropriate support to developers. In fact, our limited knowledge does not even allow us to make sure that we have been appropriately assessing techniques to support the design problem identification. As far as we know, no other study has observed developers during the identification of design problems. Even with several studies that have investigated how to help developers to identify design problems, we still do not know how developers identify design problems in practice. Therefore, our fourth research question is stated as follows:

RQ4. How do developers identify design problems in practice?

In order to answer RQ4, we should conduct a multi-trial industrial experiment with developers from different software companies. In this study, developers will have to identify design problems in their systems under development. We plan to capture data on their behavior by filming the environment, recording audio and capturing their computer screens on video. These data will allow us to conduct an in-depth qualitative analysis.

We are aware that design problem identification is a cumbersome task, which requires much effort from researchers to understand it and to propose techniques to support it. One could have expected that we skip research questions 1 and 2, and focus on questions 3 and 4. However, we cannot skip these research questions because they address research problems necessary to understand how developers identify design problems in practice. We cannot neglect investigating code smells since they have been discussed in the literature as a consistent indicator of design problems (53, 28, 47, 51, 27, 55, 29). They are also part of the developers' routine (29). Conversely, we could not only focus on code smells and pretend that developers do not use other symptoms, which makes us to look at the practice all the time.

Without investigating how developers identify design problems in practice, we may not understand the identification task. Consequently, we may not be able to provide support that is aligned with the practice. We expect that answers to the previous research questions can bring knowledge to support the identification of design problems. The two first research questions can help us to understand the state-of-art as code smells have been used in several techniques. The other two research questions can gather knowledge about the identification task in practice. We expect to achieve our research goal by answering these questions, paving the way for more effective identification techniques.

1.4 Main Contributions

This thesis presents studies aimed at understanding how developers identify design problems in practice. For each study, we defined research questions, in which we found the following results:

– In RQ1, we investigated if code smells are key symptoms to indicate relevant design problems in practice. For that, we analyzed 50 software systems in order to observe how developers refactored their source code. We found that developers tend to apply refactoring operations when the code elements contain several smells. After ensuring that these refactored elements were likely to contain design problems, we found that in most cases code smells are key symptoms to indicate relevant design problems. However, we found a few cases in which refactored elements may contain design problems even though code smells cannot indicate design problems on them. As a matter of fact, our investigation was limited in two senses. First, we only investigated the refactored code. Thus, we do not know if code smells are key symptoms to indicate relevant design problems if the element has not been refactored. Second, we did not actually investigate developers' behavior; instead, we conducted a retrospective study. Even though our results indicate that in most cases code smells likely represent key symptoms, we still have to investigate if code smells suffice to help developers in identifying design problems in practice. The need for this investigation is even more evident when we take into account related studies that focused on investigating if code smells are perceived by developers as critical for the system (51, 29), which is not always the case.

– In RQ2, we investigated if developers are able to use code smells to identify design problems. We also investigated if developers using agglomerations (a smell-based technique to support developers) obtain better precision in identifying design problems. The comparison of code smells and agglomerations helped us to find if code smells suffice in helping developers during the design problem identification. The investigation of RQ2 revealed that there was no statistically significant improvement in precision when developers used agglomerations. Only 36.36% of them found more design problems using agglomerations rather than using smells without being agglomerated. Unfortunately, we noticed that code smells are not enough to support developers in identifying design problems. This result confirmed that developers need better support to analyze elements than code smells can provide, which was something that developers mentioned in the study. Interestingly enough, the previous study, which was used to answer RQ1, already indicated the role of code smells for developers. In that study, we found that developers focus on refactoring code elements with several code smells. To a certain degree, we could claim that developers already use smells to find elements to refactor. Nevertheless, combining the results of RQ1 and RQ2, we found that in practice developers can benefit from the analysis of multiple code smells. We also found that code smells do not suffice in helping developers to identify design problems. These results motivated us even more to find the other symptoms that developers use in practice.

– In RQ3, we observed developers identifying design problems in practice in order to find out what symptoms they use in addition to code smells. The need for this study became even clearer after the study used to answer RQ2, in which we noticed the need to find other symptoms to support developers during the identification task. An appropriate way to find these symptoms was by observing what they already use in practice. The investigation of RQ3 revealed that, in practice, developers search for multiple symptoms of a design problem in source code before identifying it. In fact, they use and combine multiple ones to identify a single design problem. What is interesting about this result is its implication for the state-of-the-art. We now know that some techniques to support developers have been mistakenly assuming that developers only use a predefined, dominant indicator of a design problem. Instead, these techniques should provide multiple indicators (symptoms) and let developers combine them during the identification of design problems. This knowledge has not been reported in any other study.

– In RQ4, we kept observing how developers identify design problems in their software systems. We had to continue our investigation with more experiments because, while the previous study used to answer RQ3 provides us a finding hitherto unknown, that study did not provide enough evidence to explain the phenomenon, which is the design problem identification. The investigation of RQ4 resulted in a theory describing this phenomenon, which serves to understand the identification of design problems in practice. We found the activities and factors that influence how developers identify design problems. For example, the theory presents the steps that developers follow to identify a design problem. This is another result that has been reported nowhere else. The findings provided by the theory not only shed light on how developers identify design problems, but these results can also be used to improve the techniques to help developers.

1.5 Thesis Outline

This introductory chapter portrayed an overview of this thesis. The remainder of the thesis is structured as follows. Chapter 2 introduces an overview of basic concepts required to understand the thesis and also presents the related work. Chapter 3 presents the study in which we use refactoring to investigate if code smells are key symptoms to identify relevant design problems, which provides answers to RQ1. Chapter 4 presents a study to investigate if developers can use code smells to identify design problems, which provides answer to RQ2. In this chapter, we also discuss if code smells suffice in helping developers to identify design problems. Chapter 5 describes a theory of how developers identify design problems in practice. We answer RQ3 and RQ4 in this chapter while we explain how the design problem identification happens in practice. Finally, Chapter 6 concludes this thesis by summarizing the achieved research contributions, making final considerations, and pointing out directions for future research.

2
Background and Related Work

This chapter presents the background and related work of this thesis. Section 2.1 outlines aspects of software design and how they relate to design problems. Section 2.2 presents the types of design problems addressed in the context of this thesis. Section 2.3 discusses the relation between code smells and design problems. Finally, Section 2.4 presents related work that investigated code smells and other indicators of design problems.

2.1
Software Design and Design Problems

Software design is the result of creating a software-based solution for a specific problem (2). To target the problem, a software system needs to meet a set of requirements that, when considered and treated all together, should result in the problem solution. Therefore, software design is the description of (i) how the system is decomposed and organized into components and (ii) how these components should behave (62, 63). Hence, it is a set of models and artifacts that record the major decisions that have been taken during the software development. Accordingly, the system stakeholders should make fundamental design decisions towards the problem solution. These design decisions will affect the software system and its development process. A **design decision** comprises the "description of the choice and considered alternatives that (partially) realize one or more requirements" (64). Therefore, design decisions and the reasons behind them describe what is allowed in the software system from its designing until its deployment and maintenance.

Eventually, design decisions impact the entire software system, either positively or negatively. For instance, let us consider non-functional requirements, which are aspects of or constraints on a system that are not specifically related to the system functionality; instead, they specify properties that the system must have, such as maintainability, understandability, usability, and performance (65, 66). Some inappropriate design decisions, such as overloading an interface with several non-cohesive services or spreading a functionality over several elements, may have a negative impact on non-functional requirements. In this vein, we state that a design problem exists when these inappropriate design decisions have a negative impact on non-functional requirements.

Design problem is the manifestation of one or more design decisions that impact non-functional requirements negatively. Figure 2.1 summarizes the relation between design decisions, design problems, and non-functional requirements. On the one hand, design decisions drive how the system will be developed; thus software design can be faced as the results of a set of these decisions (1, 2). On the other hand, each design decision can impact the system non-functional requirements either positively (straight arrows) or negatively (dotted arrows) (6). In the meantime, these design decisions affect how the code elements are implemented.

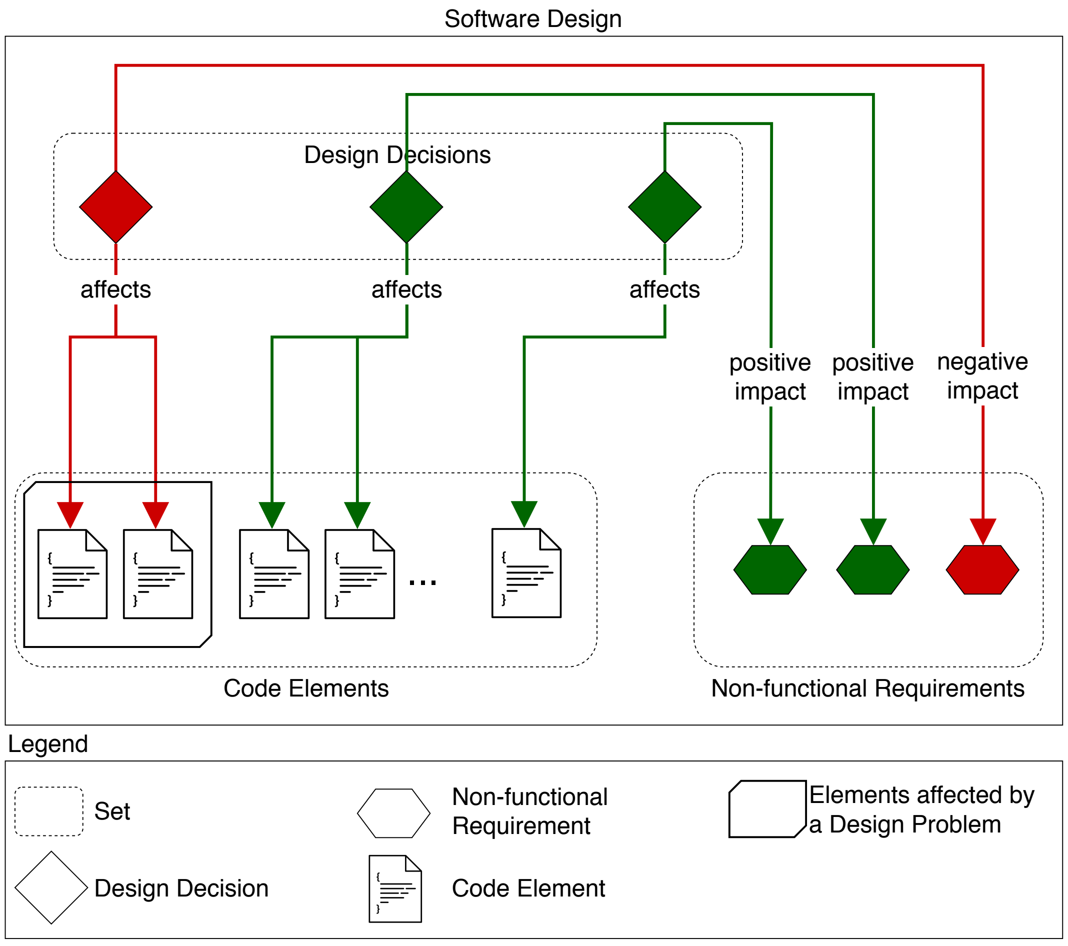


Figure 2.1: Design Problem

In the context of our research, we focused on those design problems that can be identified by the analysis of code elements affected by inappropriate design decisions, which negatively impact non-functional requirements. We focused on design decisions related to modularity aspects, *i.e.*, how the modules are designed and how they communicate in the system. Hence, we consider design decisions that include, but are not limited to, how the system is organized into subsystems and components, how and which code elements encapsulate process and data to address each functionality, and how the elements interact with each other and their execution environment (7, 8, 9). These design decisions are those related to design problems often harmful in software systems (14, 15, 16, 17).

Given the frequent lack of design documentation, developers have to rely on the analysis of code elements affected by the design decisions to identify a design problem, as illustrated in Figure 2.1. That is the reason why we focus on design problems that can only be identified from the analysis of relevant code elements for the system modularity, such as interfaces, components, hierarchies and other elements that encapsulate process and data in the system design (10). Consequently, design problems that (i) cannot be identified by the analysis of code elements, and (ii) are resulting from design decisions that are not related to modularity aspects are out of the scope of this thesis. For instance, we do not consider design problems that can only be identified with the design documentation or design problems related to the choice of an inappropriate database or library.

2.2 Types of Design Problems

Due to the type of design problems that we focus on, we use the inappropriate design decisions in addition to the affected code element (classes, hierarchies, interfaces, components, and the like) to classify the design problems. For instance, let us consider an inappropriate design decision related to overloading an element with many functionalities. If the overloaded element is a component, then we classify the design problem as a *Component Overload*. On the other hand, if an interface is the element overloaded with many functionalities, then we classify the design problem as a *Fat Interface*. In these two examples, the same inappropriate design decision led to two different design problems. However, the manifestation of the design decision was different in each code element. In other words, the combination of the design decision and the affected element was which characterized the type of design problem.

This classification allow us to focus on design problems that are domain-independent, *i.e.*, design problems that are not specific to a particular software domain (Section 2.4.1), but instead, design problems that recurrently appear in systems from different domains. Thus, we can classify some common design problems found in the literature (34, 7, 8, 31, 9, 52, 32, 11, 33). Figure 2.2 presents these design problems and how they are related to one or more design decisions. We describe these design problems next.

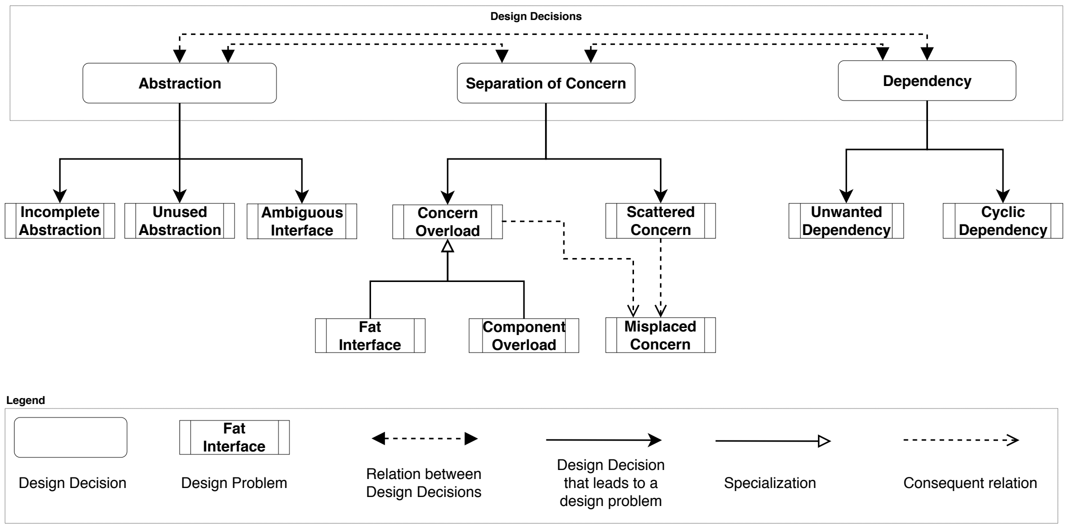


Figure 2.2: Types of Design Problems

2.2.1 Design Problems related to Abstractions

The first set of design problem types are those related to the design of abstractions. These design problems have in common an inappropriate design decision that a stakeholder made when he designed the abstraction, thereby leading to an incomplete, unused or ambiguous abstraction. Badly designed abstractions can be reified as one or more elements in a program.

The first design problem in the set is *Incomplete Abstraction* (ICA). This problem happens when a stakeholder designed an abstraction to partially implement a functionality; however, the rest of the responsibility is covered nowhere else (33). The stakeholder can also inappropriately split a responsibility among two or more code elements, which results in one or more incomplete abstractions (32). This design problem can be identified in the source code when a code element, usually a class, does not support a responsibility completely in its enclosing component (31, 32, 33).

Unused Abstraction (UA) is the second design problem related to the design of an abstraction. This design problem happens when the code element representing the abstraction is not directly used or is unreachable (34, 9). Usually, this design problem manifests in the source code due to modifications that make code elements obsolete, for instance, when the responsibility implemented by some classes ceases to be part of the system's functionality. Such scenario can happen to concrete classes as well as stand-alone interfaces and abstract classes that do not have subtypes or clients. In both cases, the system understandability decreases since these elements add an unnecessary effort to understand them even though they are unused in the system.

Ambiguous Interface (AMI) is the third design problem resulting from the inappropriate design of an abstraction. It refers to interfaces representing the abstraction that do not reveal which services it offers. AMI happens in the source when the interface of a component offers only a single, generic and ambiguous entry-point for its clients (7, 8). Usually, *Ambiguous Interface* appears in components that have several similar services to expose. However, these services are so similar to each other that they can be accessed by the same interface method. In this case, which differs a service from the other is the parameter received in the interface method. In other words, the component has an interface with only a general method with few parameters, at which the parameters dictate what service will be accessed. *Ambiguous Interface* hampers system understandability because the interface does not reveal which services a component is offering. Hence, the client has to inspect the component's implementation before using its services.

2.2.2
Design Problems related to Dependencies

The second set of design problem types are those related to the dependency between abstraction. Design problems within this set have in common dependencies that should not exist. These design problems manifest when stakeholders introduce dependencies that should not exist in the system either because they were not part of the intended system design or because they cause an undesired effect on non-functional requirements.

Unwanted Dependency (UWD) is the first design problem related to dependencies. It happens when a stakeholder creates a dependency in the software system that violates an intended design decision, *i.e.*, when there is a dependency between abstractions that does not exist in the intended set of design decisions of a system (6). In the source code, this design problem manifests when there is a dependency between code elements that was not defined in the system design or a dependency that violates architectural and design patterns. For instance, let us consider the Layer pattern (70). Specification for this pattern instructs to design a system in layers at which only adjacent layers can directly communicate with each other. However, if a layer communicates with another non-adjacent one, then there is a dependency that should not exist. This violation of the intended design decision leads to *Unwanted Dependency*. A design problem such *Unwanted Dependency* decreases the changeability of the system due to the introduction of dependencies that should not exist.

The second design problem related to dependencies is *Cyclic Dependency*

(CCD). This design problem happens when a stakeholder creates dependencies that create cycles. In the source code, this design problem manifest when two or more elements depend on each other directly or indirectly (11). This design problem can be introduced in different scenarios, for instance, when the stakeholder designs a class that calls a second one, which calls a third class; in its turn, the third class calls the first one back. These classes can even be in different components (13); in this case, there is the violation of the `Acyclic Dependencies Principle` (13). The cycle can also be created when a supertype directly or indirectly refers to one of its subtypes, forming a cycle in the hierarchy (71, 56). Despite the scenario, when there are long dependency cycles among the elements, the system might end up at a stage where these cycle dependencies compromise the understandability, testability, reusability, and maintainability of the software systems (11). Also, *Cyclic Dependency* can cause deadlock (12), which negatively impacts the system performance and availability.

2.2.3
Design Problems related to Separation of Concerns

The third set of design problem types are those related to the separation of concerns. A concern comprises anything that stakeholders of a software project may want to consider as a conceptual unit (69). Design problems in this set have in common the violation of the `Separation of Concerns Principle`, *i.e.*, when the stakeholder intentionally or not inappropriately decomposes the system concerns into dependent parts instead of independent ones (68, 67).

Concern Overload (CCO) is the first design problem related to the separation of concerns. It happens when a stakeholder creates an abstraction that fulfills to many concerns (52). In the source code, this design problem manifests in a interface, an abstract class or even a component. Depending on the element, this design problem can also violate other modularity principles. For instance, abstract classes and system components also tend to violate `Single Responsibility Principle` since they implement more than one concern (13). Interfaces, on the other hand, may also violate the `Interface Segregation Principle` once these interfaces force their clients to depend on methods they do not use (13). Despite the abstraction type, this type of design problem negatively impacts the system understandability, extensibility, reusability, and testability. As represented in Figure 2.2, *Concern Overload* can be split into other design problem types depending on the affected element. For example, when an interface is overloaded with concerns, it has the design problem named of *Fat Interface*. These specializations of *Concern Overload* are explained as follow.

Fat Interface (FTI) is a specialization of *Concern Overload*, which results from when the stakeholder designs the interface as the entry point for more than one unrelated concern (13). For instance, in the example discussed in Section 1.1, the stakeholder created the `IFacade` interface to provide access to services of three different components. However, the services (concerns) provided by these components were not related to each other directly. In this case, the stakeholder should have created an independent interface for each concern; instead, he put three unrelated concerns into only one interface. Consequently, he created dependency among them, violating the `Separation of Concerns Principle`. Furthermore, the interface clients are forced to depend on methods defined in the interface that they do not use. As a consequence, there is the violation of the `Interface Segregation Principle` as well. Design problems as *Fat Interface* impact the system understandability, extensibility, and testability since to understand, extend or test the services, it is required to handle the other services under the same interface. Similarly, the design problem also impacts the system reusability once the interface is too coupled with other modules and clients.

Component Overload (CPO) is the second specialization of *Concern Overload*, which happens when a stakeholder designs a system component to fulfill too many concerns (52). Hence, in addition to the violation of `Separation of Concerns Principle`, components that have this design problem are likely to contain classes violating the `Single Responsibility Principle` (13). This possibility exists because the classes within an affected component may be implementing more than one concern, which led to the *Component Overload* at the first place.

Conceptually, *Component Overload* and *Fat Interface* represent the same inappropriate design decision. They are the results of overloading an abstraction with too many concerns (*Concern Overload*). In fact, they represent two faces of the same decision, in which they differ in the role that the affected abstraction provides. *Fat Interface* provides an entry point to access the many concerns, whereas *Component Overload* contains implementations of these too many concerns. As an example, a system component may be implementing different services (*i.e.*, concerns), in its turn, these services can be exposed altogether through the same interface.

Scattered Concern (STC) is another type of design problem related to the separation of concerns (7, 8). This design problem type happens when the stakeholder decomposes the system concerns into dependent parts instead of independent ones (68, 67). As a consequence, multiple elements are responsible for partially implementing the same concern; thus, violating the `Separation of Concerns Principle`. To make matters worse, sometimes the scattered concern is not the predominant one in at least one of the elements. In these cases, the element implements two concerns: its predominant concern and another one, which the predominant concern can either the scattered one or not. Hence, this element also violates the `Single Responsibility Principle`. This type of design problem tends negatively to impact the system modifiability, understandability, testability, and reusability.

The last design problem related to the separation of concerns is *Misplaced Concern* (MPC). This type of design problem happens when a stakeholder creates an abstraction that implements a concern that is not its predominant one. In the source code, this design problem can be faced as the consequence of having either *Concern Overload* or *Scattered Concern*. For instance, if a component has two concerns instead of one, then one of them has been misplaced implemented in the component. Therefore, the component has at least two instances of design problems: *Component Overload* and *Misplaced Concern*. Similarly, when *Scattered Concern* affects a set of elements, there is a high chance of one of these elements to implement a misplaced concern. In this scenario, we can have two possible alternatives. In the first alternative, the predominant concern in the element is the same scattered one. Thus, the misplaced concern is another implemented concern. In the second alternative, the predominant concern in the element is not the scattered one. Thus, the misplaced concern is exactly the scattered one.

2.3 Code Smells and Design Problems

As discussed in the previous section, design problems have a negative impact on non-functional requirements. In fact, their negative consequences for software systems can increase the cost related to the maintenance of software projects. Despite they are often targets of significant maintenance effort (7, 17, 20), identification of design problem is non-trivial (26, 27). First, software systems tend to be increasingly large in size and complexity, thereby expanding the search space for problems. Second, some design problems may pervade the implementation of multiple elements (7, 28). Thus, developers may need to analyze several elements to identify a single design problem (27). Third, design documentation is often unavailable or outdated, making the source code the only artifact available for developers to identify design problems in most cases.

Due to the lack of design documentation, developers need to locate indicators of design problems directly on the source code. Some studies have been investigating information extracted from the source code that can be used as indicators of design problems, such as metrics (72), quality attributes (42), modularity principles (27, 58). In our context, any indicator that developers use in practice to identify a design problem, we call it as *symptom*. Therefore, we define a design problem **symptom** as a partial sign or indication of the presence of a design problem that is used in practice. Code smell is an example of symptom usually associated with design problem identification. A code smell is a microstructure in the system that represents a surface, sometimes partial, indication of a design problem (21). Examples of code smell types vary from method-level smells, such as `Long Method` and `Feature Envy`, to class-level smells, such as `God Class`, and `Data Class` (21, 30). Table 2.1 presents 17 types of code smells. These are code smells that are commonly studied in the literature, and some of them are closely related to design problems.

Studies based on code smells use them as indicators of design problems since code smells tend to co-occur in elements affected by design problems (53, 47, 54). The use of code smells to indicate design problems is indeed reasonable as each smell may be fully or partially associated with a design problem (41, 38, 28, 47, 73, 27, 46). For instance, `Speculative Generality`, a code smell that indicates an element, usually a class, that was created to support anticipated future features that never have been implemented (21). This code smell can be associated with *Unused Abstraction* (34, 9). Moreover, smells can be identified directly in source code, which is often the only tangible artifact available for developers. Hence, it is no surprise that the use of code smells is often associated with design problem identification in the literature (21, 41, 38, 39, 28, 47, 46).

Some of these aforementioned studies have investigated the relation between code smells and design problems (74, 16, 75, 53, 38, 28, 47, 51, 76, 27, 55, 29). Based on them, we can associate some types of code smells with some types of design problems, as we did when we associated `Speculative Generality` with *Unused Abstraction*. This association is possible due to smell patterns reported in studies that investigated the relation between smells and design problems. A **smell pattern** in our context is one or more types of code smells that are likely to indicate a design problem if they appear in code elements. Table 2.2 shows these patterns.

As an example of how this association between code smells and design problems can help developers in the identification task, let us consider the example in Figure 2.3, which uses a UML-like notation to show a partial

Table 2.1: List of Smell Types

Smell Type	Description
Brain Class	Long and complex class that centralizes the intelligence of the system
Brain Method	Long and complex method that centralizes the intelligence of a class
Class Data Should Be Private	A class exposing its fields, violating the principle of data hiding
Complex Class	A class having at least one method having a high cyclomatic complexity
Data Class	These are classes that have only fields and accessors methods
Dispersed Coupling	A method that accesses many elements, and the accessed code elements are dispersed among many classes
Feature Envy	A method that is more interested in a class other than the one it actually is in
God Class	When a class centralizes the system functionality
Intensive Coupling	A method that has tight coupling with other methods, and these coupled methods are defined in the context of few classes
Lazy Class	A class having very small dimension, few methods and with low complexity
Long Method	A method that is unduly long in terms of lines of code
Long Parameter List	A method having a long list of parameters, some of which avoidable
Message Chain	A long chain of method invocations is performed to implement a class functionality
Refused Bequest	A class redefining most of the inherited methods, thus signaling a wrong hierarchy
Shotgun Surgery	When a change performed on it demands a lot of little changes to several different classes
Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables
Speculative Generality	A class declared as abstract having very few children classes using its methods.

view of a university management system: UniM system. Service is one of the main components in the system, which is composed of classes that implement services for database transactions. AbstractService is an abstract class within the component that defines the main database operations, such as inserting element, searching an element by id and updating elements. Thus, all the service classes within the component extend the AbstractService class. Due to the relevance of Service component, a developer may search for design problem on it. Classes that extend the AbstractService is a good start point for the analysis since they should have similar characteristics due to their common methods to access the database. One of these classes is the InstitutionalEnrollmentService.

After analyzing InstitutionalEnrollmentService source code, a developer can find different instances of code smells. This class contains instances of `Long Method`, *i.e.*, methods that are very long regarding lines of code

Table 2.2: Association Between Design Problems and Smell Patterns

Design Problem	Code Smells
Ambiguous Interface	Long Method and Feature Envy and Dispersed Coupling in elements that are related to the interface
Cyclic Dependency	Intensive Coupling and Shotgun Surgery
Component Overload	Shotgun Surgery, and Divergent Change, and Feature Envy, and God Class, and Intensive Coupling, and Long Method
Concern Overload	Divergent Change, and Feature Envy, and God Class, and Intensive Coupling, and Long Method, and Shotgun Surgery
Fat Interface	Shotgun Surgery in the interface or Divergent Change, and Dispersed Coupling, and Feature Envy in elements related to the interface
Incomplete Abstraction	Lazy Class
Misplaced Concern	God Class or Dispersed Coupling, and Feature Envy, and Long Method
Scattered Concern	Dispersed Coupling, and Divergent Change, and Feature Envy, and God Class, and Intensive Coupling, and Shotgun Surgery
Unused Abstraction	Speculative Generality
Unwanted Dependency	Feature Envy, Long Method, Shotgun Surgery

(LOC), for instance, the calGradePointAverage method. This method receives a student enrollment and calculates its grade point average – the average of all grades from all classes that a student has been enrolled. To perform this computation, the class has to call other classes, such as InstitutionalEnrollment[1], StudentService and CourseService, to calculate the average. Due to this high coupling with other classes, InstitutionalEnrollmentService contains two other types of code smells: `Dispersed Coupling` and `Intensive Coupling`. In fact, methods as calGradePointAverage also contain instances of `Feature Envy`, since these methods have to access other classes to perform their computation. Consequently, they become more interested in other classes rather than in its own class (21).

The smell pattern composed of `Intensive Coupling`, `Long Method`, and `Feature Envy` in the class are associated with the *Concern Overload*, as showed in Table 2.2. Indeed, the reason for these smells appear in the InstitutionalEnrollmentService is because the class implements two distinct concerns, *query enrollment* and *management enrollment*, which are represented in the figure by green and blue, respectively. The first implemented concern, *query enrollment*, is due to the hierarchical relation with AbstractService and its enclosing component (Service). The second implemented concern, *management enrollment*, is due to a responsibility that should not have been implemented by the

[1]This class does not appear in Figure 2.3 because it belongs to another component.

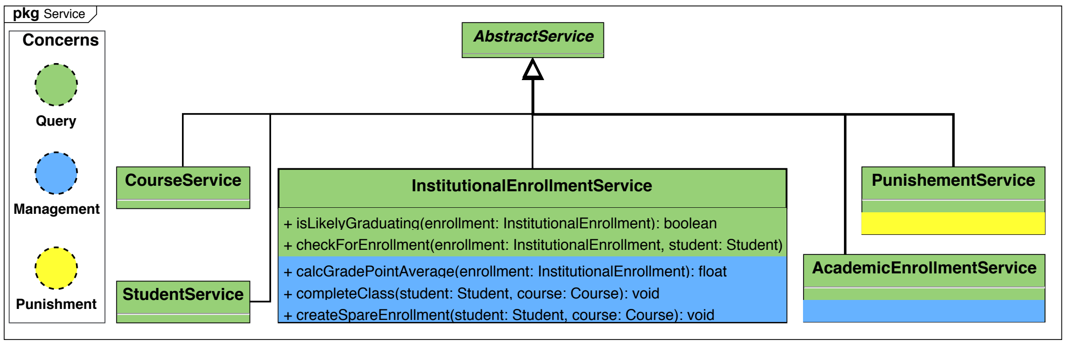

pkg Service
Concerns
Query
Management
Punishment
AbstractService
CourseService
StudentService
InstitutionalEnrollmentService
+ isLikelyGraduating(enrollment: InstitutionalEnrollment): boolean
+ checkForEnrollment(enrollment: InstitutionalEnrollment, student: Student)
+ calcGradePointAverage(enrollment: InstitutionalEnrollment): float
+ completeClass(student: Student, course: Course): void
+ createSpareEnrollment(student: Student, course: Course): void
PunishementService
AcademicEnrollmentService

Figure 2.3: Service Component in the UniM System

class. Hence, the class has *Concern Overload* design problem indeed. Thus, the code smells appear in the class due to the overload of concerns, which, in its turn, can be used to identify the design problem. Unfortunately, InstitutionalEnrollmentService is not the only class within the Service component that implements more than one responsibility; PunishmentService and AcademicEnrollmentService are some of these classes. Therefore, Service component manifests *Component Overload* as well.

This example in Figure 2.3 illustrates how code smells can be indicators of design problems. Indeed, the example also shows the association between code smells and design problems. Similar to design problems, code smells appear in source code due to inappropriate decisions, which can lead to a design problem. Even though some authors tend to classify code smells and design problems as the same (28, 73, 27), they have subtle differences. Code smells tend to affect a reduced scope, for instance, the `Long Method` that affects only the scope of calGradePointAverage method. On the other hand, some design problems may affect multiple code elements, such as *Scattered Concern*. Furthermore, design problems tend to occur in relevant elements for the system, sometimes affecting multiple elements associated with the design problem, for instance, the Service component. Additionally, code smells do not impact non-functional requirements with the same intensity as design problems. For example, a method that is long regarding LOC (`Long Method` smell) has a less impact on understandability then a component overloaded of concerns (*Component Overload*). Due to these subtle differences, code smells are usually faced as *symptoms* of the presence of a design problem. Hence, it is no surprise that they have been the center of different techniques to identify design problems (41, 38, 39, 28, 47, 46).

2.4 Related Work

This thesis encompasses the search for knowledge regarding the process of identifying design problems directly in source code. In fact, other studies have contributed to gathering this knowledge. In this section, we discuss some of these studies and how they have contributed to shed some light on the identification of design problems. Section 2.4.1 presents some studies towards the categorization of design problems. In this subsection, we discuss some particularity of each catalog. Additionally, we discuss the strengths and weaknesses of the catalog based on our point of view of what constitutes a design problem. Section 2.4.2 presents studies that explore the relation between code smells and design problems. In this subsection, we discuss how these studies contributed to show how code smells are associated with design problems, and how they can be used as indicators of design problems. Finally, Section 2.4.3 presents studies that proposed techniques to identify design problems. In this subsection, we discuss how these studies extract information from source code that can indicate design problems.

2.4.1 Catalogs of Design Problems

Some authors have proposed classification schemes to classify common design problems into a catalog. Each scheme usually represents the author's point of view of what constitutes a design problem. We discuss these catalogs here, presenting their main particularities, classification scheme and also intersections with the set of design problems that we consider in this thesis (Section 2.2).

Garcia *et al.* (8) proposed a set with four design problems: *Connector Envy*, *Scattered Parasitic Functionality*, *Ambiguous Interfaces*, and *Extraneous Adjacent Connector*. These design problems, which they call architectural bad smells (7), are the results of design decisions that negatively impact system lifecycle properties, such as understandability, testability, extensibility, and reusability. The four design problems in their catalog emerged from the reverse-engineering and re-engineering of two large industrial systems as well as case studies that the authors found in the literature. In this catalog, the authors specified the four types of design problems regarding to standard architectural building blocks (9, 79): components, connectors, interfaces, and configurations (relations between components and connectors). A disadvantage of such a specification is that it does not comprise some types of design problems, such as those related to an inappropriately designing of abstractions (*e.g.*, *Incomplete*

Abstraction). Furthermore, not all software systems can be easily represented according to these architectural building blocks, which hinders the search for these four types of design problems in these systems. Therefore, the set of design problems we consider in this thesis only includes design problems from Joshua *et al.*'s catalog that are independent of this block-based specification. They are *Ambiguous Interface* and *Scattered Parasitic Functionality*, which is similar to *Scattered Concern*.

Another catalog of design problems is the one proposed by Ganesha, Sharma and Suryanarayana (80). In this catalog, the authors used the violation of one of four fundamental object-oriented design principles (principles of abstraction, encapsulation, modularity, and hierarchy) to create a classification scheme. As a consequence of using the violation of object-oriented principles, this catalog includes code smells and design problems, which authors call them both of design smells. Posteriorly, Suryanarayana, Samarthyam and Sharma (73) extended the catalog to a total of 31 design smells. A disadvantage of this catalog is to treat design problems and code smells as part of the same group. In addition to overly simplifying the definition of code smells and design problems, considering them to be part of the same catalog compromises the notion of code smells being a type of design problem symptom. In other words, an issue in treating them all together is not to be able to use code smells as an indicator of design problems. On the other hand, an advantage of this catalog is the authors used the violation of common object-oriented principles to classify their design smells, which leads to a uniform categorization. Additionally, the names used for each type of design smell provides an intuitive understanding of the smell. Furthermore, the authors exemplified the negative impact of some of these smells. On the whole, we only considered some design smells that meet our definition of design problems.

Trifu *et al.* also proposed a catalog of design problems (27, 58). Their catalog comprises design problems that are the results of design decisions that contradict commonly accepted design practices. Their catalog contains three types of design problems, which was expanded as part of Trifu's PhD thesis to a total of ten design problems (81). Even though Trifu *et al.* used different terminology to name the design problems, their catalog includes some common design problems found in the literature, such as *Concern Overload*. On the other hand, other design problems in their catalog correspond to code smells. Such intersection happens because Trifu *et al.* used common design principles to classify the design problems. For instance, a violation of a hierarchy relation in which a subclass uses only some of the methods inherited from its parent will be catalogd as a design problem, even though there is a code smell with this characteristic: `Refused Bequest` (21). Such intersection does not represent an issue; as previously mentioned, some authors consider code smells and design problems as the same structural problem. However, in the case of Trifu *et al.*'s catalog, this intersection can be problematic because they use code smells as indicators of design problems. Therefore, this intersection can be faced as an inconsistency in their catalog. Despite this inconsistency, a strength of their catalog is to treat each design problem as a disease. Thus, they can characterize each design problem according to the symptoms they manifest in the system as well as a possible treatment for it.

2.4.2 Relation between Code Smells and Design Problems

As previously discussed, code smells have been in the center of several techniques to support the identification of design problems. The reason is due to their relation to design problems. In this sense, some studies have shown how code smells can be an indicator of software maintenance problems such as design problems. For instance, Abbes *et al.* (49) brought up the notion of code smells that interact to each other in the source code. They conducted three experiments to verify the impact of two code smells (`God Class` and `Spaghetti Code`) on software understandability. They also investigated the effects of such interaction. They concluded that classes affected by `God Class` and `Spaghetti Code` in isolation did not increase the maintenance effort; however, when these two smells appeared together, they led to a statistically significant increase in maintenance effort.

Yamashita and Moonen (50) also investigated the interaction between code smells. Nevertheless, they conducted an empirical study with a set of 12 code smells to observe how their interactions affect the software maintenance. In the study, the authors hired six software engineers to maintain four medium-sized Java systems during a month. Their results indicate that smells that interact with each other negatively affect the software maintenance. Posteriorly, Yamashita *et al.* (54) replicated the aforementioned study. In their new study, they investigated collocated smells – code smells that interact in the same source code file –, and coupled smells – code smells that interact across different source code files. In addition to corroborating their findings from the previous study, they observed that limiting the analysis to collocated smells would reduce developers capability to reveal design problems since coupled smells may reveal critical design problems that were not revealed by collocated smells.

These studies mentioned above indicated that not all code smells are related to design problems. In this context, Macia *et al.* (52) investigated the relation between code smells and design problems. They analyzed the relevance of code smells to identify design problems in six software systems. Their results showed that the majority of the design problems in the source code happened in code elements with code smells. In fact, approximately 65% of all detected code smells were related to 78% of all design problems used in the study. In this vein, Macia *et al.* (38) investigated if code smells identified by automatic detection strategies were related to design problems. Unfortunately, their result revealed that many of the code smells detected by the employed strategies were not related to design problems. Even worse, over 50% of the smells that have not been detected by the strategies were related to design problems. Such results suggest that state-of-the-art detection strategies are not able to locate relevant code smells to support the identification of design problems. In order to tackle such deficiency, Macia *et al.* (53) proposed a tool, named SCOOP, that detects code smells that are relevant to the identification of design problems. The evaluation showed that SCOOP was able to detect code smells related to 293 out of 368 design problems found in three software systems. In this study, Macia *et al.* also proposed the detection of recurring code smells patterns. Each pattern comprises a group of code smells structurally related in source code that are likely to indicate a design problem.

Following Macia's steps, Oizumi *et al.* (82) used the code smells patterns to improve the notion of group of interrelated code smells, leading them to define code smell agglomerations. An agglomeration consists of a group of code smells that are related to each other for some reason, for instance, through method calls or inheritance. Oizumi *et al.* analyzed seven systems of different sizes and found that 70% of all design problems were related to agglomerations in most of the analyzed systems. Given such result, Oizumi *et al.* (82) investigated to what extent agglomerations and design problems are associated with each other. They analyzed a total of 5418 code smells and 2229 agglomerations within seven systems. Their results indicated that most of the code smells associated with a design problem were part of one or more agglomerations. In fact, for each agglomeration associated with a design problem, an average of 2 to 4 code elements with smells were affected by a design problem. Finally, their results hinted that some types of agglomerations could be better indicators of design problems than others. Regarding such matter, Oizumi *et al.* (47) investigated which types of agglomerations are likely to be associated to a design problem. They also investigated if agglomerations were more likely to be related to design problems than single smells, *i.e.*, smells that are not part of any agglomeration. After analyzing more than 2200 agglomerations of code smells from seven software systems, they concluded that certain types of agglomerations, such as those related to the separation of concerns, are consistent indicators of design problems. They also found that the chance of each code smell within an agglomeration being related to a design problem is more than five times higher than every single smell.

These studies showed that code smells are related to design problems. Therefore, not only they brought knowledge about the identification of design problems, but they also justified why code smells have been used in several techniques. Unfortunately, these studies fell short of evaluating if code smells are key symptoms for developers in practice and if developers can use them to identify design problems. Indeed, these techniques can narrow down the number of code smells for developers to analyze, but developers may not be able to use these smells to identify a design problem, which can be even worse if code smells are not key symptoms in practice. Since these studies did not conduct such an evaluation with developers in practice, we do not know to what extent these techniques support developers during the identification of design problems. Consequently, we sill do not know what is the role that code smells play in supporting developers in identifying design problems.

2.4.3 Techniques to Identify Design Problems

Studies from previous subsection showed the relation of code smells and design problems. However, code smells are not the only indicator of design problem that can be extracted from the source code. Some studies have focused on other information that can be used to indicate design problems. For instance, Xiao, Cai and Kazman (42) have used the Design Rule Theory (83) to identify design problems. Design Rule Theory states that the system should be decoupled into mutually independent modules. Xiao, Cai and Kazman followed this theory to visualize the system as a set of independent modules and design rules, which are design decisions that decouple the system into independent modules. They called this visualization of DRSpace; based on it, they represented the systems as a square matrix, which shows all the files in source code and their connection. Using DRSpace, they observed classes that changed together, despite not being related to each other. They found that these classes contained design problems and were the cause of bugs. This result was further confirmed by Kazman *et al.* (43), who were able to identify these classes.

Based on Xiao, Cai and Kazman's study, Mo *et al.* (44) proposed and evaluated a suite of hotspot patterns, which are recurring design problems that lead to high maintenance cost. These patterns were defined using the DRSpace notion, and they were detected by the combination of structural, history and design information. In their study, they showed that these patterns might be the cause of bug-proneness and change-proneness. Furthermore, they consulted developers, who mentioned that these patterns comprised design problems. Posteriorly, Xiao *et al.* expanded the DRSpace notion to include a conditional probability of a file change (45). Thus, they built a regression model to approximate the probabilities of change propagation among files. Based on this model, they could quantify the technical debt due to design problems. Unfortunately, a downside of these studies regards the information they use to identify design problems. These studies rely on history and design information, which may not be available for many software systems, specially if these system are in their first versions.

Mo, Gueheneuc and Leduc proposed a technique that uses a domain-specific language to describe some code smells (39). Then, they used the description to generate detection algorithms automatically from the specification of each code smell. Their technique was developed to identify code smells mainly; posteriorly, Mo *et al.* extended it to detect design problems as well (40, 59). Finally, Mo *et al.* improved their previous work by proposing DECOR, a method that describes steps necessary for the specification and detection of code smells and design problems (28). A downside of Mo *et al.* technique is the dependency to the domain-specific language. A developer to use DECOR has to specify the types of design problems that he needs to identify. However, a developer may not know how to specify some types of design problems. In fact, some types of design problems may not be specified, especially considering that the characteristics of each type of design problem may change from a software system to another one. Thus, a developer would need to keep changing the specification for each new setting.

2.5 Summary

This chapter presented the main concepts addressed in this thesis. In Section 2.1, we presented aspects of software design that are related to design problems. In particular, we presented the definition of design problem adopted throughout this thesis. According to our definition, a design problem is the result of one or more inappropriate design decisions that have a negative impact on non-functional requirements. In Section 2.2, we used this definition to categorize the types of design problems. We introduced a classification scheme based on the combination of inappropriate design decisions and affected elements. We used this scheme to classify our set of design problems according to three categories.

In Section 2.3, we discussed the relation between code smells and design problems. In this section, we presented an example with two-fold purposes: (i) to illustrate how code smells can be used to indicate design problems, and (ii) to show the difference between code smells and design problems. We also discussed in this section why code smells have been used in several techniques as an indicator of design problems.

Finally, we presented some related studies in Section 2.4. We divided this section into three subsections. The goal was to explore how studies in each subsection helped to bring knowledge to the process of identifying design problems. First, we presented some studies that categorized some design problems found in the literature. Second, we presented studies that explored the relation between code smells and design problems; thus, allowing us to discuss why code smells have been the center of several techniques. Finally, we presented techniques that used other types of symptoms to identify design problems.

Based on the discussion presented in this chapter, we claim that we do not know if code smells suffice in helping developers to identify design problems. Thus, the next chapter provides the first investigation to better understand the role of code smells in helping developers to identify design problems. For this purpose, we conducted a retrospective study aimed to find if code smells are symptoms to developers' identification of design problems that are somehow relevant for them.

3
Investigating Code Smells as Key Symptoms in Practice

In the previous chapter, we discussed different types of design problems. We also discussed how a developer can use one or multiple code smells to identify a design problem. Some studies have explored the relation between code smells and design problems (53, 47, 54). The relation was established due to the co-occurrence of code smells and design problems in the same code elements, which made code smells to be considered consistent indicators of design problems (41, 38, 47, 46). Consequently, code smells became the center of different techniques to support the identification of design problems (41, 38, 39, 28, 47, 46).

These studies tend to take code smells for granted when they assume that smells are consistent indicators of design problems. Indeed, code smells can be consistent indicators of design problems, but in the end, the consistency has little or no importance if smells are not key symptoms for assisting developers to identify design problems along a software project. A key symptom is one that helps developers to identify design problems that are somehow relevant in the software system. If code smells are not key symptoms for developers, then techniques that rely on smells may fall short of supporting developers in identifying design problems. Unfortunately, we do not know if code smells are key symptoms since these studies did not conduct this investigation in practice. As a matter of fact, our knowledge about the role of code smells in the design problem identification is limited. As a first step to understand their role in the design problem identification, we need to investigate if they are indeed key symptoms for developers identifying design problems in their projects.

To conduct this investigation, we can observe if code smells indicate design problems that are relevant for developers in their projects. A relevant design problem can be one that developers ended up identifying and removing from the source code. We can consider these design problems to be relevant because developers decided to focus their effort on removing them from the source code, for instance, by applying refactoring operations. Refactoring is a program operation used for improving the code structure of a system (21). Often, it can be applied to repair a deteriorated code (25), which can be deteriorated due to the presence of design problems. Since the presence of design problems is one of the reasons why elements can reach a deteriorated state (15, 16), we can investigate elements that developers focused on during refactoring. Thus, if these elements contain any sign of design problems and code smells simultaneously, then we can assume that code smells could have been used by developers to spot relevant design problems in these elements.

To investigate if code smells are key symptoms for developers to identify design problems, we conducted a retrospective study to analyze the source code of refactored elements. Then, we analyzed if refactored elements, *i.e.* elements that were refactored, contained code smells. In fact, when we conduct a study in which we observe how developers refactored their code, we are analyzing a snapshot of developers' work. Thus, to a certain degree, we are investigating what happened in practice. Our results indicate that most refactoring operations (79.48%) are applied to elements that contain at least one code smell. In fact, almost half of the refactoring operations (47.38%) were applied to elements that contained multiple code smells. Since we noticed signs that these elements had a design problem, we concluded that in most cases, code smells are key symptoms for developers to identify design problems. Unfortunately, we found some scenarios in which code smells cannot indicate design problems. Indeed, when we analyze our results considering results found in the literature, we conclude that in most cases code smells likely represent key symptoms to identify relevant design problems; however, smells still may not suffice in helping developers to identify design problems in practice.

We describe this study in details next. In Section 3.1, we present basic concepts and terminology used throughout this chapter. In Section 3.2, we present our research question. In Section 3.3, we present the procedure used to conduct our investigation. In Section 3.4, we discuss the results of our investigation. In Sections 3.5 and 3.6, we present some related studies and threats to validity, respectively. Finally, we summarize this chapter in Section 3.7.

3.1 Background and Terminology

Software systems invariably undergo changes over the evolution that can compromise the structural quality of the systems. If these changes are carried out recklessly, software systems may reach a deteriorated state that requires either significant maintenance effort or the complete redesign (14, 15, 16, 17). Since these changes are driven by design decisions, the introduction of design problems is a reason why software systems can reach such deteriorate state (15, 16). When the software systems reach such a state, developers usually apply refactoring operations to repair the code.

Refactoring is defined as a program transformation intended at preserving the observable behavior of the software system while improving its internal structure (84, 21). Often, refactoring operations are applied to repair deteriorated code (25), which may happen by removing design problems (22, 23, 24). Refactoring is a complex code transformation, which includes a type and a goal, both associated with what the developer intends to do. We describe these particularities and others as follow.

Refactoring Type. Refactoring type indicates if the refactoring operation is applied to attributes, methods, classes, or interfaces. These types encompass changes that can involve only a single class as the *Extract Local Variable* refactoring or multiple classes as the *Extract Class* refactoring. Other examples of refactoring types involve (25): (i) restructuring or moving class members, such as *Extract Methods*, *Move Method* and *Pull Up Method*, and (ii) extracting new elements, such as *Extract Superclass* and *Extract Interface*. For this study, we have considered the 13 types of refactoring operations presented in Table 3.1. These types, defined in Fowler's catalog (21), comprise the most popular types of refactoring operations (25). They are also applied to remove the code smells that can indicate a design problem (21).

Refactored Elements. All elements directly affected by refactoring operations are considered as refactored elements. For each refactoring type, a different refactored element set is directly affected by the refactoring operation. Let us consider the *Move Method* refactoring, in which a method m is moved from class A to B. In this case, the refactored elements are: m, A and B. Although other elements can be indirectly affected by the *Move Method* on moving m from A to B, we consider only these three elements as refactored ones. Table 3.2 shows the list of elements that can be directly affected by each refactoring type. We should know what are the elements affected by the refactoring operations since these are the elements that we should search for design problems.

Refactoring Tactics. The goals of refactoring widely vary in practice (85, 86, 87). The goals include combating design degradation, reducing maintenance effort, and facilitating feature additions or bug fixes (85, 86, 87). Given these goals, developers may follow two main tactics when they refactor the source code (25): *root-canal refactoring* and *floss refactoring*. Developers apply root-canal refactoring to repair a deteriorated code, and it involves a process of

Table 3.1: List of Refactoring Types

Type	Description
Extract Interface	Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common. Then, we extract the subset into an interface.
Extract Method	A code fragment that can be grouped together. Then, we turn the fragment into a method whose name explains the purpose of the method.
Extract Superclass	Two classes with similar features. Then we create a superclass and move the common features to the superclass.
Inline Method	A method is, or will be, using or used by more features of another class than the class on which it is defined. Then, we create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
Move Class	A class that is in a package that contains other classes that it is not related to in function. Then, we move the class to a more relevant package. Or create a new package if required for future use.
Move Field	A field is, or will be, used by another class more than the class on which it is defined. Then, we create a new field in the target class, and change all its users.
Move Method	A method is, or will be, using or used by more features of another class than the class on which it is defined. Then, we create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
Pull Up Field	Two subclasses have the same field. Then, we move the field to the superclass.
Pull Up Method	Methods with identical results on subclasses. Then, we move them to the superclass.
Push Down Field	A field is used only by some subclasses. Then, we move the field to those subclasses.
Push Down Method	Behavior on a superclass is relevant only for some of its subclasses. Then, we move it to those subclasses.
Rename Class	The name of a class does not reveal its purpose. Then, we change the name of the class.
Rename Method	The name of a method does not reveal its purpose. Then, we change the name of the method.

exclusively applying refactoring operations. As the goal of this tactic is to repair deteriorated code, there is a high chance of this tactic to be applied to repair the code by removing design problems. For instance, to get rid of *Misplaced Concern*, a developer can apply a *Move Method* and *Move Field* to move the misplaced concern to the class to which it actually belongs. Conversely, developers apply floss refactoring with the intention of achieving another objective that is different from structural improvements, such as adding a new feature or fixing a bug. For example, to fix a bug related to hierarchy, a developer may need to apply the *Push Down Field* before fixing the bug. In order to add a new feature, developers may need to refactor the system to accommodate the new feature. Even in the floss refactoring, refactoring is applied in conjunction with other changes to remove a deteriorated state as a first step to achieve the objective.

Table 3.2: Refactored Elements

Type	Refactored Elements
Extract Interface	Classes implementing the new interface.
Extract Method	The method created; the method from where the new method was extracted; and class containing both methods.
Extract Superclass	Classes extending the new class; and new class created.
Inline Method	The method which received the new code; and class containing the method.
Move Class	The class.
Move Field	The two classes affected by the change: the class which the field used to reside and the class which received the field.
Move Method	The two classes affected by the change: the class which the method used to reside and the class which received the method.
Pull Up Field	The two classes affected by the change: the class which the field used to reside and the class which received the field.
Pull Up Method	The two classes affected by the change: the class which the method used to reside and the class which received the method.
Push Down Field	The two classes affected by the change: the class which the field used to reside and the class which received the field.
Push Down Method	The two classes affected by the change: the class which the method used to reside and the class which received the method.
Rename Class	The class.
Rename Method	The renamed method and the class that contains it.

3.2 Study Design

This section presents the design of our study. Section 3.2.1 presents the research question that drives the study. Section 3.2.2 presents how we categorize the refactored elements according to the probability of the element to contain a design problem.

3.2.1 Research Question

Recent studies have been using code smells as an indicator of the presence of a design problem (Section 2.4.2). In fact, code smells became the center of several techniques to support the design problem identification (Section 2.3). The reason is due to the co-occurrence of code smells and design problems, which led recent studies to explore code smells as the primary design problem indicator (53, 47, 54). Unfortunately, these studies have relied on code smells without investigating in practice if code smells are key symptoms for developers to identify design problems.

There is a subtle difference between being a consistent indicator of a design problem and being key symptoms for developers to identify relevant design problems in practice. As showed by some studies (53, 47, 54), code smells can be indeed consistent indicators of design problems due to the co-occurrence. However, when we analyze from the developers' perspective, code smells may not indicate relevant design problems in practice, *i.e.*, they are not key symptoms used in actual project settings. If so, then techniques that rely on smells are likely to fall short of expectations.

Unfortunately, whether code smells are key symptoms for developers to identify design problems is knowledge that we do not have. This lack of knowledge is worrisome, since it compromises how developers identify design problems when relying on code smells. Indeed, code smells have been proposed to support developers during the design problem identification; however, we do not know in practice to what extent code smells support developers in this task. Therefore, to better understand the role of code smells in the design problem identification, we need to investigate if they are key symptoms for developers to identify relevant design problems.

One way is to conduct a retrospective investigation about the practice. In this investigation, we need to find the elements that developers identified design problems, and we verify if these elements contain code smells that could have been used to indicate the design problem found by developers. If code smells could indicate these design problems, then we can assume that they are key symptoms for developers to identify design problems. In this investigation, the first step is to identify the code elements in which developers found design problems. We can rely on refactoring to identify them.

Refactoring is a common activity that developers use to repair a deteriorated code (21, 25), while the presence of design problems is a reason for the code to reach a deteriorated state. Thus, there is a close relation between refactoring and design problems, which makes it reasonable to rely on refactoring in our investigation. Furthermore, if we can guarantee that, to a certain extent, the refactored elements had design problems (Section 3.2.2), then we can state that these design problems are relevant for developers. They are relevant because developers focused their effort to identify and remove them from the source code through refactoring. Consequently, if we find code smells in these refactored elements, we can conclude that code smells are indeed key design problem symptoms. In the context of this investigation, we intend to answer the following research question:

RQ1. Are code smells key symptoms to indicate relevant design problems for developers?

To answer this research question, we conducted a retrospective investigation with 50 software projects. We went through the commits of each project to identify the refactored elements in the history of each system. After that, we verified if these refactored elements had code smells that could have been used to spot any design problem in the element. To make this verification, we had to assure that these refactored elements were likely to contain design problems. We explain as follows how we verified if these refactored elements could have had any sign of the presence of design problems.

3.2.2 Categorization of Refactored Elements

We have to investigate if code smells are key symptoms to indicate a relevant design problem. For this investigation, we rely on refactoring to identify elements that were likely to contain design problems. Therefore, to answer our research question, we are assuming that the refactored elements could have had design problems, which motivated the refactoring. Such assumption is plausible since refactoring and design problem identification are directly related to each other: the former activity succeeds the latter. As we can see in the steps represented in Figure 3.1, some design problems affect code elements (*Step 1*) in such a way that they can lead the elements to reach a deteriorated state. Once design problems are identified, developers can apply refactoring operations in code elements affected by design problems (*Step 2*) to repair them by removing the design problems. Since developers focused their effort on identifying and removing design problems through refactoring, then these design problems are relevant for developers. Then, we can verify if the refactored elements contain code smells (*Step 3*). Based on the relation between design problem and refactoring, we can answer our research question (*Step 4).*

Even though there is a relation between refactoring and design problems, we cannot presume that all the refactoring operations were applied to elements with a design problem. As discussed in Section 3.1, developers can refactor the elements to achieve another objective that is different from structural improvements (floss refactoring). Therefore, we need to assure that these refactored elements were likely to contain design problems. For such assurance, we categorized the refactored elements according to their probability of containing design problems. As we want to investigate if code smells are key symptoms to identify relevant design problems, our first categorization regards the presence or absence of code smells:

– **Smell-free category**: This category encompasses the refactored elements that are NOT affected by code smells.

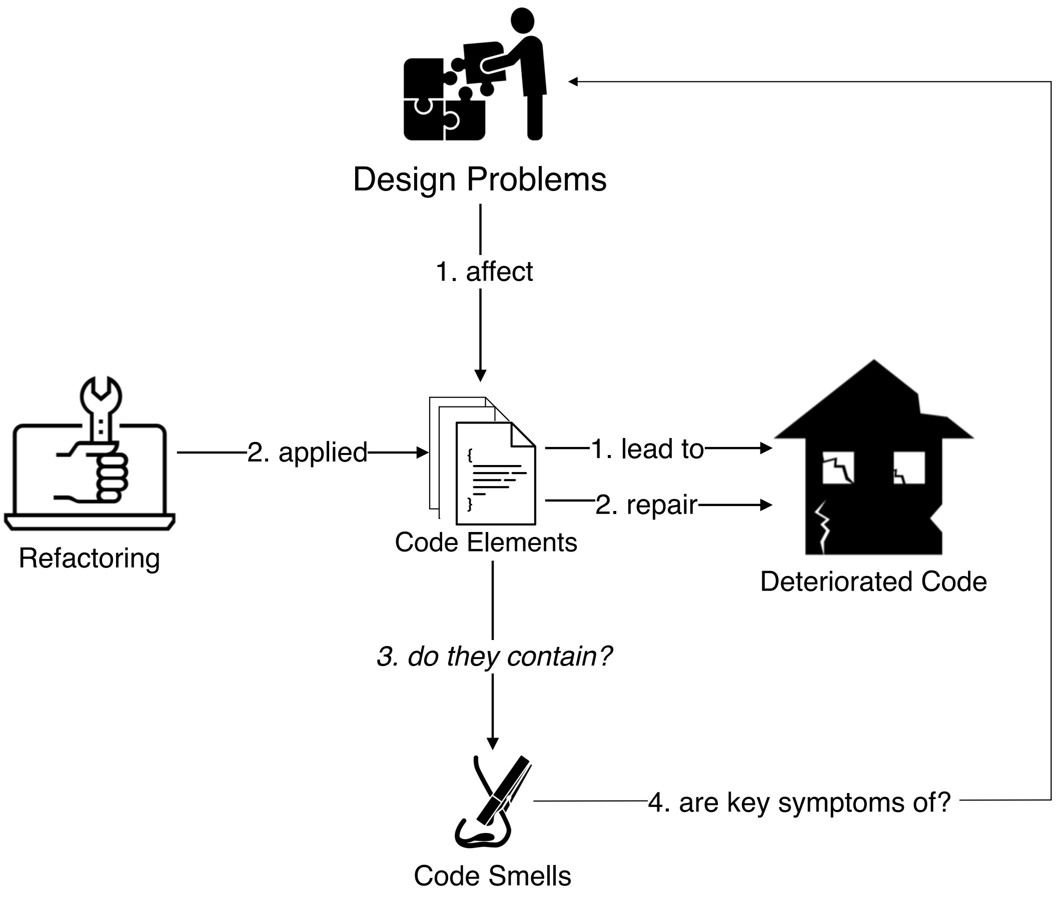


Figure 3.1: Relation between Design Problems and Refactoring

– **Smelly category**: This category comprises the refactored elements that are affected by at least one code smell.

According to these categories, we can have two possible results. First, most refactoring operations belong to the *smell-free category*. If we find this result, we can already conclude that code smells are not key symptoms to indicate relevant design problems. This conclusion is possible because we are assuming that relevant design problems are those that developers focused their effort to remove from the source code through refactoring. Therefore, if these relevant design problems affect refactored elements that do not have code smells, then code smells are not key symptoms for developers to identify relevant design problems. The second possible result is that most refactoring operations belong to the *smelly category*. In this case, there is a chance that code smells are key symptoms to indicate design problems in these elements. Nevertheless, before reaching any conclusion, we have to assure that the refactored elements had some sign of containing design problems.

We can assure that the elements had design problems based on three analyses: the number of code smells, the refactoring tactic, and the smell patterns. In the first one, we use the number of code smells as a sign of a design problem. The more smells a refactored element has, the greater the probability that it contains a design problem. Indeed, some studies have shown that elements with various code smells are likely to be related to a design problem (49, 52, 47, 46, 50). In fact, Oizumi *et al.* (47, 82) found that most of the code smells associated with a design problem were part of a group of interrelated code smells. They gave the name *agglomeration* for this group of interrelated smells (82). In this vein, we divided the *smelly category* into two subcategories according to the number of smells affecting the element: *single smell category* and *smell agglomerate category*. Inspired by Oizumi *et al.*'s studies, we used the same term, agglomeration, to name the category that indicates refactored elements that contain multiple code smells, *i.e.*, the *smell agglomerate category*. Both categories as described below.

- **Single smell category**: This category comprises the refactored elements affected by only ONE code smell.
- **Smell agglomerate category**: This category includes the refactored elements affected by more than one code smell.

In the second analysis to assure the presence of a design problem in the refactored element, we only investigate root-canal refactoring (25). Since the root-canal refactoring is explicitly used only to repair a deteriorated code, there is a high chance that developers target design problems when they apply this tactic. Nevertheless, that does not mean that developers do not target design problems when they apply floss refactoring. For instance, design problems have a negative impact on non-functional requirements such as changeability. Hence, the presence of a design problem can make it harder for a developer to make changes in the system, such as the addition of a new feature. Thus, he (or she) has to refactor the system to accommodate the changes. To better accommodate them, he (or she) has to target elements that may contain a design problem, which was the reason why the changes were hard in the first place. In other words, the code may contain a design problem that is hampering the changeability of the system. In this case, the developer has to target an element that may contain a design problem during floss refactoring.

In the third analysis, we rely on the smell patterns to assure that the refactored elements contain any sign of design problems. A smell pattern

Table 3.3: Smell Patters Used to Indicate Design Problems

Design Problem	Code Smells
Ambiguous Interface	Long Method and Feature Envy, and Dispersed Coupling in elements that are related to the interface
Cyclic Dependency	Intensive Coupling and Shotgun Surgery
Component Overload	Shotgun Surgery, and Divergent Change, and Feature Envy, and God Class, and Intensive Coupling, and Long Method
Concern Overload	Divergent Change, and Feature Envy, and God Class, and Intensive Coupling, and Long Method, and Shotgun Surgery
Fat Interface	Shotgun Surgery in the interface or Divergent Change, and Dispersed Coupling, and Feature Envy in elements related to the interface
Incomplete Abstraction	Lazy Class
Misplaced Concern	God Class or Dispersed Coupling, and Feature Envy, and Long Method
Scattered Concern	Dispersed Coupling, and Divergent Change, and Feature Envy, and God Class, and Intensive Coupling, and Shotgun Surgery
Unused Abstraction	Speculative Generality
Unwanted Dependency	Feature Envy, Long Method, Shotgun Surgery

represents one or more types of code smells that are likely to indicate a design problem if they appear in code elements. In Section 2.3, we discussed how some smell patterns are most likely to be related to some types of design problems. Based on some studies that have investigated the relation between code smells and design problems (74, 16, 75, 53, 38, 28, 47, 51, 76, 27, 55, 29), we characterized some patterns of code smells with some types of design problems. Hence, if these patterns appear in the refactored elements, there is a high chance that these elements to contain design problems. Table 3.3[1] shows the relation between each design problem (Column 1) that is most likely to be spotted by some smell patterns (Column 2).

Based on these smell patterns, we can have a sign that the refactored element was likely to contain a design problem. For instance, let us suppose that a developer applied the *Move Method* refactoring to move a method from one class to another. We can only assume that the refactored method was likely to have a design problem if the moved method had instances of `Dispersed Coupling`, `Feature Envy`, and `Long Method`. In this case, we assume that the developer applied the *Move Method* in an element with *Misplaced Concern* according to the smell pattern in the element.

We have to use the smell patterns because only relying on the number of code smells is not enough. There are some cases that the number of code smells can be misleading. For example, let us consider two elements. The first

[1]This table is a copy of Table 2.2 from Section 2.3.

one contains one instance of `Lazy Class` while the second one contains three instances of code smells: a `Message Chain`, a `Long Parameter List`, and a `Class Data Should Be Private`. According to the number of code smells, the second element is more likely to contain a design problem than the first one. Nevertheless, when we consider the type of each smell, the first element is most likely to contain a design problem, in this case, *Incomplete Abstraction*. Thus, in this example, the pattern `Lazy Class` was more appropriate to indicate the presence of a design problem than the number of code smells. The inverse case can also happen, which justifies the first analysis of the number of smells.

These analyses provide us with an approach to investigate if code smells are key symptoms for developers to identify relevant design problems. However, we highlight that this approach does not cover all scenarios. For instance, a design problem can affect elements that were not refactored. Since we look only at refactored elements, we can miss these elements. In this scenario, we cannot make any assumption that code smells are key symptoms to spot these design problems. We discuss this scenario and others when we present the results (Section 3.4) and threats to validity (Section 3.6).

3.3 Data Collection and Analysis

This section presents the data collection procedure to answer our research question. We structure the data collection procedure into four phases, which are described next.

3.3.1 Phase 1: Selection of Software Projects

The first phase consists of choosing the set of software projects to compose the sample of our study. For this purpose, we selected open source projects to be analyzed. We focused our analysis on open source projects so that our study could be replicated and extended. As GitHub is the world's largest open source community, we established GitHub to be the source of software projects selected for the study. We chose GitHub projects that match the following criteria:

– Projects with different popularity levels: projects that have been evaluated with different levels of popularity. As GitHub star is a metric to keep track of how popular an open source project is among GitHub users. We used the number of stars in each project to measure its popularity level;

- An active issue tracking system: users actively use the GitHub issue management system to maintain bug reports and improvement suggestions;
- Java projects: projects with at least 90% of the code repository effectively written in Java.

These criteria allowed us to select 50 software projects that are active and important to the software community. The selected projects are written in Java. We focused on Java projects because (i) Java is a very popular programming language[2], and Java projects were also targeted by related studies (85, 88). Furthermore, we also selected projects in Java due to the availability of tools to identify refactoring operations (89). As a result of applying these criteria, we selected Java projects with a diversity of structure, size and popularity. Table 3.4 presents these projects with their name, lines of code, number of classes, commits and stars for each selected project. The projects are organized according to their domain.

3.3.2 Phase 2: Refactoring Detection

The second phase consists of detecting refactoring operations in all subsequent pairs of versions v_i and v_{i+1} for all projects selected in Phase 1. Let $S = \{s_1, \cdots, s_n\}$ be a set of software projects. Each software s has a set of versions $V(s) = \{v_1, \cdots, v_m\}$. Each version v_i has a set of elements $E(v_i) = \{e_1, \cdots\}$ representing all methods, classes and fields belonging to it. Transformations between each subsequent pair of versions must be analyzed to detect refactoring operations. In this way, we assume R is a refactoring detection function where $R(v_i, v_{i+1}) = \{r_1(rt_1; e_1), \cdots, r_k(rt_k; e_k)\}$ gives us a set of tuples composed by two elements: the refactoring type (rt) and the set of refactored elements represented by e. So, function R returns the set of all refactoring operations detected in a pair of software versions. Thus, a tool that implements the R function can be used to detect refactoring operations in a software project.

We chose Refactoring Miner (89, 90) (version 0.2.0)[3] as the tool to detect refactoring operations. This tool implements a lightweight version of UMLDiff algorithm (91) for differencing object-oriented models. When the tool is applied between two versions, it returns the elements that changed from one version to the other. It also returns the refactoring type associated to the change. The precision of 98% reported by the authors (90) led to a very low rate of false positives, as confirmed in our validation phase. The only

[2]`https://www.tiobe.com/tiobe-index/`
[3]Available at `https://github.com/tsantalis/RefactoringMiner`

Table 3.4: Projects Used in the Study

Domain	Project	LOC	Number of Classes	Commits	Stars
Android	JARA	188,003	69	109	0
	Facebook Fresco	50,779	860	744	14,679
	OkHttp	49,739	642	2,645	27,421
	Google I/O Sched App	40,015	754	129	15,686
	Mayhem and Hell	25,043	304	148	1
	PhilJay MPAndroidChart	23,060	268	1,737	23,036
	WhatsUp (MarvinBellmann)	10,453	40	108	1
	Dagger	8,889	441	696	11,097
	Android Bootstrap	4,180	123	230	4,298
	LeakCanary	3,738	127	265	19,847
	Orhanobut Logger	887	11	68	9,423
Application	Containing	4,022,774	136	818	1
	Bublag Confetti	1,481,974	417	210	0
	Google J2ObjC	385,012	4,866	2,823	5,172
	ArgoUML	177,467	2,597	17,654	5
	Apache Ant	137,314	1,784	13,331	205
	Achilles	83,124	653	1,188	207
	Passsafe	12,203	196	150	0
	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	9,682
Framework	Spring Framework	555,727	12,715	12,974	22,052
	Ikasan	537,283	2,515	2,465	15
	Apache Dubbo	104,267	1,690	1,836	19,934
	Tap4j	34,026	123	146	16
	Alfred MPI	5,545	54	145	2
	JUnit4	2,113	1,251	309	6,935
Library	Elasticsearch	578,561	8,845	23,597	32,200
	PhiCode Philib	238,086	163	892	1
	Spring Boot	178,752	5,178	8,529	26,294
	JBoss Xerces	140,908	1,136	5,456	4
	Facebook SDK for Android	42,801	836	601	4,534
	Netflix Hystrix	42,399	1,569	1,847	14,172
	JBoss Ballroom	20,695	215	635	0
	Retrofit	12,723	554	1,349	26,557
	Drugis Common	12,195	254	240	6
	IRC Bot (c2nes/ircbot)	8,159	80	107	0
	Whydah - UserAdminService	7,454	60	249	0
	Pusher Java Client	7,029	74	352	174
	Lyra	6,603	95	192	245
	Dynamic Collections	5,955	215	180	6
	Elasticsearch Transport Thrift	4,450	48	113	80
Pluggin	Sen Word-Builder	736,148	65	120	0
	TUBAME Migration Tool	378,855	552	315	9
	GitHub Pull Request Builder	8,094	66	589	0
Web Application	Apache Tomcat	668,720	2,275	18,068	2,406
	Media Magpie	62,938	470	336	1
	Netflix SimianArmy	16,577	244	710	6,618
	OpenConext-crunche	4,574	31	108	0

drawback of this tool is the number of refactoring types detected: 13 types (Section 3.1). Fortunately, these 13 types were amongst the ones reported as the most common refactoring types (25). Refactoring Miner gives us as output a list of refactoring operations $R(v_i, v_{i+1}) = \{r_1, \cdots, r_k\}$, where k is the total number of refactoring operations identified. Each r_i is a tuple containing the refactoring type rt_i and the refactored element e_i.

3.3.3 Phase 3: Code Smell Detection

In order to verify if code smells are key symptoms for developers to identify relevant design problems, we need to detect if the refactored elements contain code smells, which is the third phase of our data collection procedure. Several strategies have been proposed for detecting code smells. For instance, there are strategies based on metrics (92, 28), based on the source code evolution information (93), based on machine learning methods (94, 48), and based on optimization algorithms such as genetic algorithms (95).

For this study, we have decided to detect code smells with metric-based strategies (92, 30). Thus, we can compare our results with related studies that have used the same detection strategies (85, 88). These strategies are based on a set of metrics and thresholds, in which the metric values are compared against predefined thresholds and combined using logical operators. For this purpose, we collected metric (values) for all source files in each selected project from Phase 1. Next, we collected the code smells by applying a set of previously defined detection rules (85, 30) used in related studies (85, 88). Unfortunately, the detection rules were not implemented by any publicly available tool. Thus, we developed a tool[4] that implements all detection rules for the 17 code smells used in our study (Table 2.1). The specific metrics, rules and thresholds for the 17 code smells implemented in our tool are defined in (85) and (30),

3.3.4 Phase 4: Manual Validation

The last phase comprises the validation of the refactoring operations. We used a tool in the second phase to detect refactoring operations (90). Even though this tool has been reported to achieve a precision of 98%, we were not sure if the tool would achieve the same precision in our set of selected software systems. Since we use refactoring to investigate code smells, our results depend on the refactoring operations detected in each system. Thus, the tool could find false positives and false negatives that could compromise our analysis.

[4]The tool is available at `https://github.com/opus-research/organic`.

Therefore, we noticed the need to validate the refactoring operations found by the tool before conducting the study. To validate the refactoring operations, we conducted two manual inspections of the refactoring operations returned by the tool.

The first inspection was to validate each one of the 13 types of refactoring operations that we use in our study (Section 3.1). For this inspection, we randomly sampled refactoring operations of each type. We decided to sample refactoring for each type since the precision of the Refactoring Miner could vary from one type to another. Such variation is due to the rules implemented in the tool to detect each refactoring type. To deliver an acceptable confidence level to the results, we calculated the sample size of each refactoring type based on a confidence level of 95% and a confidence interval of 5 points. For this inspection, we recruited ten students to validate the samples manually. The manual inspection started by presenting to the students a pair of versions of elements marked as refactored by Refactoring Miner. For each pair of elements, the student had to mark it as a valid refactoring or not. In this way, we were able to estimate the number of false positives generated by Refactoring Miner. We highlight that our goal was to ensure the trustability of the tool for our set of software systems. This is the reason why we relied on the students, who were familiar with refactoring, to validate the tool.

In general, we observed a high precision for each refactoring type, with a median of 88.36%. The precision found in all refactoring types is within one standard deviation (7.73). Applying the Grubb outlier test (alpha=0.05), we could not find any outlier, indicating that no refactoring type is strongly influencing the median precision found. Thus, the results found in the representative sample represents a key factor to provide trustability to the other results reported in this study.

The second manual inspection was to validate the refactoring tactics (Section 3.1). As discussed in Section 3.2.2, we rely on three aspects to assure that the refactored code elements were likely to contain design problems. An aspect is the refactoring tactic. Therefore, to investigate this aspect, we need to identify when developers applied root-canal refactoring and floss refactoring. To conduct this inspection, first we selected a sample containing 4,991 refactoring operations. Then, we used Eclipse and the eGit plugin[5] to classify a refactoring as root-canal refactoring or floss refactoring. Finally, we used a diff tool to analyze all the changes in the classes modified by the refactoring operations. When a behavioral change was identified, we filled a form explaining it, and we classified the change as floss refactoring. When we

[5]`http://www.eclipse.org/egit/`

did not find a behavioral change, we classified it as root canal refactoring. At the end of this inspection, we classified 4,991 refactoring operations into root-canal and floss refactoring. For this manual inspection, we did not rely on students; instead, we relied on the expertise of three researchers to conduct the inspection. We decided to conduct this inspection among us for a couple of reasons. First, we needed people who had experience with refactoring before. To determine that there was a behavior change due to a refactoring operation, one needs to have knowledge that students did not have. Second, this second inspection requires more effort and time than to validate the refactoring types.

3.3.5 Algorithm for Categorization

To answer our research question properly, we have to assure that the refactored elements were likely to contain design problems. For such assurance, we have to categorize the refactored elements into three categories: *smell-free*, *single smell* and *smell agglomerate* categories (Section 3.2.2). Algorithm 1 presents the method we used to categorize the refactored elements.

Algorithm 1 Categorization the Refactored Elements

```
Require: Set of projects P, versions V, elements E, refactoring operations R
Ensure: The categorization of the refactored elements
 1: $P \leftarrow \{p_1, p_2, \cdots, p_n\}$ {Phase 1}
 2: for all $p \in P$ do
 3:     for all $v_i \in V(p)$ do
 4:         if $v_{i+1} \in V(p)$ then
 5:             Collect all refactoring operations $R(v_i, v_{i+1})$ {Phase 2}
 6:             Collect smells for all $e \in E(v_i)$ {Phase 3}
 7:             for all $r \in R(v_i, v_{i+1})$ do
 8:                 smellCategory = categorizeAccordingToSmellPresence(r.e)
 9:                 r.targets += (smellCategory)
10:             end for
11:             saveResultDB($R(v_i, v_{i+1})$)
12:         end if
13:     end for
14: end for
```

The algorithm receives as input a set of projects, chosen in Phase 1 (Line 1). Then, it iterates over each project (Line 2), in which it retrieves two versions of the project: the i version and the subsequent one (Lines 3 and 4). The algorithm calls the Refactoring Miner tool (Line 5) to collect the refactoring operations, which comprises the first phase of the data collection process. Similarly, the algorithm calls our code smell detection tool to collect all the code smells for all the elements in the i version (Line 6). This routine comprises the third phase of the data collection process. Once the refactoring operations, smells and structural attributes are available, the next step in the algorithm is to categorize each refactoring element according to the number of code smells.

The algorithm iterates over each detected refactoring to categorize the respective refactored elements (Line 7). Thus, the algorithm calls a function to categorize the refactored elements according to the number of code smells affecting the element (Line 8). The function in Line 8 returns one of the categories defined in Section 3.2.2: *smell-free*, *single smell* or *smell agglomerate*. The categorized element is associated to the refactoring (Line 9); thus, for each refactoring the algorithm associates all the refactored elements to one category. Finally, the algorithm saves the results in the database (Line 11).

3.4 Analysis of the Results

We discuss in this section the results that we found when investigating if code smells are key symptoms to indicate design problems. In Section 3.4.1, we present the frequency with which refactoring operations are applied to code elements. In Section 3.4.2, we investigated the refactored elements according to their probability of containing design problems. Finally, we present in Section 3.4.3 how previous findings reinforce the answer to our research question.

3.4.1 Frequency of Refactoring Operations on Code Elements

To answer our RQ (*"Are code smells key symptoms to indicate relevant design problems for developers?"*), we investigated if the refactored elements contain code smells. For this investigation, we considered a broad range of refactoring and smell types, which led us to analyze 13 different refactoring types (Section 3.1) and 17 different types of code smells (Section 2.3). After applying the refactoring detection procedure (Section 3.3.2), we identified 51,227 refactoring operations. These refactoring operations had been applied to 52,667 elements. Table 3.5 presents the frequency with which refactoring operations are applied to elements. For this result, we only considered if the refactored element contains or not code smells.

Table 3.5: Frequency of Refactoring Operations

	Total	**Smell-free Category**	**Smelly Category**
Refactoring operations	51,227	10,512 (20.52%)	40,715 (79.48%)

According to our data, most refactoring operations are applied to smelly elements. From the 51,227 refactoring operations, 40,715 (79.48%) were applied to elements with code smells, while 10,512 refactoring operations (20.52%) were applied to elements without code smells. One could argue that these refactoring operations were applied to smelly elements because most of them contain code smells. In other words, the refactored elements contain code smells because their software system has a high rate of smells, thereby increasing the probability that refactoring operations are applied to smelly elements. In order to verify if most code elements contain smells, we computed the probability of randomly choosing a smelly element in our dataset ($|smelly\ elements|/|all\ elements|$), which is 0.3%. This low probability shows that, in our dataset, refactoring operations are not applied to smelly elements by coincidence. Refactoring operations indeed tend to concentrate on smelly elements, which were confined to a vast minority of the program elements. This behavior was consistently observed for both root-canal and floss refactoring.

As previously discussed, if most of refactoring operations belong to the *smelly category*, then there is a chance that code smells are key symptoms to indicate design problems in these elements. This was exactly the result that we achieved according to our data (Table 3.5). Therefore, this result is the first shred of evidence that code smells can be key symptoms to indicate relevant design problems in most cases. Thus, our next step is to conduct the analyses regarding the probability that these refactored elements have design problems. Before digging into these analyses, we need to address the 10,512 refactoring operations applied to elements without code smells.

Indeed, 20.52% of refactoring operations is a number that we cannot ignore. This number represents 10,807 elements that were refactored but did not have any code smell. According to our assumption, relevant design problems might exist in these elements, since they were refactored. Unfortunately, in this scenario, code smells are not key symptoms to identify design problems in these elements. Therefore, in practice, this result indicates that developers cannot rely entirely on code smells to identify some instances of design problems. Consequently, developers should rely on other indicators of design problems. We will discuss more about this scenario later (Section 3.4.3).

3.4.2
Investigating the Chance of Elements Containing Design Problems

According to our results, most refactored elements contain code smells. Thus, there is a chance that code smells are key symptoms to identify design problems in these elements. However, we need to investigate if these refactored elements contain any sign of design problems. For this purpose, we conducted three analyses to assure that the refactored elements were likely to contain design problems: the number of code smells, the refactoring tactic, and the smell patterns. We discuss the results for each one as follows.

3.4.2.1
Frequency of Refactoring Operations in Smelly Elements

As presented in Section 3.2.2, we have to assure that the refactored elements had some sign of containing design problems. The number of code smells affecting a refactored element comprises the first analysis that we conducted towards achieving such assurance. Thus, we have divided the *smelly category* into two other subcategories based on the number of code smells in the refactored element: *single smell category* and *smell agglomerate category.* This categorization allows us to increase the probability that refactored elements contain design problems, since the more smells a refactored element has, the greater the probability that the element contains a design problem (49, 52, 47, 46, 50). Table 3.6 details the frequency of refactoring operations applied to smelly elements according to these subcategories.

Table 3.6: Frequency of Refactoring Operations on Smelly Elements

	Smelly Category		
	Total	**Single Smell**	**Smell Agglomerate**
Refactoring operations	40,715 (79.48%)	16,443 (32.10%)	24,272 (47.38%)

From 40,715 refactoring operations, 47.38% were applied to elements with more than one smell, while 32.10% to elements with only one smell. 47.38% of refactoring operations were applied to approximately 24,953 code elements. Thus, according to our categorization, 47.38% of the refactored elements have a high chance of containing a design problem. Therefore, in most cases, code smells are key symptoms for developers to identify relevant design problems. On the other hand, 32.10% of the refactored elements (16,906 elements) contain a single code smell. Thus, these elements are less likely to contain design problems according to the number of code smells. Nevertheless, there are some types of design problems that can be identified by a single code smell, such as *Incomplete Abstraction*. Hence, we cannot assume that these elements do not have design problems without considering other analyses to assure the presence of design problems.

3.4.2.2 Frequency of Refactoring Operations Classified as Root-canal Tactic

Analysis of the root-canal refactoring is another way to assure that the refactored elements are likely to contain design problems. Root-canal refactoring consists of a process of exclusively applying refactoring operations to repair deteriorated code, which can be deteriorated due to the presence of design problems. Since root-canal refactoring is applied to repairing deteriorated code, elements refactored when developers applied this tactic are the most likely to contain design problems. To find these elements, we carried out a manual validation to classify 4,991 refactoring operations according to the applied tactic: either root-canal or floss refactoring. Table 3.7 presents the frequency of those refactoring operations classified as root-canal refactoring.

Table 3.7: Frequency of Root-canal Refactoring

Tactic	Operations	Smell-free Category	Smelly Category		
			Total	Single Smell	Smell agglomerate
Root-Canal Refactoring	1,168	134 (11.47%)	1,034 (88.53%)	248 (21.23%)	786 (67.29%)

The results in Table 3.7 show that most refactoring operations are applied to smelly elements when we consider only the root-canal refactoring. In fact, this result is expected to a certain degree. As root-canal refactoring is a tactic usually applied to repair deteriorated code, then the code should show signs of deterioration. In this case, the sign of deterioration is the code smells. Code smells are symptoms of design problems, which in their turn are one of the reasons why elements can reach a deteriorated state (15, 16). Consequently, this result indicates that indeed the developers focused on refactoring elements in a deteriorated state. Indeed, the percentage of refactoring operations applied to elements without smells, *i.e.*, in the *smell-free category*, (11.47%), is lower than the overall percentage shown in Table 3.5 (20.52%).

Most refactoring operations were applied to elements with code smells. As they happened during root-canal refactoring, there is an increase in the probability that these elements contain design problems. When we consider only the *smelly category*, we notice that most refactoring operations were applied to elements with multiple code smells (67.29%). In fact, this number is much higher than the overall refactoring operations applied to the *smell agglomerate category* (47.38%) shown in Table 3.5. This result reinforces that developers were indeed focusing on deteriorated elements as one might expect to happen during root-canal refactoring. Furthermore, this result indicates that, in most cases, code smells are key symptoms for developers to identify design problems, at least when we take into account root-canal refactoring. In summary, when developers focus their effort on repairing deteriorated code, most refactoring operations are applied to elements that are likely to contain design problems. Since these elements contain multiple code smells, we can assume that, in the context of root-canal refactoring, code smells are key symptoms for developers to identify design problems.

As we conducted the manual validation, we also have the frequency of refactoring operations when developers applied floss refactoring. Developers apply floss refactoring when they want to achieve a particular purpose, such as adding a new feature or fixing a bug. One can argue that the results presented in previous sections can be different when we consider only one tactic of refactoring. For instance, root-canal refactoring occurs when developers want to repair deteriorated code. Thus, if most of the refactoring operations are root-canal ones, then it is justified that most refactoring operations target elements with some sign of design problems. On the other hand, the number of refactored elements when developers apply floss refactoring can be different since the goal is not to repair a deteriorated code. Consequently, the frequency of refactoring operations in each category can be different when we take into account only floss refactoring. Table 3.8 presents the frequency for those refactoring operations classified as floss refactoring.

Table 3.8: Frequency of Floss Refactoring

Tactic	**Operations**	**Smell-free Category**	**Smelly Category**		
			Total	**Single Smell**	**Smell agglomerate**
Floss Refactoring	3,817	509 (13.34%)	3,308 (86.66%)	780 (20.43%)	2,528 (66.23%)

Surprisingly enough, the results for floss refactoring (Table 3.8) are very similar to those found for root-canal refactoring. Most refactoring operations were also applied to smelly elements during floss refactoring. In fact, when we analyze the *smelly category*, most refactoring operations are applied to elements with more than one code smell. This similar result is surprising, since floss refactoring has a different purpose from root-canal refactoring. Thus, repairing a deteriorated code does not play an essential factor when developers choose the element to refactor during floss refactoring. Nevertheless, that does not mean that the code elements do not contain design problems. They may have design problems, at least if we take into account the number of code smells. Unfortunately we cannot have the same guarantee as we had when we considered the root-canal refactoring. Hence, we can only rely on the number of code smells as a sign of the presence of design problems.

This comparison between refactoring tactics is also useful to mitigate bias related to one tactic overshadows the other one in the overall result (Table 3.5). Thus, comparing the results from both tactics (Tables 3.7 and 3.8), developers refactor elements with multiple code smells regardless of whether they apply root-canal refactoring (67,29%) or floss refactoring (66.23%). Therefore, assuming that the number of code smells is a sign of a design problem, this similar result between root-canal and floss refactoring can be faced as another evidence that smells may be a key symptom to indicate design problems.

In the same way that one tactic could overshadow the other one, there is a chance of a refactoring type having a strong influence on the results. Thus, we need to investigate each refactoring type to verify whether developers tend to focus on *smelly elements* independently of the refactoring type or not. Table 3.9 presents the frequency of each refactoring type applied to *smelly elements*. The table also presents the frequency of root-canal refactoring and floss refactoring. We can observe that, similar to previous results, developers tend to apply most refactoring operations to elements with multiple code smells (*smell agglomerate category*): 66.23% of floss refactoring and 67.30% of root-canal refactoring. In other words, independently of type, most refactoring operations are applied to elements that present a sign of design problem, which can be identified by code smells.

Indeed, most refactoring types are applied to elements with multiple code smells, *i.e.*, in elements with a sign of design problem. Only two refactoring types did not follow this distribution (light gray rows). On the one hand, the *Inline Method* was frequently applied to smell-free elements (65.25%) when developers performed floss refactoring. On the other hand, *Move Class* was frequently applied to smell-free elements when developers performed root-canal refactoring (51.66%). Interestingly enough, none of these two refactoring types were applied to smell-free elements simultaneously in floss refactoring and root-canal refactoring. This result can be another piece of evidence that code smells can be key symptoms for developers to identify design problems, at least in most cases.

On the other hand, this result for *Inline Method* and *Move Class* also indicates that, in some scenarios, code smells are not key symptoms for

Table 3.9: Frequency of Each Tactic by Refactoring Types

Refactoring Type	Floss Refactoring				Root-canal Refactoring			
	Total	Smell-free	Single Smell	Smell Agglomerate	Total	Smell-free	Single Smell	Smell Agglomerate
Extract Interface	32	5 (15.63%)	8 (25.00%)	19 (59.38%)	27	2 (7.41%)	14 (51.85%)	11 (40.74%)
Extract Method	1,137	40 (3.52%)	115 (10.11%)	982 (86.37%)	345	13 (3.77%)	55 (15.94%)	277 (80.29%)
Extract Superclass	213	6 (2.82%)	130 (61.03%)	77 (36.15%)	37	3 (8.11%)	23 (62.16%)	11 (29.73%)
Inline Method	469	306 (65.25%)	27 (5.76%)	136 (29.00%)	38	6 (15.79%)	5 (13.16%)	27 (71.05%)
Move Attribute	511	8 (1.57%)	223 (43.64%)	280 (54.79%)	30	3 (10.00%)	10 (33.33%)	17 (56.67%)
Move Class	48	10 (20.83%)	20 (41.67%)	18 (37.50%)	151	78 (51.66%)	26 (17.22%)	47 (31.13%)
Move Method	393	7 (1.78%)	111 (28.24%)	275 (69.97%)	107	3 (2.80%)	23 (21.50%)	81 (75.70%)
Pull Up Attribute	201	12 (5.97%)	33 (16.42%)	156 (77.61%)	136	4 (2.94%)	34 (25.00 %)	98 (72.06%)
Pull Up Method	246	2 (0.81%)	43 (17.48%)	201 (81.71%)	197	6 (3.05%)	6 (3.05%)	185 (93.91%)
Push Down Attribute	18	0 (0.00%)	3 (16.67%)	14 (77.78%)	11	1 (9.09%)	4 (36.36%)	6 (54.55%)
Push Down Method	42	1 (2.38%)	5 (11.90%)	39 (92.86%)	9	0 (0.00%)	4 (4.00%)	5 (55.56%)
Rename Class	99	24 (24.24%)	18 (18.18%)	57 (57.58%)	2	0 (0.00%)	1 (50.00%)	1 (50.00%)
Rename Method	408	88 (21.57%)	46 (11.27%)	274 (67.16%)	78	15 (19.23%)	43 (55.13%)	20 (25.64%)
Total	**3,817**	**509 (13.34%)**	**780 (20.43%)**	**2,528 (66.23%)**	**1,168**	**134 (11.47%)**	**248 (21.23%)**	**786 (67.30%)**

developers to identify design problems. We consider that these smell-free elements are likely to have design problems since they were refactored; thus in practice, this result indicates that some design problems cannot be identified by code smells. Consequently, in practice, code smells may not suffice to help developers to identify design problems.

3.4.2.3 Frequency of Refactoring Operations Applied to the Smell Patterns

In the previous subsections, we conducted two analyses to assess the probability that refactored elements contain design problems. There is a third analysis: smell patterns. As discussed in Section 3.2.2, if some types of code smells appear in code elements, there is a high chance that these elements contain design problems. Based on these occurrences reported in the literature (74, 16, 75, 53, 38, 28, 47, 51, 76, 27, 55, 29), we presented a list of recurrent smell patterns that increase the probability that a code element contains design problems (Table 3.3). Table 3.10 presents the design problems that are likely to be found in refactored elements according to these smell patterns.

Results in Table 3.10 shed lights about the role of codes smells in supporting the design problem identification. As discussed in Section 2.3, there are some design problems that can be identified by only a single smell, such as *Incomplete Abstraction* and *Unused Abstraction*, while there are other design problems that are most suitable to be identified by multiple code smells, such

Table 3.10: Design Problems Found in Refactored Elements

Design Problems	Single Smell	Multiple Code Smells
Ambiguous Interface	N/A	0
Ciclic Dependency	N/A	31
Concern Overload	490	514
Fat Interface	0	2
Incomplete Abstraction	12503	N/A
Scattered Concern	N/A	26
Unused Abstraction	2317	N/A
Unwanted Dependency	N/A	11
N/A = Not Applicable		

as *Ambiguous Interface*, *Cyclic Dependency*, *Scattered Concern* and *Unwanted Dependency*. Finally, there are design problems that can be identified by a single or multiple smells, such as *Concern Overload* and Fat *Interface*. These two last design problems provide us with some interesting discussions about the code smells.

In total, we found 1,004 classes that could have *Concern Overload* according to the smell patterns. This design problem can be identified with the pattern `God Class` (490 instances) or with multiple code smells: `Divergent Change`, `Feature Envy`, `God Class`, `Intensive Coupling`, `Long Method`, and `Shotgun Surgery` (514 instances). We randomly sampled 50 classes with only the `God Class` pattern and 50 classes that had the pattern with multiple code smells. After a manual validation of these classes, we noticed that the pattern with multiple smells is most likely to indicate a design problem. From the classes that had only `God Class`, we validated 26% of them as having the *Concern Overload* (13 classes). On the other hand, from the classes that had the pattern with multiple code smells, 64% were validated as having the design problem (32 classes). This result indicates that indeed elements with various code smells are likely to be related to a design problem (49, 52, 47, 46, 50). This result is particularly important to increase the reliability of our analyses, since we use the number of code smells to assure that refactored elements were likely to contain design problems (Section 3.2.2).

Fat Interface is another design problem that can be identified by one or by multiple code smells. However, we did not find any interface in the system that had the pattern with only one code smell (`Shotgun Surgery`), *i.e.*, the smell pattern that could indicate the design problem with only one smell. On the other hand, we found 2 instances of *Fat Interface* when we searched for elements that had the smell pattern used to identify this design problem with multiple code smells (`Divergent Change`, `Dispersed Coupling`, and `Feature`

`Envy`). Even though there were only two instances of *Fat Interface*, these are two more cases which show that the pattern with multiple smells is more likely to indicate a design problem than the pattern with only a single smell. Consequently, this result also strengthens our assumption about the number of code smells be suitable to assess the probability that elements contain design problems.

Another interesting result is the one regarding *Ambiguous Interface*. According to the smell patterns, to identify this design problem, a developer needs to find code elements that use the interface and contain `Long Method`, `Feature Envy`, and `Dispersed Coupling`. We did not find elements that are connected to the interface and contain this smell pattern. However, when we consider only `Long Method` and `Feature Envy`, then the number increases to 130 possible instances of design problems. This result is interesting to show that these elements were presenting the first signs of a design problem. Even if the developers' intention was not to remove the design problem, this result indicates that they refactored elements that had a sign of a design problem, and these elements had code smells that could indicate a (potential) design problem.

These results about *Concern Overload*, *Fat Interface*, and *Ambiguous Interface* are also useful to discuss if a single code smell is enough to identify design problems. According to our validation of these 102 instances of design problems, relying on a single code smell may not be enough to identify design problems in practice. In fact, our data suggest that if developers rely on smell patterns with multiple code smells, they have a higher chance of identifying design problems. However, there are some pros and cons in using these patterns. On the one hand, the analysis of various smells in some of these patterns may increase the developers' confidence about the presence of design problems. On the other hand, to reason about multiple code smells simultaneously may be a hard task to carry on. Nevertheless, the investigation of developers using multiple code smells is one that we need to conduct to understand if code smells suffice in helping developers to identify design problems.

Additionally, we highlight that developers cannot identify a design problem if they expect to find all the instances of the smells within a pattern. For example, a developer would not identify any *Ambiguous Interface* in our dataset if he was expecting to find instances of `Long Method`, `Feature Envy`, and `Dispersed Coupling`. As we mentioned, we did not find elements connected to the interface with all the smells in this pattern. In other words, developers cannot expect to find all the smells of a pattern to identify a design problem, especially if the affected elements are presenting the first signs of a design problem.

3.4.3 Code Smells are Key Symptoms in Some Scenarios

Results found in previous subsections can be used to reinforce the answer to our research question. We found that most refactoring operations are applied to smelly elements. In fact, developers tend to refactor elements that contain multiple code smells. When we analyze the smelly elements, we noticed that they present signs of design problems in most cases. For instance, most refactored elements contain multiple code smells. Not only are these multiple code smells signs that the elements contain design problems, but they also evidence that they can be used as key symptoms to identify design problems. This result is even more evident when we consider only the context of root-canal refactoring. In this context, most refactoring operations are applied to elements that have at least one code smell. Surprisingly enough, this same result is obtained when we analyze only the floss refactoring. This result is surprising since we were not expecting that developers focused on (floss) refactoring elements that were deteriorated. Consequently, we can view this result as another piece of evidence that code smells are key symptoms for developers to identify design problems in the context of refactoring.

Unfortunately, we found some scenarios where code smells are not key symptoms. Obviously, code smells are not key symptoms to indicate design problems in the smell-free elements. However, these elements can be the ones that also have relevant design problems. In these cases, developers cannot rely on code smells to identify design problems, which makes us wonder about other symptoms that developers can use to identify design problems in these elements. There are also design problems that manifest in elements that were not the focus of refactoring. Since we are only investigating refactored code, we missed these elements. In these cases, we cannot say anything about code smells being key symptoms to identify design problems. We even found some cases where most refactoring operations were applied to elements without code smells, *e.g.*, when we analyzed only the *Inline Method* and *Move Class* refactoring types. As we consider that these elements are likely to have design problems due to the refactoring, code smells cannot be used to indicate design problems in these elements. In practice, this result indicates that developers cannot rely on code smells to identify some design problems. These cases where the refactored elements did not have code smells also make us wonder about to what extent code smells suffice to help developers to identify design problems in practice. In summary, we can conclude that in most cases, code smells likely represent key symptoms to indicate design problems, at least when we consider the refactored code. On the other hand, in practice, code smells may not suffice to help developers to identify relevant design problems. This is an investigation that we need to conduct to understand if code smells suffice to help developers during design problem identification.

As mentioned, we conducted a retrospective study to investigate if in practice code smells are key symptoms to identify design problems. Unfortunately, our study has some limitations. To a certain degree, our results are grounded in practice, since we are retrospectively looking at snapshots of what has actually happened. However, we have not **strictly** investigated developers' behavior when using code smells to identify design problems in practice. Even though we found that in most cases code smells are key symptoms for developers to identify relevant design problems, we still do not know if code smells suffice to help developers. In fact, we have few pieces of evidence that, in practice, code smells do not suffice to indicate some design problems. For instance, there are code elements that do not have code smells. Even so, these elements were likely to contain design problems. Furthermore, our study was limited in the sense of focusing on refactored code. Hence, we cannot say if code smells are key symptoms in code elements that were not refactored. Even though this study shed light on the role that code smells play during design problem identification, we sill need to continue our investigation about code smells. A follow-up investigation is observe developers using code smells in practice. Consequently, we will be able to state if code smells suffice or not to help developers to identify design problems.

3.5 Related Work

We present in this section some studies related to refactoring, code smells and design problems.

3.5.1 Code Smells as Key Symptoms

Some studies have investigated the developers' perception of code smells (51, 29). These studies are closely related to ours. For instance, Yamashita and Moonen conducted an exploratory survey with developers about their knowledge and concern with code smells (29). In this study, the authors investigated though developers' perspective if code smells should be considered meaningful conceptualization of design problems. For this investigation, they applied a survey with 73 software developers. The survey included questions about developers' level of knowledge on code smells, their perception on the criticality of code smells, and their perception on how useful code smell information is for different software engineering tasks. Based on these questions, the authors were able to identify the code smells that developers perceived as critical, why they are critical, and what features a code smell detection tool should have. The results indicated that only 18% of respondents (13 developers) had a good understanding of code smells. However, the majority of the developers (19 out of 50 developers who finished the survey) are concerned to code smells in their code, while 14% (7 developers) were extremely concerned. `Duplicated Code`, `God Class`, and `Long Methods` were the smells that developers perceived as critical.

Palomba *et al.* reported a similar study, which aimed to analyze to what extent code smells are perceived as design problems (51). The authors validated 12 different code smells in three open source projects. Next, they showed developers code snippets affected and not affected by these 12 code smells. Then, they asked the developers if they considered that the code snippets had design problems. If so, they asked developers to explain what design problems they perceived. They reported that most code smells are, in general, not perceived by developers as design problems. However, there are some code smells (`Complex Class`, `God Class`, `Long Method`, and `Spaghetti Code`) that developers immediately perceived as design problems.

As mentioned, these studies are related to ours. Thus, one could expect to use them to answer our research question. Unfortunately, that is not possible since we have different goals and research methods. Our goal is to investigate if code smells are key symptoms to identify design problems. Yamashita and Moonen focused on investigating to what extent developers had a theoretical knowledge of code smells; while Palomba *et al.* investigated if developers perceive code smells as design problems. In summary, these studies provide general evidence that only in specific circumstances developers perceive code smells as critical structures in the system. Hence, the results in these studies may suggest that code smells may not suffice to help developers in identifying design problems in practice.

Unfortunately, these studies have limitations that prevent us to use their results in our context, *i.e.*, to state that code smells are key symptoms for developers to identify design problems in practice. The first limitation regards the research method. These previous studies rely on surveys to get evidence about the relevance of code smells for developers. However, as explained in (57), some issues appear when using questions to gather information. For instance, the questions in a survey may not be designed in a way that yields useful and valid data. Another issue is to phrase the questions insomuch that all participants understand them in the same way, especially when the target population is diverse. The second limitation of these studies is that their results have not been grounded in practice. Unfortunately, "it is possible that what people say they do in response to survey questions bears no relationship to what they actually do because they are unable to introspect reliably on their work practices" (57). In other words, we cannot rely on only surveys if the goal is to investigate a phenomenon in practice.

3.5.2 Applying Refactoring Operations

Our data showed that most refactoring operations (79.48%) are applied to smelly elements. Even though the original motivation of developers is unknown, we actually observed a similar behavior when they applied both root-canal and floss refactoring. Mainly in the former case, one would expect developers to explicitly intend to improve code structure by removing design problems. Regarding the developers' motivation, Silva, Tsantalis and Valente (87) investigated the reasons why developers refactor their code. Their results indicate that refactoring operations are mainly driven by fixing a bug or changing the requirements, such as adding a new feature. Their results show that the refactored code may contain code smells, although developers did not mention it explicitly as their intention to refactor. For instance, developers said that they apply the *Move Class* refactoring when they want to move a class to a package that is more functionally or conceptually relevant to that particular class. Even though developers do not mention code smells in this description, it is possible that the moved classes contain smells such as `Feature Envy` and `Intensive Coupling`.

Based on the assumption that refactoring operations are applied to elements with design problems, our study investigated whether the refactored elements contain or not code smells. For that, we investigated a large set of software systems. This allowed us to analyze refactorings in systems of various domains and sizes (LOC). Also, we have analyzed 17 types of code smells considered relevant in the literature. Our results indicate that most refactored elements (79.48%) contained at least one code smell. When we analyzed only the refactored code elements that contain code smells, we noticed that 47.38%% of them contain more than one code smell (smell agglomerate). Even when developers applied only floss refactoring, *i.e.* when their goal was not to repair a deteriorated code, they applied most refactoring operations to smelly elements (86.66%). These results suggest that developers are also motivated (at least implicitly) by design problems. Developers might not be applying refactoring to remove design problems; nonetheless, they are still applying refactoring to elements that present signs of design problems.

Bavota *et al.* (85) investigated whether refactoring operations occur on code elements where certain indicators suggest that might be a need for refactoring. Their indicators include structural quality metrics and the presence of code smells. According to their results, quality metrics do not show a clear relationship with refactoring, and 42% of the refactorings are applied to smelly elements. Different from Bavota *et al.* (85), we investigated a large set of software systems, and we also performed the validation and classification of a subset of refactorings. Also, we have considered all the 11 types of code smells used by Bavota plus 6 other types of code smell deemed relevant in the literature. Thus, our data sample is much larger than the sample analyzed by Bavota *et al.* (85). Surprisingly, our results indicate that most refactorings are applied to smelly elements.

Cedrim *et al.* (88) also investigated the frequency with which refactoring operations are applied to smelly elements. They also found that 79.4% of refactoring operations are applied to smelly elements. We found the same percentage of refactoring operations on smelly elements, which is interesting. We followed the same data collection procedure, we used the same tools, and we considered the same 23 systems that Cedrim *et al.*. Therefore, we were expecting to achieve similar results but, instead, we found the same result, practically. What makes this similarity interesting is that we have evolved the study of Cedrim *et al.*. For instance, we added 27 other systems, four other types of code smells, and we also added two other refactoring types. Despite these differences, both studies found the same percentage of refactoring operations applied to smelly elements. Consequently, the same result in two studies can indicate the reliability of the result.

3.6 Threats to Validity

We discuss in this section the study limitations based on the threats to validity. We also present the measures that we took to mitigate these threats.

Internal Validity The data collection using the Refactoring Miner tool represents a threat to internal validity because it may find some false-positives and false-negatives. To minimize this threat and check the precision of the tool, we randomly selected samples of each refactoring type and manually validated them. In this sense, we could improve confidence regarding the Refactoring Miner precision.

We could not reach the developers to ask their intentions (root canal or floss) in all refactorings detected. Therefore, we included the validation whether the refactoring is root or floss in our manual task, which is another threat to internal validity. Notice that such analysis is limited to two versions of the source code directly related to the refactoring, not considering all versions in the repository. Moreover, the manual analysis only considers behavior preservation in the refactored elements.

Code smells are fundamentally important for this study since our goal is to investigate their relevance for developer in practice. We did not validate the detected code smells, which is a threat. Thus, the results are sensitive to code smell detection rules. As mentioned before, such rules are based on thresholds. The risk is that different thresholds can lead to results completely distinct. Therefore, the detection rules and the choices of thresholds can pose a threat to this study. To mitigate such risk, we used thresholds previously validated by others researchers (85, 30).

External Validity We selected a set of 50 software projects to analyze. Thus, the representativeness of these projects is a recurrent external threat to validity. We mitigate this threat by establishing a systematic process to sample a set of valid projects from GitHub. As a result, we obtained relevant Java projects with an interesting diversity of structure and size metrics.

We theorized that we can investigate if code smells are relevant for developers in practice by analyzing maintenance activities. Such theorization is a threat. One may argue that the analysis of maintenance activity is not appropriate to investigate if code smells are relevant in practice. Consequently, we would be susceptible to finding artificial results that are not consistent with practice. We do acknowledge this threat; however, we took measures to guarantee that our approach to investigating the relevance of code smells is appropriate. First, we selected a maintenance activity that is closely associated with code smells. Thus, we can expect that code smells are used in such activity. Second, we could have conducted our investigation using a different methodology. For instance, we could have surveyed developers about the use of code smells in practice. However, we deem that the use of surveys is not appropriate to our investigation. As reported in (57), what people say they do in response to surveys may not represent what they do in their work practices. Consequently, we cannot rely on only a survey without evidence about the use of code smells observed directly in practice. To work around this situation, we decided to investigate software projects in the GitHub repository. These projects are the results of developers' work after all. Thus, to a certain degree, these projects represent what happened in practice.

In our study, we focused on refactored code elements since they are likely to contain design problems. However, design problems can appear in any code element regardless of it having been refactored or not, thus, we are missing these elements. Consequently, we could (erroneously) conclude that code smells are not key symptoms for developers to identify relevant. In other words, only considering refactored elements imposes a threat to our study. It is a threat because code smells and design problems can appear in elements that were not refactored. Nevertheless, we found that refactoring operations are not applied to smelly elements by coincidence. The chance of randomly choosing a smelly element in our dataset is only 0.3%. Thus, refactoring operations indeed tend to concentrate on smelly elements. Therefore, to use refactoring to investigate code smells was appropriate. Unfortunately, we cannot say the same about design problems. We only can say that in the context of refactored code, code smells are key symptoms to indicate design problems in most cases.

3.7 Summary

This chapter presented the first study to gather knowledge about how developers identify design problems in practice. In this study, we investigated if code smells are key symptoms for developers to identify design problems in practice. The reason for such investigation is due to the role that code smells play in several techniques to support the design problem identification. Several studies explore code smells as the primary design problem symptom since they tend to co-occur with design problems (53, 47, 54). Thus, related studies have proposed techniques based on code smells.

These studies assume that code smells can indicate design problem. Unfortunately, these studies did not investigate if code smells are indeed key symptoms for developers to identify design problems in practice. In this sense, we conducted a retrospective study to investigate if code smells can indicate relevant design problems in practice (Section 3.2). For this study, we relied on refactoring to find elements affected by relevant design problems. We relied on refactoring due to its close relation to design problems. Thus, if the refactored code has simultaneously signs of design problems and code smells, then we can assume that code smells are key symptoms to identify design problems.

To conduct our investigation, we selected 50 software projects, in which we analyzed if the smelly elements were the focus of refactoring operations

(Section 3.3). Our results indicate that most refactoring operations were applied to smelly elements (Section 3.4). From 52,667 refactoring operations, 79.48% were applied to elements that contained at least one code smell. In fact, 47.38% of them were applied to elements with more than one smell, while 32.10% happened to elements with only one code smell. In other words, when we consider only refactoring operations applied to smelly elements, most of them occurred in elements with multiple code smells. This distribution repeated itself when we analyzed each refactoring type and tactic. These results indicate that code smells likely represent key symptoms for developers to identify design problems in most cases.

Unfortunately, we found cases where code smells are not key symptoms. In fact, the design of our study limited us to the context of refactoring. Therefore, there are still cases where we cannot state whether code smells are key symptoms to identify relevant design problems. Based on these results and to carry on the investigation about the role of code smells in the design problem identification, our next step is to investigate if code smells suffice in helping developers to identify design problems. In other words, our next study is to investigate if developers can use code smells to identify design problems in practice.

4 Investigating the Support of Code Smells to Identify Design Problems

Although code smells have been a well-researched topic over the last decade, most studies tend to focus on proposing solutions to support developers in identifying design problems (52, 53, 38, 28, 47, 46). Unfortunately, little effort has been made towards investigating to what extent code smells support developers during the design problem identification, which is worrisome. If there is no understanding about the support provided by smells, these proposed solutions may fall short of expectation. In light of this topic, we presented in the previous chapter a retrospective study to investigate if code smells likely represent key symptoms for developers when identifying relevant design problems in their projects. In that study, we analyzed the refactored code to investigate if code smells can spot design problems in the refactored elements. That study was our first step in our quest to understand the role that code smells play in supporting developers during the identification of design problems. We found that code smells likely represent key symptoms to identify relevant design problems in most cases, at least in the context of refactored code. However, we found some scenarios at which code smells cannot be used as key symptoms for developers. Based on these scenarios, we wondered if in practice code smells suffice to support developers in design problem identification.

Indeed, the assumption that code smells can support developers during the design problem identification is a reasonable one. Some studies have shown that code smells can be a consistent indicator of design problems (53, 47, 54). For instance, Oizumi *et al.* showed that a design problem is often related to an agglomeration, which is a group of code smells that are somehow related to each other (47). Even though Oizumi *et al.* found that code smell agglomerations can be a consistent indicator of design problems, they did not investigate if developers can actually use and reason about code smells (agglomerations) to identify design problems. It may be possible that, in practice, code smells do not suffice to help developers to find a design problem. Similarly, other studies (53, 28, 51, 27, 55, 29) did not investigate if code smells, or their techniques based on smells, support developers during the identification of design problems. Such investigation is necessary to understand the role of code smells in the design problem identification. For instance, if code smells do not suffice to support developers during the design problem identification, these smell-based techniques may be doomed to fall short of expectations.

Therefore, to investigate if code smells suffice to support developers to identify design problems, we designed and executed a multi-method study with 11 professional developers. Our goal was to analyze if developers can effectively find design problems when using code smells. As we found in our previous study that developers focus on refactor code elements that contain either a single smell our multiple smells (Section 3.4), we divided the developers into two groups. In the first group, we asked them to identify design problems using only single smells. In the second group, we asked them to use multiple code smells – in this group, we asked them to use agglomerations, as defined by Oizumi *et al.* (47). As agglomeration is a smell-based technique to support developers during the identification of design problems, we can investigate if developers achieve better precision in identifying design problems using agglomeration. Comparing developers using single smell and multiple smells (*i.e.,* agglomerations) allows us to investigate to what extent code smells suffice to help developers along the identification tasks. For instance, in our previous study, we discussed that developers may have a higher chance of identifying design problems when reasoning about multiple smells (Section 3.4.2.3). On the other hand, to reason about multiple code smells simultaneous may be a hard task to carry on. Thus, we expect to find out whether developers benefit of reasoning about multiple smells.

Our analysis revealed that only 36.36% of developers found more design problems when explicitly reasoning about agglomerations as compared to code smells. On the other hand, 63.63% of developers reported the lowest number of false positives. In summary, we noticed that code smells are not enough to support developers in identifying design problems, even when developers use agglomerations. Developers actually mentioned that they need better support to identify design problems; a support that code smells cannot provide. When we combine this finding with the results from the study reported in Chapter 3, we realize that in practice code smells do not suffice to help developers to identify design problems. Consequently, developers need to rely on other symptoms in addition to smells.

We describe this study in details next. In Section 4.1, we describe agglomerations and we explain how developers can use the smells within an agglomeration to identify a design problem. In Section 4.2, we explain why we are investigating whether code smells suffice to support developers during design problem identification. We also present in this section the procedure used to conduct our investigation. In Section 4.3, we discuss the results that we found. In Sections 4.4 and 4.5, we present some related studies and threats to validity, respectively. Finally, we summarize this chapter in Section 4.6.

4.1 Background

This section presents the background required to understand our study settings. We intend to investigate whether code smells suffice to help developers to identify design problems. For this investigation, we compare developers using code smells with developers using agglomeration (47). Agglomeration is a smell-based technique to group code smells that are likely to indicate a design problem. We explain this technique in this section.

4.1.1 Code Smell Agglomerations

Developers can rely on the analysis of code smells to identify design problems (Section 2.3). In fact, some studies (49, 38, 47, 54) suggested that elements with several code smells are likely to have a design problem. Thus, it is reasonable to expect that developers will be able to find design problems if they analyze the smells in these elements. Unfortunately, a developer often has to analyze various code elements to confirm the presence of a design problem. Furthermore, it is hard and time-consuming to identify which code smells he should focus. Even for small software systems, there are hundreds of smells (38). Consequently, analyzing them to identify design problems may not be trivial for developers.

Some studies try to soften this scenario (52, 47, 46). For instance, Oizumi *et al.* (47) proposed a set of heuristics to group code smells that are interrelated in source code, and named such group an agglomeration. They found that most code smells associated with a design problem were part of one or more agglomerations (82). In fact, they found that the chance of each code smell within an agglomeration being related to a design problem is more than five times higher than every single smell that is not part of any agglomeration (47). Furthermore, this result is also aligned to the one found in our previous study about code smells (Chapter 3). We found that refactored elements with multiple code smells are likely to contain a design problem, and developers can benefit of reasoning about these multiple smells to identify it. Thus, these results suggest that an agglomeration can better help developers to identify design problems. Unfortunately, Oizumi *et al.* did not investigate

Table 4.1: Agglomeration Categories

Category	Description
Intra-method	Agglomerations of code smells that are located in single methods. There is an agglomeration in a method if the method is affected by at least *T + 1* code smells, where *T* is a threshold that can be defined by the developer.
Intra-class	Just like methods, classes may also be affected by diverse code smells. There is an agglomeration in a class if the class is affected by at least *Y + 1* code smells, wehre *Y* is a threshold that can be defined by the developer.
Intra-component	Agglomerations of code smells that occur inside of a single design component. This agglomeration comprises code elements that are located within a single component, and the elements are affected by the same type of code smell. The elements also must be connected by method calls or type references.
Hierarchical	Agglomerations composed by code elements that are affected by the same type of code smells, and these elements implement the same interface or inherit from the same code element.

whether developers can use agglomerations to identify design problems. Such an investigation is necessary, since the relation between design problems and code smells is often complex.

An agglomeration is determined in the program by the co-occurrence of two or more code smells in the same method, class, hierarchy or package (or component). Code smells that co-occur in these cases are only considered part of an agglomeration if they are syntactically related to each other (47). A syntactic relation happens when code smells are explicitly related in the source code. An explicit relation is established between two smelly elements (*i.e.*, elements with code smells) whenever they have at least one of the following dependencies: a shared attribute, a method call, class extension and method overload. Based on these dependencies, a syntactic agglomeration is classified according to their scope in the program. Table 4.1 presents four categories of agglomeration: *intra-method*, i*ntra-class*, *hierarchical*, and *intra-component*

4.1.2 Identifying Design Problems with Agglomerations

We define *non-agglomerated code smells* to be those smells that did not undergo any type of grouping. Conversely, we define agglomeration as the group of code smells that were grouped according to Oizumi *et al.*'s study (47). As previously discussed, developers can benefit from using agglomerations instead of using non-agglomerated code smells. For instance, the chance of each code smell within an agglomeration being related to a design problem is higher than every non-agglomerated smell. Furthermore, agglomerations can narrow down the number of code smells that developers need to analyze. Let us consider

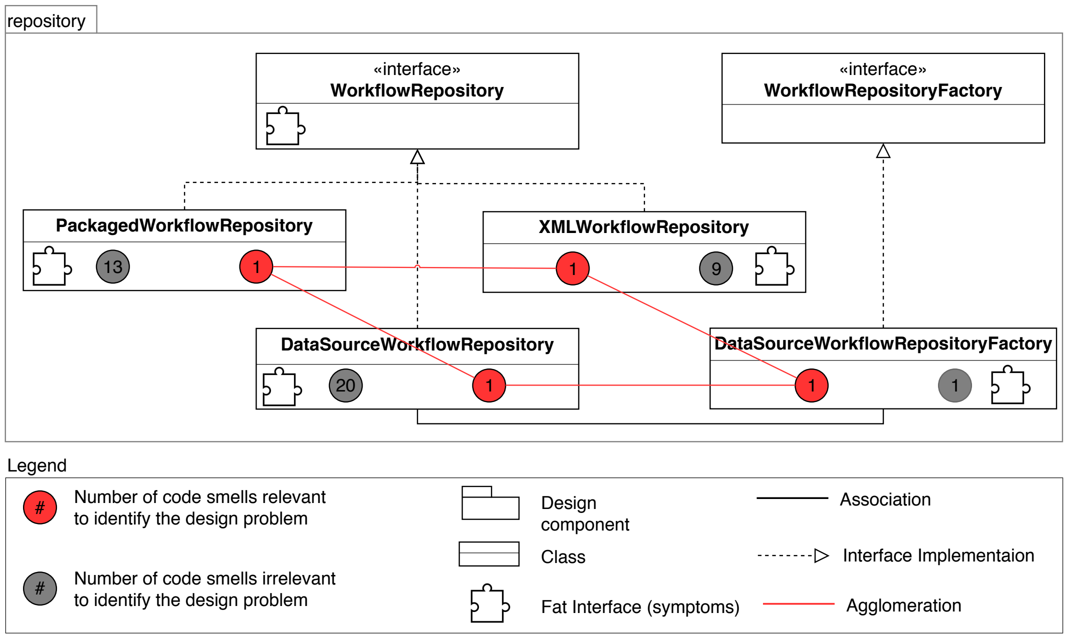

repository
«interface»
WorkflowRepository
«interface»
WorkflowRepositoryFactory
PackagedWorkflowRepository
13
1
XMLWorkflowRepository
1
9
DataSourceWorkflowRepository
20
1
DataSourceWorkflowRepositoryFactory
1
1
Legend
Number of code smells relevant to identify the design problem
Number of code smells irrelevant to identify the design problem
Design component
Class
Fat Interface (symptoms)
Association
Interface Implementaion
Agglomeration

Figure 4.1: Example of Agglomeration in the Workflow System

the following example in Figure 4.1 to discuss how developers can benefit from using agglomerations to identify design problems.

This figure presents some classes that belong to the Workflow Manager subsystem – a subsystem of the Apache OODT (Object-Oriented Data Technology) system (96). It is responsible for describing, executing, and monitoring workflows. Suppose that a developer is in charge of identifying design problems in the repository component. For this task, he (or she) can rely on agglomerations to spot elements that may contain a design problem. Thus, he can run, for instance, the Organic tool to find these agglomerations. The Organic tool (78) is a plug-in developed for the Eclipse IDE, which contains the algorithms that implement the detection of agglomerations. After running the tool, he will notice that the repository contains several code smells (represented by circles in the figure), however only four of them compose an agglomeration.

This agglomeration is formed by four instances of the `Feature Envy` smell in this example. As illustrated by Figure 4.1, each of the `Feature Envy` occurrences affects a different class. In this case, three classes implement the WorkflowRepository interface. When the developer analyzes these classes based on the `Feature Envy` smell, he can realize that these classes contain the smell because one of their methods is more interested in other classes than in its own hosting class. This happens because these methods are forced to implement a method that was defined in the WorkflowRepository interface. The smells in the agglomeration are indicating that (the corresponding method in) the interface may contain a design problem. In fact, this interface has the *Fat Interface* problem since it declares more than one concern. As a consequence, the design problem is scattered into the classes that use or implement the interface (they are indicated in the figure by the puzzle symbol); forcing the other classes to use or implement these other concerns. This "forced implementation" becomes a problem because these methods are implementing a concern that should have not been implemented in their hosting classes. That happens because the WorkflowRepository interface processes multiple services; thus, any class that implements this interface needs to handle more services than it actually should have to, affecting the system maintainability and understandability negatively.

In this example, the developer knows that the code smells in the agglomeration have the same type (`Feature Envy`). Also, he knows that three classes affected by the agglomerated smells implement the same interface, as reified in a *hierarchical* agglomeration. This interface, in its turn, seems to provide non-cohesive services. Thus, the developer can infer that *Fat Interface* is affecting the WorkflowRepository interface. On the other hand, if he did not reflect upon the code smell agglomeration, it would be harder for him to identify the design problem. One of the reasons is that the number of code smells spread over the 6 classes and two interfaces within the package. Although the package contains only eight classes (Figure 4.1 only shows some of them), it has more than 50 code smells. Thus, he would have to analyze many smells to discard, postpone or further consider them in the identification of design problems.

Let us assume that the developer only reasons about each non-agglomerated code smell, *i.e.*, without taking into consideration the agglomeration. Thus, he can choose to analyze the DataSourceWorkflowRepository class first because it contains the highest number of smells. Analyzing the 21 instances of code smells in the class, the developer may notice that the class has smells related to high coupling with other classes (`Intensive Coupling` and `Dispersed Coupling`), low cohesion (`Feature Envy`), and an overload of responsibilities (`God Class`). However, all these smells may indicate other different problems. Thus, he would have to extend the analysis to other classes to gather more information that can potentially indicate a design problem. Unfortunately, the other classes also have different instances of code smells, and these instances may not be related to any design problem. Therefore, the developer can face difficulties to find the relevant code smells that can help him to identify a design problem. Thus, the analysis of agglomerations seems to be a better strategy. However, there is limited empirical understanding if agglomerations improve developers' precision in identifying design problems.

4.2 Study Design

This section presents the design of our study. In Section 4.2.1, we present the research questions that we intend to answer. In Section 4.1.1, we present the types of agglomerations that we are considering in our study. Finally, in Section 4.1.2, we explain through an example how developers can use an agglomeration to identify a design problem.

4.2.1 Research Questions

The identification of design problems in the source code is not a trivial task (26, 27). A reason is that developers are forced to analyze several elements in the source code to identify some design problems. For instance, if a interface does not have a code smells to indicate the presence of a *Fat Interface*, developers would need to analyze the source code of a suspicious interface, but also the classes that either implement or depend on this interface (Section 1.1). This explains why the occurrence of a code smell in isolation often does not indicate a design problem (Section 3.4.2.3).

In fact, recent studies revealed that design problems are often located in code elements affected by many smells (49, 38, 47, 50, 54). The more code smells a code element has, the most likely it contains a design problem (47), which may help to explain why developers focus their effort on refactoring these elements (Chapter 3). Hence, one may expect that developers can analyze a smelly element and, then, identify a design problem with in it. Unfortunately, we do not know whether developers are able to use code smells to identify design problems in practice, even though several studies have presented techniques based on code smells to help developers to identify design problems (41, 38, 39, 28, 47, 46). In this context, we defined the following broader research question:

RQ2. Are developers able to use code smells to identify design problems?

To answer this research question, we need to conduct a study in which developers have to analyze the source code of software system with the support of code smells. Then, we can verify whether developers were able to use the smells to identify design problems in these systems. In the context of this study, we also need to investigate to what extent smells help developers in identifying design problems. This investigation is necessary since some studies have proposed smell-based techniques to help developers to identify design problems (53, 28, 47, 51, 27, 55, 29). If we find out throughout this investigation that code smells do not suffice to support developers during the design problem identification, techniques that rely on code smells may fall short of expectation.

As a matter of fact, we cannot neglect these techniques if we also want to understand whether code smells suffice to help developers. Code smell agglomeration is an example of a technique that intends to help developers to identify design problems (47, 82). According to Oizumi et al. (82), agglomerations may help developers to identify design problems. They found that most code smells associated with a design problem were part of at least one agglomeration. Second, they found that the chance of each smell within an agglomeration being related to a design problem is more than five times higher than every smell that is not part of any agglomeration (47). Based on these results, there is a chance of developers being most effective in identifying design problems using agglomerations. Unfortunately, the authors did not investigate if agglomerations improve the precision of developers during the design problem identification. Thus, in the context of our research question, we intend to answer the following specific research question:

SRQ1. Does the use of agglomerations improve the precision of developers in identifying design problems?

This specific research question allows us to analyze whether code smell agglomerations help developers to identify design problems with high precision. Agglomeration is a technique that exclusively relies on code smells. Different from other smell-based techniques (39, 28, 27, 58), an agglomeration only groups smells, *i.e.*, there is no other symptom used together with code smells. Therefore, investigating agglomerations does not deviate from the context of our research, which is the investigation of code smells. In fact, investigating agglomerations helps us to find out if code smells suffice to help developers to identify design problems since agglomeration is completely based on code smells.

One could assume that developers would often benefit from the use of agglomerations in their quest for finding design problems. However, it is through the analysis of SRQ1 that we will be able to verify whether developers can correctly identify more design problems using agglomerations. Regardless of the result, another question that should be investigated is about how to better support developers in exploring code smells. Even though previous studies (53, 28, 47, 51, 27, 55, 29) have shown the strong relation between design problems and code smells, we do not know whether and how the identification of design problems with smells can be improved. The second specific research question addresses this matter.

SRQ2. How can the identification of design problems with code smells be improved?

To answer our research questions, we conducted a controlled experiment with 11 professional developers. In this study, precision is measured based on the percentage of true positives indicated by the developers – *i.e.*, the percentage of design problems validated as true design problems (Section 4.2.5). In order to measure if there was an improvement or not in the precision, we are comparing the participants using agglomerations with participants identifying design problems with a list of non-agglomerated smells, *i.e.,* code smells that were not grouped as agglomeration. Thus, we asked the participants to identify design problems using agglomerations and non-agglomerated smells. After the identification, we conducted an analysis over the identified design problems. In this analysis, we used a ground truth to confirm or refute each design problem indicated by participants. Then, we compared the number of false positives and true positives produced with the developers using agglomerations against developers using non-agglomerated smells.

As part of the study, we also applied a post-experiment questionnaire to all developers. The objective of this questionnaire was to identify the main advantages and barriers of reflecting upon multiple smells along the task of identifying design problems. The outcomes of this analysis could help us to understand better ways to improve the support for developers identifying design problems. Moreover, the response for the questionnaire could help us to understand whether code smells suffice or not to help developers to identify design problems. Indeed, the combination of quantitative and qualitative analyses can help us to draw more well-grounded conclusions about how developers identify design problems in practice, specifically with the support of code smells.

4.2.2 Experiment Procedures

In our study, we needed to ensure that participants meet certain requirements (Section 4.2.3). As a consequence, we obtained a small sample of participants. Thus, we opted for conducting a *quasi-experiment* (98). A *quasi-experiment* is an experiment in which the units or groups are not assigned to conditions randomly. This allowed us to assign each participant to different treatments during the experiment steps. The experiment was conducted individually with each participant. They had to perform the experiment in two steps with four tasks in each one. Both steps comprise the same set of tasks the only difference between the steps was regarding the treatment, *i.e.*, usage of agglomerations or non-agglomerated code smells.

As explained before, we need to compare developers using agglomeration with them using non-agglomerated smells. This comparison allowed us to verify whether there was an improvement in the precision when developers used agglomerations. Thus, we divided the participants into two groups. The first group would identify design problems using agglomerations in the first step. After that, they would identify design problems using a list of non-agglomerated smells in the second step. The second group of participants would make the identification inversely: using the non-agglomerated smells in the first step and, then, using the agglomerations in the second one. Thus, in each step, we have two groups of participants.

As each participant identified design problems twice (first and second step), we had to select two software projects. Thus, each participant could identify design problems using a different project in both steps. Another reason for providing two software projects is to avoid bias with the learning curve. For example, supposing that the participant uses the same project in both steps. He could find more problems in the second step than in the first one. That could happen because he can identify in the second step the same problems that he identified in the first step, plus other design problems identified only in the second step. This increase in the number of design problems found in the second step would not be due to the use of agglomerations, but rather due to the knowledge acquired by the participant.

There are four possible combinations of participants based on the distribution between steps and software projects. Therefore, all participants were divided into four mutually exclusive arranges to promote a fair comparison. Table 4.2 presents the cross design for the four arranges. The agglomeration group represents the group of participants that identified design problems using the agglomerations, and the NA-smells group comprises the participants that identified design problems using the list of non-agglomerated smells.

The study was composed of a set of six activities distributed into three phases, as represented in Figure 4.2 described as follows.

Activity 1: Apply the questionnaire for subject characterization. The subject characterization questionnaire is composed of questions to characterize each participant, including academic degree, professional experi-

Table 4.2: Combinations of Groups, Projects and Steps

	Step 1		**Step 2**	
Arrange	**Group**	**Project**	**Group**	**Project**
1	Agglomeration	Project 1	NA-smells	Project 2
2	Agglomeration	Project 2	NA-smells	Project 1
3	NA-smells	Project 1	Agglomeration	Project 2
4	NA-smells	Project 2	Agglomeration	Project 1

NA-smells = Non-agglomerated code smells

ence with Java programming, background on code smells and Eclipse IDE. This questionnaire is available at Appendix B.

Activity 2: Training Session. After defining the order of execution of each step, the next step was to provide a training session for the participants. The main objective of the training session was to ensure a common knowledge base for all participants, which was required to understand and properly execute the experimental tasks. Thus, they received training about basic concepts and terminologies. This training was given only once for each participant individually before the first steps of the experiment. The training consisted of a 15-minute presentation that covered the following topics: software design, code smells, and design problems. The training session took approximately 15 minutes, and the participants could make any question throughout it. The presentation is available at Appendix B.

After the training session, subjects received some artifacts that could be used during the experiment (Section 2.3). They received a list with a brief description of the types of design problems presented in the training session. They also received a list with the description of basic principles of object-oriented programming and design. They received a document containing: (i) a brief description of both project systems, and (ii) a very high-level description of their design blueprint (Appendix B). We gave these documents because when they have to conduct perfective maintenance tasks, they need to have some minimal information about the systems to be maintained. The design blueprint represented the high-level design in the view of the project managers, but it was not detailed enough to support the identification of design problems. As it often occurs in practice, the analysis of the source code is inevitably required to identify a design problem.

Activity 3: System Introduction. We asked participants to read the document containing the description of the project in which they would identify design problems. They had 20 minutes to read the description and the design blueprint of the system. Thus, they could start the identification with a certain level of familiarity with the software project.

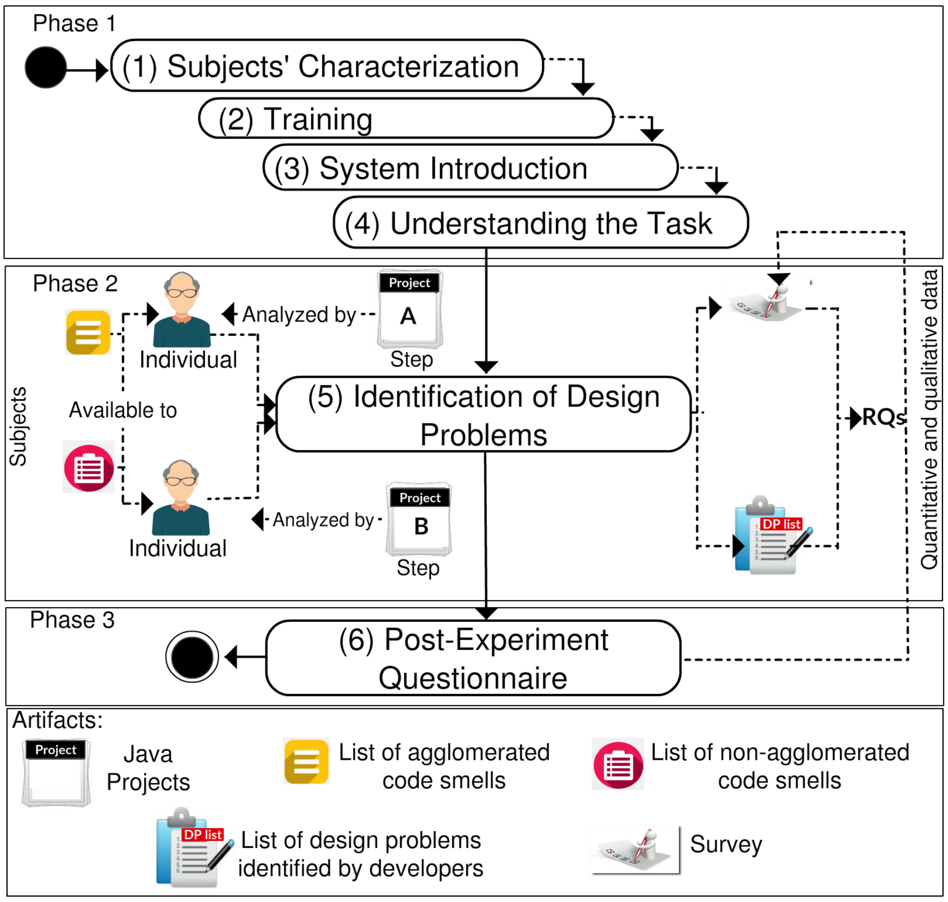

Phase 1
(1) Subjects' Characterization
(2) Training
(3) System Introduction
(4) Understanding the Task
Phase 2
Subjects
Individual
Available to
Individual
Analyzed by
Project A
Step
Analyzed by
Project B
Step
(5) Identification of Design Problems
RQs
Quantitative and qualitative data
DP list
Phase 3
(6) Post-Experiment Questionnaire
Artifacts:
Java Projects
List of agglomerated code smells
List of non-agglomerated code smells
List of design problems identified by developers
Survey

Figure 4.2: Experimental Design

Activity 4: Understanding the Task. In this activity, we explained how the participant could use the Organic tool to collect either the list of agglomerations or the list of (non-agglomerated) code smells. As the Organic tool was developed as an Eclipse plug-in, we explained each one of the sections displayed in the Eclipse IDE and that was related to the Organic tool. This activity lasted approximately 10 minutes.

Activity 5: Identification of Design Problems. In this activity, the participant had 45 minutes to identify design problems in the project. We emphasized to the participant the importance of achieving the key goal of finding design problems. For each identified design problem, the participant was asked to provide in a form (Appendix B) the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods or packages realizing the design problem in the source code, and (iv) the category (or categories) of agglomerations (Section 4.1.1) that helped him to identify the design problems. If the participant was identifying design problems with the list of non-agglomerated smells, he needed to provide almost the same information; the difference was that instead of providing the agglomeration (and its category), he needed to provide the code smells that he used to identify the design problem. For conducting this task, participants were instructed to use only the information provided by Organic in the current phase. This means that neither the non-agglomerated group had access to the list of agglomerations, nor the agglomeration group had access to the list of non-agglomerated smells. Nevertheless, both the project source code and the information provided by Organic (agglomerated or non-agglomerated smells) could be freely explored and analyzed during the design problem identification.

Activity 6: Post-experiment Questionnaire. In this activity, the participant received a questionnaire to provide feedback. This questionnaire provides a list of questions, which enables the participant to expose his opinion on the identification of design problems. This questionnaire is available at Appendix B. More details about this activity are provided in Section 4.2.6. After the sixth activity had been completed, we asked the same participant to repeat all tasks in the second phase.

4.2.3 Software Projects and Participant Selection

In order to conduct the experiment, we selected two software systems in which developers had to identify design problems. We selected two programs that represent components of the Apache OODT project (96): *Push Pull* and *Workflow Manager*. We selected subsystems of the OODT project since it is a large heterogeneous system; then, we could choose subsystems based on their diversity. Also, the Apache OODT project has a well-defined set of design problems previously identified by developers who actually implemented the systems (Section 4.2.5); thus, avoiding the introduction of false positive design problems in the ground truth. In addition, the OODT project was developed for NASA, used in other studies (41, 52, 53, 38, 47) and with a global community involved in its development. A brief description of the project systems are presented as follows:

– Push Pull: it is the OODT component responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area.

– Workflow Manager: it is a component that is part of the OODT client-server system. It is responsible for describing, executing, and monitoring workflows.

After choosing the projects, our next step was to recruit developers for the experiment. Thus, we sent a characterization questionnaire for a group of developers of our network. Their answers were analyzed to determine which

Table 4.3: Knowledge Classification

Classification	**Description**
None	I have never heard about it
Minimum	I have heard about it, but I do not use it
Basic	I have a general understanding, but almost never use it
Intermediary	I have a good understanding, and use basic features sometimes
Advanced	I have a deep understanding, and often use advanced features
Expert	I am a specialist in this topic, and use many features almost every day

Table 4.4: Characterization of the Participants

Id	**Experience in years**	**Education Level**	**Knowledge**		
			Java	**Code Smells**	**Eclipse**
P1	5	PhD	Advanced	Advanced	Advanced
P2	6	Graduate	Advanced	Basic	Advanced
P3	8	Master	Advanced	Intermediary	Advanced
P4	4	Graduate	Intermediary	Basic	Basic
P5	5	Master	Advanced	Intermediary	Intermediary
P6	5	Graduate	Intermediary	Intermediary	Intermediary
P7	12	Graduate	Expert	Advanced	Expert
P8	5	Graduate	Advanced	Advanced	Advanced
P9	10	Graduate	Intermediary	Intermediary	Intermediary
P10	4	PhD	Advanced	Intermediary	Advanced
P11	5	PhD	Advanced	Intermediary	Advanced

of them were eligible to participate in the study based on the following requirements:

R1. Four years or more of experience with software development and maintenance. We have chosen four years because this is the average time used by companies such as Yahoo (99) and Twitter (100) to classify a developer as experienced.

R2. No previous knowledge about Push Pull and Workflow Manager.

R3. At least basic knowledge about code smells.

R4. At least intermediary knowledge of Java programming and Eclipse IDE.

We defined the knowledge of each topic based on a scale composed of five levels: *none*, *minimum*, *basic*, *intermediary*, *advanced* and *expert*. Table 4.3 shows the description of such classification. We included in the questionnaire the same description of each level, allowing the subjects to have a similar interpretation of the answers. Table 4.4 summarizes the characteristics of each developer selected for the experiment.

4.2.4 Quantitative Analysis Procedures

In order to answer our research questions, we asked the experiment participants to analyze two systems with the aim of identifying design problems as described above. For each system, we analyzed the precision of participants regarding the identification of design problems. The precision of participants was measured based on *true positives* (TP) and *false positives* (FP). In this context, a true positive is a candidate of design problem, as indicated by the participant, that was confirmed by a ground truth analysis. On the other hand, a false positive is a candidate of design problem that was not confirmed in the ground truth analysis. Thus, the precision is calculated using the following formula:

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

Precision is an important aspect of the identification task. Through the correct identification of design problems, developers are able to optimize their work by solving problems that really impact the system design. On the other hand, the lack of precision would lead software development teams to spend time and budget with irrelevant tasks. For example, in companies adept of code review practices (97), the lack of precision can lead developers to waste time on refactoring tasks that do not contribute to system maintainability. The precise identification of design problems is also important in open source projects. For instance, the contributions of eventual collaborators via pull requests are often rejected by core developers due to the presence of design problems (19). Therefore, in this case, a lack of precision could lead core developers to reject relevant contributions due to “false design problems”.

In this study, we did not measure recall because of the high number of design problems in the analyzed systems. Together with the system’s original developers, we created a ground truth of design problems (Section 4.2.5) with more than 150 instances of design problems. Hence, it would be impracticable for participants to find all them due to the time constraints in the study (45 minutes). Consequently, they were expected to reach a low recall value. Therefore, we focused on precision.

4.2.5 Ground Truth Analysis

We had to validate the identified design problems as true positives or false positives for each one of the analyzed systems. However, we could not argue that a design problem was a “true design problem” or not since we were not involved with the design of each system. Thus, we relied on the knowledge of the systems' original designers and developers to help us in validating the design problems. We certified they were the people who had the deepest knowledge of the design of the investigated projects. We highlight these designers and developers were not subjects of this experiment.

We performed two steps to incrementally develop the ground truth. First, we asked original OODT designers and developers to provide us a list of design problems affecting the systems. They listed the problems and explained the relevance of each one through a questionnaire (Appendix B). They also described which code elements were contributing to the realization of each design problem. Second, we identified some design problems using a suite of design recovery tools (101). We asked developers of the systems to validate and combine our additional design problems with their list. The procedure for the additional identification was the following: (i) an initial list of design problems was identified using a method presented in (53), (ii) the developers had to confirm, refute or expand the list, (iii) the developers provided a brief explanation of the relevance of each design problem, and (iv) when we suspected there was still inaccuracies in the list of design problems, we discussed with them. In the end, we had the ground truth of design problems validated by the original designers and developers.

4.2.6 Qualitative Analysis Procedures

The experiment with professional developers aimed to help us to assess the precision of developers in the identification of design problems with agglomerations. However, we also need to investigate whether there is room for improvements regarding the use of code smells. Therefore, we conducted a qualitative analysis to investigate what should be improved from the perspective of professional software developers. Besides identifying possible improvements, this analysis also helped us to understand if code smells suffice to help developers to identify design problems.

As described in Section 4.2.2, we asked the participants to provide us feedback about the identification of design problems. They answered a post-experiment questionnaire (Appendix B), and we used their answers to conduct a qualitative analysis. The objective of the questionnaire was to gather participant's opinion regarding (i) the (dis)advantages of using the agglomerations or code smells to identify design problems, (ii) whether the provided information could be easily understood, (iii) which types of information were fundamental to identify design problems, (iv) what he believes that should be done to improve the identification of design problems, (v) what he thought about the use of the code smells for the identification of design problems, (vi) how the visualization mechanism provided by the Organic tool affected his performance, and (vii) which types of code smells and categories of agglomerations were the most useful for identifying design problems. The results of this questionnaire helped us to answer research question SRQ2.

By applying the questionnaire, we were able to gather the opinion of developers regarding the use of code smell agglomerations. However, as reported by (57), what is reported in the questionnaire may not be what actually happens in practice. Therefore, to obtain more reliable results, we also observed the participants of our experiment during the identification of design problems. This observation was performed during the experiment and also in an analysis after the experiment, through video and audio recorded during the experiment. This analysis allowed us to look at code smell agglomerations from the standpoint of professional software developers. It is important to note that the observation of participants during the experiment does not replace nor invalidate the questionnaire responses. In fact, the combination of observations and responses helped us to obtain a deeper understanding and interpretation on the results observed in the experiment.

4.3 Results and Analysis

The results of this study are organized in three sub-sections. In Section 4.3.1, we present the results of our quantitative analysis regarding research question SRQ1. In Section 4.3.2, we provide the results of our qualitative analysis to answer research question SRQ2. In Section 4.3.3, we use the results of the quantitative and qualitative analyses to discuss our broader research question RQ2. We also discuss if code smells suffice in supporting developers during the design problem identification.

4.3.1 Does the Use of Agglomerations Improve Precision?

As described in Section 4.2.4, we conducted a quantitative analysis to answer our first specific research question (*Does the use of agglomerations improve the precision of developers in identifying design problems?*). Table 4.5 presents the precision results for each participant (rows). The first column (*ID*) shows the identification number of each participant. The second column (*Agglomeration Group*) presents the true positives (*TP*), false positives (*FP*) and

Table 4.5: Developers' Precision

Id	**Agglomeration Group**			**NA-Smells Group**		
	TP	**FP**	**Precision**	**TP**	**FP**	**Precision**
1	2	1	66.67%	1	1	50%
2	0	3	0%	1	4	20%
3	3	2	60%	1	4	20%
4	2	0	100%	1	3	25%
5	4	0	100%	3	1	75%
6	1	0	100%	1	0	100%
7	1	1	50%	1	1	50%
8	3	0	100%	3	0	100%
9	0	1	0%	0	6	0%
10	0	0	-	1	1	50%
11	0	1	0%	0	0	-
All	**16**	**9**	**64%**	**13**	**21**	**38.24%**

precision for the participants when they were provided with agglomerations to identify design problems. Similarly, the third column (*NA-smells Group*) presents the true positives (*TP*), false positives (*FP*) and precision for the participants when they were provided with a list of non-agglomerated code smells.

Developers identified a few more true positives using agglomerations. We can see in Table 4.5 that the developers identified a few more design problems (TPs) when they were in the agglomeration group (16 TP design problems) than when they were in the NA-smells group (13 TP design problems). As far as the per-subject analysis is concerned, four developers (light gray rows) identified more true positives when they used agglomerations than when they used the list of non-agglomerated code smells in the NA-smells group. The use of agglomerations outperformed the use of smells in these four cases. On the other hand, two participants (2, 10) did not identify any true positive using the agglomerations, but they identified a true positive each in the NA-smells group. The rest of the participants (6, 7, 8, 9 and 11) identified the same number of true positives (5 TP design problems) regardless of the group.

Upon qualitative analysis, we were able to reveal the main reason why the four developers in the light gray rows identified more true positive design problems in the agglomeration group than in the NA-smells group. As illustrated in the example in Figure 4.1 (Section 4.1.2), these four participants systematically used each agglomeration's smell as an indicator of the presence of a design problem. They analyzed each one of the code smells as a complementary symptom of the presence of a design problem, which gave them confidence to confirm the occurrence of the design problem. Surprisingly, we noticed the same behavior for the participant 8 even being in the NA-smells group. He was capable of agglomerating the code smells on his own, starting from the individual smells given in the list of non-agglomerated smells. Then, he used such agglomerations to identify design problems in the NA-smells group. This is the reason why he reached a precision value of 100% in both groups.

Agglomerations help developers to avoid false positives. In general, developers identified less false positives when they used agglomerations (9 FP design problems) than when they used the list of non-agglomerated smells (21 FP design problems). With the exception of participant 11, all others identified either fewer or an equal number of false positives when they were in the agglomeration group than when they were in the NA-smells group. When we analyze the NA-smells group, we can notice that more than half of the identified design problems are false positives (61,76%) while the agglomeration group identified only 36% of false positives.

After observing how developers identify design problems in the NA-smells group, we noticed that they did not go further with the analysis of the elements. Usually, a developer needs to analyze other classes in order to gather more information that can potentially indicate a design problem as discussed in Section 4.1.2. When the participants used the agglomerations, they analyzed multiple elements because they analyzed each code smell within the agglomeration even when the smells were in different elements. This behavior did not happen when participants were in the NA-smells group. In most of the cases, the participants in the NA-smells group analyzed only one code smell, which increased the likelihood of reporting false positives. Then, they reported a design problem in the class due to the presence of one smell only. However, some code smells are not related to any design problem; thus, the developer can report a false positive if he mistakenly considers a smell that is not related to a design problem. That explains why developers in the NA-smells group found so many false positives. As developers tend to look at all agglomeration' smells before reporting a design problem, the likelihood of reporting a false positive decreases, even when there is a code smell that is not related to a design problem.

Agglomerations improve the precision. In this study, participants achieved higher precision (64%) when they used agglomerations than when they used code smells (38,24%). Besides calculating the precision for the participants, we also used a statistical test to verify if there is a statistically significant difference between the two groups – *i.e.*, agglomerations and NA-Smells. Firstly, we applied the *Shapiro-Wilk normality test* to verify whether the precision data follows a normal distribution. This test resulted in p-values of 0.04133 and 0.5148 for the agglomeration group and the NA-smells group, respectively. Thus, it is not possible to claim that our sample follows a normal distribution. Based on this result, we opted for the *Wilcoxon Rank Sum Test*. The Wilcoxon test revealed a p-value of 0.3737 (with a confidence level of 95%) for our population. This means that the difference between the two groups is not statistically significant.

We cannot claim a statistical significance in our results due to the sample size of this study. However, someone could expect that participants using agglomerations would significantly outperform the NA-smells group. As a matter of fact, we noticed some factors that explain, at least partially, why participants did not find many more design problems when they were in the agglomeration group than when they were in the NA-smells group. These factors are presented in the next subsection, and they are useful to discover improvements for the identification of design problem with the analysis of smells.

4.3.2 How to Improve Design Problem Identification?

This section presents the answer to our second specific research question (*How can the identification of design problems with code smells be improved?*). We conducted a qualitative analysis to answer this question. As described in Section 4.2.6, this analysis was mainly based on the analysis of responses to our post-experiment questionnaire.

Where to start? As discussed in the previous section, the participants identified few more true positives using agglomerations. Someone could expect that *all* developers using agglomerations would significantly outperform the control group. However, we observed that participants spent much more time analyzing the agglomerations than analyzing the non-agglomerated smells. That happened because they analyzed each code smell in the agglomeration, as previously explained in Section 4.3.1. Furthermore, sometimes the participants analyzed agglomerations that were not related to any design problem. That is another factor that explains the almost equal number of true positives between both groups.

Unfortunately, almost the participants analyzed irrelevant agglomerations *i.e.*, agglomerations that do not lead to a design problem. Participants 6, 9, 10 and 11 were the ones that suffered the most from the analysis of irrelevant agglomerations. Since these four participants faced such issue, they suggested in our post-experiment questionnaire that the Organic tool should provide means to prioritize relevant agglomerations and code smells. Hence, they would not spend time with the analysis of irrelevant code smells. This issue helps us to explain why they fell short of identifying design problems through the analysis of agglomerations.

Need for prioritizing smells. The aforementioned need for prioritization shows that the time and effort required to identify design problems is a key factor for developers; thus, prioritization should be taken into consideration. As a matter of fact, the prioritization of smelly elements has been the focus of recent research (102, 103, 46). For example, in (46), the authors proposed and assessed prioritization criteria for smell agglomerations. As they have observed, the prioritized list of agglomerations would help developers to progressively analyze the agglomerations that have more chance to represent design problems, discarding the irrelevant ones. This would be especially useful in large legacy systems, in which thousands of agglomerations may be detected. Nevertheless, there is no prioritization criterion that is effective for any system (46).

Based on our qualitative analysis, we noticed that existing criteria for prioritization should select agglomerations that are cohesive. A *cohesive agglomeration* in our context is an agglomeration in which all code smells are related to the same design problem. If there is one code smell that is not related to the design problem, such smell may direct the developer away from the design problem in the worst case. In the best case, the developer will spend time analyzing a code smell that is useless to identify the design problem. This fact suggests that developers need accurate algorithms to find cohesive agglomerations and to discard the less cohesive ones. However, prioritization algorithms based on existing criteria are unable to do this as far we are concerned. Consequently, the prioritization of smells and agglomerations still poses a challenging research topic.

Analysis of multiple code smells is challenging. Besides the prioritization issue, participants also suffered to analyze multiple code smells. This difficulty was even worse when developers had to identify design problems using agglomerations. Some developers had to analyze agglomerations affecting larger program scopes, *i.e.*, agglomerations crosscutting implementation packages or class hierarchies. We noticed that a large agglomeration requires that developers reason about a wide range of smells scattered over different elements. As they tend to use each code smell as a symptom of design problem, they have difficulties to correlate the multiple symptoms of an agglomeration. This is a challenging task because the higher the number of smells involved in an agglomeration, the greater is the number of code elements that must be analyzed. Consequently, developers will have more code to analyze, which increases the complexity of the analysis.

Need for proper visualization mechanisms. In order to alleviate the analysis of code smells, some participants suggested the adoption of visualization mechanisms for smells and agglomerations. For instance, participant number eight suggested the visualization of agglomerations through a graph-based representation (104). He mentioned that such visualization would provide an abstract and general view of each agglomeration. The main advantage of this form of visualization is that the more abstract a representation is, the fewer details will be displayed for analysis. Consequently, the developers would not be overloaded with details. At the same time, an abstract representation such as the graph-based visualization would help developers to see the full extent of an agglomeration (*i.e.,* all the code elements involved in the agglomeration). After providing an abstract view, a visualization mechanism could allow developers to progressively explore the agglomeration details such as the types of smells, location of stinky elements and relationships among smells. Such details could be displayed in the graph itself, in the source code, or in complementary views.

Identification of the design problem type. The difficulty in analyzing code smells also raised the need for recommendations on which types of design problem each smell or agglomeration is most likely to indicate. These recommendations would reduce the effort required to decide whether the elements are affected by design problems or not. For example, the agglomeration in Figure 4.1 occurs in classes of the same hierarchy that are implementing the WorkflowRepository interface (Section 4.1.2). All code elements in the agglomerations presented the same type of smell, which was the `Feature Envy`. The occurrence of multiple `Feature Envies` in a unique hierarchy, suggests that there is a problem in the interface, which is spreading through all classes of the hierarchy. Therefore, to help developers to decide whether there is a problem or not, heuristics could suggest that this hierarchical agglomeration may indicate problems like *Ambiguous Interface* (7) and *Fat Interface* (13), for example.

In Section 2.3, we presented some smell patterns that can be used to identify design problems. Those patterns include code smells that are likely to indicate a design problem if they appear in code elements. Suggestions of design problem types based on those patterns can help developers to focus their attention in specific characteristics of the suggested design problems (Section 2.2). However, this kind of recommendation algorithm requires multiple case studies to understand how and when each form of smell pattern may represent specific types of design problem. As reported in a previous study (82), this is a challenging research topic.

4.3.3 Do Code Smells Suffice to Support Design Problem Identification?

Based on the results for our two specific research questions, we can answer the broader research question (*Are developers able to use code smells to identify design problems?*). The data analysis showed that developers were able to find design problems using both agglomerations and non-agglomerated smells. Not developers are only able to use code smells but they also find most design problems when they use code smell agglomerations. We found that agglomerations helped developers to identify more design problems and to avoid false positives. When we analyzed the questionnaire answers and how the developers identified design problems, we noticed that developers tended to have a higher confidence to identify the occurrence of some design problems when using agglomerations. That happens because developers tend to analyze each agglomeration's smell before reporting a design problem. Consequently, the likelihood of reporting false positive decreases. In summary, our results indicate that developers are indeed able to use code smells in practice.

However, we noticed that code smells, either as agglomerations or not, still do not suffice to help developers to identify design problems. This result is most evident when we compare participants who used agglomerations with the participants who used the non-agglomerated smells. The use of agglomerations outperformed the use of smells in only four cases (Table 4.5). In the other five cases, developers identified the same number of true positives when they used agglomerations and non-agglomerated code smells. To make matters worse, in two cases, the developers did not identify any true positive using the agglomerations, but they identified a true positive each when they used non-agglomerated smells. As mentioned in the previous subsection, there are some explanations for these results. Even so, these results indicate that code smells do not suffice to help developers to identify design problems, at least not in all cases.

Indeed, the results of our study encourage the use of smell agglomerations to identify design problems. However, some issues should be addressed before developers can explore smell agglomerations in a time-effective manner. For instance, there is a need to provide mechanisms for better prioritizing and visualizing code smells and agglomerations. Although we have found some room for improvement, we cannot guarantee that, after addressing these improvements, developers will be able to effectively identify design problems when using code smells. Maybe, we have been expecting too much from code smells. As we found in Chapter 3, code smells are not key symptoms to identify design problems in some scenarios, which made us wonder what other symptoms developers use to identify design problems in these scenarios. When we take such result into consideration and combine it with the results we found here, we can conclude that code smells do not suffice in supporting developers during the identification of code smells. Since code smells do not suffice to help developers to find design problems, we need to investigate other symptoms that developers use in addition to code smells.

In this context, we can ask developers to use other symptoms to identify design problems, and then to investigate whether these symptoms suffice to help them during the design problem identification. In other words, we can conduct the same type of investigation that we did with code smells. Unfortunately, this research strategy may not be the most suitable for our general goal. We know that code smells are used in practice (29), however we do not know what other symptoms developers use in practice, which does not allow us to conduct the same research strategy that we did with code smells. Therefore, if we intend to understand how developers identify design problems in practice, the most reasonable research is to investigate what else developers use to identify design problems. In light of this investigation, we already found that code smells do not suffice to help developers to identify design problems. Therefore, our next study should be towards understanding what are the other symptoms that developers use in practice and how they use them to identify design problems.

4.4 Related Work

We found few studies that investigated the detection of agglomerations (78, 103). In this context, Vidal, Marcos and Díaz-Pace (103) presented a tool for detecting code smells and agglomerations of a (Java-based) system and ranking them according to different criteria (103). The main benefit of using this tool is that developers can configure and extend the tool by providing different strategies to identify and rank the smells and agglomerations. However, this tool represents agglomerations without showing the relations that could exist between code smells.

Regarding detection and visualization of non-agglomerated smells, Van Emden and Moonen (105) presented a tool that detects and visualizes code smells in source code, displays the code structure as a graph and maps code smells onto the attributes of that graph. This tool can be problematic for several reasons. The visualization is built assuming that code smells are concentrated in a particular region of the code, and that software metrics will point developers there. This assumption does not always hold; many code smells require understanding the relationships between many interacting code elements. These relationships cannot be represented by a simple mapping between code structure and the attributes of a graph.

Other studies (52, 54) have investigated the effects of code smells on software design. For instance, Yamashita *et al.* (54) studied collocated smells – code smells that interact in the same source code file –, and coupled smells – code smells that interact across different source code files. Regarding software design, they observed that limiting the analysis to collocated smells would reduce their capability to reveal design problems, as coupled smells may reveal critical design problems.

We also found studies that have investigated the use of information other than code smells to identify design problems (44, 45). In this case, Mo *et al.* (44) proposed and evaluated the combination of structural, history and design information to identify potential design problems. Xiao *et al.* (45) introduced an approach that uses a history coupling probability matrix to identify and quantify design problems. However, one disadvantage of such studies is they rely on design information, which may not exist for many software systems. In addition, they have not evaluated their work from the perspective of software developers.

Based on these related studies, we observed that they did not investigate whether in practice developers are able to use code smells. In fact, related works propose techniques for supporting the detection and visualization of code smells (105, 52, 53, 38, 28, 77, 78, 47, 106, 103, 46, 107). Consequently, we did not know heretofore whether developers could use code smells to identify design problems, nor whether smells suffice to help them during the identification. Therefore, our research covers this gap by investigating whether smells suffice to help developers to identify design problems in practice, which they do not. Additionally, our results indicate the need to observe in practice the symptoms that developers use in addition to code smells.

4.5 Threats to Validity

This section presents some threats that could limit the validity of our main findings. For each threat, we present the actions taken to mitigate their impact on the research results.

The first threat to validity is related to the number of participants in the study. We have selected a sample of 11 participants, which may not be enough to achieve conclusive results. However, instead of drawing conclusions based on only the quantitative results, we conducted a qualitative analysis as well. In addition to conducting a qualitative analysis, we defined a set of requirements for selecting developers suitable for the study. Also, we conducted training sessions with all participants. Such sections aimed to resolve any gaps in the participants' knowledge and any terminology conflicts, allowing us to increase our confidence in the results.

The second threat is related to possible misunderstandings during the study. As we asked developers to conduct a specific software engineering task and to answer a questionnaire, they could have conducted the study differently from what we asked. To mitigate this threat, we assisted the participants during the entire study, and we helped them to understand the experiment tasks and the questionnaire. We highlighted that our help was limited to only clarifying the study in order to avoid some bias on our results.

Finally, there are two threats concerning the selected projects. The first one is about the difficulty of the participants in understanding the source code used in the experimental tasks. This difficulty appears due to the complexity of the source code and time constraints to complete each task. The second threat is related to the possibility of one software project being easier to identify design problem than the other. We minimized the first threat by running a pilot-experiment to define a experimental time reasonable to perform the tasks. To minimize the second threat, we selected projects with similar size, complexity, and number of known design problems. We also have trained all participants about each project. In addition, our results suggest no variation in difficulty to identify design problems in the two projects.

4.6 Summary

This chapter presented the second study to investigate whether developers are able to identify design problems based on code smells. This study is also the second one to understand the role that code smells play to help developers to identify design problems. We assessed whether developers are effective in revealing design problems when they reason about code smells (Section 4.2). We conducted this investigation because recent studies have shown that design problems are likely to be related to code smells. Unfortunately, these studies did not evaluate whether developers can identify design problems using code smells. Thus, we conducted a multi-method study with 11 developers. In the study, we asked them to identify design problems using code smells and agglomerations. An agglomeration is a group of code smells that are somehow related to each other in the source code. After the experiment, we compared their results using agglomerations with the results of when they used the non-agglomerated smells to identify design problems. This comparison allowed us to better understand whether code smells suffice to help developers to identify design problems

The data analysis showed that developers find most design problems when they use code smell agglomerations to identify design problems. In fact, we noticed that agglomerations help developers to avoid false positives. Consequently, we found that agglomerations may improve the precision of developers in identifying design problems. In our qualitative analysis, we found that developers tended to have higher confidence to identify the occurrence of some design problems when using agglomerations. That happens because some developers analyzed each agglomeration's smell before reporting a design problem. Consequently, the likelihood of reporting false positive decreases. This behavior did not happen when participants used non-agglomerated code smells. In most of the cases, they analyzed only one code smell, which increased the likelihood of reporting false positives.

Our results also indicate that developers need better heuristics to support the analysis of code smells and agglomerations. For instance, the developers need to prioritize smells and agglomerations that are most likely to indicate a design problem. Prioritization algorithms are required because the analysis of smells is difficult and time-consuming. Thus, developers should focus on those that are most likely to indicate design problems. In addition, we also noticed that developers need proper visualization mechanisms to support the analyses of hierarchies and packages. Additionally, some agglomerations are widely spread in the source code; a single agglomeration may contain code smells located in multiple class hierarchies. Thus, developers have a large program scope to analyze. They may face difficulty to visualize how the code smells are related in the agglomeration. A graph-based visualization can help developers to figure out how the code smells are related to each other in the agglomeration.

On the one hand, the results of our study encourage the use of smell agglomerations to identify design problems. On the other hand, the results indicate the need to investigate other symptoms. We found that code smells do not suffice to help developers during the identification of design problems. Since smells may not be the only type of symptom that developers use in practice, we should investigate whether developers can identify design problems using other symptoms. However, instead of proposing developers to use other symptoms, we should investigate what else they use to identify design problems. Thus, our next step is to investigate in practice how developers identify design problems. We expect from this investigation to find the symptoms that developers use and how they use these symptoms during the design problem identification.

5
Investigating How Developers Identify Design Problems

In the previous chapter, we investigated whether developers can use code smells to identify design problems. For such investigation, we asked them to identify design problems in two software systems. In that study, we were interested in finding out if code smells suffice to help developers to identify design problems. For that investigation, we compared developers using code smells with developers using agglomerations. Our results indicate that agglomerations improve developers' precision in identifying design problems. Despite the increase in precision, we noticed that agglomerations, and consequently code smells, do not suffice to help developers during design problem identification.

Discussions in our two previous studies (Chapters 3 and 4) shed light on how developers identify design problems with the support of code smells. For instance, we found that developers can reason about multiple smells to identify a design problem. Even though smells do not suffice to support them, the analysis of multiple smells gave developers confidence to confirm the occurrence of the design problem, which may help to explain why they focus their effort on refactoring elements with multiple smells (Section 3.4.1). Indeed, we found that when developers reason about multiple smells, they analyzed each smell as a complementary symptom of the presence of a design problem (Section 4.3.1). Hence, the more symptoms (smells) a code element has, the more confident developers were regarding the existence of a design problem. These findings led us to wonder whether the analysis of multiple symptoms helps developers in design problem identification. However, we need to find out first what other symptoms developers use in addition to code smells. To identify these other symptoms, we need to observe developers identifying design problems in their own systems. Such observation will help us to better understand the phenomenon, which is the design problem identification.

Unfortunately, there is limited understanding about how developers identify design problems in practice, in particular when the source code is the only artifact available in a project. Existing studies tend to focus on proposing solutions for assisting developers in identifying design problems (26, 52, 53, 38, 28, 47, 27, 46, 56, 45). However, such proposed solutions may be misaligned with how developers identify design problems in practice. For instance, most of these studies make oversimplified assumptions about the process of identifying design problems. They consider that developers would rely on a single type of symptom (*e.g.*, either code smells (53, 28, 47) or design principle violations (56, 45)) to infer the occurrence of a design problem. However, this assumption might not hold in real project settings, especially after we found that code smells do not suffice to help developers to identify design problems. Thus, it is very likely that developers use other symptoms in addition to code smells. In summary, we know little about how developers identify, in practice, design problems in source code.

To provide such necessary understanding, in this chapter we investigate how developers identify design problems in source code. To do so, we conducted a multi-trial industrial experiment with professional software developers from five different companies, where they had to identify design problems in their systems under development. In the experiment, we captured data on their behaviour by filming the environment, recording audio and capturing their computer screens on video. These data allowed us to conduct an in-depth qualitative analysis based on Grounded Theory (60). As a result, we have built a theory of design problem identification.

According to Stol and Fitzgerald, nascent research areas typically take the research-then-theory approach, whereas more mature areas rely on (and refine) theories to further advance the field (108). Aligned with this statement, the theory presented here offers insightful propositions and explanations on how design problems are identified, which can serve as a basis to improve the state-of-art. For example, while most studies address only one type of design problem symptom, the theory reveals that, in practice, developers rely on a heterogeneous set of symptoms. Thus, previous studies are misaligned not only for assuming that developers will use a single, dominant type of symptom, but also for not considering how they use these symptoms. Based on the theory, researchers can build solutions most suitable to help developers. For instance, we identified cases when developers consider a symptom useful to identify design problems. Thus, researchers can use this knowledge to build tools that prioritize helpful symptoms for developers.

The remainder of this chapter is organized as follows. Section 5.1 presents our research design. Section 5.2 summarizes the results in which our theory is grounded. Section 5.3 complements the theory with additional propositions concerning the developer. Section 5.4 presents how the theory can be used to drive research on identifying design problems. Section 5.5 and 5.6 present related work and threats to validity, respectively. Section 5.7 summarizes this chapter.

5.1 Research Design

This section presents the design of our study. Section 5.1.1 presents our research questions. Section 5.1.2 discusses the process to select systems and developers to participate in the study. Section 5.1.3 describes the experiment to answer our research questions. Section 5.1.4 presents the data that we provided for developers during the experiment. Finally, Section 5.1.5 describes the procedure to collect and analyze the data.

5.1.1 Research Questions

Several researchers have proposed solutions to help developers during the identification of design problems (26, 52, 53, 38, 28, 47, 27, 46, 45). However, they do not focus on explaining how developers identify those design problems. In general, the existing studies propose solutions that will help developers to identify design problems. Despite their contribution, they do not clarify "the mechanisms through which and the conditions under which [the cause-effect relationship] holds" (98, p. 8). Therefore, the support they provide for developers to identify design problems may be somewhat misaligned with developers' current practice.

We highlight that before proposing solutions that will help developers to identify design problems, first, we need to understand how they conduct the identification task in practice. By understanding this task, researchers will be able to build mechanisms most suitable for helping developers during the identification of design problems. Therefore, we need to investigate how developers analyze the source code in their quest for identifying design problems. Such investigation is necessary since we know little about how developers actually identify design problems in source code. Our knowledge is so limited that we do not even know what symptoms developers use to identify design problems. Indeed, one might expect that developers use other symptoms to identify design problems; especially after we found that code smells do not suffice to help developers to identify design problems (Chapters 3 and 4). Consequently, the first step to understand how developers identify design problems is finding what symptoms they use in practice, which leads us to the following research question.

RQ3. What are the design problem symptoms that developers use in practice?

Answering this research question provides us with knowledge about how developers identify design problems in practice. However, RQ1 does not suffice to shed light on the process of identifying design problems. In fact, understanding a phenomenon like the identification of design problems requires more investigation. For instance, we still need to investigate how developers use and analyze these symptoms, what factors influence them during the analysis, what activities occur during the design problem identification, and the like. Such understanding can be provided by a theory with explanations and understanding of concepts and factors that go beyond the mere observation of a phenomena (109, 110, 111, 112). To build this theory, we need to answer the following research question:

RQ4 How do developers identify design problems in practice?

By answering both research questions, we expect to determine and understand the symptoms, activities and factors that influence how developers identify design problems. To answer these questions, we conducted a multi-trial controlled experiment in different software companies. We then analyzed the collected data and derived a theory using Grounded Theory (GT). Such theory provides an overview, explanation and understanding on *how developers identify design problems in source code.*

5.1.2 Software Systems and Developers' Selection

We searched for software companies that could provide us with software systems and developers to conduct the experiments. We defined the following criteria to select the companies: experience of their developers, size in terms of number of developers in a project, application domain of their projects, and development process. We defined these criteria in order to achieve some heterogeneity while selecting companies from our industrial collaboration network, thereby balancing contextual diversity and convenience (113). Based on these criteria, we chose the following five software companies from the North and Northeast of Brazil:

- **Company 1:** The company is incubated at a northeastern university, to which it provides the management of academic registrations.
- **Company 2:** The company deals with renting and selling print devices, and it has a department specialized in software development.
- **Company 3:** The company develops technological solutions that vary from vehicle tracking to management software for small businesses.

Table 5.1: Companies Description

Company	Type	Domain	Programming Language	Provided Systems
1	Government	University Administration	Java	S1
2	Private	Software Factory	Java	S2
3	Private	Software Factory	Java	S3
4	Private	Industrial Automation	Java, Android, iOS	S4, S5, S6
5	Government	Government Administration	Java	S7, S8

– **Company 4:** The company provides software systems for industrial automation, which vary from Java Desktop systems to Android and iOS applications.

– **Company 5:** The company develops and maintains systems for the government administration of a state in the North of Brazil. The company provided us with two systems.

Table 5.1 shows more details about the companies. The first column indicates the company identification for the experiment. The second column indicates if the company is private or from the government. The third column shows the domain in which the company creates software systems. The fourth column shows all the programming languages used in the company. The last column indicates the systems that each company provided to be used in the experiment. We asked the companies' managers, some of whom were software designers, to suggest specific systems that met the following criteria:

1. Systems in different stages of design degradation;

2. Systems from different domains and with different sizes with respect to the number of modules and developers;

3. Systems that were not in their initial versions;

4. Systems developed in Java.

The first and second criteria allowed us to select systems with a diversity of design degradation and structure. The third criterion was to ensure that the system would have design problems since the likelihood that a system has design problems in their initial version is lower than when the system in some versions later. The last criterion was to ensure the consistency among the systems. As each selected company has to provide software systems, we selected Java projects given the popularity of the Java programming language[1]. Thus,

[1]`https://www.tiobe.com/tiobe-index/`

it would be easier to keep the consistency among the provided systems: all of them implemented in the same programming language. The selected systems are:

– S1 (71,327 LOC): It is responsible for handling academic registrations for all undergraduate courses in the university. It was developed using the Spring, JBoss Seam and Hibernate frameworks. It went into production in the second half of 2010, and it is still in use today. It requires maintenance tasks daily.
– S2 (11,729 LOC): It manages the company's services. It tracks arrivals and departures of print devices, and manages contract deals with the clients. It also controls the replacement and reusability of compatible machinery parts.
– S3 (9,666 LOC): It allows users to create personalized e-mails and websites for their own company.
– S4 (72,683 LOC): It supports the management of registry offices for audit and control from the Justice Court of Brazil.
– S5 (657,901 LOC): It is a computational solution for maintaining information on the patients' health status and their medical records.
– S6 (27,939 LOC): It is a system developed for keeping track of products in a production line.
– S7 (475,644 LOC): It is a legacy system to process tax and to control the entrance of products from a state in the North of Brazil.
– S8 (237,249 LOC): Developed for standardizing budget of a state in the North of Brazil.

After the companies' managers provided us with the systems, we asked them to indicate developers familiar with each one and who could act as subjects in the study. For conducting our study, the subjects were divided into teams. Table 5.2 presents the subject characterization and the corresponding teams. All teams are composed of two developers, except for T10, whose company asked us to involve three developers.

Table 5.2: Characterization of the Developers

Team	ID	Experience (years)	System	Company
T1	D1	3	S1	1
	D2	5		
T2	D3	13	S1	1
	D4	14		
T3	D5	14	S1	1
	D6	6		
T4	D7	7	S2	2
	D8	2		
T5	D9	4	S2	2
	D10	4		
T6	D11	10	S3	3
	D12	8		
T7	D13	12	S4	4
	D14	13		
T8	D15	4	S5	4
	D16	8		
T9	D17	4	S6	4
	D18	10		
T10	D19	7	S7	5
	D20	7		
	D21	9		
T11	D22	12	S8	5
	D23	9		

5.1.3 Experimental Tasks

The experiment comprised the following four activities: subjects characterization, training, problem identification, and follow-up questionnaire.

Activity 1: Subjects characterization. We asked the developers to fill out a questionnaire (Appendix C) to gather their information, including educational level, professional experience with software development, Java programming, and knowledge about design problems. Their responses allowed us to understand their individual characteristics, helping us to identify any gap in their knowledge or misunderstandings about main concepts used in the study, and to prepare for the training activity.

Activity 2: Training. We conducted a training session with all the developers about software design and design problems, with examples of problems pertaining to different categories. The following design problems were included in the training session: *Ambiguous Interface*, *Unwanted Dependency*, *Component Overload*, *Cyclic Dependency*, *Scattered Concern*, *Fat Interface*, and *Unused Abstraction* (Section 2.2). We selected these design problems together with the project managers, who suspected that these represented common cases of design problems in the selected projects. However, we made it clear to the developers that they were also allowed to identify other types of design problems with which they were already familiar. The 40-minute training session was organized in two parts: a Powerpoint-based presentation; and discussions and questions. The presentation used in the activity is available in Appendix B.4.

Activity 3: Problem identification. We asked developers to identify design problems in their software systems. They had 90 minutes to analyze the source code to identify all the design problems they could find. At the beginning of this activity, we asked them to explain aloud what they were doing while we recorded the task on video. Thus, we could triangulate the results from the questionnaire and the video recording to improve the data analysis.

Activity 4: Follow-up questionnaire. Developers filled out a questionnaire about their perception of the task. We also asked them to indicate whether each symptom was useful to identify a design problem. The questionnaire used in the activity is available in Appendix C. The answers were also used to complement the qualitative analysis.

5.1.4 Provided Data

For the identification of design problems, developers need to locate *symptoms* of design problems directly on the source code, such as code smells (21). Similar to what happens with code smells, other types of symptoms may manifest in source code due to the presence of a design problem. Consequently, many developers use tools to identify these symptoms. Thus, to make Activity 3 more realistic, we provided our subjects with a set of possible symptoms we had detected in the code of the analyzed systems after running and manually combining the output of some tools (114, 28, 78); simulating the use of tools, but not limiting the developers to the output of a specific one.

However, we highlight that we did not know beforehand what symptoms developers use in practice. In fact, we have a research question to address this matter. Thus, the selection of symptoms that developers would receive happened gradually, as we were noticing them using other symptoms. First, we provided only the code smells, since we knew that developers can use code smells to identify design problems. Additionally, code smells have been discussed in the literature as a consistent indicator of design problems (53, 28, 47, 51, 27, 55, 29). They are also part of the developers routine (29). Therefore, code smells were the first type of symptoms that we provided to developers. After running the experiment with some developers, we noticed that they start to use other symptoms. Thus, we provided these other symptoms when we ran the experiment with other developers. We discuss the symptoms that developers used in practice in the context of our first research question (Section 5.2.2).

We summarized and presented these symptoms to developers through a web page based on SonarQube (114) (Appendix C). We provided a visualization similar to SonarQube because it is a well-known platform for inspection of software systems, and it was familiar to most subjects. Thus, we could reduce the learning curve to how the symptoms are presented. The main difference of our mechanism is that it presents all symptoms in a single page. It is noteworthy that, while SonarQube provides several pieces of information unrelated to design problems, our mechanism provides only symptoms that may help developers to identify design problems.

5.1.5 Data Collection and Analysis

We used different instruments to collect data. The developers had to answer characterization and follow-up questionnaires. They also had to write any observation in a specific field at the web page in which we presented the symptoms. They could write anything in the observation field, such as: the name of a design problem affecting the elements; whether he agreed with the suggested symptoms; or even comments about the code. They used either the Eclipse or IntelliJ IDEs to analyze the source code, and they used the browser to access the web page with the symptoms. We used the think-aloud method (116), asking the developers to verbalize their thoughts during the experiment. All their procedures were recorded on audio and video. We used Techsmith's Camtasia[2] to record audio and screenshots of their computer. In addition, a video camera was installed in the room to record the developers during the study.

Figure 5.1 summarizes the process to collect and analyze data, which can be divided into three phases. The two first phases concern to collect and analyze data using Grounded Theory and the third phase concerns the representation of the resulting theory. These phases are explained next.

[2]Camtasia is available at www.techsmith.com/camtasia.html

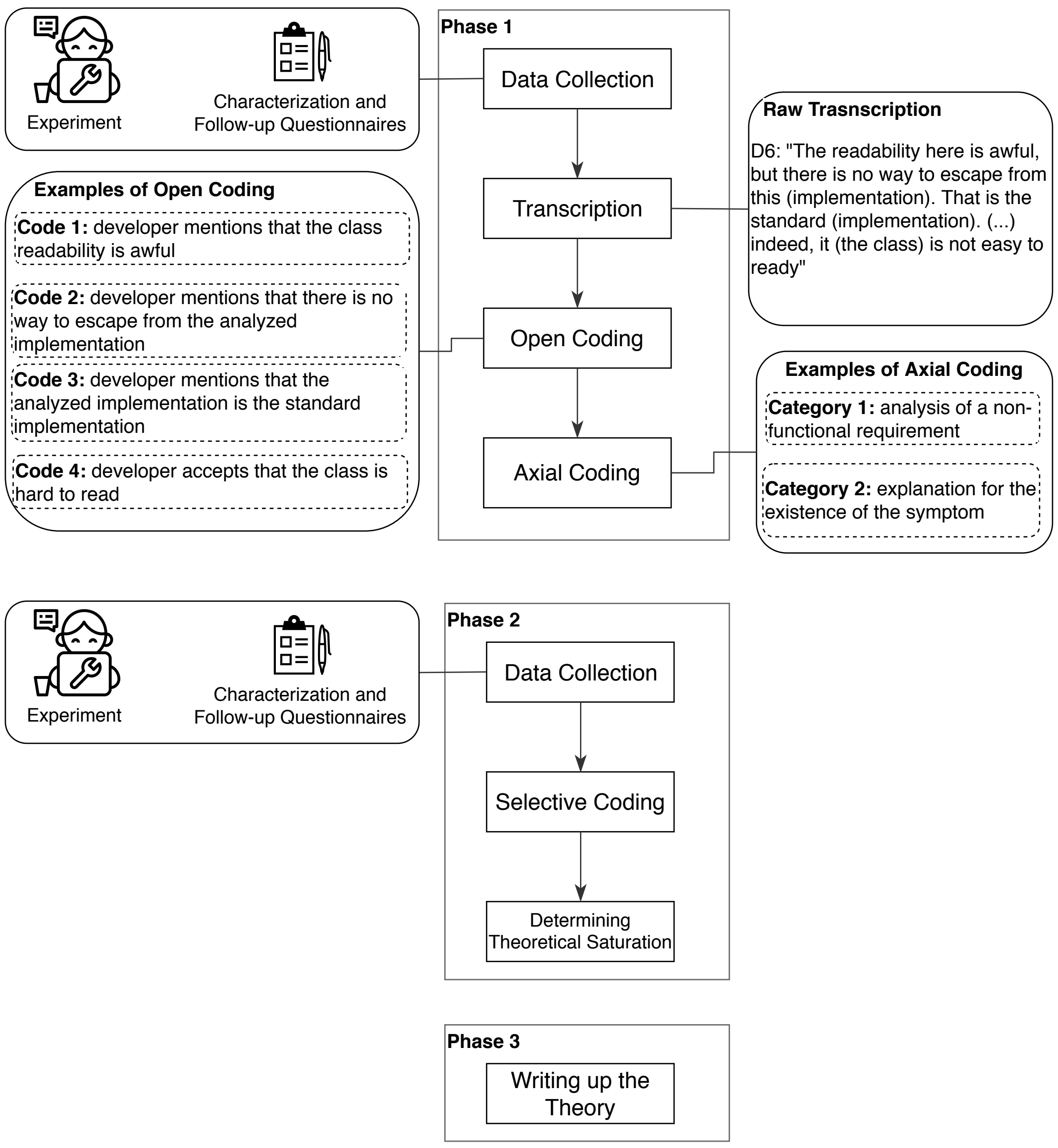

Experiment
Characterization and Follow-up Questionnaires
Phase 1
Data Collection
Transcription
Open Coding
Axial Coding
Raw Trasnscription
D6: "The readability here is awful, but there is no way to escape from this (implementation). That is the standard (implementation). (...) indeed, it (the class) is not easy to ready"
Examples of Open Coding
Code 1: developer mentions that the class readability is awful
Code 2: developer mentions that there is no way to escape from the analyzed implementation
Code 3: developer mentions that the analyzed implementation is the standard implementation
Code 4: developer accepts that the class is hard to read
Examples of Axial Coding
Category 1: analysis of a non-functional requirement
Category 2: explanation for the existence of the symptom
Experiment
Characterization and Follow-up Questionnaires
Phase 2
Data Collection
Selective Coding
Determining Theoretical Saturation
Phase 3
Writing up the Theory

Figure 5.1: Research Process to Collect and Analyze Data

5.1.5.1 Grounded Theory

We applied the principles of Grounded Theory (GT) to further understand and explain how developers identify design problems in source code. GT is a qualitative research method that uses a systematic set of procedures to inductively develop a theory about a phenomenon (60), which the theory emerges from the data. GT is used to understand the action in a substantive area from the point of view of the actors involved in a phenomenon. (117). There are different versions of GT (118), and we chose to adopt Strauss's and Corbin's GT (60), since it allows us to ask questions about the conditions upon which a phenomenon occurs (118).

GT contains three coding procedures: *open coding*, *axial coding*, and *selective coding*. Coding refers to the task of data analysis (60, 118). Open coding involves the breakdown, analysis, comparison, conceptualization, and categorization of the data. Axial coding consists in examining the identified categories to establish conceptual relations between them. Finally, in selective coding, we further refine the categories and relations, and identify the core category to which all others are related. In this procedure, we aim to reach the theoretical saturation, *i.e.*, the point at which the theory is well supported and new data do not longer trigger reinterpretations.

In the first phase (Figure 5.1), we collected the data through the experiment as explained in Section 5.1.3. In this phase, we conducted the experiment with two companies. After the data collection, we employed GT procedures to analyze the data. We first transcribe all the video and audio recordings. We then performed *open coding* to associate codes with quotations of developers' utterances, as shown in the example below:

Raw Transcript. *"D6: The readability here is awful, but there is no way to escape from this (implementation). That is the standard (implementation). (...) indeed, it (the class) is not easy to ready"*
Code 1. developer mentions that the class readability is awful
Code 2. developer mentions that there is no way to escape from the analyzed implementation
Code 3. developer mentions that the analyzed implementation is the standard implementation
Code 4. developer accepts that the class is hard to read

We related the codes through *axial coding*. In this procedure, the codes were merged and grouped into more abstract categories, and the type of relation (60) was established. For instance, the previous codes were grouped into the following two categories:

Category 1. analysis of a non-functional requirement
Category 2. explanation for the existence of the symptom

After the axial coding in Phase 1, we generated 201 codes (these codes and the relation among them are available in Appendix C.3). We noticed that some codes did not change anymore, for instance, the codes representing the type of symptoms that developers use. However, we also noticed that these codes did not explain completely how developers used these symptoms. Hence, if we ran the experiment with other teams, we could find more information to explain how developers used the symptoms. Therefore, we concluded that we did not reach theoretical saturation. Consequently, we had to conduct additional experiments with more companies.

In Phase 2, we selected more companies that could provide us with systems and developers (Section 5.1.2). Hence, we replicated the experiment with three other companies. First, we collected the data, then we transcribed and conducted the open and axial coding (omitted in the figure). Next, we used *selective coding* to identify core categories that best explain how developers identify design problems. We conducted Phase 2 until we reached the theoretical saturation. In the end of this phase, we generated a total of 1,161 codes (all codes and the relation among them are available in Appendix C.3).

To determine the theoretical saturation, we had to decide whether the theory's components were well supported and new data would no longer need revisions or reinterpretations of the theory. We reached this conclusion after running the experiment with companies 3 and 4. When we analyzed how the T8 team identified design problems, we started to notice that the codes did not change anymore. Actually, we found that the new codes only provided further details about the analysis of symptoms. We confirmed that new data would not trigger the reinterpretations of the theory after analyzing the T9 team. Before claiming to have reached the theoretical saturation, we decided to run the experiment with another company. We have not found any other code that would trigger revisions or reinterpretations of the theory after we analyzed T10 and T11 teams. Thus, in agreement with the other reviewers, we concluded that we have reached the saturation. The refinement of the codes among the companies is available in Appendix C.3.

After determining the theoretical saturation, our next step was to write up the theory in a way that could be described according to constructs (basic elements) and propositions (the interaction between constructs).

5.1.5.2 Peer Review Process

For each transcript, the codes, memos, and networks showing the relations in the categories and codes, we peer-reviewed and changed upon agreement with some researchers, who collaborated with the study. Figure 5.2 shows an overview of the peer review process to collect and analyze data, which can be divided into four activities. First, four researchers ran the experiments. These researchers were grouped in pairs; thus, a researcher could help the other during the experiment and support each other to avoid bias. The next activity consisted of the transcription of the videos. This activity was conducted by two developers. Eleven videos, one for each team, were divided between the two researchers, who had to watch the video and transcribe it. After the transcription, each researcher reviewed the transcription of the other – which is represented in Figure 5.2 by the dotted arrow. Eventually, a third researcher reviewed all transcriptions.

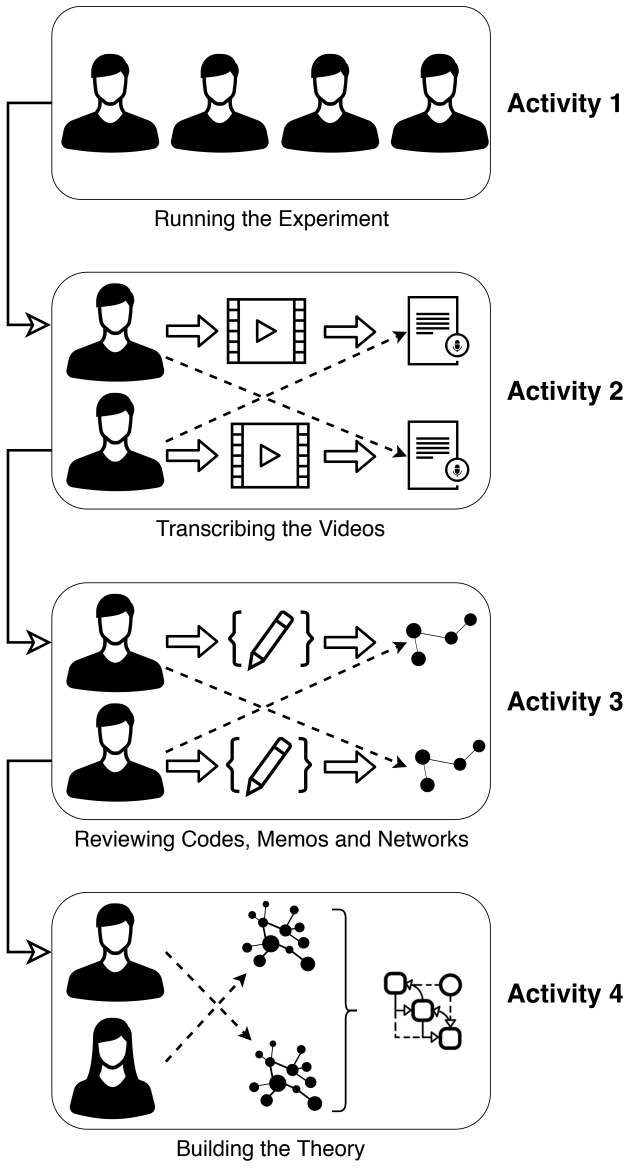


Figure 5.2: Peer Review Process to Collect and Analyze Data

In the third activity, the two previous researchers conducted the open coding, axial coding and selective coding as discussed in the previous subsection. After the coding, each researcher reviewed the code, memos, and networks created during the data analysis. This activity was mainly conducted by two researchers, however, they had the support of other two researchers to avoid bias during the interpretation of data. Several meetings happened to discuss the codes, memos, and networks. Finally, in the fourth activity, the researchers built the theory. In this activity, two researchers, who were the main leaders on the theory construction, compared their resulting theory. This comparison was made together with other researchers who were involved in previous activities. Thus, we were able to merge both resulting theories to define the final one. We conducted several meetings to build the theory, which was important to avoid distortions when interpreting the data.

5.1.5.3 Writing Up the Theory

In the end of Phase 2, we had artifacts (transcripts, codes, memos, and networkings) to explain how developers identify design problems. Our next step was to represent these artifacts as a theory. According to Sjøberg *et al.* (112), Software Engineering (SE) theories should not be useful only as an academic exercise, but also to the software industry. Thus, in Phase 3, we used their framework to writing up the theory in a way that both research communities and industry may benefit from using it.

Following their framework, our first step was to divide the description of our theory into four parts: *constructs* (basic particles that compose a theory), *propositions* (interaction among constructs), *explanations* (factors behind propositions) and *scope* (the universe of discourse in which the theory is applicable). For this step, we had to map the artifacts found after applying the GT into these four parts. The categories identified in the axial and selective coding became eligible candidates to become constructs and propositions. The memos, codes and the interaction among the codes became the baseline for the explanations. Since we experimented with software systems from different companies, the scope is delimited by closed-source software systems.

After the mapping, our next step was to use Sjøberg *et al.*'s diagrammatic notation to describe the constructs, propositions, and scope of a theory. In this notation, we relate the constructs to one of four archetype classes: *actor*, *technology*, *activity*, and *software system*. According to their framework, the typical SE situation is one which an actor applies technologies to perform specific activities on an (existing or planned) software system. At the end of this step, we were able to write up the data into a theory.

5.2 A Theory on How Developers Identify Design Problems

In order to answer our research questions, we conducted a multi-trial controlled experiment to investigate how developers identify design problems in practice. As a result, we derived a theory describing the activities and factors that influence how developers identify design problems. Next, we answer our research questions. The first research question ("*What are the design problem symptoms that developers use in practice?*") is answered in Section 5.2.2. The second research question ("*How do developers identify design problems in practice?*") is answered while we explain the theory's constructs and propositions throughout the next sections.

As aforementioned, we used Sjøberg's framework (112) to describe the theory, which is summarized by a visual representation in Figure 5.3. Adopting their terminology, the diagnosis is the process (*technology*) that the developer (*actor*) applies to identify design problems (*activity*) in the source code (*software system*). Regarding the *scope*, the theory is supposed to be applicable in closed-source software systems in which developers intend to identify design problems by analyzing symptoms that manifest themselves in source code.

According to Sjøberg *et al.*'s framework, our theory fits in the *Explanation type* since it describes and explains how the identification of design problems is conducted (Section 5.2.1), the symptoms and their characteristics (Section 5.2.2), and how the symptoms are used to diagnose design problems (Section 5.2.3). In the framework, they also describe criteria to evaluate theories. Testability is one criterion, which indicates "the degree to which a theory is constructed such that empirical refutation is possible." Regarding such criterion, our theory has high testability since empirical refutation of its propositions is possible by replicating the study. In fact, such replication is feasible given that we first conducted the experiment with two companies and then replicated it with three more companies until reaching theoretical saturation.

When describing the theory, we introduce the constructs and propositions, identifying them in the text with **C** and **P**, respectively. We answer our research questions while we discuss the propositions and their constructs next. We also present explanations for propositions that are aligned with findings of previous studies and explanations that comprise findings that have not been presented elsewhere. Complete description of the constructs and propositions is available in Table C.1 (Appendix C.4).

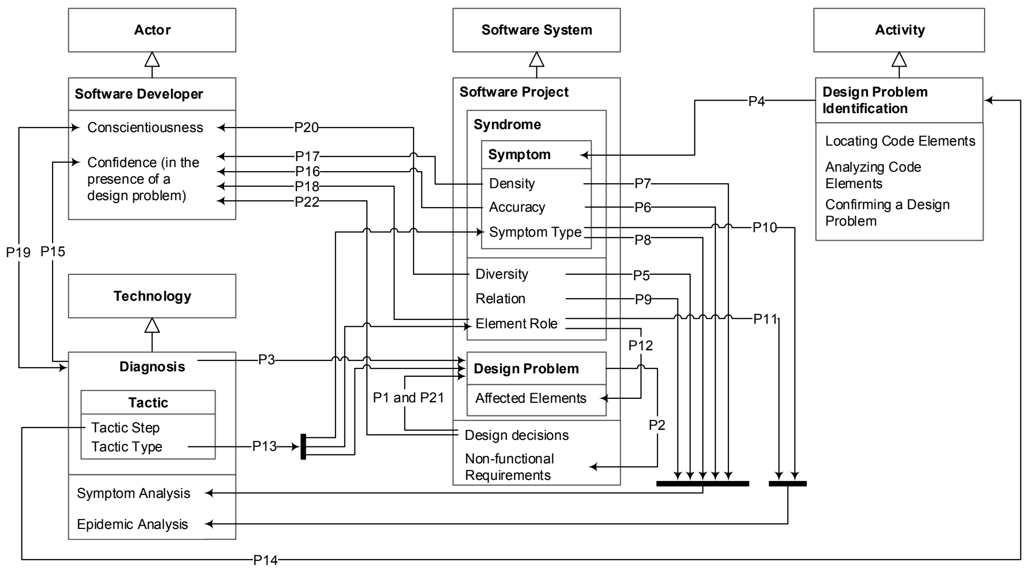


Figure 5.3: Visual Representation of the Theory

5.2.1 Identification of Design Problems

A *design problem* (**C1**) arises in code elements due to one or more *design decisions* (**C2**), made intentionally or accidentally (**P1**). In fact, when design decisions impact non-functional requirements negatively, we state that a design problem exists (**P2**). A design problem may affect one or more elements (**C3**) in such a way that these elements manifest symptoms of its presence. A *symptom* (**C4**) is an indication of the presence of a design problem.

Three Steps to Identify Design Problems Using Symptoms. A code element may contain several design problem symptoms. Thus, we define a *syndrome* (**C4**) as a set of symptoms affecting the same code element. In this context, we refer to *diagnosis* (**C6**) as the process of identifying a design problem through the analysis of symptoms that manifest themselves in source code (**P3**). From the data collected during the subjects' diagnostic activities, we noticed that the identification of design problems was often divided into three steps: (i) locating code elements, (ii) analyzing the elements, and (iii) confirming or rejecting the presence of a design problem. In all these three steps, developers rely on design problem symptoms that manifest in source code (**P4**).

5.2.2 Design Problem Symptoms

Towards understanding how developers identify design problems in the source code, we defined our first research question ("*What are the design problem symptoms that developers use in practice?*"). At the end of the study, we found that developers use the following symptoms:

1. **Violation of Non-functional Requirements**: Information of non-functional requirements (*e.g.,* `readability`, `testability`, `robustness`, `security`), which were possibly being violated;

2. **Code Smells**: A code smell is a microstructure in the system that represents a surface, sometimes only partial, indication of a design problem (21);

3. **Violation of Architectural and Design Patterns**: Information on the use of architectural and design patterns (4), to help identify misused patterns;

4. **Poor Structural Quality Attributes**: A structural quality attribute is a characteristic or feature that reflects properties of a software system. Examples include `coupling`, `cohesion`, `size` and `complexity`. These attributes can be measure using software metrics;

5. **Violation of Object-Oriented Principles**: Information about object-oriented principles (115) that were possibly being violated, which may indicate a problem. These principles have been pointed out as guides to avoid design problems. The principles include `encapsulation`, `abstraction`, `inheritance` and `polymorphism`. Other principles are also taking into account, for instance, the S.O.L.I.D principles (13);

As far as we know, we were the first one to investigate what the symptoms that developers use in addition to code smells. Interesting enough, this investigation happened gradually as we observed each developer during the design problem identification task. As explained in Section 5.1.5, we ran the experiment individually with each team. We made clear that they were free to use any design problem symptom in their quest for design problems. As we experimented with more teams, we noticed that they started to use other symptoms that have not been provided by us. After that, we made these other symptoms available for other teams. In other words, the selection of symptoms that developers would receive happened gradually, as we were noticing them using other symptoms. For instance, the two first teams received only the

Table 5.3: Helpfulness According to Developers

Symptom	Applied times	N°. of contributions	Percentage of success
Design Pattern Violation	43	34	79.07%
Quality Requirements	43	31	72.09%
Violation of Non-functional Requirements	62	46	74.19%
Code Smells	37	17	45.95%
Violation of Object-oriented Principles	38	20	52.63%

code smells. Then, we noticed that they started to use other symptoms, such as poor quality attributes and violation of object-oriented symptoms. Since they used these symptoms, when we ran the experiment with other teams, we added these other symptoms in addition to the code smells. Here in this section, we will discuss about all these symptoms in general. We will discuss each one individually when we explain how developers use them to identify design problems (Section 5.2.3.3).

Symptom helpfulness. We noticed that developers do not always consider all the symptoms of a syndrome when identifying design problems. Instead, they only consider those symptom instances that they judge helpful during the identification. We could identify when developers judge a symptom helpful because we asked them to evaluate the symptom based on how helpful it was to identify the problem (Section 5.1.3). Table 5.3 presents the percentage of helpfulness of each type of symptom. This percentage is calculated only when the team used the symptom. The first column indicates the name of the symptom, while the second column shows the number of times that the symptom was used by developers. The third column shows the number of occasions that the developers mentioned the symptoms were helpful to identify a design problem. Finally, the last column indicates the percentage of helpfulness, *i.e.*, the percentage that developers used the symptom and evaluated it as helpful.

Symptom attributes that drive developers to select what to analyze. Based on the helpfulness mentioned by developers, we performed a qualitative analysis to investigate which symptom attributes *i.e.*, characteristics of the symptom (such as its accuracy or type), developers take into consideration when they choose the symptoms most likely to help them. We observed that the following symptom attributes are most helpful for developers to identify design problems: symptom type, accuracy, density, relation (among the symptoms), and diversity. Symptom *type* (**C7**) indicates a category to which a set of symptoms with common characteristics belongs (*e.g.*, code smell). *Accuracy* (**C8**) is the degree to which a symptom is correct in indicating a design problem, while *density* (**C9**) is the number of symptom instances in a syndrome. Regarding these attributes, we were already expecting that accuracy and density would be attributes that developers take into account to consider a symptom helpful. However, we had not expected that they would take into consideration the relation among symptoms and the diversity of symptoms in the syndrome.

Diversity of a syndrome. *Relation* (**C10**) is how two or more symptoms are connected to each other. For instance, both `Intensive Coupling` smell and `violation of the layered pattern` (119) measure the degree to which elements are undesirably coupled with others. Since they measure similar (albeit complementary) properties of an element, they are related to each other. We noticed that developers use the relation among the symptoms to discover other types of symptoms that can indicate a design problem (Section 5.2.3.1). We also noticed that developers frequently located elements that manifested several different types of symptoms (**P5**). In fact, we observed that *diversity* (**C11**) is another attribute that developers consider. Diversity is the degree to which a syndrome contains a variety of symptom types. Upon analysis, we found that the more different types of symptoms a syndrome has, the greater the chance the developer will identify at least one design problem in the element (Section 5.3.2).

Indeed, the diversity of a syndrome has a strong influence on the diagnosis. As this finding had not been observed before, studies that assume that developers rely on only one type of symptom (44, 47, 56, 45) may be misaligned with how diagnosis is conducted in practice, in two ways: (i) they may assume that developers will use a predefined, dominant type of symptom; and (ii) they may not consider the diversity of symptoms as another indicator for identifying design problems. We discuss in Section 5.3.2 how diversity influences human aspects.

Considering the attributes that influence the developers. As mentioned before, developers do not consider all symptom instances to identify a design problem. They take into account only those symptoms that they consider helpful to identify a problem. We showed the attributes developers consider to assume that a symptom is helpful. For instance, if a syndrome has several types of symptoms, developers consider the density and diversity to select the symptom. In other words, developers select a symptom when they are satisfied with these attributes. Conversely, when attributes do not satisfy the developers (*e.g.*, the syndrome does not contain diverse types of symptoms), the symptom would be ignored and not considered helpful, possibly leading to missing a design problem.

Knowing about how developers consider a symptom helpful is useful for researchers since they can propose solutions that emphasize helpful symptoms for the developers. For instance, some studies propose solutions to prioritize smells that can help developers to identify design problems (49, 47, 46). As code smells are a type of symptom, they also present some of the attributes discussed above. However, some studies on code smells may not consider attributes as the density of smells or diversity. Therefore, developers may neglect some code smells for not considering them helpful for the identification. These studies could use the attributes that developers take into account as a mechanism to prioritize smells (Section 5.4).

5.2.3 Design Problem Diagnosis

As aforementioned, diagnosis is the process of identifying a design problem through the analysis of symptoms. We noticed that developers diagnose a design problem based on two types of analyses: a *symptom analysis* (**C12**), and an *epidemic analysis* (**C13**).

5.2.3.1 Symptom Analysis

In symptom analysis, developers choose and analyze a set of symptoms affecting a single element, *i.e.*, they do not analyze multiple elements. This happens because they usually rely on the aforementioned symptom attributes: type, accuracy, density, relation (among symptoms), and diversity. In this analysis, developers verify, based on these attributes, whether the symptoms affecting the analyzed element indicate a design problem. If so, then they do not proceed to analyze other elements.

Incorrectly ignoring symptoms. Someone can expect that the accuracy (**P6**), the density (**P7**), and the type (**P8**) of the symptoms influence problem identification. For instance, let us consider code smells. Palomba *et al.* investigated to what extent code smells are perceived as design problems (51). They noticed that developers consider the type of the code smell to decide whether it is a problem. Surprisingly, developers tend to incorrectly associate the type of symptom with its accuracy or density, and that does not happen only with code smells. Thus, if they rely on the accuracy or density to disagree that a symptom indicates a design problem, they tend to not consider the same type of symptom in the other elements, even when they actually

Table 5.4: Combining Symptoms

Symptoms	Instances	Design Problems	Teams
1	16	11	T7, T8, T9
2	13	10	T1, T3, T7, T8, T9
3	14	11	T3, T4, T5, T6, T7, T8, T9
4	10	6	T2, T3, T5, T8, T9
5	3	1	T4

indicate a design problem. For instance, if a developer analyzes the violation of a design pattern such as `Data Access Object` and concludes that this type of symptom is irrelevant for identifying a design problem, then he is less likely to consider a violation of a design pattern in the elements he analyzes next. This happened, for instance, with the T3 developers.

Combining multiple related symptoms. Someone can argue that analyzing a single element is not enough to identify a design problem. Nevertheless, we noticed that they combine symptoms in a single element in order to confirm the presence of a design problem. Table 5.4 shows the frequency with which developers either used only one symptom or combined multiple symptoms. Its first column indicates the number of symptoms that developers combined to identify design problems. Its second column indicates how many times the symptom or combination of symptoms happened. Its third column indicates the number of design problems found when the subject used a symptom or a combination of symptoms. Its last column indicates the teams that used or combined symptoms. We obtained these data after applying the GT. Table C.2 (Appendix C) shows a full version of this table containing (i) which symptoms were combined and (ii) which design problems the team found.

We can see in Table 5.4 that most developers tend to combine symptoms to identify a design problem. Also, we noticed that developers identify more design problems when they combine symptoms. Based on this result, we investigated how the combination takes place. We noticed that developers use symptom relations to identify the symptoms to combine. Thereby, the relation helps developers to identify other helpful symptoms in the syndrome. Therefore, the more related to others a symptom is, the greater the likelihood of a developer selecting it for combination (**P9**).

As an example of how developers use the symptom relation to find other helpful symptoms, let us consider the developers of the T2 team. They were analyzing the code smells, and they noticed that the class had the `Dispersed Coupling` smell. Due to the presence of this smell, they analyzed the `coupling` quality requirement. When analyzing this type of symptom, they noticed it was indicating a high coupling with other classes. This finding increased their confidence that the class contained a design problem. These developers also noticed that the coupling was related to a third type of symptom: violation of non-functional requirements. When they analyzed this symptom, they noticed that the high coupling was making the class harder to read. In this example, the developers used the relation among the three symptoms (code smells, quality requirements, and violation of non-functional requirements). Then they combined these symptoms to identify that the element was involved in the Concern Overload design problem (7).

The relation among the symptoms is what drives the combination, by helping to identify other related symptoms. The combination of the symptoms is another evidence that previous studies (44, 47, 56, 45) may have proposed solutions for the problem identification that do not fit the developers' needs. In other words, developers consider multiple symptoms (Section 5.2.2) and they also combine these symptoms to increase their confidence in the presence of a design problem. Therefore, forcing the developers to use only a reduced set of symptoms is likely to go against the way in which developers identify design problems in practice.

5.2.3.2 Epidemic Analysis

When developers analyze an element, they do not consider only the symptoms affecting that element; sometimes they also consider whether other elements are affected by the same set of symptoms. We name this an epidemic analysis. Analogously to the way in which attributes influence the selection of symptoms in a single element, there are attributes that developers consider before choosing elements for an epidemic analysis. In addition to considering the types of symptoms (**P10**), developers also take into account the *element role* (**C14**) to choose the epidemic elements most likely to help them to identify a design problem (**P11**). Element role is the function that an element plays in the software system, *e.g.*, the role of Service.

Complementary analysis. The reason why developers use the element role to identify epidemic elements is that each design problem may be scattered over several elements. Since those elements share the same symptoms, developers assume that they may help them to identify the design problem, which justifies the epidemic analysis. However, a surprising finding is that developers analyzed epidemic elements only when they had used the symptom analysis but had not succeeded in identifying a design problem. Since they are not confident about the presence of a design problem during the symptom analysis, they proceed to the epidemic analysis of other elements in order to decide whether there is a design problem in the elements under analysis.

Prioritization of key elements. Developers tend to prioritize epidemic elements that provide a central functionality in the system. This happens because they associate the role played by the element with the probability of the element containing a design problem (**P12**). As an example, the T2 developers were analyzing the symptoms of an element. During the analysis, they noticed that the element was playing the Service role in the system. At this point, the developers included other Service classes in the analysis. When they focused the analysis on all the classes that play a service role, this change of focus led them to identify a design problem. They mentioned that all the service classes in the systems are affected by the Scattered Concern design problem (7). Curiously, these developers had already analyzed other Service classes before, without identifying any design problem. In that case, the T2 developers were not applying the epidemic analysis; thus, they did not take into account the element role to select elements with similar set of symptoms. Developers are more likely to accept that elements playing an important role have a design problem. However, we found cases in which subjective factors influenced their decision, as discussed in Section 5.3.

5.2.3.3 Identification Tactics

Regardless of the type of analysis that developers conduct, they rely on symptoms in the code elements to identify a design problem. Indeed, as previously discussed, the identification of design problems was often divided into three steps. Developers locate a code element in the first step; then, they analyze the element (or elements) in the second one; and finally, they confirm or reject the presence of a design problem in the element(s). In these steps, the action is the same: the analysis of code elements and their symptoms. *Tactic* (**C15**) is how we name this action, *i.e.*, a tactic is a specific action, in which developers rely on the analysis of the elements and their symptoms towards to the identification of a design problem.

A tactic can be characterized by two attributes: the type of the tactic and the moment (step) to which the developer applies it. The *tactic type* (**C16**) indicates the action that developers apply in their quest for design problems. We noticed two main actions: either (i) developers search for a specific design problem or a specific element, or (ii) they rely on the analysis of a specific symptom to identify a design problem (**P13**). After the qualitative analysis, we noticed that the developers used six types of tactics: **smell-**

based, **problem-based**, **principle-based**, **element-based**, **NFR-based**, and **pattern-based tactics**. They apply each tactic in a specific moment during the analysis. The moment that developers apply the tactic – which we named of *tactic step* (**C17**) – refers to one of the three steps in the design problem identification (**P14**). In other words, developers apply the tactic either (i) to locate code elements, (ii) to analyze the elements, or (iii) to confirm or reject the presence of a design problem.

Figure 5.4 presents the six types of tactics, represented by a yellow rectangle, and how they are related to the constructs in our theory. We represent the action that defines the tactic type in green, and we represent the moment that the tact is usually applied (*i.e.*, the tactic step) in blue. As we can notice in the figure, the tactic type is either associated with the search for a specific design problem our element or associated with the analysis of a symptom type. We describe each tactic as follow.

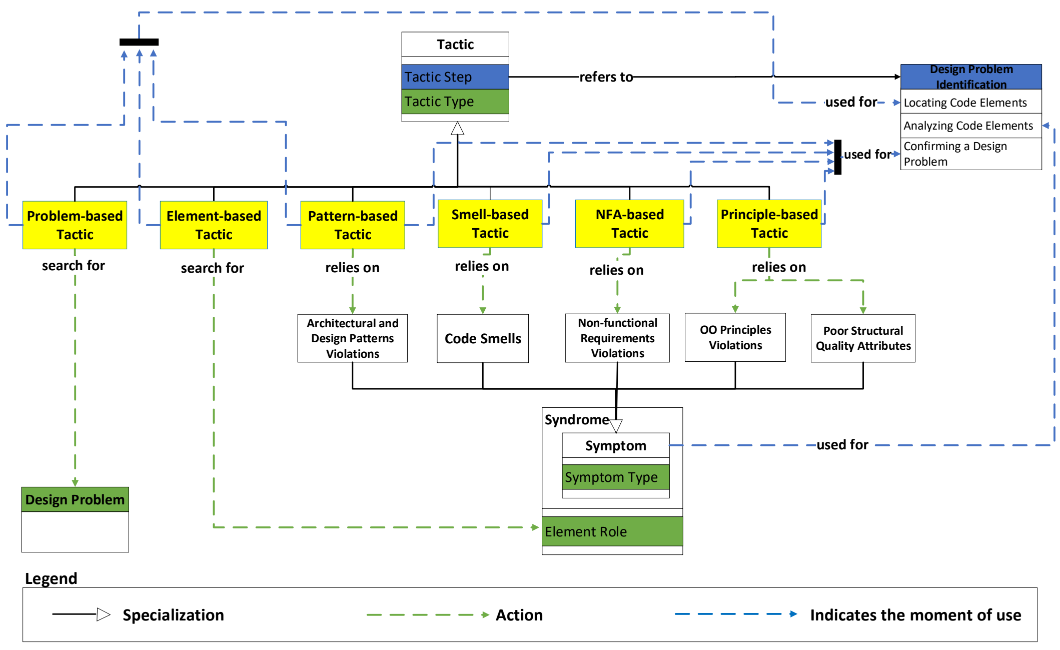


Figure 5.4: Identification Tactics

Smell-based tactic is the tactic in which the developers rely on code smells to identify design problem. As mentioned by other studies (53, 52, 38), developers can use smells as a symptom of design problems. Developers used the smell-based tactic to confirm the existence of a design problem. They marked elements as having a design problem whenever they were analyzing an element, and they noticed that it had a code smell. We also observed that developers explored some types of code smells that were not in the provided initial list. For example, they explored the number of switch statements in methods (`Switch Statements` smell) when they were analyzing certain classes. Similarly, they mentioned that some classes have similar code snippets (`Duplicate Code` smell). As an example, D21 developer identified a `God Class` smell even though the instance of the smell was not in the provided list. After finding the smell, the developer confirmed the occurrence of a design problem in the class.

Principle-based tactic is the tactic that the developers used poor structural quality attributes (*e.g.* `cohesion`, `coupling` and `complexity`) or the violation of object oriented principles (13) (*e.g.* `open-closed principle` and `information hiding`) to identify design problems. They mainly used this tactic to confirm if an element under analysis has a design problem. In this case, they marked the element as having a design problem when they noticed that a class under analysis was violating a design principle. This case happened with D22 developer.

Problem-based tactic is the tactic in which the developers searched for occurrences of a specific type of design problem they already had in their mind in the source code. We classify that a subject used the problem-based tactic when he explicitly mentions he was looking for a specific type of design problem across the system. We observed that they tended to focus on searching for design problems related to interfaces and components (realized as packages in the source code). For instance, *Fat Interface* (13) and *Component Overload* (47) were problems that developers identified with high frequency. On the other hand, we did not observe developers looking for problems related to abstract concepts, such as *Delegating Abstraction* (47) and *Unused Abstraction* (47).

Element-based tactic is the tactic in which the developers selected specific code elements to investigate if it is affected by a design problem. They do not necessarily reason about specific types of design problems, but they look for any sort of indication (*e.g.*, frequent modifications or the element role in the system) in those elements that may signal the manifestation of a design problem. In this tactic, the developers focused their reasoning on code elements – such as core classes, interfaces, and hierarchies – that represent key design abstractions in the program. Given the relevance of such elements to the system, developers knew these elements could form structures realizing a design problem in the implementation. Thus, they directly started inspecting these code elements and reasoning about their symptoms.

Developers often knew already which code elements they should analyze first. Interestingly, most of these cases were classes: we expected developers would also analyze often interfaces and packages given their relative importance to the design. However, such elements were rarely analyzed. Moreover, there were a few cases in which they had to determine a criterion to choose such elements explicitly. For example, one of the developers chose a class based on the number and nature of variables and methods located in the class. Another subject decided to limit the search to classes within specific subsystems. He picked a subsystem that was visibly large regarding the number of classes. The same subject also suggested restricting the search to a generic subsystem. All classes that did not belong to any other specific subsystem were created in or moved to this subsystem.

NFR-based tactic is the tactic where the developers reasoned about non-functional requirements that are negatively affected by certain design decisions. They reasoned how a design decision explicitly hinders one or more quality attributes. Again, they did not necessarily reason about specific types of code smells or design problems. The developers used this tactic when they were analyzing a code element, and they noticed that its implementation impacted one or more non-functional requirements. The most cited non-functional requirement was maintainability. However, developers also mentioned flexibility, readability, adaptability, performance, security, and robustness.

Pattern-based tactic is the tactic that developers searched for instances of a design or architectural pattern in the source code and verify if their implementation violates the pattern rules. This tactic was frequently used both to locate elements and to confirm the existence of design problems. In this tactic, developers analyzed code structures potentially violating a pattern rule. Whenever developers could confirm the violation, they marked the element as having a design problem. developers discussed a wide range of patterns, including *Adapter*, *Builder*, *Facade*, *SOA* and *MVC*.

In Section 5.2.3.1, we discussed how developers combine multiple related symptoms. In fact, not only most developers combine symptoms, but also they identify more design problems when they combine the symptom. Developers combine the symptoms because they apply multiple tactics, each one in a different step of the design problem identification. For instance, whenever they were looking for violations of a design or architectural pattern (pattern-based tactic) to locate elements, they did not only rely on the violation itself to support the confirmation of a design problem. They often confirmed the existence of a design problem when they noticed other symptoms, *e.g.*, the element was either explicitly affecting a non-functional requirement (NFA-based tactic) or hosting one or more code smells (smell-based tactic). As previously explained, these combinations happened because an element may contain several symptoms that indicate design problems. For instance, if an element violates a pattern, it is likely that the element also contains smells and violations of design principles. Consequently, these symptoms may impact non-functional requirements negatively.

5.3 Propositions Concerning the Developer

In this section, we provide some additional propositions concerning the developer, observed through the think-aloud method (116) with the support of video and audio recordings.

5.3.1 Confidence in the Presence of a Design Problem

The confirmation or rejection of a design problem in a group of elements is mainly influenced by the developers' *confidence* (**C18**), which is the degree to which they are convinced about the presence of a design problem. The most confident the developer is, the greater the likelihood of confirming a design problem.

Attributes that increase developers' confidence. The attributes that influence a design problem diagnosis also affect the developers' confidence (**P15**). According to our study, the attributes that influence the developers' confidence the most are: accuracy, density, element role, and diversity. It is not a surprise that the more accurate (**P16**) and denser (**P17**) the developer believes that the symptom is, the more confident he will be in the presence of a design problem. Nevertheless, the element role plays an even greater influence on the developers' confidence (**P18**).

Developers' divergence regarding element role. At first glance, when most developers analyze an element that plays an important role in the system, they tend to assume that the element contains a design problem. Examining further, we observed two behaviors. When developers analyzed element role together with other attributes, they tended to confirm the corresponding design problem. Conversely, whey they only considered the element role (ignoring other attributes), they tended to reject the design problem, arguing it is acceptable to have design problem symptoms in elements that play an important role in the system.

These two behaviors happened with T2 and T4, respectively. T2 developers confirmed the design problem in the element because, among other attributes, the element played an important role in the system. On the other hand, T4 developers said that, due to the element role, it is acceptable that the element contains the design problems symptoms. According to them, if the element were not an important class for the system, it would not be acceptable to have a design problem or its symptoms in the class.

Pondering about the number of symptoms. When a developer analyzes individual symptoms, the number of symptoms with which he agrees or disagrees influences his confidence in the design problem identification. When analyzing each symptom, the developer decides whether it indicates a design problem. In the end, he counts the number of symptoms he judged as indicating a problem and the number of symptoms he judged as irrelevant. If the former is greater than the latter, then he confirms that the element has a design problem. T3 developers used this strategy to increase their confidence in the presence of a design problem in some elements.

5.3.2 Conscientiousness

Conscientiousness (**C19**) is a personality trait related to being careful, responsible, and persevering (120). The more conscientious the developer is, the greater the likelihood of identifying a design problem. Likewise, when developers diagnose more design problems, they become more conscientious. As these attributes have a circular effect between them (**P19**), it would be interesting to find ways to increase the developers' conscientiousness.

Diversity as an attribute to increase conscientiousness. The diversity of symptoms is the attribute that most influences the conscientiousness of the developers (**P20**). The higher the diversity of a syndrome, the greater the chance the developer will identify a design problem in the element. That happens because the diversity not only increases the confidence of the developers, but it can also help the developers to decide whether the element contains a design problem. In fact, the diversity had a great influence on developers of the T7, T9, T10 and T11 teams, because they tended to assume diversity as a strong indicator of a design problem (Section 5.2.2). Therefore, this finding is another evidence the studies with an assumption that developers rely on only one type of symptom may be misaligned with the developers' practice (47, 45, 56, 44). Even worse, these studies are not taking advantage of the impact that the diversity attribute has on developers' conscientiousness.

Side effect of only considering the diversity attribute. We noticed a side effect when developers rely too much on the importance of the diversity of a syndrome without further analyzing other attributes. For instance, after the T4 developers had analyzed a set of elements with diverse symptoms, they later judged an element as free of a design problem because it did not have the same diversity of symptoms as the ones analyzed previously. Although this behavior was not very frequent, it brings out another issue that studies that rely on only one type of symptom do not take into account.

5.3.3 Incapability of Providing an Alternative

Justifying the presence of a design problem with design decisions. Sometimes the developers are convinced that an element contains several symptoms that indicate a design problem, even though they do not confirm the presence of a design problem. Although such behavior seems contradictory, they argue that they do not consider the element as containing a design problem because they see no other way to implement the element. In these cases, developers use the concept of design decision to justify why they do not consider the presence of a design problem (**P21**). Consequently, the design decision that developers use as an argument influences their confidence in the presence of a design problem (**P22**).

Developers justified the presence of a design problem mostly when they could not provide an alternative implementation. This behavior is aligned with the theory discussed by March and Simon (121), who theorized that developers typically do not choose an optimal solution because such solution would require that all alternatives to a problem be perceived. However, they argue that in practice it is unlikely for developers to know all alternatives. Hence, the known alternatives represent the boundaries that developers face before making a decision. Therefore, developers stop searching for further solutions when one that satisfies their needs is found.

Justifying the presence of a design problem with the lack of an alternative implementation. March and Simon's (121) theory also manifests in the context of identifying design problems, as we observed in our study. The developers used the limited known alternatives to justify why a specific implementation does not present a design problem. In these cases, they mentioned that they could not find any alternative solution (optimal or not) to implement the element. According to them, the element should not be considered as an element involved in a design problem. In other words, the known alternatives are not only boundaries that developers face, but also used to justify the presence or absence of a design problem.

5.4 Towards Improving Design Problem Diagnosis

As mentioned in Section 5.1.5.3, in the third phase of data collection and analysis, we used the Sjøberg *et al.* framework to write up the theory in a way that both research community and industry could benefit from it (112). Thus, the theory presented here provides an explanation of how developers identify design problems. Researchers and industry practitioners can use the discussions presented here as an underlying mechanism to drive solutions for supporting developers during design problem identification. For instance, in this section, we present some solutions that emerged from the theory, and that can improve design problem identification.

5.4.1 Supporting Multiple Symptoms

Providing multiple design problem symptoms. Most studies rely on a single, predefined, dominant type of symptom (44, 47, 56, 45), which may be limiting how developers identify design problems in practice. Thus, there is a need for solutions that provide developers with multiple symptoms, and then help them to navigate among these symptoms and to combine them. In fact, we noticed that developers would benefit from mechanisms to automatically provide symptoms for combination. For instance, a solution in this sense is to provide other symptoms that are complementary to the one being analyzed. Such tool, for instance, could have helped the T2 developers to identify a design problem (Section 5.2.3.1). They used the Dispersed Coupling to choose the coupling attribute to analyze next. Later, they chose the readability non-functional requirement to complement their analysis. In this example, a tool could provide the coupling attribute and the readability requirement as soon as the developers indicate the Dispersed Coupling code smell as helpful.

Filtering relevant symptoms. Developers consider the diversity of symptoms. However, if an element manifests several symptoms, the developers could have a hard time to choose the most helpful one. For instance, D16 (T8 team) mentioned the difficulty that he had to choose helpful symptoms:

D16: *"Since I was not familiar with each type of symptom and design problem, it was hard for me to match them. Even with the provided symptoms, I could not figure out which one was actually related to the design problems."*

To address this issue, an automatic tool could help them to filter those symptoms that are most likely to indicate a design problem. In the same way that a tool could propose complementary symptoms to the one being analyzed, it could hide symptoms that are least similar to the one under analysis. Such tool could make the analysis of multiple symptoms less cumbersome.

Visualization support. Another solution to help developers to deal with multiple symptoms is to provide visualization mechanisms. For instance, *Scattered Concern* (7) problem occurs when multiple code elements implement a functionality that should have been implemented by only a few elements. In this case, developers have to analyze multiple elements that may have the scattered functionality. These elements are likely to share some symptoms. Perhaps if developers could visualize how the multiple symptoms interact in the system, they could identify these elements more easily. In fact, D14 (T7 team) mentioned in the follow-up questionnaire that a visualization mechanism would help him to identify some design problems:

D14: *"For some design problems e.g., Cyclic Dependency, Scattered Concern, it's hard to find by looking at the source code manually, which is too low level when we don't have a higher level architecture view."*

5.4.2 Prioritization of Similar Elements

Prioritizing epidemic elements. Developers tend to prioritize elements that play an important role in the system. In addition, if these elements have diverse symptoms, then they should be the first elements to be analyzed by the developers. Researchers could therefore use the attributes presented here to build tools that prioritize elements. For instance, developers of the T2 team used the element role during the epidemic analysis (Section 5.2.3.2). In two cases they relied on the element role to select epidemic elements. However, in one case they could identify a design problem, whereas in the other case, they could not. The difference between these two cases was related to the number of epidemic elements playing the same role. While in the first case all the epidemic elements played the Service role, in the second case only few epidemic elements played the Controller role. The following quotations illustrate this.

D4: *"I think that all the service classes will have (the design problem)"*
D3: *"Indeed, the service (classes)"*
D4: *"I guess that (they) are similar to each other. In fact, I believe that the next service (class) will be similar"*

5.4.3 Additional Support for the Developer

Based on the propositions concerning the developers (Section 5.3) we suggest providing the following additional support.

Providing an alternative implementation. It is often difficult for developers to provide an alternative implementation for an element that may contain a design problem (Section 5.3.3). In this context, a tool could indicate an alternative implementation that could remove the design problem symptoms. Hence, a developer would not be able to use the lack of an alternative implementation as justification for not confirming a design problem.

Personalizing the detection of symptoms. The accuracy of the symptom is also influenced by the developers' subjectivity. Developers mentioned that a certain type of symptom was accurate in indicating a design problem in some elements, but not in others. Thus, most developers mentioned that they need tools that allow them to personalize the detection of symptoms according to their software systems. Allowing developers to adjust thresholds and detection rules would minimize how the (low) accuracy influences their confidence in the presence of a design problem. D11 (T6 team) mentioned in the follow-up questionnaire the need for such feature:

> D11: *"The symptoms suggest a possible design problem. However, none of them should be rigid rules. Often, it makes sense to have long methods, message chains or many parameters (in the method). In some cases, we could replace a long string of conditional (statements), but it would make it difficult to understand. A method was considered long, but its readability was very clear, which did not justify a refactoring."*

5.5 Related Work

Along this chapter, we presented some studies about the identification of design problems. We have not found studies that present the diagnosis of design problems as a theory. Instead, we found studies that focus on presenting the phenomenon rather than explaining it (26, 52, 53, 38, 44, 28, 47, 27, 46, 45). For instance, several researchers proposed techniques to identify design problems (52, 53, 38, 47, 45). Although these studies had encouraging results, they did not conduct experiments with software developers or they have not taken into account the attributes that affect design problem identification.

For instance, Mo *et al.* (44) proposed the detection of recurring design problems by the combination of structural, history and design information. Xiao *et al.* (45) introduced a solution – based on a history coupling probability matrix – to identify and quantify design problems. The proposed solution uses 4 patterns of design flaws that show the correlation between design problems and reduced software quality. The aforementioned techniques depend on design information, which may not exist for many software systems. In addition, these studies rely on only a predefined, dominant type of symptom. However, their technique does not match with the practice as developers consider and combine multiple symptoms to diagnose a design problem.

Vidal *et al.* (46) presented and evaluated criteria for prioritizing groups of code smells that are likely to indicate design problems in evolving systems. Their results provide evidence that one of the proposed criteria helped to correctly prioritize elements with design problems. However, only one of the criteria helped to prioritize elements. We showed some attributes that developers take into consideration to select symptoms likely to help them 5.2.2. Researchers could use the knowledge presented here to build tools that prioritize elements in a similar way that we showed how developers select the elements to analyze.

Oizumi *et al.* (47) investigated to what extent code smells could "flock together" to realize a design problem. These code smells that flock together and are related to each other composed what they authors called agglomeration. After analyzing more than 2,200 agglomerations of code smells from seven software systems with different sizes and from different domains, the researchers concluded that certain forms of agglomerations are consistent indicators of design problems. Although we also have investigated multiple instances of code smells as indicators of design problems, our findings are more grounded on the in-depth observation of the developers' behavior than in quantitative results of retrospective studies. Moreover, similar results found on both studies helps to strength evidence that developers often reason about multiple symptoms to identify design problems in the implementation.

There is recently a growing interest in studying the relevance of code smells to support the identification of design problems (47, 51, 46). Palomba *et al.* (51) reported an empirical study aimed at analyzing to what extent code smells are perceived as design problems. In their study, they showed developers code snippets affected and not affected by code smells. In an affirmative case, they asked developers to explain what problem they perceive. They reported that some code smells are, in general, not perceived by developers as design problems. We go beyond Palomba *et al.* (51). We investigated various strategies followed by developers to identify a design problem. We found that there are other strategies than a smell-based strategy to identify design problems. Moreover, we noticed that when developers use multiple instances of smells, they succeed to identify design problems.

5.6 Threats to Validity

This section presents and discusses threats to validity.

Construct Validity. We provided some symptoms for developers to use during design problems diagnosis. These data could have biased the experiments. However, we provided these data considering the literature (115, 4, 53, 52, 38, 47) and considering the companies' managers. They mentioned that some of the developers not only were familiar with some symptoms but also had the culture of using them. Furthermore, we noticed developers using other symptoms that we have not provided; thus, we decided to provide these other symptoms for the other developers. However, developers were free to use these symptoms or not. The time allocated for the tasks 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. However, in the context of applying an analysis through GT, we saw this diversity as an opportunity to strengthen the evidence supporting the depicted propositions. Moreover, we provided training to mitigate this threat.

External Validity. The number of subject represents a threat. All of them worked for companies located in Brazil. However, it is important to note that this is a multi-company study involving five different working environments and eight different systems. Finally, the presented study covered only systems developed in Java. Using other programming languages with different core characteristics may influence developers in identifying design problems.

Conclusion Validity. This threat concerns the relation between treatment and outcome. We tried to mitigate it by combining data from different resources: quantitative and qualitative data obtained with videos, and questionnaires. We believe data collection and analysis were properly built to answer our questions. The participation of the researcher 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. Moreover, the identification of the constructs and the depicting of propositions were performed separately by researchers. In fact, three researchers conducted the Grounded Theory procedures independently; then we merged their results to shape the theory (Section 5.1.5.2). Thus, the contents were compared and discussed by the researchers until reaching a consensus.

5.7 Summary

A design problem is the result of one or more inappropriate decisions that negatively impact non-functional requirements. Despite their harmfulness, the identification of each design problem is not trivial. One of the main reasons is that design documentation is often unavailable or outdated. Thus, developers often have to rely on the source code to identify design problems, which may quickly turn into a complex task. Although researchers have investigated techniques to help developers, there is little knowledge on how developers actually proceed to identify design problems in practice.

In order to address this limitation, this chapter presented a study to investigate how developers identify design problems in practice. We conducted a multi-trial industrial experiment with developers from different companies, where they had to identify design problems in their systems (Section 5.1). As a result, we derived a theory describing the activities and factors that influence on how developers identify design problems, which can serve to further understand the identification of design problems (Section 5.2). For example, the theory reveals that developers rely on a heterogeneous set of symptoms, and they tend to combine them. The theory also presents the characteristics of symptoms that developers consider helpful. We also discuss the theory taking into account propositions concerned to developers (Section 5.3). For instance, we found some factors that influence their confidence in the presence of a design problem in an element. Finally, we discussed how the knowledge revealed by our theory can be used to advance the state-of-art (Section 5.4).

Future steps in this work involve the execution of new empirical studies to assess in more depth the theory's propositions and explanations. For instance, we intend to address some findings described at Section 5.4 and verify whether they have positive effects on design problem identification. The goal of these studies is to use the theory to implement a novel family of solutions that are more effective than the current ones to help developers to identify design problems.

6 Conclusion

Software systems have been discontinued or reengineered due to the prevalence of certain design problems in the source code (14, 15, 16, 17). A design problem is the manifestation of one or more inappropriate design decisions that impact non-functional requirements negatively. In practice, we delimited design problems to those resulting from design decisions that include (i) how the system is organized into subsystems and components, (ii) how and which code elements encapsulate process and data to address each functionality, and (ii) how the elements interact with each other and their execution environment (7, 8, 9).

Given the harmfulness of certain design problems, developers should identify and remove them (with refactoring operations) as early as possible (7, 17, 20). However, before removing a design problem, developers have to identify them. As design documentation is often nonexistent, informal or not up-to-date, developers need to identify design problems directly on the source code. For this identification, they need to locate symptoms of the presence of a design problem, such as code smells. Several studies have presented techniques based on code smells to help developers to identify design problems (41, 38, 39, 28, 47, 46). However, these studies have not investigated if code smells suffice to help developers to identify design problems in practice.

As a matter of fact, we had limited knowledge about how developers identify design problems in practice. For instance, we di not know if code smells suffice to help developers during design problem identification. Indeed, we did not know if code smells are key design problem symptoms for developers in practice (Research Problem 1). A key symptom is one that helps developers to identify a design problem that is relevant in the software system. We also did not know if developers are able to identify design problems in practice through the use of code smells (Research Problem 2). Since we did not know to what extent code smells suffice to help developers to identify design problems, smell-based techniques could not support developers entirely during the identification.

Lack of knowledge was not restrict to not knowing only if code smells suffice to support developers. In fact, we did not even know how to assess existing techniques since we do not know how the design problem identification takes place in practice. Additionally, we did not know if developers use other symptoms in addition to code smells (Research Problem 3). Lack of sufficient knowledge of how developers identify design problems in practice (General Problem) could have been preventing researchers and tool engineers from providing the required support for developers to identify design problems. In this context, the goal of this thesis was to stepwisely understand how developers identify design problems in practice. To achieve such a goal, we conducted several studies, which are summarized as follows.

6.1 Revisiting the Thesis Contributions

In our quest to understand how developers identify design problems in practice, our first step was to investigate the role that code smells play in the identification. Code smells have been discussed in the literature as a consistent indicator of design problems (53, 28, 47, 51, 27, 55, 29). Notwithstanding, we did not know if in practice code smells are key symptoms for developers identifying relevant design problems. A relevant design problem is one that developers had to identify and remove from the source code. Thus, we conducted a retrospective study (Chapter 3) to answer the following research question: *Are code smells key symptoms to indicate relevant design problems for developers?*

To answer this research question we analyzed the source code of 50 software projects. In this analysis, we verified if the refactored elements were likely to contain design problem. Then, we analyzed if these elements contain at least one code smell closely related to the design problem. If the code elements contain code smells, then smells have the potential to serve as key symptoms to help developers to identify relevant design problems in practice. As a result of this analysis, we found that most refactored elements had code smells that could be used to identify design problems in the elements. In this context, the first contribution of this thesis was:

1st Contribution. *First evidence that code smells likely represent key symptoms for developers to identify relevant design problems. Furthermore, certain design problems bear no relationship with code smells*

Even though our results indicate that in most cases code smells can represent key symptoms, we found some scenarios in which code smells could not indicate design problems. For instance, we found some refactored elements that did not have code smells but they were likely to be affected by a design problem. This result led us to investigate if in practice developers can use and reason about code smells to identify design problems.

In this follow-up study (Chapter 4), we aimed to answer the following research question: *Are developers able to use code smells to identify design problems?* To answer this research question, we asked 11 developers to identify design problems using single code smells. In this study, we also asked the same developers to use multiple code smells. As we found that developers can benefit from analyzing multiple smells, we also wanted to investigate whether they could reason about multiple smells in practice. In this sense, we asked developers to use agglomerations of code smells. Comparing developers using single smells and using multiple smells (*i.e.*, agglomeration) allowed us to find whether code smells suffice to help developers during the design problem identification. As the result of this study, we found that not only developers are able to use code smells but also the use of agglomerations improves developers' precision in finding design problems. Unfortunately, we also found that code smells do not suffice to help developers to identify all design problems. In this context, the second contribution was:

2nd Contribution. *Code smells do not suffice to assist developers along the task of identifying design problems in practice.*

Even though smells do not suffice to support them, we found that developers can benefit from the analysis of multiple smells. In fact, such analysis gave developers confidence to confirm the occurrence of the design problem, which may help to explain why they refactor refactoring elements with multiple smells (Section 3.4.1). Interesting enough, we found that although the analysis of multiple smells can be cumbersome, developers use each smell as a complementary symptom of the presence of a design problem. This result led us to wonder if developers also benefit from the analysis of other symptoms in addition to smells. To identify what symptoms developers use, we conducted a multi-trial study (Chapter 5) to answer the following research question: *What are the design problem symptoms that developers use in practice?*.

To answer this research question, we asked professional software developers to identify design problems in their own systems. In this study, we captured data of developers identifying design problems by filming the environment, recording audio and capturing their computer screens on video. These data allowed us to conduct an in-depth qualitative analysis based on Grounded Theory procedures (60), which also allowed us to answer the following research question: *How do developers identify design problems in practice?* As the result for the first research question in this study, we found the five preeminent symptoms that developers use in practice. They are violation of non-functional requirements, code smells, violation of architectural and design patterns, poor structural quality attributes, and violation of object-oriented principles. Thus, the third contribution of this thesis was:

> **3rd Contribution.** *The symptoms that developers use in practice in addition to code smells.*

In addition to finding what symptoms developers use in practice, we also found out how they use these symptoms. In fact, we found out how they select the symptoms that they consider useful to identify a design problem. We found out when they use these symptoms, and which factors influence them during their use towards the identification of design problems. All this knowledge heretofore unknown was wrapped up in a theory that provides an overview, explanation, and understanding on how developers identify design problems in source code. Therefore, the fourth contribution of this thesis was:

> **4th Contribution.** *A theory on how developers identify design problems in practice.*

This theory explains how software developers identify design problems through the analysis of symptoms that manifest in the source code. It has constructs about how developers conduct the identification, for instance, which steps they follow. The theory also contains constructs about the symptoms that developers use in practice. For instance, developers rely on a heterogeneous set of symptoms, and they tend to combine those five preeminent symptoms. The theory also presents the characteristics of symptoms that developers consider helpful, such as the symptom type, its relation with other symptoms and the frequency that it emerges in an element. These and other characteristics are used by developers to prioritize elements and symptoms during design problem identification. The theory also has constructs and propositions concerning developers. For instance, we found some factors that influence their confidence in the presence of a design problem in an element, such as the diversity of symptoms, which increases the developers' conscientiousness.

We highlight that before proposing solutions that will help developers to identify design problems, first, we need to understand how they conduct the design problem identification in practice. Therefore, the contributions listed

Table 6.1: Papers Produced in the Context of this Thesis

Paper	Chapter
Leonardo da Silva Sousa. 2016. *Spotting design problems with smell agglomerations.* In Proceedings of the 38th International Conference on Software Engineering Companion – Doctoral Symposium (ICSE '16).	4
*Willian Oizumi, **Leonardo Sousa**, Alessandro Garcia, Roberto Oliveira, Anderson Oliveira, O. I. Anne Benedicte Agbachi, and Carlos Lucena. 2017. *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).	4
Leonardo Sousa, Roberto Oliveira, Alessandro Garcia, Jaejoon Lee, Tayana Conte, Willian Oizumi, Rafael de Mello, Adriana Lopes, Natasha Valentim, Edson Oliveira, and Carlos Lucena. 2017. *How Do Software Developers Identify Design Problems?: A Qualitative Analysis.* In Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17).	5
***Leonardo Sousa**, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, Carlos Lucena, and Rodrigo Paes. 2018. *Identifying design problems in the source code: a grounded theory.* In Proceedings of the 40th International Conference on Software Engineering (ICSE '18).	5
Leonardo Sousa, Isabella Ferreira, Diego Cedrim, Alexander Chávez, Alessandro Garcia, and Carlos Lucena. 2018. *The Structural Quality of Refactored Code: A Study of 50 Software Projects.* Journal. under submission process.	3
* Distinguished Paper Award	

here can advance both state-of-the-art and state-of-the-practice. Researchers and industry practitioners can use the discussions in this thesis to define mechanisms that drive news solutions for supporting developers during design problem identification. The knowledge provided here can help researchers in building more suitable mechanisms that are aligned with how developers identify design problems in practice.

Finally, to facilitate future references to works that resulted from this thesis, Table 6.1 presents the papers produced in the context of this thesis and the respective chapters to which they are related.

6.2 Future Work

New challenges and opportunities for improvement have emerged along the studies conducted in the context of this thesis. Based on them, further directions for future work are presented next.

Investigating Heuristics to Prioritize Smells and Agglomerations. In Chapter 4, we relied on code smells and their agglomerations to investigate to what extent they suffice to help developers to identify design problems. Even though we concluded that smells do not suffice, they are still preeminent symptoms used in practice. We noticed the need to prioritize smells and agglomerations, which was also confirmed by developers. A software system can have thousands of code smells (38), which can lead to hundreds of possible agglomerations. However, not all code smells and, consequently, agglomerations in a system will be related to a design problem. Thus, in order to support developers during the design problem identification, we need to investigate heuristics to prioritize smells and agglomerations that are most likely to indicate a design problem.

These heuristics for prioritization can be based on the knowledge provided by the theory. Since we know how developers select symptoms that may be useful to identify a design problem, we can apply this knowledge to create heuristics to select code smells and agglomerations. We can also explore other types of relations that can be established between code smells. For instance, Oizumi *et al.* proposed agglomerations that explore code smells that are semantically related to each other (47). We can also explore other types of relations. For instance, code smells that are related to each other due to exception flow. Thus, if elements that handle exceptions contain code smells, and they are in the same exception flow, then we can create an agglomeration with these smells. In this vein, we can also explore data flow to create agglomerations.

Exploring the Theory to Improve the Design Problem Diagnosis. In Section 5.4, we discussed some solutions that can emerge from the theory and can be used to improve the design problem identification. Each one of those discussed solutions requires extensive investigation. Among these investigations, the prioritization of elements is essential. In the same way that a system can have thousands of code smells, it can have thousands of design problem symptoms. Thus, developers need to prioritize code elements that have symptoms that indicate a design problem.

Based on how developers select symptoms that they consider useful, we can propose heuristics to select elements that contain these useful symptoms. For instance, we can select elements that have a high *density* or *diversity* of symptoms. We can also investigate heuristics to find the *relation* among the symptoms, and use these relations to prioritize elements that contain symptoms that are related to each other. We also need to investigate heuristics to identify elements that play the same *role* in the system, and that have a similar set of symptoms. These elements can be grouped based on the element role and if they have the same *symptom types*. Thus, the more common symptom types a group has, the higher its priority.

Validating the Theory Propositions. As discussed in Section 5.2, the theory has high testatibility since it is possible to replicate the study in a way that empirical refutation is possible. In this sense, the theory offers several propositions that can be tested individually. Testing these propositions not only serves to validate the propositions but can also reveal further details about the design problem identification. This new knowledge can be used to improve the support for developers to identify design problems. For instance, propositions concerning the developer can be validated. This validation can reveal methods to reduce the impact that human factors have on the identification of design problems.

Extending the Theory. The design problem identification is an activity associated with maintainability. Thus, we wonder whether the theory can be extended to other activities in the maintenance context, for instance, refactoring. As mentioned, refactoring and design problem identification are two activities closely related to each other: the former succeeds the later. Thus, the theory could also be extended to include refactoring. In this context, other studies can also be conducted to investigate what propositions can be applied in another maintenance context, such as, debugging.

Bibliography

[1] 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.

[2] TAYLOR, R. N.; VAN DER HOEK, A.. **Software design and architecture the once and future focus of software engineering**. In: 2007 FUTURE OF SOFTWARE ENGINEERING, FOSE '07, p. 226–243, Washington, DC, USA, 2007. IEEE Computer Society.

[3] BRUNET, J. A.; MURPHY, G. C.; TERRA, R.; FIGUEIREDO, J. ; SEREY, D.. **Do developers discuss design?** In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, MSR 2014, p. 340–343, New York, NY, USA, 2014.

[4] 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.

[5] LARMAN, C.. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[6] PERRY, D. E.; WOLF, A. L.. **Foundations for the study of software architecture**. SIGSOFT Softw. Eng. Notes, 17(4):40–52, Oct. 1992.

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

[8] 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.

[9] LIPPERT, M.; ROOCK, S.. **Refactoring in Large Software Projects: Performing Complex Restructurings Successfully**. Wiley, 2006.

[10] TAYLOR, R.; MEDVIDOVIC, N. ; DASHOFY, E.. **Software Architecture: Foundations, Theory, and Practice**. Wiley Publishing, 2009.

[11] 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.

[12] CHEN, X.; DAVARE, A.; HSIEH, H.; SANGIOVANNI-VINCENTELLI, A. ; WATANABE, Y.. **Simulation based deadlock analysis for system level designs**. In: PROCEEDINGS OF THE 42ND ANNUAL DESIGN AUTOMATION CONFERENCE, DAC '05, p. 260–265, New York, NY, USA, 2005. ACM.

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

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

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

[16] 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.

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

[18] CURTIS, B.; SAPPIDI, J. ; SZYNKARSKI, A.. **Estimating the size, cost, and types of technical debt**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL WORKSHOP ON MANAGING TECHNICAL DEBT, MTD '12, p. 49–53, Piscataway, NJ, USA, 2012. IEEE Press.

[19] 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.

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

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

[22] KUMAR, M. R.; KUMAR, R. H.. **Architectural refactoring of a mission critical integration application: A case study**. In: PROCEEDINGS OF THE 4TH INDIA SOFTWARE ENGINEERING CONFERENCE, ISEC '11, p. 77–83, New York, NY, USA, 2011. ACM.

[23] SAMARTHYAM, G.; SURYANARAYANA, G. ; SHARMA, T.. **Refactoring for software architecture smells**. In: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON SOFTWARE REFACTORING, IWoR 2016, p. 1–4, New York, NY, USA, 2016. ACM.

[24] TAHVILDARI, L.; KONTOGIANNIS, K.. **A metric-based approach to enhance design quality through meta-pattern transformations**. In: SEVENTH EUROPEAN CONFERENCE ONSOFTWARE MAINTENANCE AND REENGINEERING, 2003. PROCEEDINGS., p. 183–192, March 2003.

[25] 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.

[26] CIUPKE, O.. **Automatic detection of design problems in object-oriented reengineering**. In: PROCEEDINGS OF TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS - TOOLS 30 (CAT. NO.PR00278), p. 18–32, Aug 1999.

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

[28] 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.

[29] 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.

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

[31] KERIEVSKY, J.. **Refactoring to Patterns**. Pearson Higher Education, 2004.

[32] PAGE-JONES, M.. **Fundamentals of Object-oriented Design in UML**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[33] SIMON, F.; SENG, O. ; MOHAUPT, T.. **Code-Quality-Management - technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht.** dpunkt.verlag, 2006.

[34] BUDD, T. A.. **An Introduction to Object-Oriented Programming**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2001.

[35] GREENWOOD, P.; BARTOLOMEI, T.; FIGUEIREDO, E.; DOSEA, M.; GARCIA, A.; CACHO, N.; SANT'ANNA, C.; SOARES, S.; BORBA, P.; KULESZA, U. ; RASHID, A.. **On the impact of aspectual decompositions on design stability: An empirical study**. In: PROCEEDINGS OF THE 21ST EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP'07, p. 176–200, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] SOARES, S.; LAUREANO, E. ; BORBA, P.. **Implementing distribution and persistence aspects with aspectj**. In: PROCEEDINGS OF THE 17TH ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS; SEATTLE, USA, p. 174–190. ACM Press, 2002.

[37] BOOCH, G.; RUMBAUGH, J. ; JACOBSON, I.. **The Unified Modeling Language User Guide**. Addison-Wesley, Boston, 2005.

[38] 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.

[39] 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.

[40] MOHA, N.; GUÉHÉNEUC, Y.-G.; LE MEUR, A.-F. ; DUCHIEN, L.. **A domain analysis to specify design defects and generate detection algorithms.** In: Fiadeiro, J. L.; Inverardi, P., editors, FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, p. 276–291, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[41] 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.

[42] XIAO, L.; CAI, Y. ; KAZMAN, R.. **Design rule spaces: A new form of architecture insight.** In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 967–977, New York, NY, USA, 2014. ACM.

[43] 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.

[44] 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.

[45] XIAO, L.; CAI, Y.; KAZMAN, R.; MO, R. ; FENG, Q.. **Identifying and quantifying architectural debt.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '16, p. 488–498, New York, NY, USA, 2016. ACM.

[46] 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.

[47] 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.

[48] KHOMH, K.; PENTA, M. D. ; GUEHENEUC, Y.. **An exploratory study of the impact of code smells on software change-proneness**. In: PROCEEDINGS OF THE 16TH WORKING CONFERENCE ON REVERSE ENGINEERING; LILLE, FRANCE, p. 75–84, 2009.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; DI PENTA, M.; DE LUCIA, A. ; POSHYVANYK, D.. **When and why your code starts to smell bad**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '15, New York, NY, USA, 2015. ACM.

[56] WONG, S.; CAI, Y.; KIM, M. ; DALTON, M.. **Detecting software modularity violations**, May 2011.

[57] EASTERBROOK, S.; SINGER, J.; STOREY, M.-A. ; DAMIAN, D.. **Selecting Empirical Methods for Software Engineering Research**. Springer London, London, 2008.

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

[59] 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.

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

[61] 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.

[62] BUDGEN, D.. **Software Design**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[63] LISKOV, B.; GUTTAG, J.. **Program Development in Java: Abstraction, Specification, and Object-Oriented Design**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[64] 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.

[65] JACOBSON, I.; BOOCH, G. ; RUMBAUGH, J.. **The Unified Software Development Process**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[66] WIEGERS, K. E.. **Software Requirements**. Microsoft Press, Redmond, WA, USA, 2 edition, 2003.

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

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

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

[70] BUSCHMANN, F.; HENNEY, K. ; SCHMIDT, D.. **Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing**. John Wiley & Sons, 2007.

[71] DIETRICH, J.; MCCARTIN, C.; TEMPERO, E. ; SHAH, S. M. A.. **Barriers to modularity - an empirical study to assess the potential for modularisation of java programs**. In: Heineman, G. T.; Kofron, J. ; Plasil, F., editors, RESEARCH INTO PRACTICE – REALITY AND GAPS, p. 135–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[72] MO, R.; CAI, Y.; KAZMAN, R.; XIAO, L. ; FENG, Q.. **Decoupling level: A new metric for architectural maintenance complexity**. In: 2016 IEEE/ACM 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 499–510, May 2016.

[73] SURYANARAYANA, G.; SAMARTHYAM, G. ; SHARMA, T.. **Refactoring for Software Design Smells: Managing Technical Debt**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[74] 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.

[75] 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.

[76] SARKAR, S.; RAMACHANDRAN, S.; KUMAR, G. S.; IYENGAR, M. K.; RANGARAJAN, K. ; SIVAGNANAM, S.. **Modularization of a large-scale business application: A case study**. IEEE Softw., 26(2):28–35, Mar. 2009.

[77] MURPHY-HILL, E.; BLACK, A. P.. **An interactive ambient visualization for code smells**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON SOFTWARE VISUALIZATION; SALT LAKE CITY, USA, p. 5–14. ACM, 2010.

[78] OIZUMI, W.; GARCIA, A.. **Organic: A prototype tool for the synthesis of code anomalies**, 2015.

[79] SHAW, M.; GARLAN, D.. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[80] GANESH, S.; SHARMA, T. ; SURYANARAYANA, G.. **Towards a principle-based classification of structural design smells**. Journal of Object Technology, 12(2):1:1–29, June 2013.

[81] TRIFU, A.. **Towards automated restructuring of object oriented systems**. PhD thesis, Karlsruhe Institute of Technology, 2008.

[82] OIZUMI, W.; GARCIA, A.; COLANZI, T.; FERREIRA, M. ; STAA, A.. **When code-anomaly agglomerations represent architectural problems? An exploratory study**. In: PROCEEDINGS OF THE 2014 BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES); MACEIO, BRAZIL, p. 91–100, 2014.

[83] BALDWIN, C. Y.; CLARK, K. B.. **Design Rules: The Power of Modularity Volume 1**. MIT Press, Cambridge, MA, USA, 1999.

[84] BOURQUIN, F.; KELLER, R. K.. **High-impact refactoring based on architecture violations**. In: 11TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR'07), p. 149–158, March 2007.

[85] BAVOTA, G.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring**. Journal of Systems and Software, 107:1 – 14, 2015.

[86] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **A field study of refactoring challenges and benefits**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, FSE '12, p. 50:1–50:11, New York, NY, USA, 2012. ACM.

[87] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor? confessions of github contributors**. In: PROCEEDINGS OF THE 2016 24TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE 2016, p. 858–870, New York, NY, USA, 2016. ACM.

[88] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects**. In: PROCEEDINGS OF THE 2017 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2017, p. 465–475, New York, NY, USA, 2017. ACM.

[89] 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.

[90] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 483–494, New York, NY, USA, 2018. ACM.

[91] XING, Z.; STROULIA, E.. **Umldiff: An algorithm for object-oriented design differencing**. In: PROC. OF ASE '05, p. 54–65, 2005.

[92] 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.

[93] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D. ; LUCIA, A. D.. **Mining version histories for detecting code smells**. IEEE Transactions on Software Engineering, 41(5):462–489, May 2015.

[94] MAIGA, A.; ALI, N.; BHATTACHARYA, N.; SABANÉ, A.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G. ; AÏMEUR, E.. **Support vector machines for anti-pattern detection**. In: PROCEEDINGS OF THE 27TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE 2012, p. 278–281, New York, NY, USA, 2012. ACM.

[95] OUNI, A.; GAIKOVINA KULA, R.; KESSENTINI, M. ; INOUE, K.. **Web service antipatterns detection using genetic programming**. In: PROCEEDINGS OF THE 2015 ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, GECCO '15, p. 1351–1358, New York, NY, USA, 2015. ACM.

[96] MATTMANN, C.; CRICHTON, D.; 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: SOFTWARE ENGINEERING ACHIEVEMENTS TRACK; SHANGHAI, CHINA, p. 721–730, 2006.

[97] MCINTOSH, S.; KAMEI, Y.; ADAMS, B. ; HASSAN, A. E.. **The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects.** In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 192–201, Hyderabad, India, 2014.

[98] SHADISH, C.; 2001, C.. **Experimental and Quasi-Experimental Designs for Generalized Causal Inference.** Houghton Mifflin, 2 edition, 2001.

[99] YAHOO!. **Explore career opportunities**, April 2017. Available at `https://careers.yahoo.com/us/buildyourcareer`.

[100] TWITTER. **Working at twitter**, April 2017. Available at `https://about.twitter.com/careers`.

[101] 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.

[102] ARCOVERDE, R.; AES, E. G.; 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.

[103] VIDAL, S. A.; MARCOS, C. ; DÍAZ-PACE, J. A.. **An approach to prioritize code smells for refactoring.** Automated Software Engg., 23(3):501–532, Sept. 2016.

[104] HERMAN, I.; MELANCON, G. ; MARSHALL, M. S.. **Graph visualization and navigation in information visualization: A survey.** IEEE Transactions on Visualization and Computer Graphics, 6(1):24–43, Jan 2000.

[105] EMDEN, E.; MOONEN, L.. **Java quality assurance by detecting code smells**. In: PROCEEDINGS OF THE 9TH WORKING CONFERENCE ON REVERSE ENGINEERING; RICHMOND, USA, p. 97, 2002.

[106] RATZINGER, J.; FISCHER, M. ; GALL, H.. **Improving evolvability through refactoring**, volumen 30. ACM, 2005.

[107] WETTEL, R.; LANZA, M.. **Visually localizing design problems with disharmony maps**. In: PROCEEDINGS OF THE 4TH ACM SYMPOSIUM ON SOFTWARE VISUALIZATION, p. 155–164. ACM, 2008.

[108] STOL, K.-J.; FITZGERALD, B.. **Theory-oriented software engineering**. Science of Computer Programming, 101:79 – 98, 2015. Towards general theories of software engineering.

[109] HANNAY, J. E.; SJOBERG, D. I. K. ; DYBA, T.. **A systematic review of theory use in software engineering experiments**. IEEE Transactions on Software Engineering, 33(2):87–107, Feb 2007.

[110] JEFFERY, R.. **Paths to Software Engineering Evidence**, p. 133–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[111] JOHNSON, P.; EKSTEDT, M. ; JACOBSON, I.. **Where's the theory for software engineering?** IEEE Software, 29(5):96–96, Sept 2012.

[112] SJØBERG, D. I. K.; DYBÅ, T.; ANDA, B. C. D. ; HANNAY, J. E.. **Building Theories in Software Engineering**. Springer London, London, 2008.

[113] RUNESON, P.; HOST, M.; RAINER, A. ; REGNELL, B.. **Case Study Research in Software Engineering: Guidelines and Examples**. Wiley Publishing, 2012.

[114] CAMPBELL, G.; PAPAPETROU, P. P.. **SonarQube in action**. Manning Publications Co., 2013.

[115] MARTIN, R.. **Agile Principles, Patterns, and Practices**. Prentice Hall, New Jersey, 2002.

[116] ERICSSON, K. A.; SIMON, H. A.. **Protocol Analysis: Verbal Reports as Data**. MIT Press. A Bradford Book, 2 edition, 1993.

[117] GLASER, B.. **Doing Grounded Theory: Issues and Discussions**. Sociology Press, 1998.

[118] STOL, K.-J.; RALPH, P. ; FITZGERALD, B.. **Grounded theory in software engineering research: A critical review and guidelines.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '16, p. 120–131, New York, NY, USA, 2016. ACM.

[119] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. ; STAL, M.. **Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.** Wiley Publishing, 1996.

[120] NORMAN, W. T.. **Toward an adequate taxonomy of personality attributes: replicated factors structure in peer nomination personality ratings.** Journal of abnormal and social psychology, 66:574–583, June 1963.

[121] MARCH, J.; SIMON, H.. **Organizations.** Wiley, 1958.

A
Glossary

– **Agglomeration** is a group of inter-related code smell that can be related to a design problem.

– **Ambiguous Interface** is a design problem that happens when a component interface is ambiguous and provides non-cohesive services. It usually appears on systems where components have interfaces implementing public methods with generic types as parameter.

– **Code smell** is a microstructure in the system that represents a surface indication of a design problem.

– **Component Overload** is a design problem that happens when a component is overloaded with responsibilities/interests. Usually the component deals with too many functionalities, some of which are not related to each other.

– **Concern** is anything that stakeholders of a software project may want to consider as a conceptual unit.

– **Concern Overload** is a design problem that happens when a code element fulfills too many responsibilities.

– **Cyclic Dependency** is a design problem that happens when one or more elements depend on each other, creating a cycle.

– **Delegating Abstraction** is a design problem that happens when an abstraction exists only for passing messages from one abstraction to another.

– **Design Decision** comprises a set of additions, subtractions, and modifications to the software design; the rationale, the design rules, design constraints and additional requirements that (partially) realize one or more requirements (64). In this context, rationale comprises the reasons behind a decision; the design rules are the mandatory guidelines for further design decisions and design constraints describe what is not allowed in the future of the design.

- **Design Problem Aspect** comprises a characteristic presented in a symptom that stands out for developers.

- **Design Problem** is a manifestation of one or more inappropriate design decisions that affect non-functional requirements.

- **Design Problem Symptom** is a partial sign or indication of the presence of a design problem (61).

- **Fat Interface** is a design problem that happens when an interface exposes many funcionalities and many of those functionalities are not related to each other. They usually are generic interfaces that could be split on specific interfaces

- **Floss refactoring** is a tactic that developers apply with the intention of achieving another objective that is different from structural improvements, such as adding a new feature or fixing a bug.

- **Incomplete Abstraction** is a design problem that happens when an element does not support a responsibility completely in their enclosing component.

- **Misplaced Concern** is a design problem that happens when an element implements a concern (e.g. functionality), which is not the predominant one of their enclosing component

- **Non-agglomerated Smell** is a code smell that has not being grouped according to Oizumi et al. heuristics (47).

- **Non-functional requirements** are aspects of or constrains on a system that are not specifically concerned with the functionality of a system, but specify properties that the system must have, such as performance, usability and failure recovery.

- **Refactored Elements** are all elements directly affected by refactoring operations are considered as refactored elements.

- **Refactoring** is a program transformation used for improving the code structure of a system.

- **Refactoring type** indicates if the refactoring operation is applied to attributes, methods, classes, or interfaces.

- **Root-canal refactoring** is a tactic used to repair a deteriorated code and that cinvoves a process of exclusively applying refactoring operations.

- **Scattered Concern** is a design problem that happens when elements are responsible for the same functionality, but some of which are responsible for functionalities that cross-cut the system.

- **Separation of Concern** is a modularity principle that comprises the subdivision of a problem into independent parts.

- **Smelly Element** represents a code element (method, class, package, and the like) that contains at least one code smell.

- **Software Design** can be both a verb and a noun. When it is consider as a verb, it refers to the process of designing software, at which the design decisions are taken. On the other hand, when it is considered as a noun, it refers to the product that is generated from the design process.

- **Symptom** is a partial sign or indication of the presence of a design problem that developers use in practice.

- **Unwanted Dependency** is a design problem that happens when a dependency violates a rule defined on the system design. It usually occurs when elements should not communicate with each other, but do.

B
Study about Design Problem Identification with Code Smells

This appendix presents the subject characterization questionnaire and the experiment and post-experiment questionnaire (these two last questionnaires were given together in the same file for developers). These questionnaires were used in the study reported in Chapter 4. This appendix also includes the Workflow and PushPull blueprints gave during the experiment

B.1
Developers Characterization

Following we present the questionnaire used to characterized the developers.

Diagnosing Design Problems: Pre-experiment

* Required

1. **Name:** *

   ____

2. **ID Number:**

   ____

3. **Date:** *

   ____

   Example: December 15, 2012

4. **Where do you Live?** *

   examples: Curitiba, Brazil - London, UK

   ____

5. **What is your current position?** *

   examples: Software Developer, Project Manager, Technical Leader, Consultant

   ____

6. **Experience with software development (in years):** *

   ____

7. **Do you have formal education in computer science?** *

   (College, University, etc)

   Mark only one oval.

   ◯ Yes

   ◯ No *Skip to question 11.*

Formal Education

8. **select your current degree:** *

   Mark only one oval.

   ◯ Technologist

   ◯ Graduate

   ◯ Master

   ◯ PhD

Apache OODT

9. **Do you know Apache OODT (Object Oriented Technology)?** *
Mark only one oval.

- Yes
- No

10. **Have you ever read Apache OODT's source code?** *
Mark only one oval.

- Yes
- No

Experience

“Software Design: The overall organization of functionalities into methods, classes, relationships and components (or packages).” Given the provided definition for software design, answer the question below:

11. **Have you ever been responsible for design decisions on any object-oriented system?** *
Mark only one oval.

- Yes
- No

Please fill the following fields with information about the most complex project in which you were/are responsible for design decisions (you can use approximated values if you don't know exactly)

12. **Number of Developers:** *

____

13. **Number of Versions:** *

____

14. **Size (lines of code):** *

____

Knowledge

Rate your knowledge about the following concepts and technologies, using the criteria described in Table I:

Table I. Classification criteria

Classification	Description
None	I have **never heard about it**
Minimum	I have heard about it, but **do not use it**
Basic	I have a **general understanding**, but **almost never use it**
Intermediary	I have a **good understanding**, and **use it sometimes**
Advanced	I have a **deep understanding**, and **often use it**
Expert	I am a **specialist** in this subject, and **use it almost every day**

15. **Source Code Metrics:** *

Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

16. **Code Anomalies (a.k.a. Code Smells):** *

Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

17. **Software Anti-patterns:** *

Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

18. **Refactoring:** *

Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

19. **Object-Oriented Design:** *

Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

20. **Design Patterns:** *

Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

21. **UML:** *

Mark only one oval.

- None
- Minimum
- Basic
- Intermediary
- Advanced
- Expert

22. **Java 6 (Programming Language):** *

Mark only one oval.

- None
- Minimum
- Basic
- Intermediary
- Advanced
- Expert

23. **Java EE:** *

Mark only one oval.

- None
- Minimum
- Basic
- Intermediary
- Advanced
- Expert

24. **Eclipse IDE:** *

Mark only one oval.

- None
- Minimum
- Basic
- Intermediary
- Advanced
- Expert

25. **Other OO Programming Language:**
Mark only one oval.

- ◯ None
- ◯ Minimum
- ◯ Basic
- ◯ Intermediary
- ◯ Advanced
- ◯ Expert

26. **Specify:**

____

Powered by
Google Forms

B.2 Study Questionnaires

Next we present the questionnaire used during the experiment and the questionnaire post-experiment.

Diagnosing Design Problems: Tasks

* Required

1. **Name:** *

2. **Phase:** *
   Mark only one oval.
   - ◯ 1 *Skip to question 3.*
   - ◯ 2 *Skip to question 4.*

Basic Training

Task 1) You will receive a basic training about code anomalies and design problems. The training will be 15 minutes long.

3. **Task 1: before starting this task, indicate here what time it is now:** *

   Example: 8:30 AM

Configuration

4. **Component:** *
   Mark only one oval.
   - ◯ PushPull
   - ◯ Workflow

Task 2) You will have 20 minutes to understand the component above. For this task, you should read the component's documentation and source code, which were provided to you before you start.

5. **Task 2: before starting this task, indicate here what time it is now:** *

   Example: 8:30 AM

6. **Technique:** *
   Mark only one oval.
   - ◯ Traditional Technique *Skip to question 7.*
   - ◯ Synthesis Technique *Skip to question 18.*

Tasks with Traditional Technique

Task 3) You will understand how to use traditional technique to diagnose design problems. In this task, you will receive a guide and a basic training about the traditional technique. The training will be 10 minutes long.

7. **Task 3: before starting this task, indicate here what time it is now:** *

   Example: 8:30 AM

Task 4) In this task, you will use traditional technique to diagnose DESIGN PROBLEMS. For each problem found, you have to provide the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods and components realizing the problem in the source code, and (iv) name(s) of code anomaly(ies) that helped you to diagnose the design problem. You will have 40 minutes to finish this task.

8. **Task 4: before starting this task, indicate here what time it is now:** *

   Example: 8:30 AM

9. **Design problems:** *

   (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods and components realizing the problem in the source code, and (iv) name(s) of code anomaly(ies) that helped you to diagnose the design problem

Answer the following questions regarding Task 4:

10. **A - Which were the main challenges to diagnose design problems?** *

11. **B - Did you understand all information provided by the technique? Please provide details about this.** *

12. **C - Which types of information were fundamental to diagnose design problems? Please rank these types of information according to their relevance.** *

13. **D – Was there any piece of provided information that was useless to perform Task 4? Why and Which one(s)?** *

14. **E - Do you feel there is any non-provided information that could help to diagnose design problems? Please explain what additional information would be helpful to diagnose the design problems.** *

15. **F - Have you used all types of code anomaly? Please provide details about this.** *

16. **G – Which was the most useful type of code anomaly?** *

17. **H – How the graphical interface provided by the technique affected Task 4?** *

Stop filling out this form.

Tasks with Synthesis Technique

Task 3) You will understand how to use synthesis technique to diagnose design problems. In this task, you will receive a guide and a basic training about the synthesis technique. The training will be 10 minutes long.

18. **Task 3: before starting this task, indicate here what time it is now:** *

Example: 8:30 AM

Task 4) In this task, you will use synthesis technique to diagnose DESIGN PROBLEMS. For each problem found, you have to provide the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods and components realizing the problem in the source code, and (iv) name(s) of agglomeration(s) that helped you to diagnose the design problem. You will have 40 minutes to finish this task.

19. **Task 4: before starting this task, indicate here what time it is now:** *

Example: 8:30 AM

20. **Design problems:** *
(i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods and components realizing the problem in the source code, and (iv) name(s) of code anomaly(ies) that helped you to diagnose the design problem

Answer the following questions regarding Task 4:

21. **A - Which were the main challenges to diagnose design problems?** *

22. **B - Did you understand all information provided by the technique? Please provide details about this.** *

23. **C - Which types of information were fundamental to diagnose design problems? Please rank these types of information according to their relevance.** *

24. **D – Was there any piece of provided information that was useless to perform Task 4? Why and Which one(s)? ***

25. **E - Do you feel there is any non-provided information that could help to diagnose design problems? Please explain what additional information would be helpful to diagnose the design problems. ***

26. **F - Have you used all categories of agglomerations? Please provide details about this. ***

27. **G – Which was the most useful category of agglomeration? ***

28. **H – How the graphical interface provided by the technique affected Task 4? ***

Powered by

B.3
Push Pull and Workflow Blueprints

Next we present the high-level description of their design blueprint for the Push Pull and Workflow softoware systems.

Push-Pull-Framework

The Push Pull framework is responsible for downloading remote content (pull), or accepting the delivery of remote content (push) to a local staging area. Content in the staging area is ingested into the File Manager system by the Crawler Framework. The Push Pull framework is extensible and provides a fully tailorable Java-based API for the acquisition of remote content.

Architecture

This section describes the architecture of the Push Pull framework, including its constituent components, object model, and extension points.

Architectural Components

The major components of the Push Pull Framework are the Daemon Launcher, the Daemon, the Protocol Layer, and the File Retrieval System, to name a few. The relationship between all of these components are shown in Figure 1.

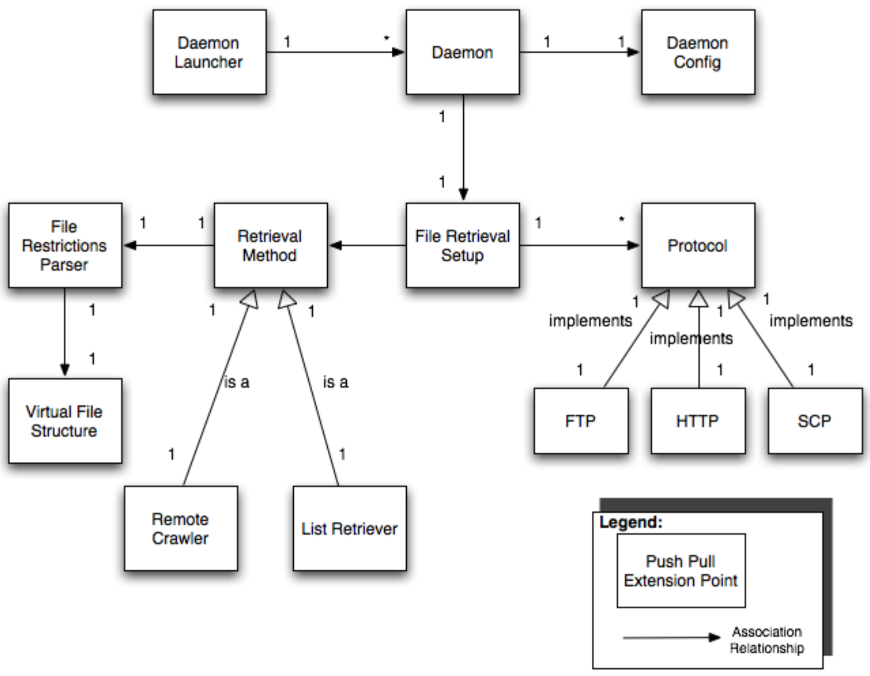


Figure 1. Push Pull Components

The Push Pull Framework provides a Daemon Launcher, responsible for creating new Daemon instances. Each Daemon has an associated Daemon Configuration, and has the ability to use a File Retrieval Setup extension point. This class is responsible for leveraging both a Protocol and a File Retrieval System to obtain ProtocolFiles, based on a File Restrictions Parser that yields eventually a VirtualFileStructure (VFS) model. The VFS defines what files to accept and pull down from a remote site.

Object Model

The critical objects managed by the Push Pull Framework include:

- **Protocol –** A means of obtaining content over some file acquisition method, e.g., FTP, SCP, HTTP, etc.
- **Protocol File** - *Metadata* information about a remote file, including its *ProtocolPath*.
- **Protocol Path** - A pointer to a remote *Product* file's (or files') location, which can be used to derive metadata and determine where to place the file in the local staging area built by the Push Pull Framework.
- **Remote Site** - Descriptive information about a remote site, including the username/password combination, as well as an origin directory to start interrogating.

Each *Protocol* delivers one or more Protocol Files. Each *ProtocoFile* is associated with a single *RemoteSite*, and each *ProtocolFile* is associated with a single *ProtocolPath*. These relationships are shown in Figure 2.

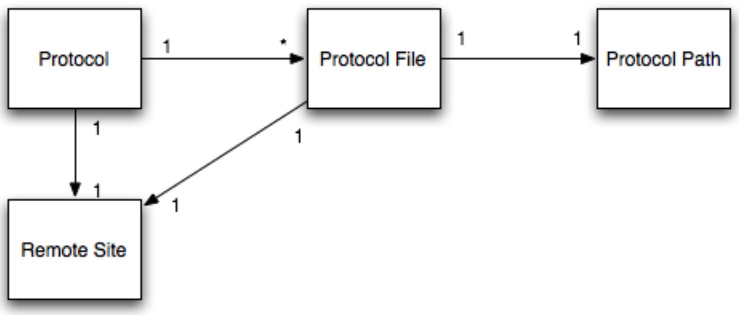


Figure 2. Push Pull Object Model

Extension Points

Push Pull Framework was constructed by making use of the Factory Method pattern. The use of this pattern was intended to provide multiple extension points for the Push Pull Framework. An extension point is an interface within the Push Pull Framework that can have many implementations. This is particularly useful when it comes to software component configuration because it allows different implementations of an existing interface to be selected at deployment time. Each of the core extension points for the Push Pull Framework is described in TABLE I.

TABLE I. Push Pull Extension Points

Extension Point	Description
Protocol	The Protocol extension point is the heart of the Push Pull framework. It is responsible for modeling remote sites and for obtaining their content via different Retrieval Methods, using different File Restrictions Parsers.
Retrieval Method	The Retrieval Method extension point is responsible for orchestrating download (pull) and acceptance (push) of remote content.
File Restrictions Parser	The File Restrictions Parser extension point is responsible for defining how to accept or decline files encountered by a Retrieval Method, in essence modeling remote file and directory structures.
System	The extension point that provides the external interface to the Push Pull Framework services. This includes the Daemon Launcher interface, as well as the associated Daemon interface, that is managed by the Daemon Launcher.

Workflow Management Component

The Workflow Manager component is responsible for description, execution, and monitoring of Workflows, using a client-server system. Workflows are typically considered to be sequences of tasks, joined together by control flow and data flow, which must execute in some ordered fashion. Workflows typically generate output data, perform routine management tasks (such as sending emails, etc.), and/or describe a business's internal routine practices. The Workflow Manager is an extensible software component that provides an XML-RPC external interface, and a fully tailorable (i.e. adaptable) Java-based API for workflow management.

Architecture

This section describes the Workflow Manager architecture, including its constituent components, object model, and extension points.

Architectural Components

The major architectural components of Workflow Manager are the Client and Server, the Workflow Repository, the Workflow Engine, and the Workflow Instance Repository. The relationship between all of these components are shown in Figure 1.

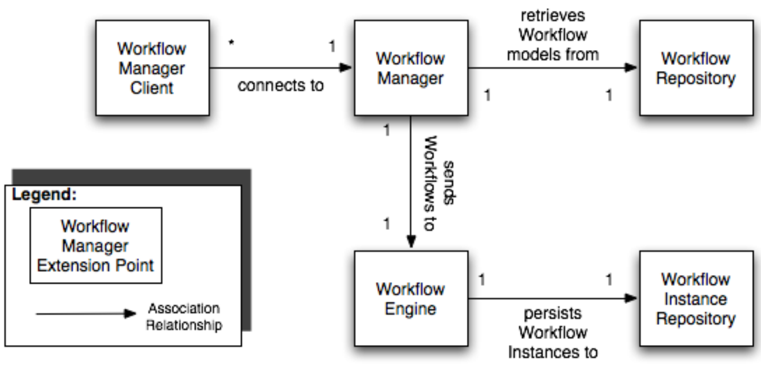


Figure 1. Architectural components of Workflow Manager

The Workflow Manager Server contains both a Workflow Repository that manages workflow models, and Workflow Engine that processes workflow instances. The Workflow Engine also has a persistence layer, called a Workflow Instance Repository, which is responsible for saving workflow instance metadata and state.

Object Model

The critical objects managed by Workflow Manager include:

- **Events** - are the stimuli that trigger Workflows to be executed. Events are named and contain dynamic Metadata information passed in by the user.
- **Metadata** - a dynamic set of properties and values provided to a *WorkflowInstance* via an user-triggered Event.

- **Workflow** - a description of both the control flow and data flow of a sequence of tasks (or stages) that must be executed in some order.
- **Workflow Instance** - an instance of a *Workflow*, typically containing additional runtime descriptive information, such as start time, end time, task clock time, etc. A *WorkflowInstance* also contains a shared *Metadata* context passed in by the user who triggered the *Workflow*. This context can be read/written to by the underlying *WorkflowTasks* present in a *Workflow*.
- **Workflow Tasks** - descriptions of data flow, and an underlying process, or stage, that is part of a *Workflow*.
- **Workflow Task Instances** - the actual executing code, or process, that performs the work in the *WorkflowTask*.
- **Workflow Task Configuration** - static configuration properties that configure a *WorkflowTask*.
- **Workflow Conditions** - any pre (or post) conditions on the execution of a *WorkflowTask*.
- **Workflow Condition Instances** - the actual executing code, or process, that performs the work in the *WorkflowCondition*.

Each Event kicks off one or more *Workflow* Instances, providing a *Metadata* context (submitted by an external user). Each *WorkflowInstance* is a run-time execution model of a *Workflow*. Each *Workflow* contains one or more *WorkflowTasks*. Each *WorkflowTask* contains a single *WorkflowTaskConfiguration*, and one or more *WorkflowConditions*. Each *WorkflowTask* has a corresponding *WorkflowTaskInstance* (that it models), as well as it does each *WorkflowCondition* has a corresponding *WorkflowConditionInstance*. These relationships are shown in Figure 2.

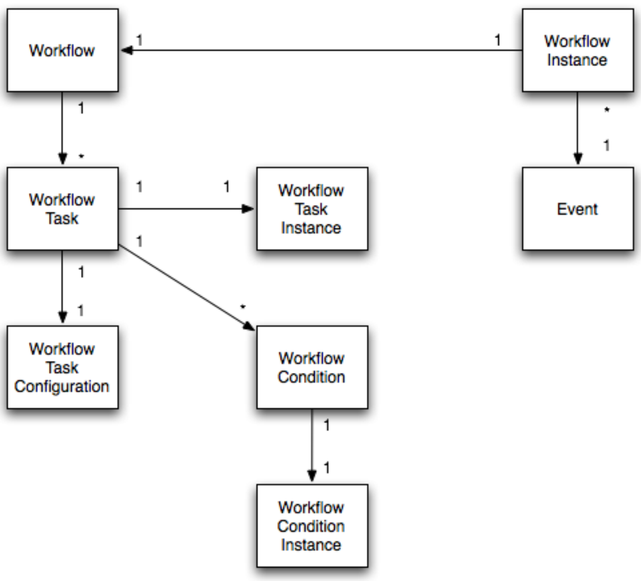


Figure 2. Object Model for Workflow Manager

Extension Points

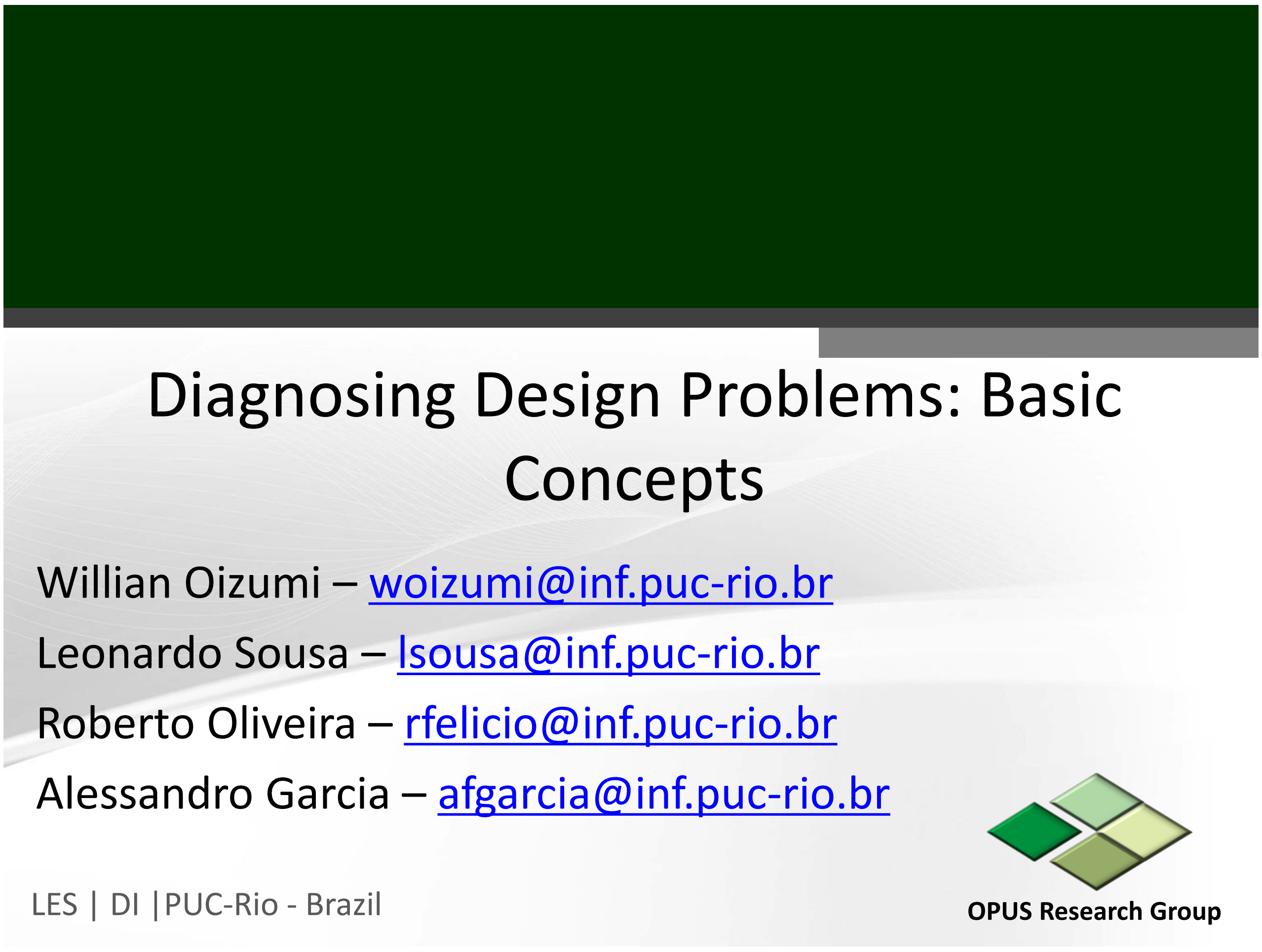
The Workflow Manager was built with the Factory Method pattern to provide multiple extension points for the Workflow Manager. An extension point is an interface within the Workflow Manager that can have many implementations. This is particularly useful when it comes to software component configuration because it allows different implementations of an existing interface to be selected at deployment time. Each of the core extension points for Workflow Manager is described in TABLE I.

TABLE I. Extension Points of Workflow Manager

Extension Point	Description
Workflow Instance Repository	The Workflow Instance Repository extension point is responsible for storing all the instance data for Workflow Instances, including shared context metadata, and runtime properties, such as start date time and end date time.
Workflow Repository	The Workflow Repository extension point is responsible for managing Workflow models, storing control flows and data flows. The Workflow Repository also stores Workflow Condition information, and Workflow Task Configuration. In essence, the Workflow Repository is a repository of abstract Workflow models, which get mapped to Workflow Instances by the Engine extension point.
Workflow Engine	The Workflow Engine's responsibility is to map abstract Workflow models to executing Workflow Instances. The Workflow Engine tracks and monitors execution of Workflow Instances, and provides the ability to start, stop and pause executing Workflow Instances.
System	The extension point that provides the external interface to the Workflow Manager services. This includes the Workflow Manager server interface, as well as the associated Workflow Manager client interface, which communicates with the server.

B.4 Presentation

Following we present the slides that we used during the training session.

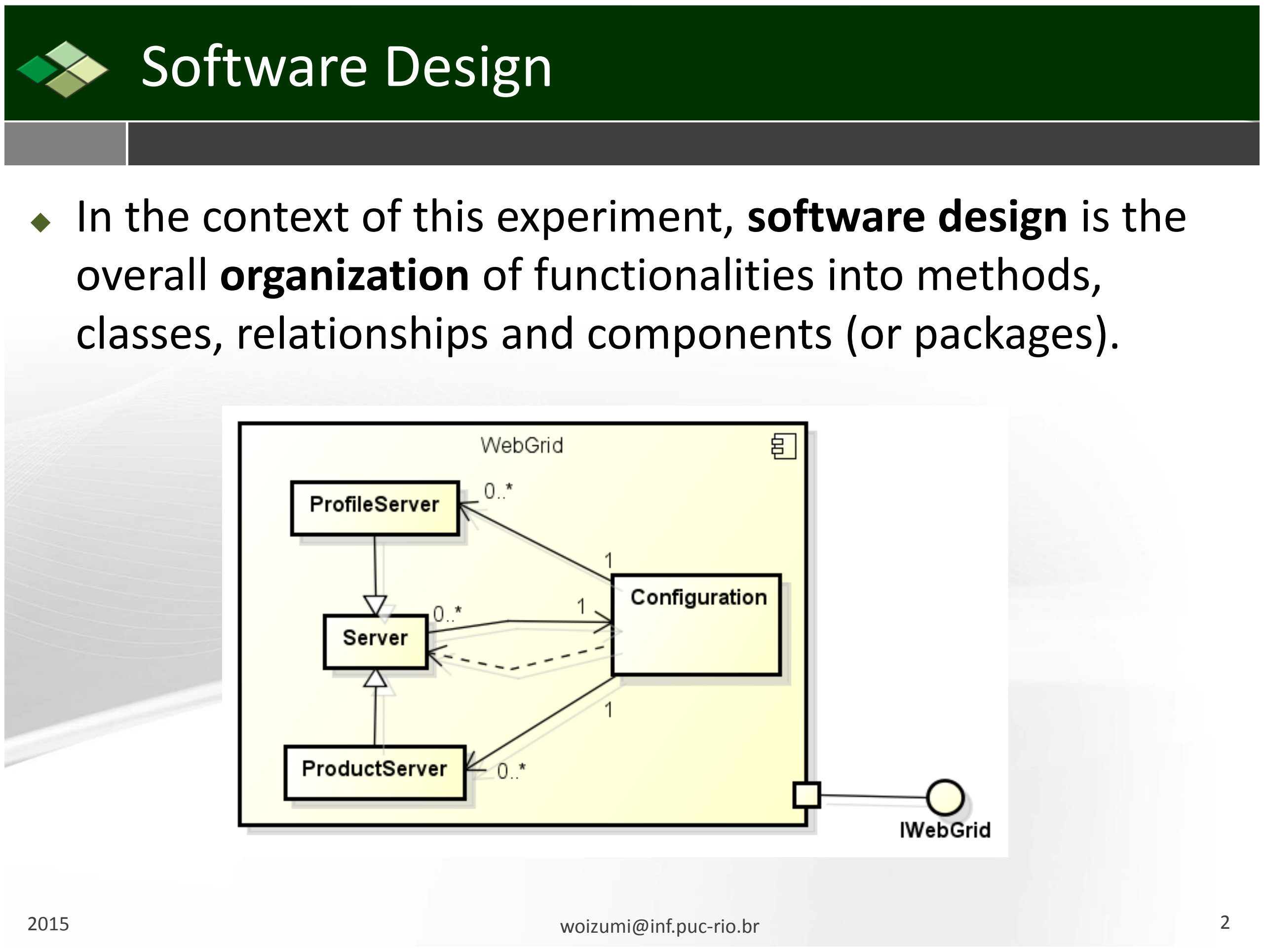


Diagnosing Design Problems: Basic Concepts

Willian Oizumi – woizumi@inf.puc-rio.br
Leonardo Sousa – lsousa@inf.puc-rio.br
Roberto Oliveira – rfelicio@inf.puc-rio.br
Alessandro Garcia – afgarcia@inf.puc-rio.br

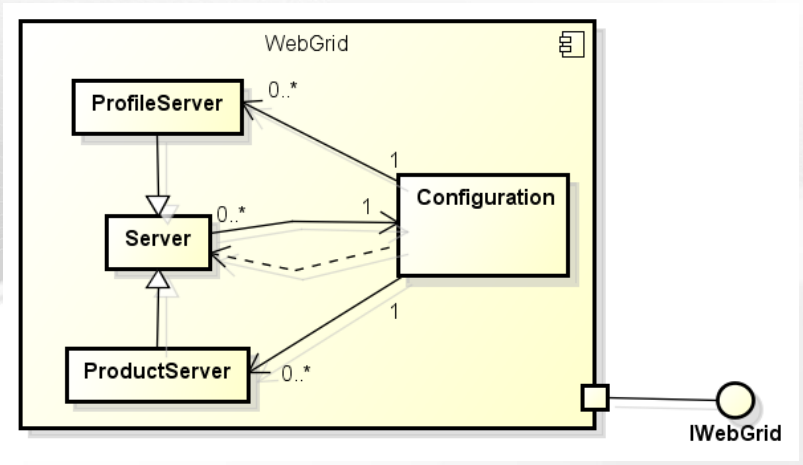
LES | DI |PUC-Rio - Brazil

OPUS Research Group

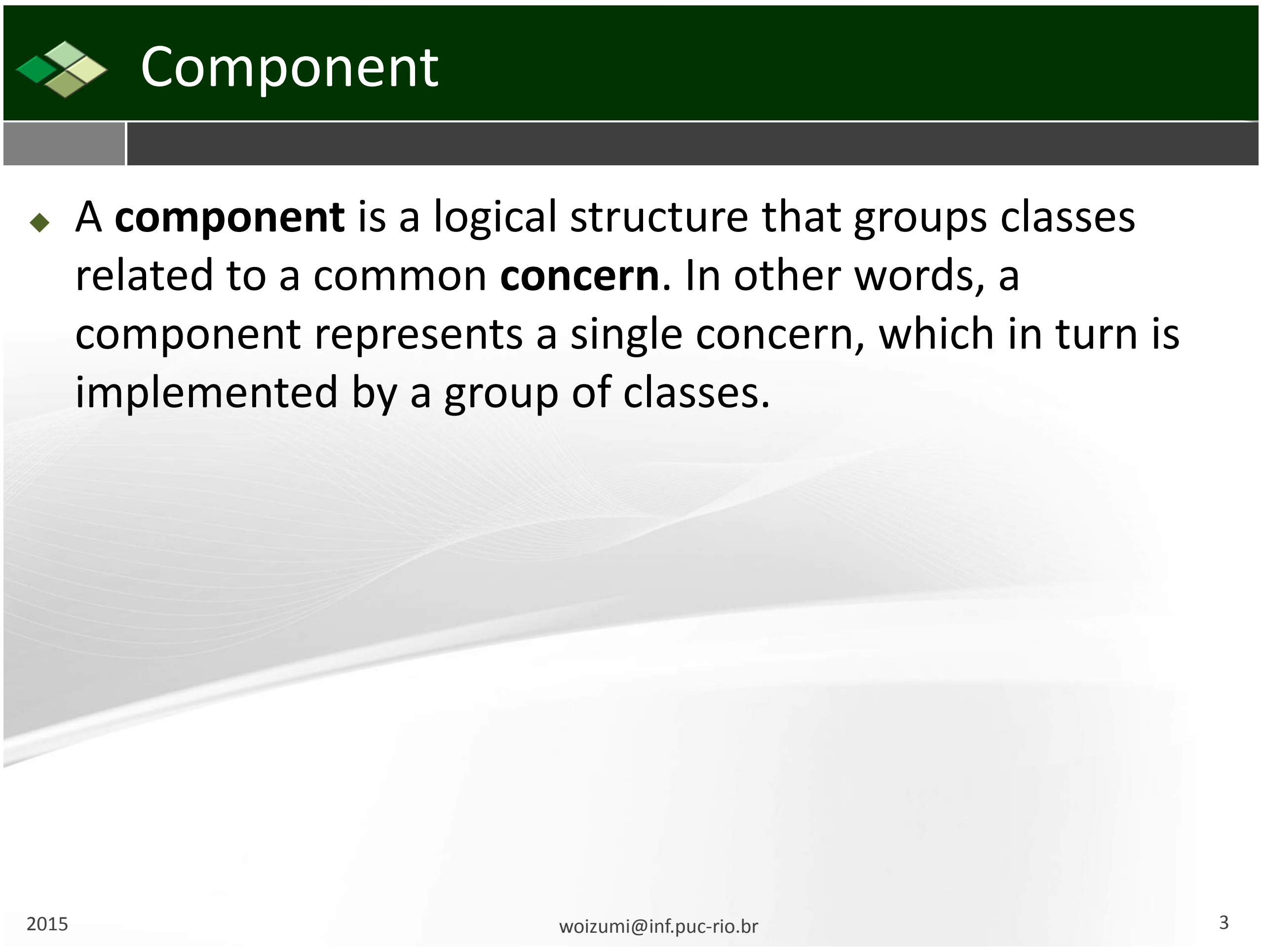


Software Design

- In the context of this experiment, **software design** is the overall **organization** of functionalities into methods, classes, relationships and components (or packages).



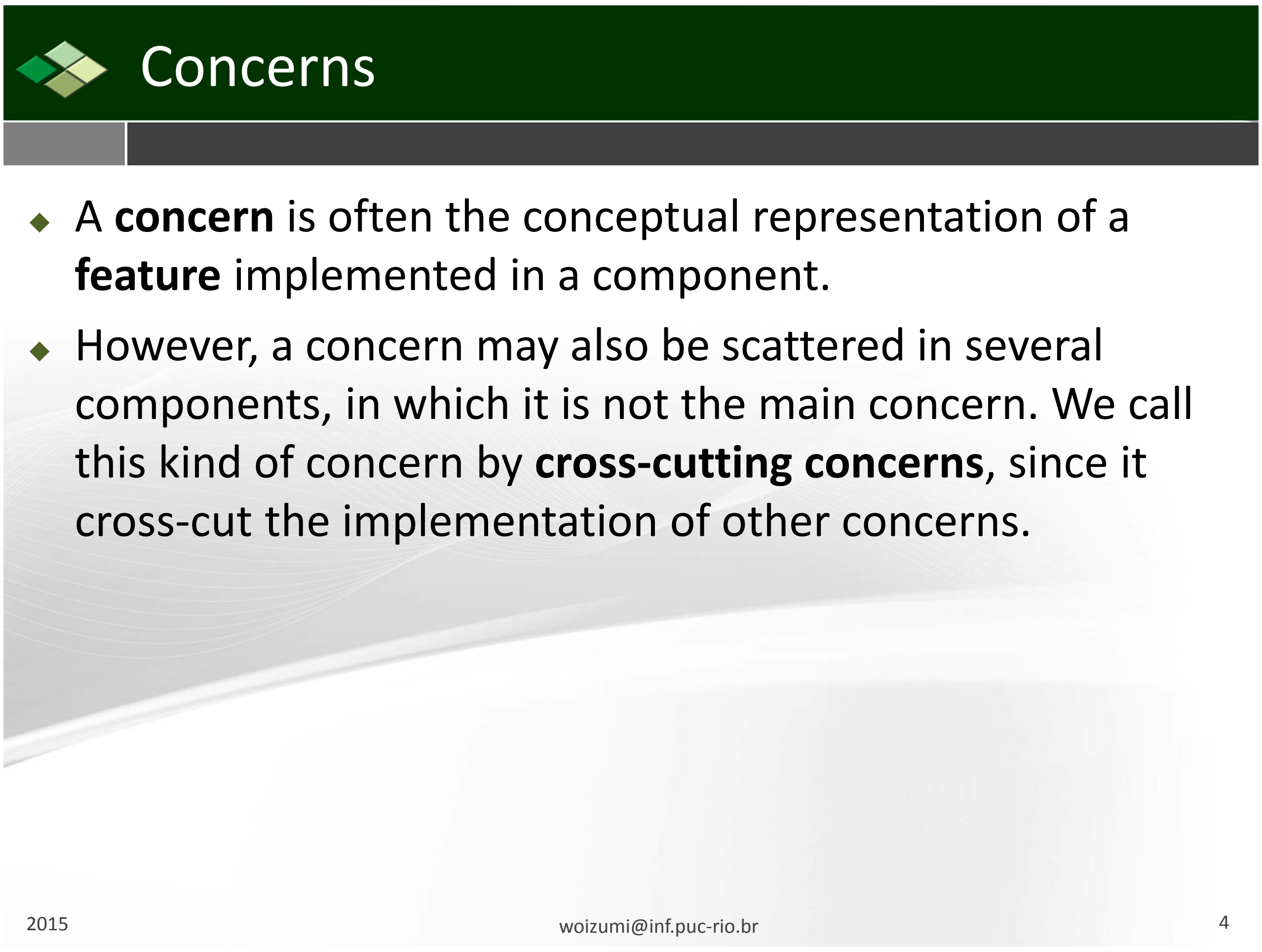
2015 woizumi@inf.puc-rio.br 2



Component

- A **component** is a logical structure that groups classes related to a common **concern**. In other words, a component represents a single concern, which in turn is implemented by a group of classes.

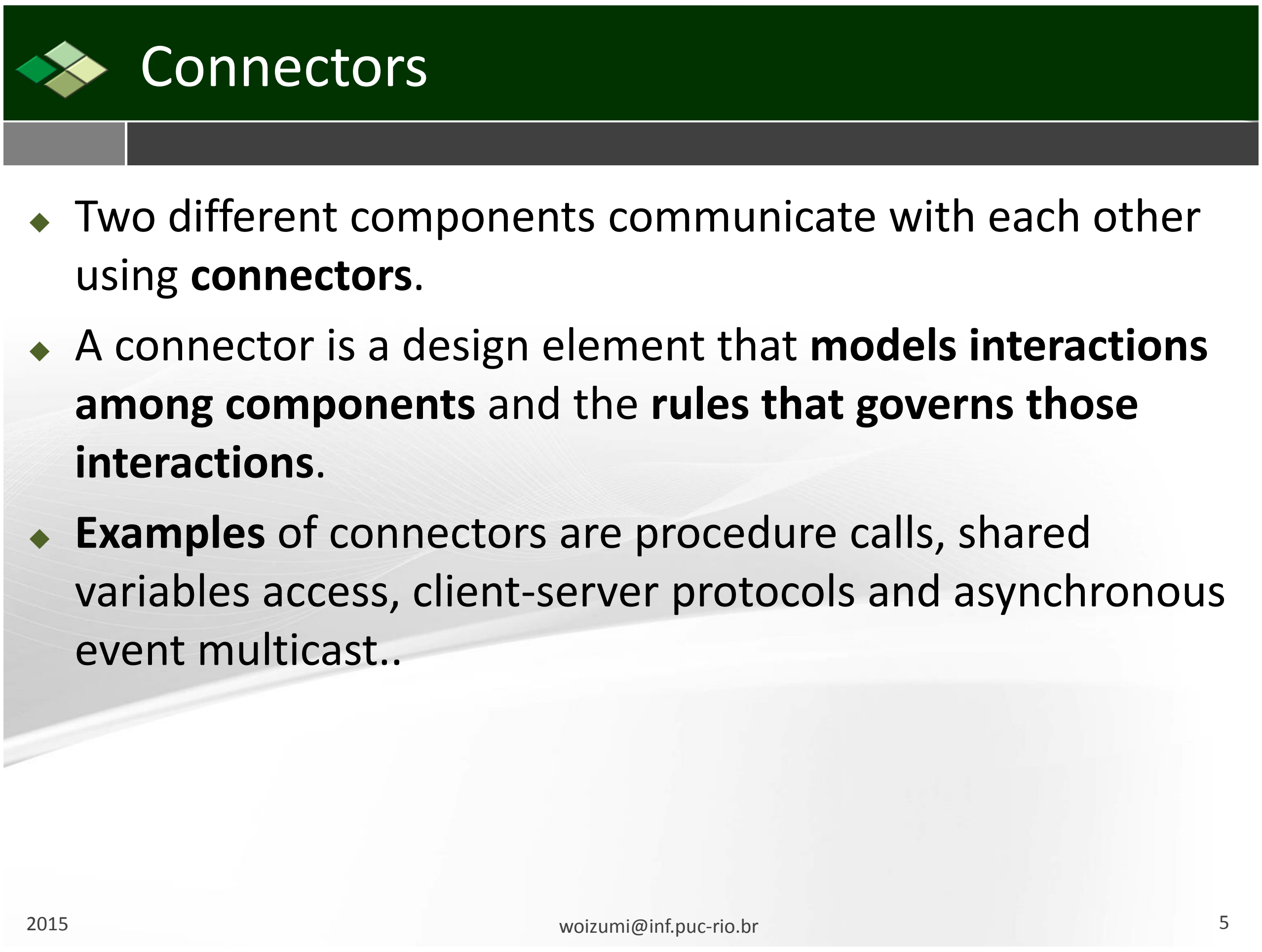
2015 woizumi@inf.puc-rio.br 3



Concerns

- A **concern** is often the conceptual representation of a **feature** implemented in a component.
- However, a concern may also be scattered in several components, in which it is not the main concern. We call this kind of concern by **cross-cutting concerns**, since it cross-cut the implementation of other concerns.

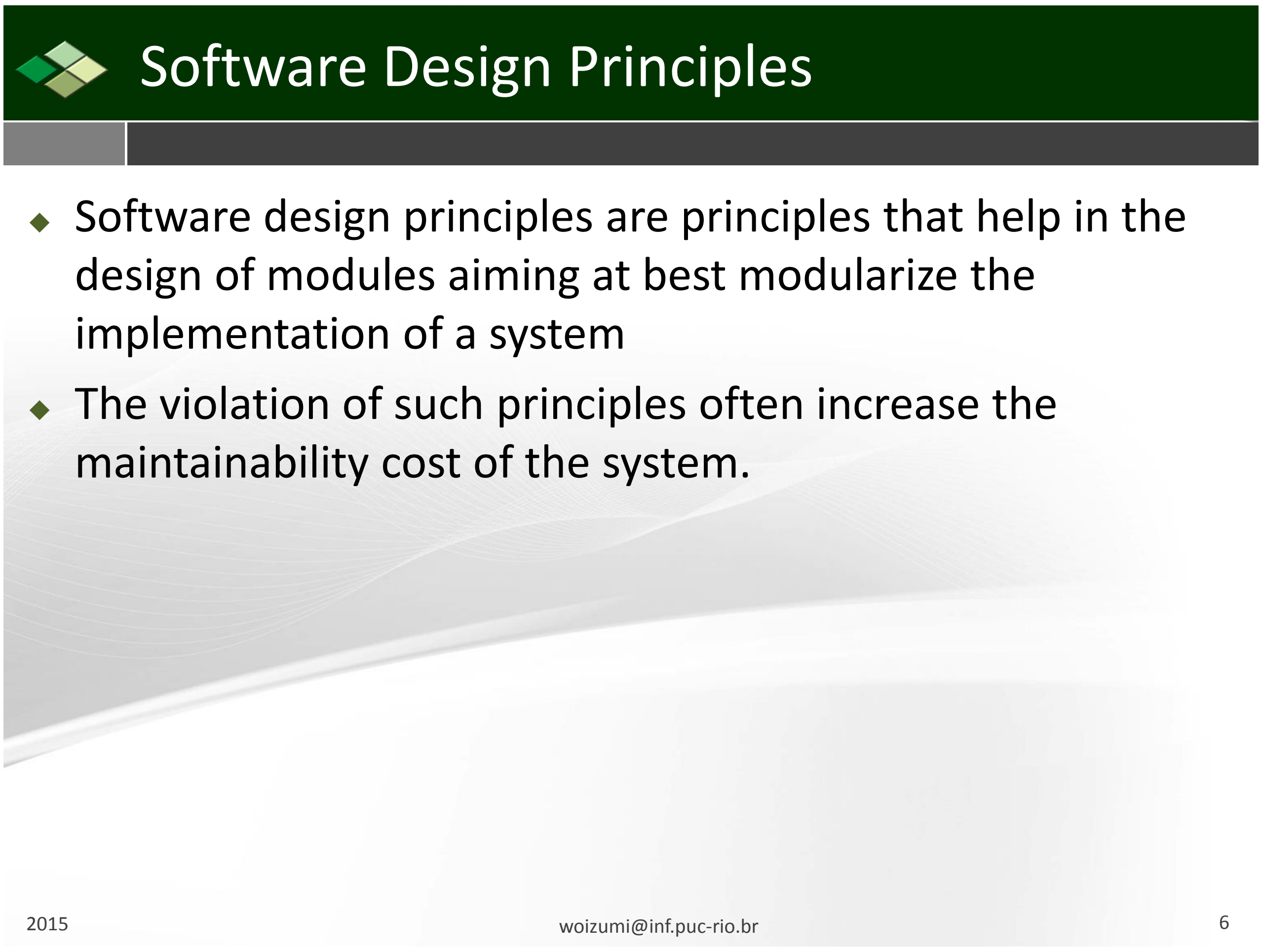
2015 woizumi@inf.puc-rio.br 4



Connectors

- Two different components communicate with each other using **connectors**.
- A connector is a design element that **models interactions among components** and the **rules that governs those interactions**.
- **Examples** of connectors are procedure calls, shared variables access, client-server protocols and asynchronous event multicast..

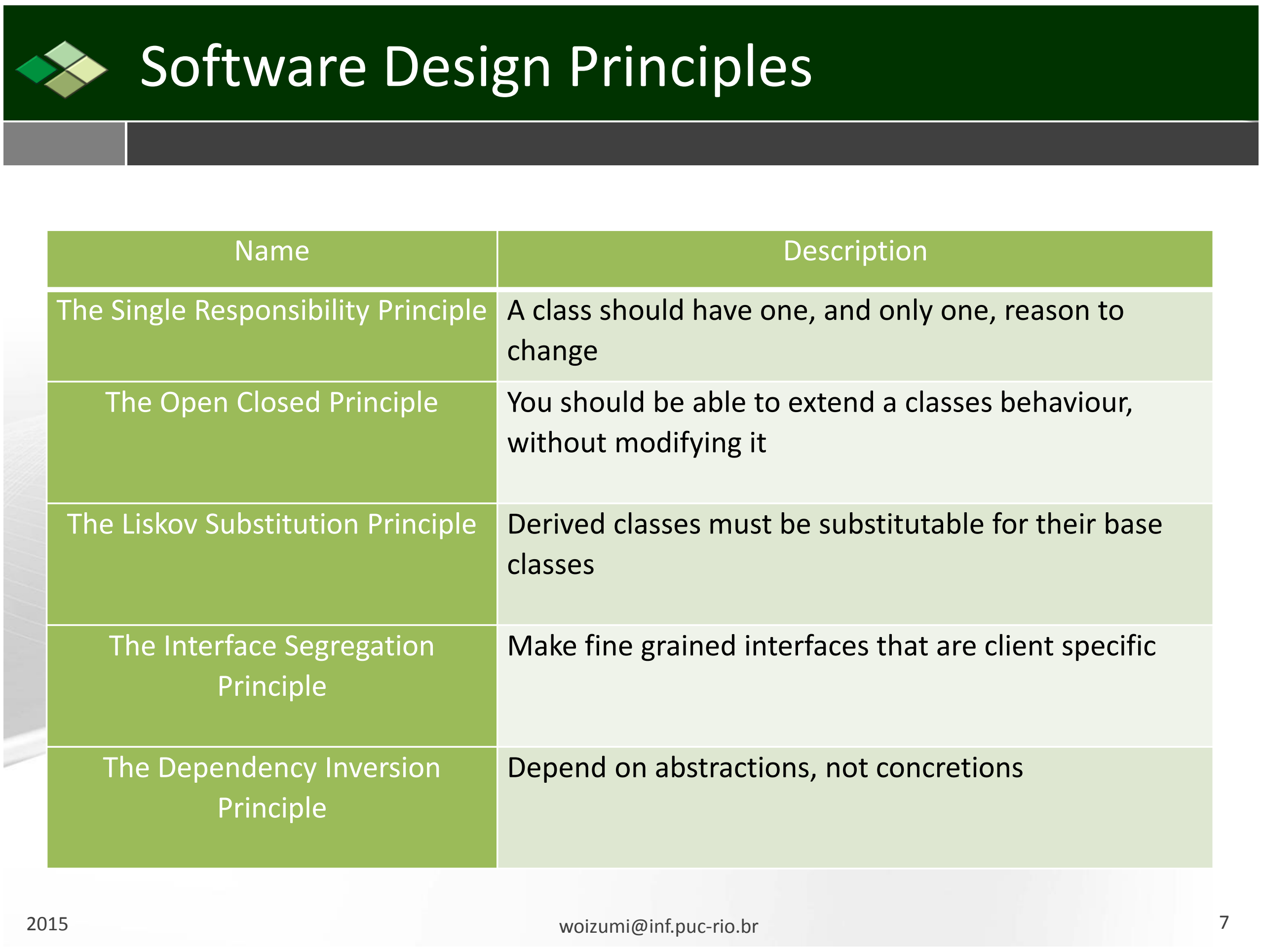
2015 woizumi@inf.puc-rio.br 5



Software Design Principles

- Software design principles are principles that help in the design of modules aiming at best modularize the implementation of a system
- The violation of such principles often increase the maintainability cost of the system.

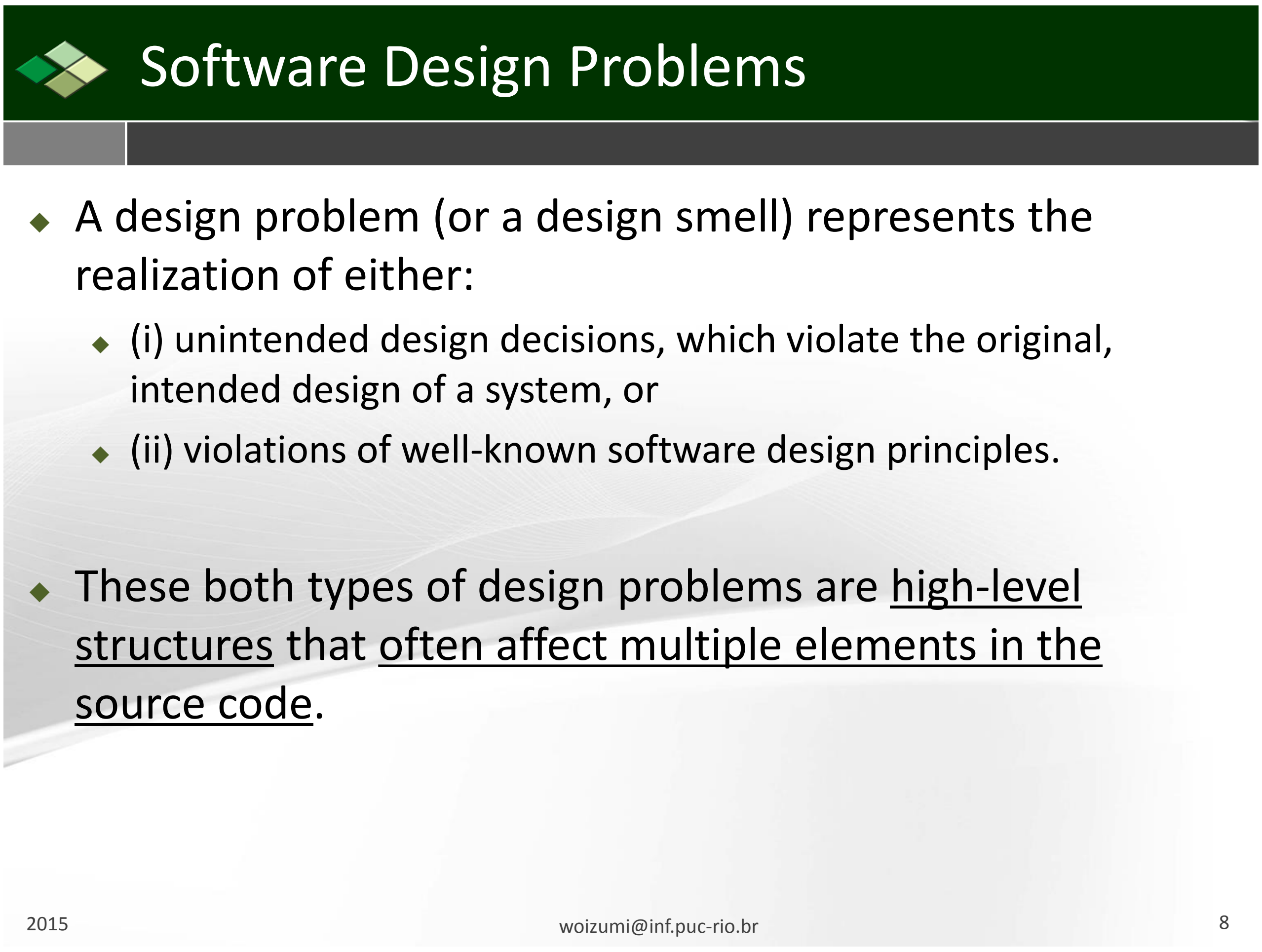
2015 woizumi@inf.puc-rio.br 6



Software Design Principles

Name	Description
The Single Responsibility Principle	A class should have one, and only one, reason to change
The Open Closed Principle	You should be able to extend a classes behaviour, without modifying it
The Liskov Substitution Principle	Derived classes must be substitutable for their base classes
The Interface Segregation Principle	Make fine grained interfaces that are client specific
The Dependency Inversion Principle	Depend on abstractions, not concretions

2015 woizumi@inf.puc-rio.br 7

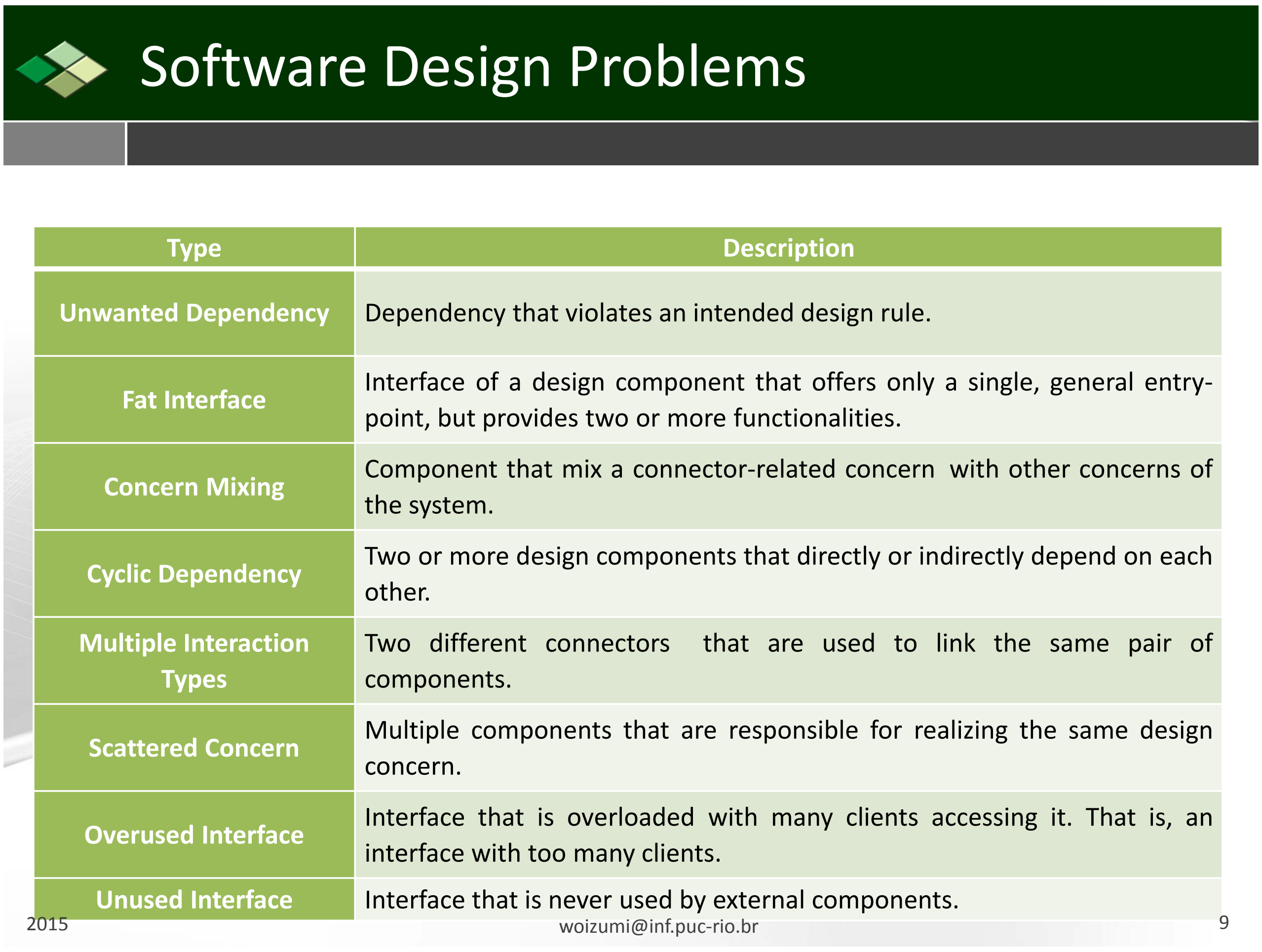


Software Design Problems

- A design problem (or a design smell) represents the realization of either:
  - (i) unintended design decisions, which violate the original, intended design of a system, or
  - (ii) violations of well-known software design principles.

- These both types of design problems are <u>high-level structures</u> that <u>often affect multiple elements in the source code</u>.

2015 woizumi@inf.puc-rio.br 8



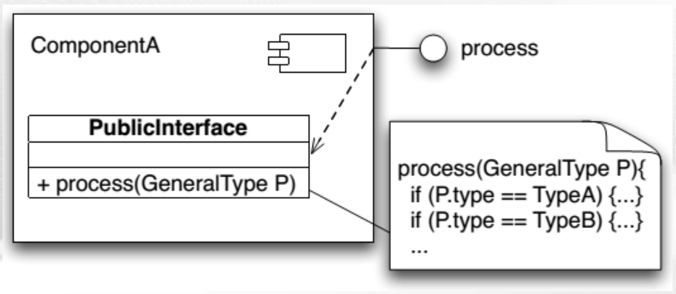
Software Design Problems

Type	Description
Unwanted Dependency	Dependency that violates an intended design rule.
Fat Interface	Interface of a design component that offers only a single, general entry-point, but provides two or more functionalities.
Concern Mixing	Component that mix a connector-related concern with other concerns of the system.
Cyclic Dependency	Two or more design components that directly or indirectly depend on each other.
Multiple Interaction Types	Two different connectors that are used to link the same pair of components.
Scattered Concern	Multiple components that are responsible for realizing the same design concern.
Overused Interface	Interface that is overloaded with many clients accessing it. That is, an interface with too many clients.
Unused Interface	Interface that is never used by external components.

2015 woizumi@inf.puc-rio.br 9

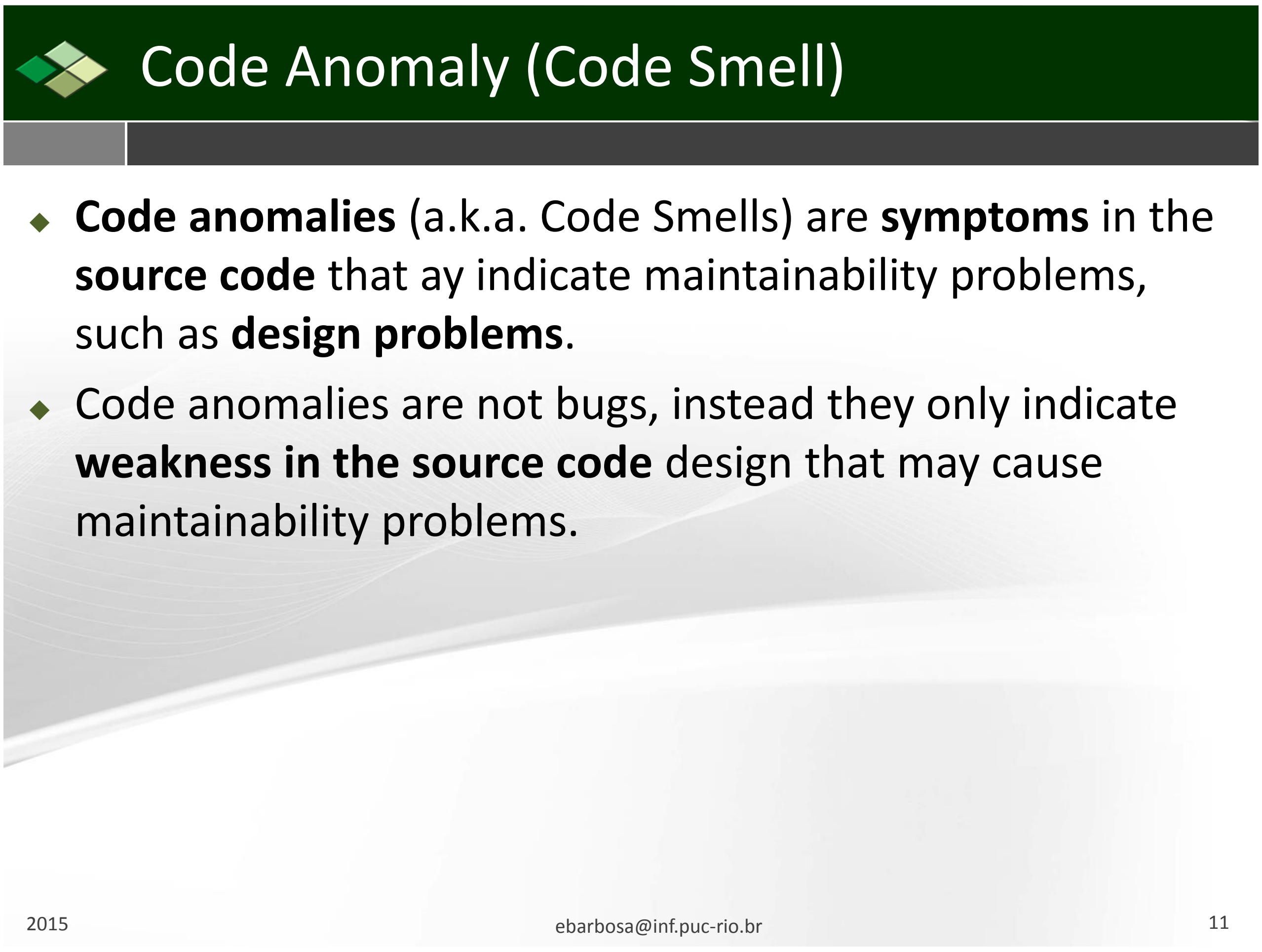
Fat Interface

- Interface of a design component that offers only a single, general entry-point, but provides two or more functionalities.



```
process(GeneralType P){
  if (P.type == TypeA) {...}
  if (P.type == TypeB) {...}
  ...
```

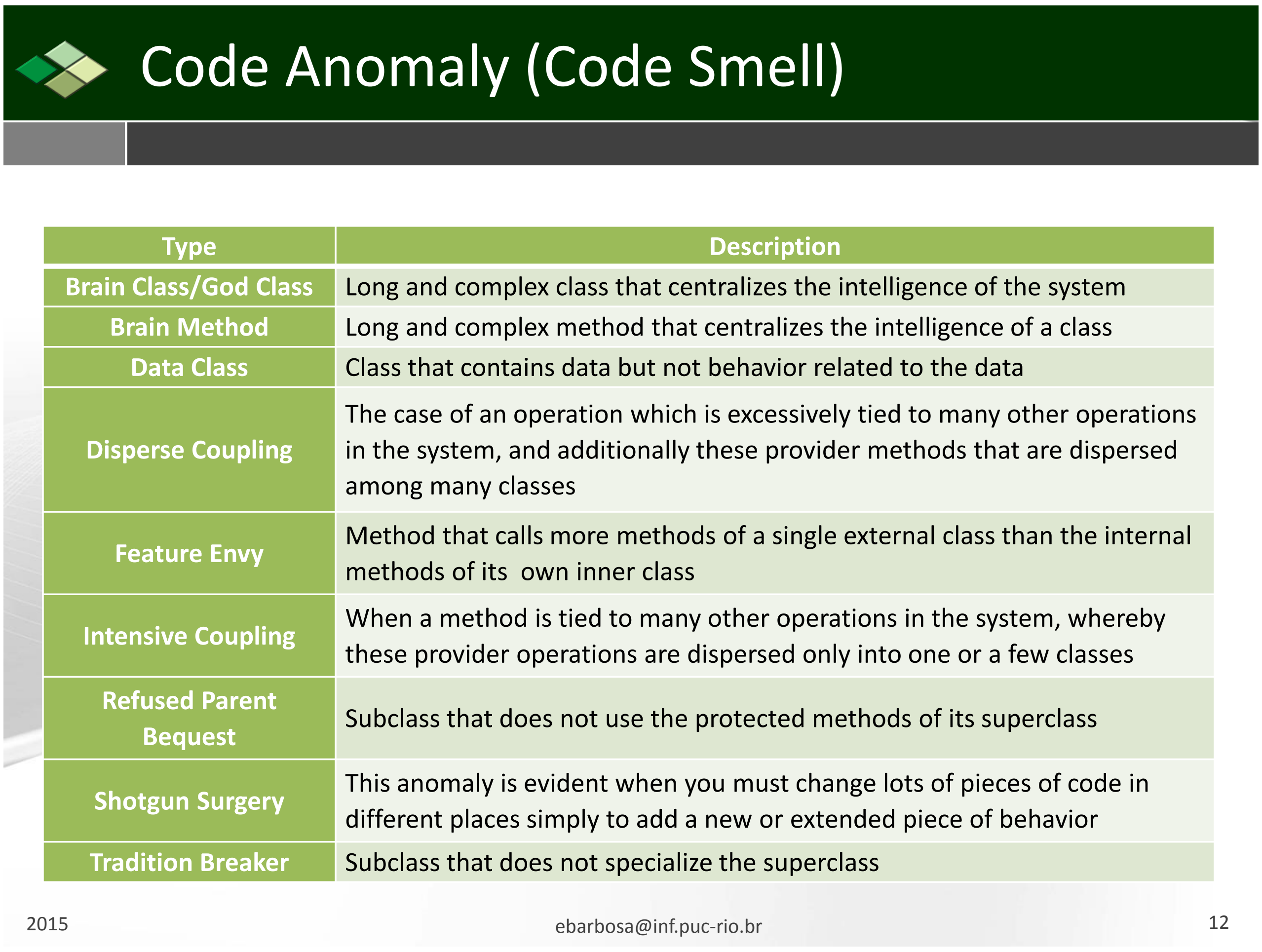
2015 woizumi@inf.puc-rio.br 10



Code Anomaly (Code Smell)

- **Code anomalies** (a.k.a. Code Smells) are **symptoms** in the **source code** that ay indicate maintainability problems, such as **design problems**.
- Code anomalies are not bugs, instead they only indicate **weakness in the source code** design that may cause maintainability problems.

2015 ebarbosa@inf.puc-rio.br 11



Code Anomaly (Code Smell)

Type	Description
Brain Class/God Class	Long and complex class that centralizes the intelligence of the system
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Disperse Coupling	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature Envy	Method that calls more methods of a single external class than the internal methods of its own inner class
Intensive Coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused Parent Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	This anomaly is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior
Tradition Breaker	Subclass that does not specialize the superclass

2015 ebarbosa@inf.puc-rio.br 12

C
Study about Design Problem Identification in Practice

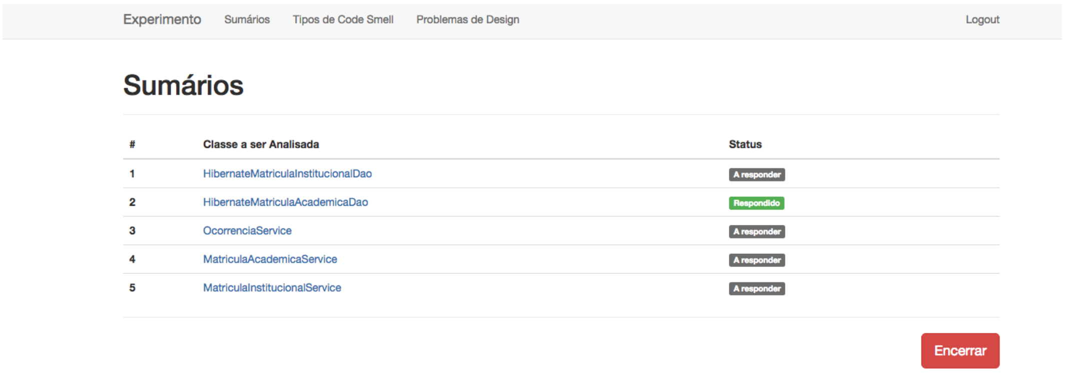
This appendix presents the subject characterization questionnaire, the summary of symptoms and the characterization of the theory.

C.1
Characterization and Follow-up Questionnaires

These are similar to the questionnaires presented in Appendix B.

C.2
Summary of Symptoms

We summarized and presented all symptoms affecting an element to developers through a web page based on SonarQube (114). Figure C.1 shows the initial web page to which the developers select an element to analyze.



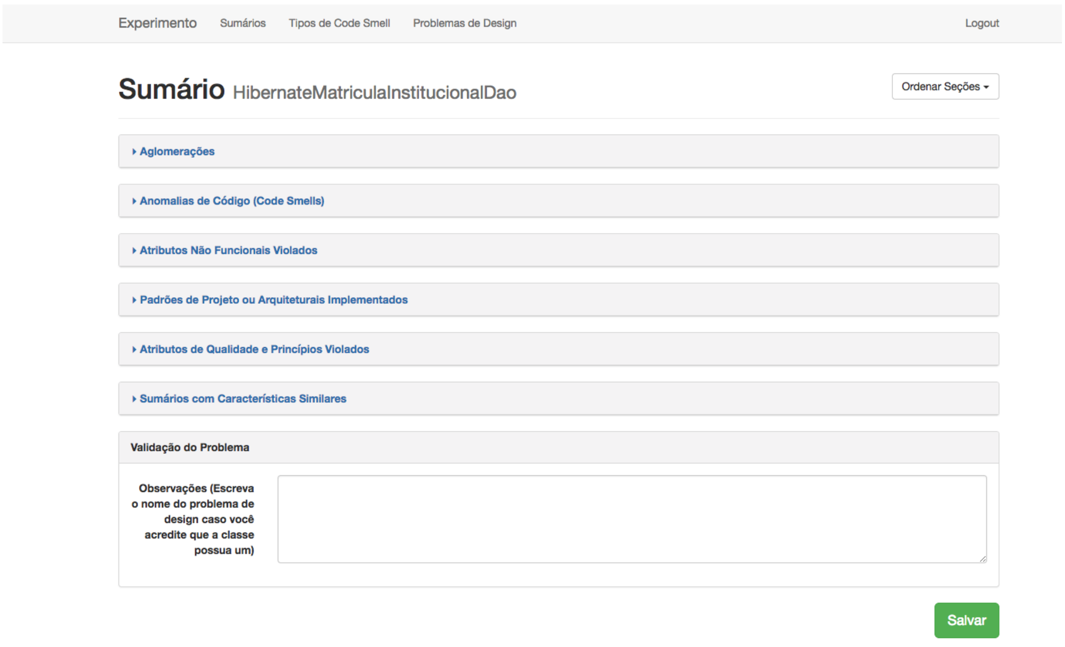
Sumários

#	Classe a ser Analisada	Status
1	HibernateMatriculaInstitucionalDao	A responder
2	HibernateMatriculaAcademicaDao	Respondido
3	OcorrenciaService	A responder
4	MatriculaAcademicaService	A responder
5	MatriculaInstitucionalService	A responder

Figure C.1: Home Page to Access the Summary

Figure C.2 shows all the design problems symptoms affecting the element selected by the developer. The figure also shows the specific field at the web page that developers could write any observation about the symptoms and the identified design problems.

Figures C.3 and C.4 show the information about two symptoms. Figure C.3 shows the code smells organized as an agglomeration (47), and Figure C.4 shows a non-functional requirement that might be violated.


Experimento Sumários Tipos de Code Smell Problemas de Design Logout

Sumário HibernateMatriculaInstitucionalDao

Ordenar Seções

- Aglomerações
- Anomalias de Código (Code Smells)
- Atributos Não Funcionais Violados
- Padrões de Projeto ou Arquiteturais Implementados
- Atributos de Qualidade e Princípios Violados
- Sumários com Características Similares

Validação do Problema

Observações (Escreva o nome do problema de design caso você acredite que a classe possua um)

Salvar

Figure C.2: Summary of Symptoms Affecting an Element

C.3 Generated Codes

Figure C.5 shows the first codes generated after analyzing the teams of Company 1 and 2.

Figure C.6 shows the codes generated after analyzing the teams of Company 3 and 4.

Figure C.7 shows the codes generated after analyzing the teams of Company 5.

C.4 Characterization of the Theory

Table C.1 presents the constructs and propositions that characterize the theory.

C.5 Design Problems

In order to support developers during the identification of design problems, we provided them a guide characterizing each one. In the following, we present all descriptions for the design problems used in our study.

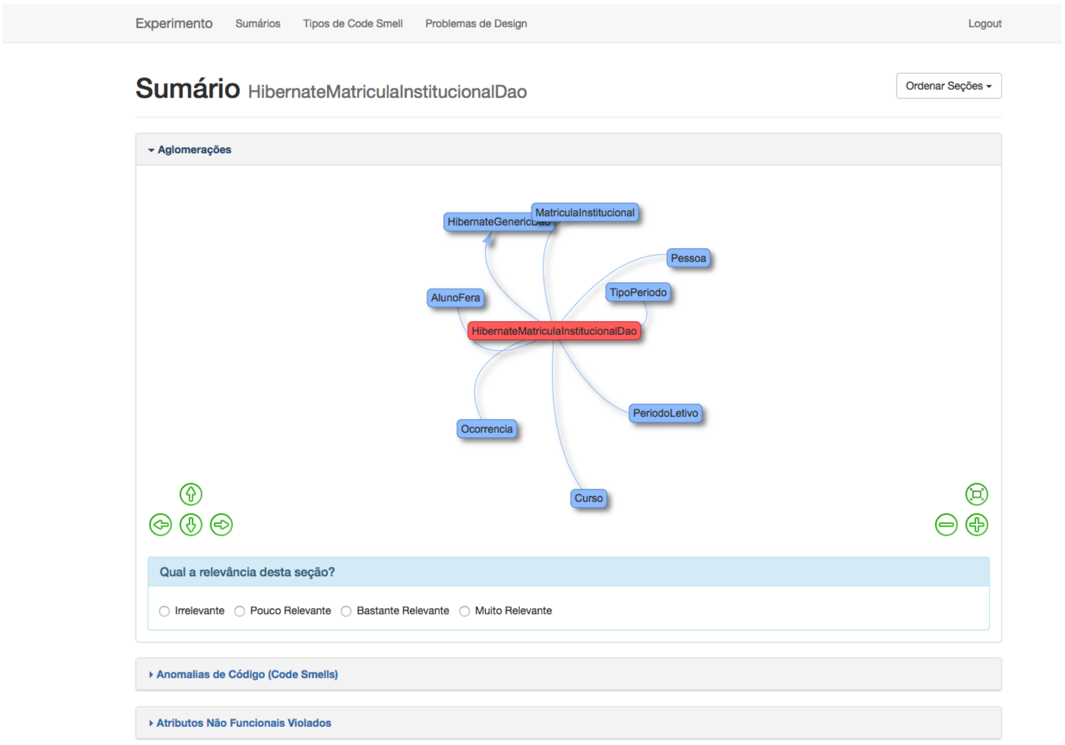
– Ambiguous Interface (AMI): Component interface that is ambiguous and provides non-cohesive services. It usually appears on systems where

Table C.1: Constructs and Propositions of the Resulting Theory

Constructs		
C1	Design Problem	A design decision that negatively impacts quality attributes
C2	Design Decisions	Decisions made during the software development process
C3	Affected Elements	Elements that manifest the presence of a design problem. These elements usually contain design problem symptoms.
C4	Symptom	An indication of the presence of a design problem
C5	Syndrome	A set of symptoms affecting the same code element
C6	Diagnosis	The process of identifying a design problem through the analysis of symptoms that manifest themselves in the source code
C7	Symptom Type	A category to which a set of symptoms with common characteristics belongs
C8	Accuracy	The degree to which a symptom is correct in indicating a design problem
C9	Density	The number of symptoms instances in a syndrome
C10	Relation	How two or more symptoms are connected
C11	Diversity	The degree to which a syndrome contains a variety of symptom types
C12	Symptom Analysis	The process of analyzing a set of symptoms affecting a single element
C13	Epidemic Analysis	The process of analyzing elements affected by the same set of symptoms
C14	Element Role	The function that an element plays in the software system
C15	Identification Tactic	A specific action, in which developers rely on the analysis of the elements and their symptoms, towards to the identification of a design problem.
C16	Tactic Type	The action that developers apply in their quest for design problems
C17	Tactic Step	The moment to which the developer applies the tactic
C18	Confidence	The degree to which they are convinced about the presence of a design problem
C19	Conscientiousness	A personality trait related to being careful, responsible, and persevering

Propositions	
P1	Inappropriate design decisions lead to design problems
P2	Design problems impact non-functionality requirements negatively
P3	The diagnosis affects the identification of design problems
P4	Identification of design problem has three steps in which developers relies on design problems symptoms in all of them
P5	The diversity of symptoms influences which symptoms the developer will use during the diagnosis
P6	The symptom accuracy influences which symptoms the developer will use during the diagnosis
P7	The density of symptoms influences which symptoms the developer will use during the diagnosis
P8	The type of symptom influences which symptoms the developer will use during the diagnosis
P9	The relation among the symptoms influences which symptoms the developer will use during the diagnosis
P10	The type of symptom affects the developer's choice of a epidemic element to be analyzed
P11	The role that the element plays on the system affects the developer's choice of a epidemic element to be analyzed
P12	The role that the element plays on the system is associate with the confirmation of the presence of a design problem
P13	Developers search for a specific design problem or a specific element, or they rely on the analysis of the symptoms to identify a design problem
P14	The moment to which the developer applies the tactic refers to one of the three steps in the design problem identification
P15	The diagnosis affects the developer's confidence in the presence of a design problem
P16	The symptom accuracy affects the developer's confidence in the presence of a design problem
P17	The density of symptoms affects the developer's confidence regarding the presence of a design problem
P18	The role that the element plays on the system influences the developer's confidence
P19	The consciousness affects the likelihood of a developer diagnoses a design problem
P20	The diversity of symptoms affects the conscientiousness of the developers
P21	The design decisions affects the confirmation of the presence of a design problem
P22	The design decisions affects the developer's confidence in the presence of a design problem

Scope
The theory is supposed to be applicable in systems in which developers intend to identify design problems by analyzing symptoms that manifest themselves in the source code.



Sumário HibernateMatriculaInstitucionalDao

Qual a relevância desta seção?

○ Irrelevante ○ Pouco Relevante ○ Bastante Relevante ○ Muito Relevante

Figure C.3: Agglomeration of Code Smells in the Element

components have interfaces implementing public methods with generic types as parameter.

– Cyclic Dependency (CCD): A relation between one or more elements that depend on each other.

– Concern Overload (CCO): Code element that fulfills too many responsibilities.

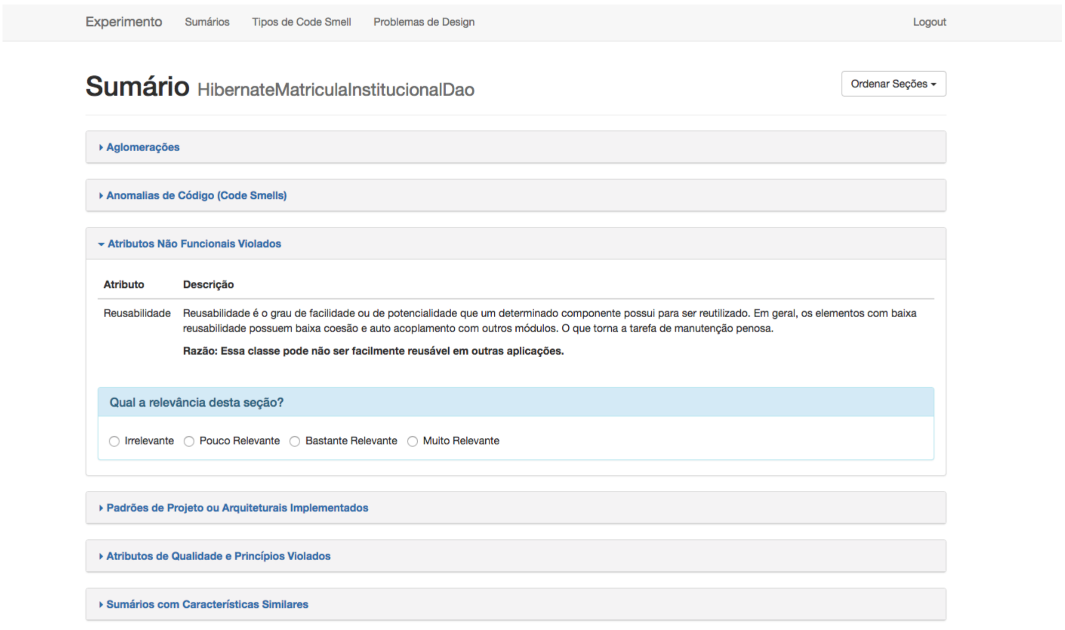
– Component Overload (CPO): Component overloaded with responsibilities/interests. Usually the component deals with too many functionalities, some of which are not related to each other.

– Delegating Abstraction (DLA): An abstraction that exists only for passing messages from one abstraction to another.

– Fat Interface (FTI): Interface that exposes many funcionalities and many of those functionalities are not related to each other. They usually are generic interfaces that could be split on specific interfaces

– Incomplete Abstraction (ICA): When an element does not support a responsibility completely in their enclosing component.

– Misplaced Concern (MPC): Element that implements a concern (*e.g.*, functionality), which is not the predominant one of their enclosing component



Sumário HibernateMatriculaInstitucionalDao

Aglomerações

Anomalias de Código (Code Smells)

Atributos Não Funcionais Violados

Atributo	Descrição
Reusabilidade	Reusabilidade é o grau de facilidade ou de potencialidade que um determinado componente possui para ser reutilizado. Em geral, os elementos com baixa reusabilidade possuem baixa coesão e auto acoplamento com outros módulos. O que torna a tarefa de manutenção penosa. **Razão: Essa classe pode não ser facilmente reusável em outras aplicações.**

Qual a relevância desta seção?

○ Irrelevante ○ Pouco Relevante ○ Bastante Relevante ○ Muito Relevante

Padrões de Projeto ou Arquiteturais Implementados

Atributos de Qualidade e Princípios Violados

Sumários com Características Similares

Figure C.4: Non-functional Requirement Information

- Unwanted Dependency (UWD): Dependency that violates a rule defined on the system design. It usually occurs when elements should not communicate with each other, but do.
- Scattered Concern (STC): Components responsible for the same functionality, but some of which are responsible for functionalities that cross-cut the system.

C.6 Symptoms Combination

Someone can argue that analyzing a single element is not enough to identify a design problem. Nevertheless, we noticed that developers often combine symptoms in a single element in order to confirm the presence of a design problem. Table C.2 shows how often developers either used only one symptom or combined multiple symptoms. Its first column indicates the symptoms that developers combined to identify design problems. Its second column indicates how many times the symptom or combination of symptoms happened. Still on second column, the number in parenthesis indicates the number of design problems found when the subject used a symptom or a combination of them. Its third column shows the design problems found. It last column indicates the teams who used or combined the symptoms.

Table C.2: Symptoms Combination

Symptoms	Instances	Design Problems	Subjects
Design Pattern Violation	4 (2)	UWD, DLA	T7, T9
Quality Requirements	3 (3)	FTI, CCO	T8, T9
Violation of Non-functional Requirements	8 (5)	STC, UWD, CPO, CCO, ICA, FTI	T7, T9
Code Smells	1 (1)	FTI	T7
Design Pattern Violation, Violation of Non-functional Requirements	8 (6)	UWD, CPO, CCO	T1, T7, T9
Quality Requirements, Violation of Non-functional Requirements	1 (1)	ICA	T7
Violation of Non-functional Requirements, Code Smells	2 (2)	FTI, CCO	T3, T7
Violation of Non-functional Requirements, Violation of Object-oriented Principles	3 (1)	UWD, AMI, CPO	T8, T9
Design Pattern Violation, Quality Requirements, Violation of Non-functional Requirements	3 (2)	UWD, CCO	T7, 8
Design Pattern Violation, Quality Requirements, Violation of Object-oriented Principles	2 (1)	CCO	T6
Design Pattern Violation, Violation of Non-functional Requirements, Code Smells	2 (2)	UWD	T9
Quality Requirements, Violation of Non-functional Requirements, Code Smells	1 (1)	CCO	T8
Quality Requirements, Violation of Non-functional Requirements, Violation of Object-oriented Principles	1 (1)	CCO, STC	T3
Quality Requirements, Code Smells, Violation of Object-oriented Principles	3 (2)	CCO	T4, T5
Violation of Non-functional Requirements, Code Smells, Violation of Object-oriented Principles	2 (2)	UWD, MPC	T7, T9
Design Pattern Violation, Quality Requirements, Violation of Non-functional Requirements, Violation of Object-oriented Principles	5 (3)	CCO, STC, UWD	T2, T9
Design Pattern Violation, Quality Requirements, Code Smells, Violation of Object-oriented Principles	2 (1)	UWD, CCO	T8
Quality Requirements, Violation of Non-functional Requirements, Code Smells, Violation of Object-oriented Principles	3 (2)	CCO	T3, T5
Design Pattern Violation, Quality Requirements, Violation of Non-functional Requirements, Code Smells, Violation of Object-oriented Principles	3 (1)	CCO	T4

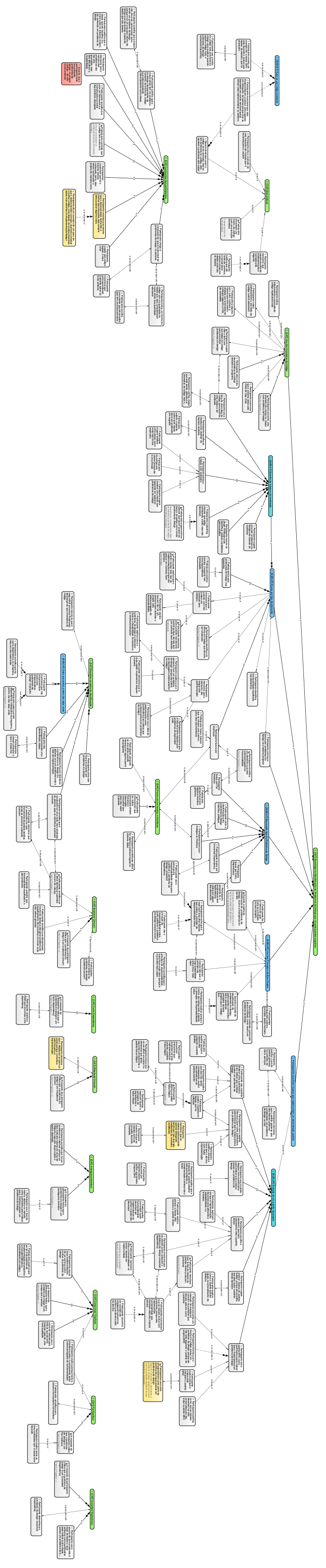


Figure C.5: Codes for the Companies 1 and 2

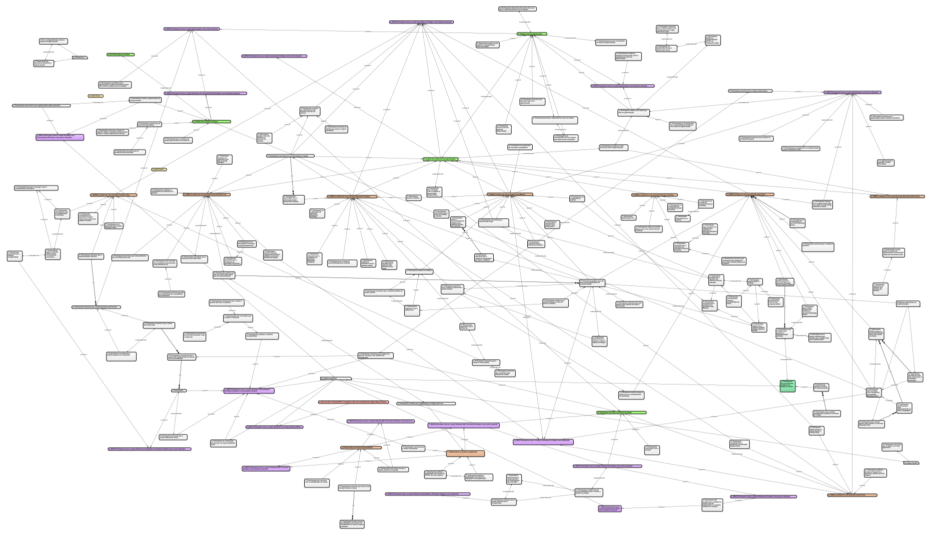


Figure C.6: Codes for the Companies 3 and 4

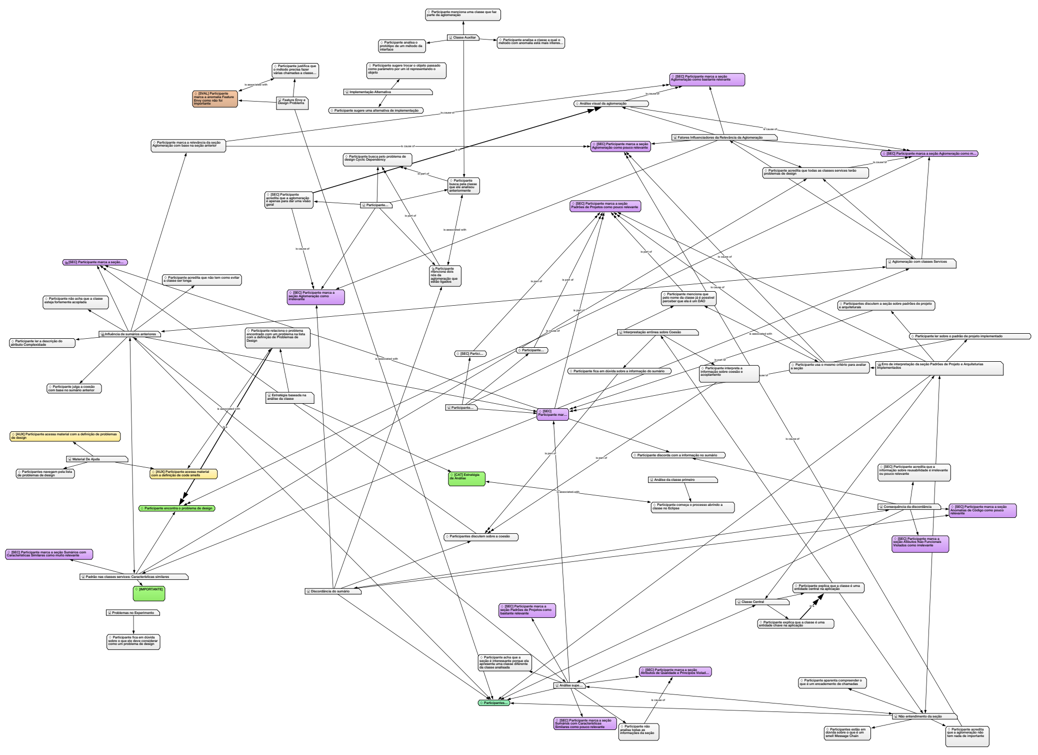


Figure C.7: Codes for the Company 5