ANÁLISE DE COOCORRÊNCIAS DE ANOMALIAS E PADRÕES DE PROJETO EM SISTEMAS DE SOFTWARE

BRUNO CARDOSO

ANÁLISE DE COOCORRÊNCIAS DE ANOMALIAS E PADRÕES DE PROJETO EM SISTEMAS DE SOFTWARE

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

ORIENTADOR: EDUARDO FIGUEIREDO

Belo Horizonte

Novembro de 2015

BRUNO CARDOSO

CO-OCCURRENCE ANALYSIS OF DESIGN PATTERNS AND BAD SMELLS IN SOFTWARE SYSTEMS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: EDUARDO FIGUEIREDO

Belo Horizonte

November 2015

Cardoso, Bruno

D1234p Co-occurrence Analysis of Design Patterns and Bad Smells in Software Systems / Bruno Cardoso. — Belo Horizonte, 2015
xxii, 67 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Eduardo Figueiredo

1. design patterns. 2. bad smells. 3. co-occurrences. I. Título.

CDU 519.6*82.10

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

FOLHA DE APROVAÇÃO

Co-occurrence analysis of design patterns and bad smells in software systems

BRUNO DOS SANTOS AZEVEDO CARDOSO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Eduardo M Figueiredo

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

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

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

Belo Horizonte, 13 de novembro de 2015.

Agradecimentos

A trajetória de um curso de mestrado é uma tarefa árdua, mas certamente é uma experiência desafiadora e enriquecedora. A conclusão desta dissertação marca o fim desta importante etapa. Gostaria, portanto, de agradecer àqueles que compartilharam comigo deste momento.

Primeiramente gostaria de agradecer à toda minha família. Em especial a meus pais, Orlando e Jucélia, que são para mim exemplos de dedicação e perseverança, que sempre me apoiaram e me deram o alicerce necessário para minha educação. Agradeço também ao meu irmão, Guilherme, pelo carinho, amizade e incentivo.

Agradeço também de forma especial à minha noiva, Raquel, que me deu grande apoio e incentivo ao longo deste período, me dando forças, principalmente nos momentos de desânimo, para que eu pudesse finalizar esta etapa.

Agradeço a todos os meus amigos. Obrigado pela paciência, incentivo e compreensão durante este período.

Ao professor Eduardo Figueiredo, que além de orientador foi um grande incentivador deste trabalho. Obrigado por acompanhar de perto minha evolução ao longo do mestrado e sempre se mostrar disponível para me ajudar a superar as dificuldades do curso. Agradeço também aos demais professores da UFMG e aos colegas do labo ratório de Engenharia de Software (LabSoft) pelas opiniões e contribuições valiosas que ajudaram a enriquecer este trabalho.

Agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) pela oportunidade de realizar um curso de alto nível de excelência e pelo suporte acadêmico e financeiro, que possibilitou a publicação de alguns dos trabalhos desenvolvidos ao longo do curso.

Por fim, agradeço a todas as pessoas que contribuiram, de forma direta ou indireta, para que eu concretizasse esta importante etapa.

Resumo

Sistemas de software são muitas vezes desenvolvidos de forma que as boas práticas do paradigma orientado a objetos não são satisfeitas, causando a ocorrência de anomalias. Anomalias são sintomas ou características estruturais de uma parte do código que podem sugerir a presença de um problema mais profundo no projeto do sistema ou no código. Por outro lado, padrões de projeto têm a intenção de catalogar as me lhores práticas para o desenvolvimento de sistemas de software orientados a objetos. Embora aparentemente muito divergentes, pode haver uma coocorrência de padrões de projeto e anomalias, visto que este fenômeno algumas vezes é citado em estudos na área de engenharia de software, embora discretamente. Portanto, este trabalho realiza uma análise exploratória, a fim de identificar a coocorrência de padrões de projeto e anomalias em sistemas de software, que pode acontecer devido ao uso inadequado de padrões de projeto. Para atingir este objetivo, primeiro realizamos uma revisão da literatura a fim de compreender o estado atual da arte relativo a padrões de projeto e anomalias e, em seguida, foi realizado um estudo exploratório para identificar instâncias concretas de usos indequados de padrões de projeto que levam a ocorrência de anomalias. Neste estudo, ferramentas foram usadas para a detecção de padrões de projeto e anomalias em cinco sistemas de médio a grande porte. Os resultados do estudo indicam coocorrências interessantes, como Command com God Class e Template Method com Duplicated Code. Nós analisamos porque esses padrões provocaram o surgimento das anomalias. Discutimos também a gravidade das anomalias e, quando apropriado, propomos alterações ao código do sistema, visando solucionar as anomalias detectadas. Esses achados indicam que, apesar de padrões de projeto catalogarem as me lhores práticas para o desenvolvimento de sistemas de software, seu uso inadequado pode ter feitos indesejados.

Palavras-chave: Padrões de Projeto, Anomalias, Coocorrências.

Abstract

Software systems are often developed in a way that good practices of the object-oriented paradigm are not fulfilled, causing the occurrence of bad smells. Bad smells are symptoms or structural characteristics in a region of code that may suggest the presence of a deeper problem in the system design or code. On the other hand, design patterns are intended to catalog the best practices for developing object-oriented software systems. Although apparently widely divergent, there may be a co-occurrence of design patterns and bad smells, since this phenomenon is sometimes cited in studies in the software engineering field, albeit discreetly. Therefore, this dissertation performs an exploratory analysis in order to identify the co-occurrence of design patterns and bad smells in software systems, that may happen due to the inadequate use of design patterns. To achieve this goal, we first perform a literature review in order to understand the current state of art concerning design patterns and bad smells and, then, we accomplish an exploratory study to identify concrete instances of design patterns inadequate usage that lead to bad smells occurrences. In this study, we use tools for detecting design patterns and bad smells in five medium to large size systems. The study results indicate interesting co-occurrences, as Command with God Class and Template Method with Duplicated Code. We analyse why these patterns provoked the bad smells arising. We also discuss the severity of bad smells and, when appropriate, propose changes to the system code in order to address the detected bad smells. These findings indicate that, although design patterns catalog the best practices for developing software systems, their inadequate use may have undesired effects.

Palavras-chave: Design Patterns, Bad Smells, Co-occurrence.

List of Figures

List of Tables

List of Acronyms

DPDSS: Design Pattern Detection using Similarity Scoring.
MARPLE: Metrics and Architecture Reconstruction Plug-in for Eclipse.
SLR: Systematic Literature Review

Contents

Chapter 1

Introduction

Design patterns are general reusable solutions to recurring problems in software design [Gamma et al., 1994]. They are descriptions of communicating objects and classes that need to be customized to solve a general design problem in a particular context. Therefore, a design pattern is a description or template of how to solve a problem that often appears in different systems. The Gang-of-Four (GoF) book [Gamma et al., 1994] of design patterns has highly influenced the field of software engineering and it is regarded as an important source for object-oriented design theory and practice. The GoF book is organized as a catalogue of 23 design patterns.

Since their inception, design patterns have been employed by designers and developers who realized how patterns could improve development. There is ample evidence that patterns can have a beneficial impact on software quality and maintainability [Cline, 1996] [Prechelt et al., 2001] [Beck et al., 1996] [Buschmann et al., 2007]. Therefore, design patterns are typically beneficial, provided that they are properly implemented. On the other hand, as design patterns became popular many software developers applied them in an exploratory way. In particular, beginners, who are delighted by design patterns elegance, and tend to employ them in a careless way, ignoring their original proposal and disregarding some basic principles like code simplicity.

This careless employment of design patterns can lead software system code to a set of flaws, like bad smells. Bad smells are symptoms or structural characteristics in a region of code that may suggest the presence of a deeper problem in the system design or code [Fowler et al., 1999]. One of the main references on this topic is the Fowler's book which has catalogued 22 bad smells. In this book [Fowler et al., 1999], bad smells are defined as code fragments that need refactoring. Other authors have also contributed to expand the set of bad smells [Kerievsky, 2005] [Marinescu and Lanza, 2006]. Kerievsky [2005] emphasizes the use of design patterns as a refactoring

technique to remove bad smells. Marinescu and Lanza [2006] presented a catalog of bad smells called "disharmonies". In this context, the main goal of this work is to identify inappropriate employment of design patterns. And, then, we aim to analyse these cases in order to aid designers and developers to better use design patterns. Our ultimate goal is to warn professionals about possible patterns misuse.

1.1 Motivation and Related Work

At first glance, while design patterns are associated with good software design and code, bad smells define lack of design or any code flaw. Design patterns may degrade the system performance and also turn code more complex unnecessarily when they are not used in a proper manner [Wendorff, 2001] [Bieman et al., 2003] [Prechelt et al., 2001] [Vokáč et al., 2004]. This work aims at identifying design patterns inappropriate employments and their consequences. In order to focus on an specific scope, this work exploits the occurrence of bad smells in code that is part of a design pattern.

Design patterns and bad smells are subject of recurring research and typically appear in software systems. Despite this frequency, they rarely are investigated in the same research context since they represent antagonistic structures. Therefore, the first step was understand how studies relate these topics. Among these studies, some tools [Jebelean et al., 2010] [Carneiro et al., 2010] [Marinescu and Lanza, 2006] [Tsantalis and Chatzigeorgiou, 2009] have been proposed to identify code fragments that can be refactored to patterns. Other studies [Bouhours et al., 2010] [Khomh, 2009] establish a structural relationship between design patterns and bad smells.

In addition, there are reported cases in the literature where the use of design patterns may not always be the best option and the wrong use of a design pattern can even introduce bad smells in code [Wendorff, 2001] [McNatt and Bieman, 2001] [Cline, 1996]. For instance, McNatt and Bieman [2001] assessed the positive and negative effects of design patterns on maintainability, factorability, and reusability. Wendorff [2001] performed a study and identified some questionable use of design patterns, like Proxy, Observer, Bridge, and Command. Cline [1996] states that design patterns might be overly hyped in some cases.

Design patterns have their place, but their inadequate use may increase complexity and decrease some quality attributes. The main goal of this dissertation is to aid software engineering professionals, such as designers and developers, by showing some concrete cases in which design patterns are careless employed and lead to bad

smells arising. We also present some lessons learned within this work that may serve as examples for software engineering professionals.

1.2 The Proposed Study

Despite the definition of design patterns and bad smells are antagonistic in Software Engineering filed, the inappropriate use of design patterns can cause bad smells in some classes or methods of the system. Therefore, the first step was performing a systematic literature review in order to understand how studies investigate these two topics, design patterns and bad smells, together [Cardoso and Figueiredo, 2014]. The results showed that, in general, studies have a narrow view concerning the relation between these concepts. Most of them focus on refactoring opportunities whereas some others state an structural comparison between design patterns and bad smells. A few studies mention that design patterns and bad smells may co-occur.

To further investigate the topic, we performed an exploratory study in order to identify instances of co-occurrences between design patterns and bad smells [Cardoso and Figueiredo, 2015]. In this study, we applied a design pattern detection tool and two bad smell detection tools over five systems. This way, it was possible to analyze in which situations design patterns and bad smells co-occur. As far as we are concerned, co-occurrences of bad smells and design patterns have not been deeply investigated in the software engineering literature and there are only a few references concerning this topic [Fontana and Spinelli, 2011] [Seng et al., 2006].

Results of this exploratory study indicate some correlations between design patterns and bad smells. We applied association rules [Agrawal et al., 1993] [Brin et al., 1997] to indicate how strong a correlation between a design pattern and a bad smell is. For instance, association rules indicate how often Factory Method and Feature Envy are present in a class at same time. After applying the association rules, we manually analyzed in which situations these associations are due to inadequate use of Factory Method or merely coincidence.

Based on detaching values of association rules and manual inspection, we analyzed two situations that called the attention: co-occurrences of Command and God Class and co-occurrence of Template Method and Duplicated Code. The results provided by this exploratory study aim to extent knowledge on inadequate use of design patterns and to understand why specific design patterns and bad smells may co-occur.

At last, using the systematic literature review and the exploratory study results as a baseline, we present some lessons learned. We analyze some possible co-occurrence

instances by exploring definitions of bad smells and design patterns. Although the examples are not real co-occurrence instances, but situations that may happen due to design patterns inadequate employment, these co-occurrence examples may aid designers and developers to better use design patterns and avoid bad smells.

1.3 Contributions

As a result of the work presented in this dissertation, the following contributions can be highlighted:

- A literature review detaching how studies relate design patterns and bad smells, revealing the lack of studies focusing on the co-occurrence between design patterns and bad smells. This provides the academic community with the state of the art in this field.
- The definition, planning, quantitative and qualitative analysis of an exploratory study that showed real instances of design pattern and bad smell co-occurrences.
- The presentation of lessons learned, in which we explore design patterns and bad smells definitions to analyze some possible co-occurrence instances.

Two papers presenting the systematic literature review and the exploratory study were published, and include part of our results

- Cardoso, B. and Figueiredo, E. Co-Occurrence of Design Patterns and Bad Smells in Software Systems: A Systematic Literature Review. In proceedings of the Workshop on Software Modularity (WMod), co-allocated with CBSoft. Maceio, Brazil, 2014, pages 82–93.

- Cardoso, B. and Figueiredo, E. Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study. In proceedings of the Brazilian Symposium on Information Systems (SBSI). Goiania, GO, 26-29 May 2015, pages 347–354.

1.4 Dissertation Outline

While this chapter introduced this dissertation, the rest of this document is organized as follow:

Chapter 2 details the theoretical foundation necessary for the understanding of the main concepts of this work: design patterns and bad smells. Besides, it presents some design pattern and bad smell detection tools. Some of these tools are used within this work.

Chapter 3 presents the related work, including the description of a literature review of the primary studies associated to the theme of this dissertation, focusing on how studies relate design patterns and bad smells.

Chapter 4 details the exploratory study that was performed during this master work. It details the settings of this exploratory study and the decisions made during its execution.

Chapter 5 presents some lessons learned during this master work. We analyze some challenges in design patterns usage and discuss how the inadequate use design patterns may lead to bad smells appearance.

Chapter 6 concludes this dissertation by presenting the final remarks, limitations and suggesting directions for future work.

Chapter 2

Background

The correct understanding of a research work begins with the understanding of the concepts related to it. This chapter presents relevant concepts for this research work. Section 2.1 introduces the concepts and also presents some examples of design patterns whereas Section 2.2 presents two design pattern detection tools. Section 2.3 introduces the concepts and also presents some examples of bad smells, whereas Section 2.4 presents three bad smell detection tools.

2.1 Design Patterns

Design patterns were presented by Gamma et al. [1994] as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". Since then, their use has become increasingly popular. The purpose of design patterns is to capture design knowledge in a way that it can be easily reused by providing tested and suitable solutions.

Design patterns aim at reducing coupling and improving flexibility within software systems. As an example , many design patterns postpone decisions until runtime [Prechelt et al., 2001], which makes code more flexible and, this way, it is easier to add new functionality without deep changing existing code. They also make it easier to think clearly about a design and encourage the use of best practices [Prechelt et al., 2001]. Design patterns aim to improve communication among designers and developers, by providing a common terminology [Beck et al., 1996]. By using well-known and well-understood concepts, it eases code readability and the system design in general becomes better understood and easier to maintain. For instance, a solution is better understood by saying that the Decorator pattern was employed in the system [Gamma et al., 1994] than saying that some extra responsibilities need to be dynamically added to an object.

We briefly explain below two common design patterns that are explored in this work: Command and Template Method.

The Command pattern has the intent of encapsulating a request as an object; thereby letting the designer parameterizes clients with different requests, queue or log requests, and supports undo operations [Gamma et al., 1994]. To implement this pattern, it is necessary to create an interface that is the abstract command. Concrete commands implement this interface. The client instantiates a concrete command and the invoker queues the requests for this command. By the time this command is executed, it is done by an object of the receiver class. Command is used when it is necessary to issue requests to objects without knowing anything about the operation that is requested or the receiver of the request [Gamma et al., 1994]. It is useful when supporting activities that require the execution of a series of commands, as the orders of customers in a restaurant. This solution allows that the requisitions are executed in different moments in time, according to availability. The command objects can be held in a queue and processed sequentially, as in a transaction.

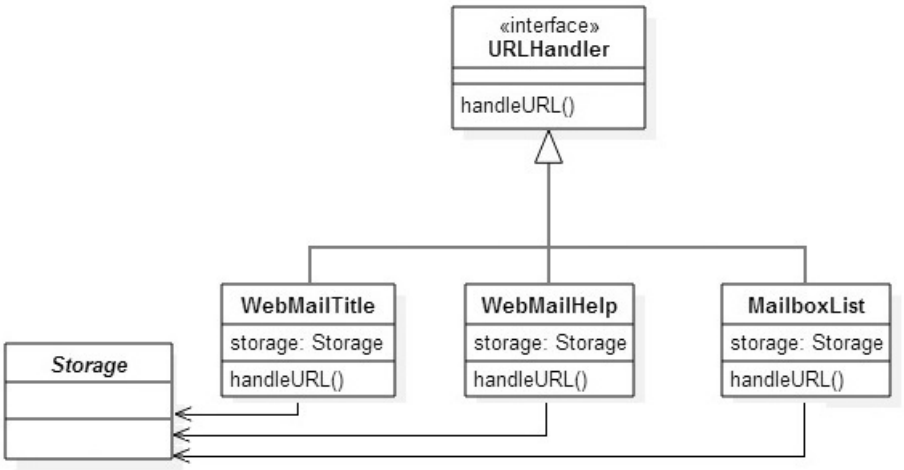

Figure 2.1. Partial Class Diagram of the Command Pattern in WebMail System

Figure 2.1 shows an example of a real instance of this design pattern extrated from the WebMail system [Terra et al., 2013]. In this example, the *URLHandler* interface is the abstract command and the three classes that inherit from it are the concrete commands. The concrete commands are associated to the *Storage* class. As shown in Figure 2.1, the three concrete commands instantiate an object of the *Storage* class and implement the interface *URLHandler*. The *Storage* class plays the role of the receiver in the Command pattern. Its attributes and methods are hidden because they are not

necessary for the understanding of this pattern. In this example, the *handleURL()* method is the classic execute method proposed in the Command pattern definition.

The Template Method pattern defines the skeleton of an algorithm in an operation, deferring some steps to client subclasses. This design pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm structure [Gamma et al., 1994]. It aims at solving the problem that occurs when two or more different components have significant similarities, but snippets of the same method may differ. In this case, merely extending a base class or implementing an interface is not enough to solve the problem. Another alternative is duplicating this method in both classes even though they have high similarity. Considering this alternative, if there is a change that targets the algorithm, then duplicated effort is necessary.

Figure 2.2 exemplifies the use of the Template Method design pattern. This figure shows a partial Class Diagram of a real instance of this pattern extracted from the JHotDraw system. The abstract class *AbstractAttributeCompositeFigure* defines a drawing template method, called *drawFigure()*, that can be modified by the specialized classes. As shown in Figure 2.2, the template method is responsible for calling other methods. The concrete classes of this example have very similar methods and perform similar tasks, as expected in the Template Method use. However, considering that they perform some tasks differently, the inherited methods that they implement – *drawFill()* and *drawStroke()* - should be different in order to perform these different tasks. Therefore, if a designer does not use a template method in this situation, it would be necessary to either duplicate all the other methods besides *drawFill()* and *drawStroke()* or it would not be possible to implement different tasks in the concrete classes.

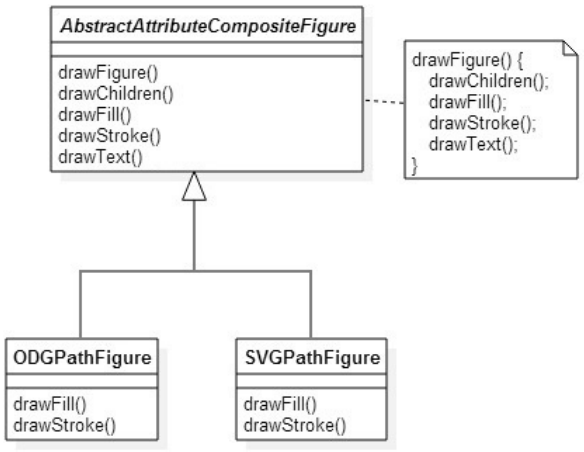

Figure 2.2. Partial Class Diagram of Template Method in JHotDraw

2.2 Design Pattern Detection Tools

By finding design patterns from source it is possible to bring the code understanding to a higher level by revealing the architectural design of a system [Shi and Olsson, 2005]. This practice is important and widely employed when it is necessary to perform a reverse engineering [Raibulet and Arcelli, 2005] [Shi and Olsson, 2005] [Lee et al., 2008]. However, this is an expensive work and detected tools are frequently used [Guéhéneuc, 2005] [Niere et al., 2002] [Shi and Olsson, 2005] [Mc Smith and Stotts, 2003] [Tsantalis et al., 2006] [Arcelli et al., 2008].

In this section, two design pattern detection tools are described: Design Pattern Detection using Similarity Scoring (DPDSS) [1] and Metrics and Architecture Reconstruction Plug-in for Eclipse (MARPLE) [2]. These tools were chosen because they are both user friendly and well known in academia [Dong et al., 2009] [Fontana and Zanoni, 2011].

DPDSS tool [Tsantalis et al., 2006] applies the algorithm that names it: Similarity Scoring, in which the modeling is performed by using directed graphs that are mapped into square matrices [Tsantalis et al., 2006]. According to the authors the two main advantages of this approach are that the matrices can be handled easily and that this type of representation is intuitive to computer scientists. Before the pattern detection process itself, the DPDSS tool defines the structure of the target system and also the structure of design patterns to be identified. This structure should include all necessary information for the identification of patterns. Thus, the relationships between the classes as well as other static information are mapped. The collected information depends on the design patterns characteristics that the user wants to detect. These information includes associations, generalizations, abstract classes, creating objects and invoking abstract methods, to name a few.

Figure 2.3 shows the default class diagram of Decorator as presented by the original authors [Tsantalis et al., 2006]. Figure 2.4 illustrates how the graph is based on the Decorator pattern class diagram for three relevant pieces of information taken from this diagram: associations, generalizations, and abstract classes as explained by Tsantalis et al. [2006]. Besides each graph is the square matrix structure that belongs to each of this information. One may have the impression that it would be sufficient to apply an exact graph matching algorithm. However, the comparison between subgraphs for each relevant available information in a design pattern structure led some authors to consider this problem NP-complete [Welling, 2011].

[1] http://java.uom.gr/∼nikos/pattern-detection.html
[2] http://essere.disco.unimib.it/reverse/Marple.html

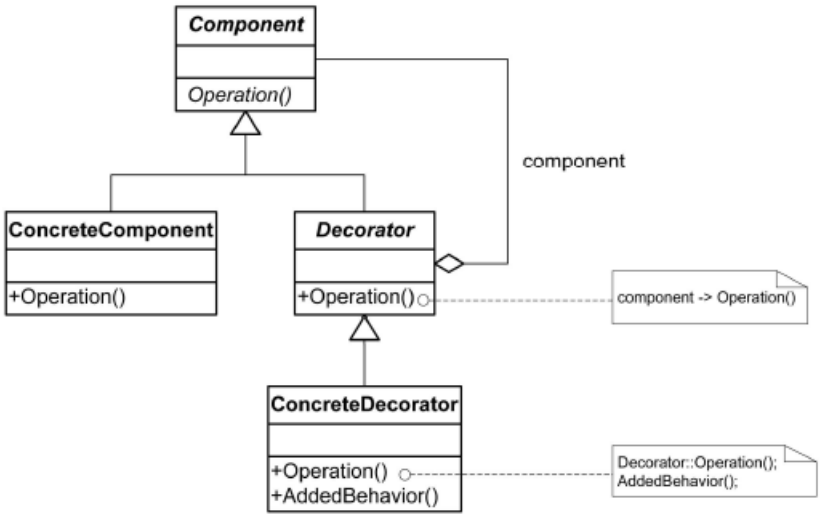

Figure 2.3. Sample Class Diagram of Decorator

Table 2.1 lists the design patterns that can be detected by the used tool, following the same classification used by Gamma et al. [1994]: by purpose and scope. In Table 2.1, the Command design pattern is shown in the second column, which lists the structural patterns, despite it is a behavioral one. This misclassification happens because the DPDSS tool cannot differentiate Object Adapter and Command patterns, putting the latter in the group of structural patterns. The same happens with Strategy and State patterns, whose instances cannot be differentiated by the tool.

Table 2.1. Design Patterns Detected by the Tool DPDSS

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method		Template Method
	Object	Prototype Singleton	Object Adapter/Command Composite Decorator Proxy Proxy 2	Observer Strategy/State Visitor

Another relevant issue is the fact that Table 2.1 displays not only the default structural Proxy design pattern, but also a variant pattern called Proxy 2 [Tsantalis et al., 2006]. Proxy 2 combines both Proxy and Decorator. In order to simplify the analysis, when this tool finds an instance of the Proxy 2 pattern, we considered that

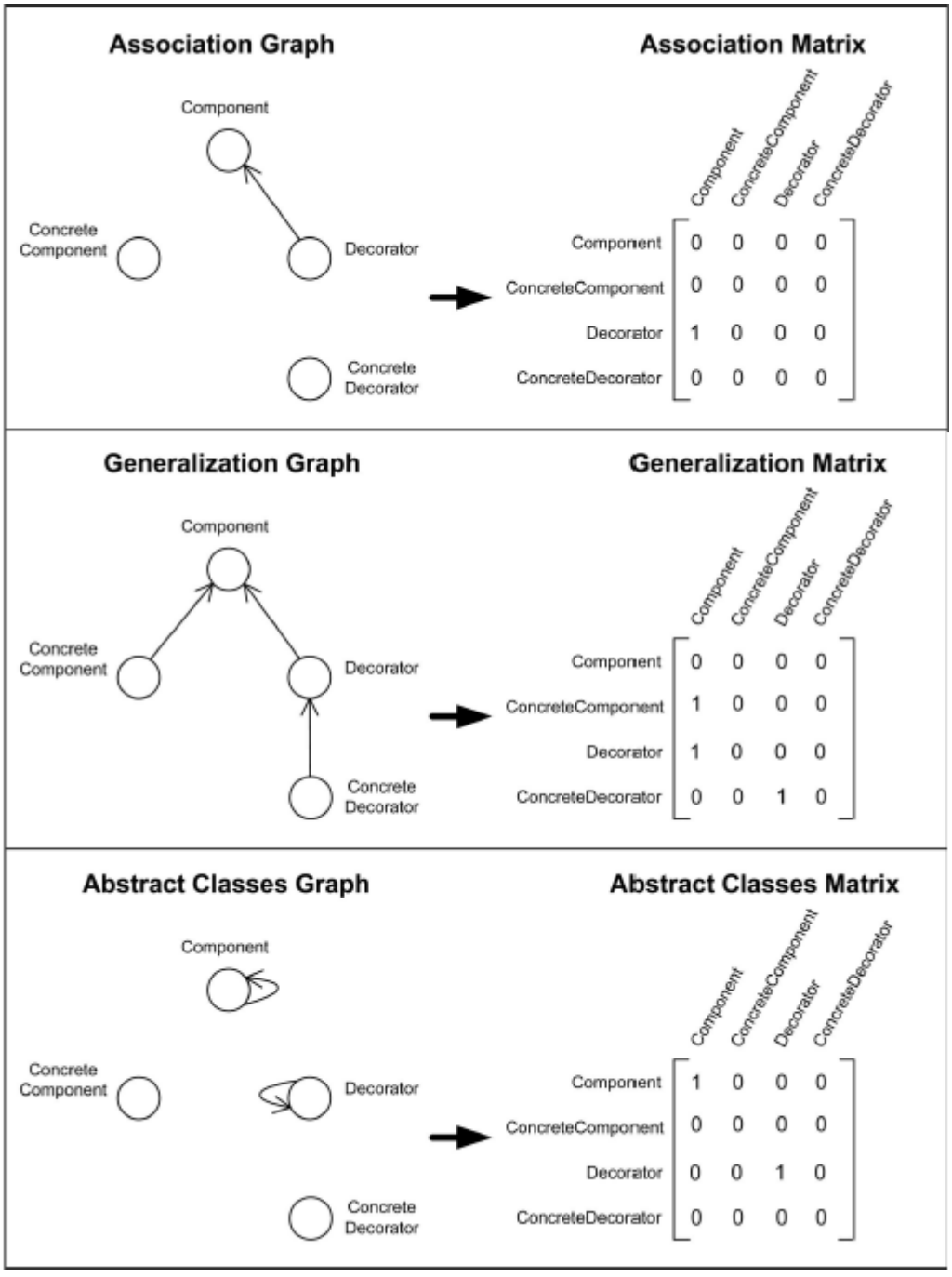

Figure 2.4. Information Mapped by the Algorithm of DPDSS Tool [Tsantalis et al., 2006]

it is an instance of the default Proxy pattern in the study we performed (Chapter 4). Considering the DPDSS tool limitations and also this simplification about the Proxy pattern variation, a total of eleven design patterns can be detected by this tool. The exploratory study presented in Chapter 4 relies on DPDSS because, according to the

authors, the tool algorithm seeks not only the basic structure of the design patterns, but it also seeks for modified pattern instances. The DPDSS tool also has a friendly user interface and it registers all classes that participate in an instance of a detected design pattern in a log file, easing the tracking and analysis of the code.

Another design pattern detection tool is MARPLE, which is an Eclipse plug-in. It supports both the detection of design patterns (DPD) and software architecture reconstruction (SAR) activities through the use of basic elements and metrics that are mechanically extracted from source code [Arcelli et al., 2008]. When it comes to design pattern detection, the MARPLE tool approach is based on the detection of design pattern subcomponents, which can be considered indicators of the presence of patterns. It uses static source code analysis. That is, the ASTs (Abstract Syntax Trees) of the target systems are parsed in order to obtain the structures necessary for the elaboration, which is called basic elements [Arcelli et al., 2008]

Figure 2.5 depicts an overview of the principal activities performed through MARPLE as presented by the original authors [Arcelli et al., 2008]. The main activities are the general process of data extraction, DPD, SAR and consequent results visualization. The information that is used by MARPLE is obtained by an AST representation of the analyzed system.

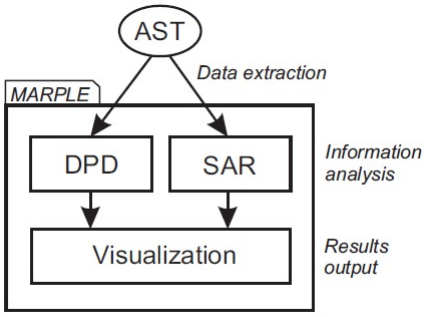

Figure 2.5. An Overview of MARPLE Process [Arcelli et al., 2008]

DPD and SAR receive both the same set of basic elements and metrics that have been found inside the system, collected in an XML file. Basic elements are formed by elemental design patterns [Smith and Stotts, 2002], design pattern clues [Maggioni, 2006] and micro patterns [Gil and Maman, 2005]. Basic elements are intended as the basic information Arcelli et al. [2008] exploit as hints for the presence of design patterns inside the code, and as basic relationships that may connect two or more classes in terms of object creation, method invocation or inheritance.

2.3 Bad Smells

Bad smells are symptoms or structural characteristics in a region of code that may suggest the presence of a deeper problem in the system design or code. They are defined as code fragments that need refactoring [Fowler et al., 1999]. Although bad smells can harm the development, maintenance, and evolution of the software system, they are not necessarily bugs since they are not incorrect implementations. They do not even necessarily harm the proper functioning of the system. Some long methods, for instance, are just fine [Fowler et al., 1999]. It is necessary to look deeper to see if there is an underlying problem there since bad smells are not inherently bad on their own. They are often an indicator of a problem rather than the problem itself.

Bad smells receive different classifications and naming schema in papers, such as bad smells, code smells, and design smells [Vale et al., 2014]. Some authors treat these classifications interchangeably, while others establish minor differences among them. Design smells are associated with smell caused during the system design while code smells are linked to the source code, as suggested the nomenclature of both. Bad smell is the general term used for both code smell and design smell.

Another related concept is anti pattern which is understood and defined in different ways in the literature [Luo et al., 2010] [Brown et al., 1998]. For example, Dodani [2006] considers that if a design pattern has the best solution to solve a problem, then the anti pattern presents the lesson learned. The author describes the effects of bad design practices and gives the procedure to be followed in order to achieve better quality software. Therefore, an anti pattern makes it possible to check or supervise bad practices.

Luo et al. [2010] make an analysis of the relationship of bad smells and anti patterns, stating that anti patterns are solutions that may cause problems and that bad smells can be related to these problems. In fact, bad smells are generally considered more low-level and symptomatic anti patterns. In fact, the presence of some specific bad smells can, in turn, manifest in anti patterns, of which code smells are parts of [Khomh, 2010].

An anti pattern can also represent good design practices, but when used in an excessive manner, produces more damaging consequences instead of the good results expected by the employ of these patterns [Brown et al., 1998]. Since anti patterns are also bad practices employed in a software system, within this dissertation anti pattern is considered a synonym of bad smell. For instance, the Blob anti pattern [Brown et al., 1998] is very similar to the Large Class smell [Fowler et al., 1999].

Fowler et al. [1999] catalogued 22 bad smells and indicated appropriate refactoring techniques for each of them. Though this book is the main reference of bad smells, there are other catalogues of bad smells in the literature [Kerievsky, 2005] [Marinescu and Lanza, 2006]. For instance, Kerievsky [2005] proposes a new set of bad smells, giving emphasis to the use of design patterns as a refactoring technique to remove them. In some sense, Kerievsky extends the set of bad smells cataloged by Fowler et al. [1999]. Other authors have proposed a new set of bad smells, such as Marinescu and Lanza [2006], who presented a catalog of bad smells called "disharmonies". In addition to the definitions of the bad smells, the authors proposed detection strategies and recommendations for their identification and correction. However, 6 out of 11 disharmonies are similar to those bad smells listed by Fowler et al. [1999].

Two examples of common bad smells investigated in this dissertation are God Class [Riel, 1996] [Marinescu and Lanza, 2006] and Duplicated Code [Fowler et al., 1999]. God Class performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes. This smell has a negative impact on reusability and understandability of this part of the system. God Class is similar to the Fowler's Large Class bad smell [Fowler et al., 1999]. Within this dissertation these terms are used interchangeably. Therefore, God Class is the term used to describe a certain type of classes which "know too much or do too much". It is also common to refer as God Class the ones that control too much. Often God Class arises by accident as functionalities are incrementally added to a central class over the course of the system evolution.

Duplicated code [Fowler et al., 1999] is the most pervasive and pungent bad smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code and it may happen in large variety of ways. Duplicated code can be in different methods of the same class, which is the simpler situation both to detect and eliminate duplication. Another common duplication problem is when the same code appears in two sibling subclasses. This problem is more difficult to find, but not so complicated to eliminate. Code duplication can also be in two or more totally unrelated classes or even in unrelated software systems [Oliveira et al., 2015]. This situation is the hardest to be detected and definitely the most difficult to be refactored. All these scenarios may also happen with subtle duplication, which exists in structures or processing steps that are outwardly different, yet essentially the same.

2.4 Bad Smell Detection Tools

We analyzed three tools for detecting bad smells: JDeodorant, PMD and inFusion. JDeodorant[3] [Tsantalis et al., 2008] was developed as a plug-in for the Eclipse IDE and it is able to automatically detect four different kinds of bad smells by using software metrics. JDeodorant detects four bad smells. This tool applies automatic refactoring, since it provides suggestions to solve the bad smells that it detects. PMD[4] [Copeland, 2005] is also a plug-in installed on the Eclipse IDE. PMD also detects four bad smells. It performs detection through the use of simple metrics such as number of lines of code. This tool offers the possibility of manual configuration of the parameters for the metrics used in automatic detection. Besides the bad smells detected by this tool, it is also able to identify a large variety of other programming flaws. inFusion[5] is the one that can detect the largest number of bad smells, eight among those described by Fowler et al. [1999]. It uses refined detection techniques, combining different software metrics, visualization and other statistical analysis techniques, allowing to present a range of relevant data for system analysis.

Table 2.2 presents some information about these three bad smell detection tools. Whereas the first column shows the name of the tool, the second column present the programming language that the tool is able to recognize. The third column shows the techniques the tools employ in order to detect bad smells. The fourth column indicates whether the tool is able or not to present and apply suggested refactorings for the detected bad smells. At last, the fifth column indicates if the tools show or detach the code snippet where the bad smell was detected. From the information presented in Table 2.2, we may notice that the three tools detect bad smells in systems written in Java language, but inFusion also supports bad smells detection in systems written in C and C++ languages. JDeodorant is the only tool which applies automated refactoring and inFusion is the only tool in this set that does not show or detach the detected code.

Table 2.3 lists the bad smells detected by JDeodorant, PMD and inFusion. In this table, only the bad smells that are detected by at least one of the three tools are shown. This way, as it can be observed in Table 2.3, a total of ten bad smells, out of 22 listed by Fowler, are detected combining the three tools. Long Method and God Class are the only bad smells that are detected by the three tools, which is expected given that these two bad smells are evidenced only by checking the method or class size. Duplicated

[3]http://www.jdeodorant.com/
[4]http://pmd.sourceforge.net/
[5]http://www.intooitus.com/products/infusion

Table 2.2. Bad Smells Detection Tools

Tools	Supported Programming Languages	Detection Strategy	Apply Refactoring	Show Detected Code
JDeodorant	Java	Metrics	Yes	Yes
PMD	Java	Metrics	No	Yes
inFusion	C, C++, Java	Metrics	No	No

Code and Feature Envy are detected by two tools, whereas Shotgun Surgery, Long Parameter List, Data Clumps, Switch Statements, Data Class, and Refused Bequest are detected by only one tool. The exploratory study presented in Chapter 4 relies on JDeodorant and PMD mainly because they are easy to install and configure and because they detach the detected code, different from inFusion.

Table 2.3. List of Bad Smells Detected per Tool

Bad Smell	JDeodorant	PMD	inFusion
Duplicated Code	-	X	X
Long Method	X	X	X
Large/God Class	X	X	X
Long Parameter List	-	X	-
Shotgun Surgery	-	-	X
Feature Envy	X	-	X
Data Clumps	-	-	X
Switch Statements	X	-	-
Data Class	-	-	X
Refused Bequest	-	-	X

2.5 Final Remarks

This chapter presented the basic concepts for understanding this work: (i) design patterns and (ii) bad smells. Design Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [Gamma et al., 1994]. Bad Smells are symptoms or structural characteristics in a region of code that may suggest the presence of a deeper problem in the system design or code. They are defined as code fragments that need refactoring [Fowler et al., 1999].

Besides these concepts, five design pattern and bad smell detection tools were presented. For design pattern detection we discussed DPDSS [Tsantalis et al., 2006] and MARPLE [Arcelli et al., 2008]. For bad smell detection we presented JDeodorant [Tsantalis et al., 2008], PMD [Copeland, 2005] and inFusion. The next chapter presents

some related work. We present a Systematic Literature Review concerning how studies portrait the relationships between Design Patterns and Bad Smells. This literature review found a total of 10 studies. Besides these 10 papers, next chapter also presents some studies that are not identified in the systematic review, despite they are related to the dissertation theme.

Chapter 3

Literature Review

Since their popularisation in the Software Engineering community by the publication of the GoF book [Gamma et al., 1994], design patterns have gained importance for software design and they have been target of a large variety of studies. It is advocated that design patterns improve the quality of systems [Prechelt et al., 2001] [Venners, 2005] and, in fact, there are some reported cases that support it [Beck et al., 1996]. On the other hand, there are reported cases claiming that design patterns may not be the best option in some situations due to their complexity, time consuming, and lack of understandability [Wendorff, 2001] [McNatt and Bieman, 2001] [Wydaeghe et al., 1998].

Bad smells [Fowler et al., 1999] are poor solutions and are also recurring target of studies [Vale et al., 2014] [Blonski et al., 2014] [Paiva et al., 2014]. Bad smells are not fully opposite to design patterns because they are related to the implementation phase while design patterns are related to design phase. However, insofar as design patterns are ultimately coded then bad smells and design patterns can be considered as opposite structures.

In this chapter, we present some work that are related to the topics of this dissertation. Section 3.1 presents some studies concerning design patterns and software quality. In the sequence, we narrow our research and perform a systematic literature review addressing design patterns and bad smells. Design patterns and bad smells are frequent targets of research, but they are rarely studied together, although there are some cases [Bouhours et al., 2010] [Trifu and Reupke, 2007]. Therefore, we aim at investigating the relationships that primary studies establish between design patterns and bad smells.

The rest of the chapter presents the systematic literature review. Section 3.2 introduces the systematic literature review. Section 3.3 presents the SLR planning,

by highlighting the research questions, selected sources, selection criteria, and then we detail the process of selection of studies. Section 3.4 presents the results of the systematic literature review. In Section 3.5 we present the threats to validity to the systematic literature review performed and Section 3.6 presents the final remarks of the whole chapter.

3.1 Studies on Design Patterns and Quality

Design patterns [Gamma et al., 1994] are recognized solutions to common design problems. The use of the most commonly referenced design pattern is often associated to good software system design. On the other hand, design patterns are also recognized by its complexity and integrating a design pattern in a real system may be a tough task. There are some studies in software engineering field that, despite they acknowledge design patterns benefits, they also investigate and analyze their impact on software systems quality.

Wendorff [2001] performed a study in which he could identify some questionable use of design patterns, like Proxy, Observer, Bridge, and Command. In order to illustrate Wendorff [2001] conclusions, we are going to summarize his findings on Proxy pattern [Gamma et al., 1994]. According to the author, Proxy is a simple pattern with an easy to grasp rationale behind it. Its simplicity makes it a typical "beginner's pattern", and beginners tend to use it freely. In many cases, developers justified the application of the proxy pattern with expected future needs for flexibility, access control, and performance, which in most cases these future needs never materialised [Wendorff, 2001]. It is known that, in general, the use of design patterns tend to increase the number of classes and methods. With Proxy it is not different. Therefore, its unnecessary employment makes the size and complexity of the software increases considerably.

Wendorff [2001] claims that it was found a Proxy instance in which considerable and complicated functionality was scattered over a hierarchy of proxy classes without a documented or conceivable rationale. Of course, the lack of clear and good documentation is not a specific design pattern flaw. However, when it comes to a complicated design, the lack of documentation is even worse since the code is bigger and more complex.

Bieman et al. [2003] examine five evolving systems and look at the relationship between design structure and software changes. Design structure is characterized by class-size, and class participation in inheritance relationships and design patterns.

Changes is measured in terms of a count of the number of times that a class is modified over a period of time. They quantify the design structure of an early version of each system, and study the relationship between the design attributes of this version and future system changes. The results of this study showed that, in general, classes that play roles in design patterns are changed more often than other classes. However, the authors informally analyzed this fact and realized that pattern participant classes provide key functionality to the system, which may explain why these classes tend to be modified relatively often. As the authors themselves state, changes to a design pattern should be made by adding new concrete classes to the existing pattern rather than by modifying existing pattern classes. Although it is necessary further investigation, change-proneness in patterns participating classes may be an indication of design patterns co-occurrences with either Shotgun Surgery or Divergent Change, which are bad smells defined by Fowler et al. [1999].

Prechelt et al. [2001] perform an experiment which investigates software maintenance scenarios that employ various design patterns and compares designs with patterns to simpler alternatives. In this study, there is an interesting section right on Introduction named "Isn't This Just Obvious?". In this section, the authors state that "clearly patterns do have the advantages claimed for them!" However, our intuition may mislead us and words such as "clearly" and "obviously" do not constitute confirmation.

The authors find that, in many cases, the design solutions in which GoF's patterns are employed are easier to maintain than their corresponding simpler solutions. Nonetheless, the authors detach that there are situations in which the use of design patterns made the program harder to maintain. They also suggest that these cases can serve as a starting point for empirically grounded development of guidelines for the use of patterns. Besides that, they confirm that compared to a straightforward solution, design patterns may provide unnecessary flexibility. Vokáč et al. [2004] and Krein et al. [2011] replicated the experiment performed by Prechelt et al. [2001]. Krein et al. [2011] state that from a practical perspective the experiment replication demonstrates a need for deeper and more meticulous studies on the topic of design patterns

Vokáč et al. [2004] found that the Visitor pattern, which has a fairly complicated structure, led to high cost in development time and poor correctness. Besides, they found that Decorator eases maintenance, despite being more difficult to trace the flow of control in a program, and increasing the time needed to understand it. Therefore, although its maintainability is good, it decreases understandability. Decorator also requires some training, just like Observer, but this one is then easy to understand and shortens maintenance. The authors also state that the Composite pattern, with its

reliance on recursion, caused some problems. Vokáč et al. [2004] explain that it may be that recursion is no longer in general use in this kind of software, and a possible cause is the availability of predefined container classes in most languages. They conclude that design patterns are not universally good or bad, but must be used in a way that matches the problem and the people. For instance, when approaching a program with documented design patterns, even basic training can improve both the speed and quality of maintenance activities.

3.2 Introduction to the Literature Review

A systematic literature review is a mean to identify, evaluate, and interpret research available to a particular research question, area or phenomenon of interest. The studies that contribute to the systematic review are called primary studies, while a systematic review is a form of secondary study. A review process can be summarized in three main phases: planning, implementation, and analysis of results [Kitchenham and Charters, 2007] [Biolchini et al., 2007].

Within the planning, it is established the reason of the review, the need for undertaking such revision. The most important activities are based on formulating the research questions in which the review is guided and then produce a review protocol, defining the basic procedures. In the implementation phase, once the review protocol was agreed, we should identify the primary studies through some documented research strategy in the sources defined in the planning phase. Then, one should evaluate and rank the studies found through the inclusion and exclusion criteria. By analyzing the results, we can collect and organize the data in order to publish the results. The main steps followed in this literature review are:

1. Formulation of the Question: consists in establishing the focus of interest and to formulate research questions that matches this interest;

2. Selection of Sources: consists in establishing repositories of research, other characteristics such as language of the studies, strings and methods research, among others;

3. Criteria Selection: consists in defining and explaining criteria for inclusion and exclusion of studies;

4. Selection of Studies: consists in applying inclusion and exclusion criteria and detail the process of selection of studies;

5. Summarization of Results: consists of presenting and analyzing the extracted information.

3.3 Systematic Review Planning

Systematic literature reviews have three main goals: (i) summarize existing evidence on a particular concept, e.g., a new treatment or technology; (ii) identify any gaps in current research in order to suggest areas for investigation; and (iii) provide a structure or a background in order to provide adequate support to new research activities [Kitchenham and Charters, 2007].

In the context of this dissertation, the systematic literature review performed during this work aims to find studies that address the relationship between design patterns and bad smells. A specific goal is to verify whether there are also studies that focus on the co-occurrence of bad smells with design patterns. If there are not studies with such focus, this might be a gap of the current studies. Another goal of this secondary study is also to provide a background and support the whole dissertation, which is related to these terms. The conduction of this systematic review is therefore necessary and it is the starting point to the whole work.

3.3.1 Research Questions

The following research questions are established in order to find studies that establish a relationship between design patterns and bad smells, more specifically if there are studies that focus on the co-occurrence of design patterns and bad smells.

RQ 1. How is the relationship between design patterns and bad smells investigated in literature?

RQ 2. Are co-occurrences between design patterns and bad smells analyzed in literature? If so, how are these co-occurrences analyzed?

3.3.2 Selected Sources

At this stage, we decided that it would be used two scientific repositories, namely IEEE Xplore and ACM Digital Library. Table 3.1 shows the two repositories, IEEE Xplore and ACM Digital Library, and the repositories direct links. The search was conducted in full text and not only metadata. This decision was taken so that the results were not very limited, given that the goals established in this secondary study require greater flexibility in the search of studies.

Table 3.1. List of Sources of Searching

Sources	Links
IEEEXplore	http://ieeexplore.iee.org/
ACM Digital Library	http://dl.acm.org/

IEEE Xplore and ACM Digital Library were chosen because they are widely known and used in academia. Moreover, unlike Springer (for instance), the authors have access to the papers in these repositories. Both repositories present the advanced search option with use of keywords in the meta-information, or full text, or both. To perform the search, the following keywords and variations were defined:

- *design pattern*: design patterns
- *bad smell*: bad smells, design smell, design smells, code smell, code smells, antipattern, antipatterns, anti-pattern, anti-patterns

It was decided that the primary studies should present the two keywords above - or one of their variations. It is important to highlight that, although anti-patterns and bad smells have different definitions and some distinct characteristics, for the goal of this dissertation they are considered synonyms and therefore anti-patterns is included in this systematic literature review as an alternative for bad smells.

The language chosen for the research is English, since the vast majority of relevant papers in the area of software engineering are in this language. Therefore, from the terms defined and the AND and OR constraints provided in the advanced search of both IEEE Xplore and ACM Digital Library it was built the following search string:

Table 3.2. SLR Search String

("code smell" OR "code smells" OR "bad smell" OR "bad smells" OR "design smell" OR "design smells" OR "antipattern" OR "antipatterns" OR "anti-pattern" OR "anti-patterns") AND ("design patterns" OR "design pattern")

3.3.3 Selection Criteria

In order to select the studies, Kitchenham and Charters [2007] suggest that clear inclusion and exclusion criteria should be established. These criteria are presented in this subsection. The aim of the systematic review is clear and the main inclusion criterion follows the same line: All the papers that establish a relationship between the two terms defined in this systematic review were considered valid.

Furthermore we established that papers have to relate design patterns and bad smells necessarily. How the relationships between these terms are presented in the studies may vary. Although we had a former idea on what relationships could exist, they are not strictly fixed, which means that the relationships established between design patterns and bad smells do not belong to a closed list. That is, any valid relationship was accepted in this systematic literature review, among them, we include the following possible relationships.

1. Structural relationship relation: It is the relation in which a comparison is made between the structure of the terms. For example, a comparison of certain classes that may compose both a design pattern role and be regarded as a bad smell.

2. Refactoring relation: It is the relation in which a refactoring strategy is applied to eliminate a bad smell by applying a design pattern.

3. Cause-consequence relation: It is the relation in which the existence of one of the terms (e.g., design patterns) may cause the emergence of another (e.g., bad smells).

We detach that this is not a closed list, but just some ways we wonder authors may relate design patters and bad smells. We anticipate that during our analysis, we only identified studies that relate design patterns and bad smells through refactoring or structural comparison. In order to obtain only papers that fit the goals of this review we defined four exclusion criteria.

Table 3.3 summarizes the inclusion and exclusion criteria we followed in this study. The first column shows the two inclusion criteria whereas the second column presents the exclusion criteria. As defined in this table, studies that focus on the analysis and comparison of detection tools were excluded. On the other hand, if the primary study not only presents, compares or analyzes detection tools, but also clearly shows any relationship between these relevant terms (as the ones cited in the inclusion criteria), this primary study was included in the systematic review.

3.3.4 Selection of Studies

Given the inclusion and exclusion criteria, the primary studies were extracted directly from the IEEE Xplore and ACM Digital Library repositories following the protocol below. It is important to detach that this systematic literature review was undertaken during the first semester of 2014.

Table 3.3. Inclusion and Exclusion Criteria

Inclusion criteria	Exclusion criteria
Papers that clearly present a relationship between design patterns and bad smells	Papers that focus on the analysis and comparison of detection tools
	Duplicated papers
Papers must be at least 4 pages long	Papers that are not primary studies
	Technical reports, manuals, and conference proceedings prefaces

1. Selection of studies through the search string;

2. Selection by analyzing the abstracts returned in step 1;

3. Selection by scanning the studies returned in step 2.

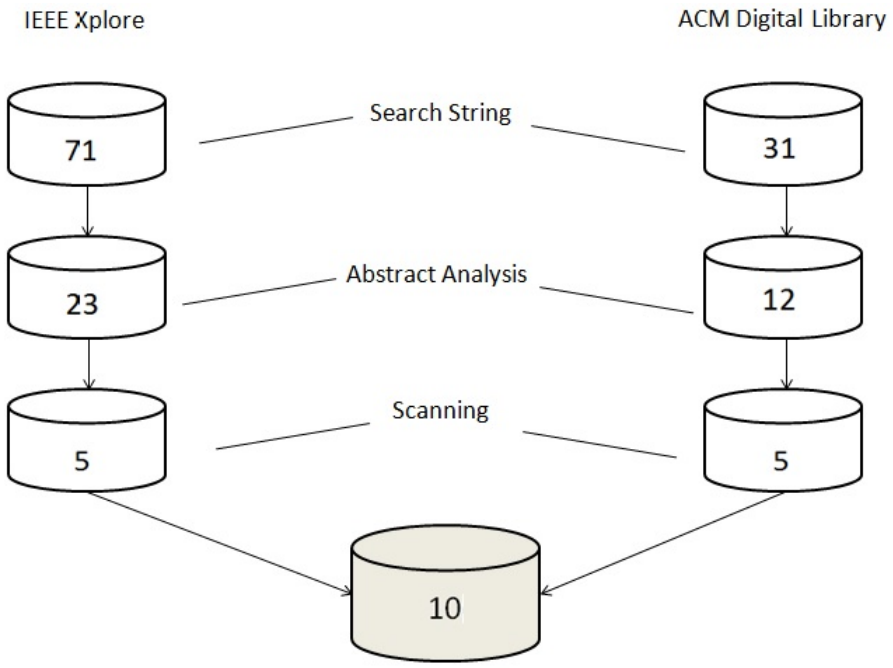

Figure 3.1. Evolution of Selected Studies

Figure 3.1 shows the evolution of the filter used by showing the number of primary studies retrieved on each step of the protocol in both repositories. Although Figure 3.1 shows the evolution of retrieved studies per repository, this does not mean that the repositories themselves indicate an inclusion or exclusion criteria. This division per repository is merely a way of organizing the retrieved studies. The application of the search string returned 71 studies in IEEE Xplore and 31 in ACM Digital Library. Therefore, the number of studies retrieved at the first step was obtained automatically.

The second step was the reading of the abstracts of the 102 primary studies found. At this step the inclusion and exclusion criteria were applied. Therefore, we analyzed if a relationship between design patterns and bad smells could be verified just by reading the abstract. Besides that, when the primary study clearly meets one of the exclusion criteria, then this study was removed from the set of studies. So, we excluded studies that are not primary studies (e.g., systematic literature reviews), technical reports, manuals, conference proceedings prefaces, and duplicated papers.

At the end of this phase, 23 studies remained from the IEEE Xplore repository and 12 from ACM Digital Library. What caused major doubt during this step was studies which abstract focuses on the detection tools, but development of the text could present a valid relation between the terms defined in this systematic review. It is important to say that, at this step, in case of doubt, the primary study was included as a valid study in order to further analyze it during the third step of the protocol. This decision was taken in cases where by simply reading the abstract of the study could not determine whether or not this study met the goals of this systematic review.

Finally, a scanning was taken over the remaining 35 studies in the third step. By using the technique of scanning, we verified whether the study helps to answer the question established in this secondary study. At this step, the inclusion and exclusion criteria were applied more strictly to the remaining studies so that, in the end, the ten studies retrieved indeed meet the goals of the systematic review. Therefore, 10 primary studies were considered in this systematic review as presented in the next section.

3.4 Results of the Systematic Review

As shown in previous section, after applying all the protocol steps for retrieving the primary studies, this systematic literature review found a total of 10 studies that met its goals and fit the inclusion and exclusion criteria. Table 3.4 presents all primary studies retrieved and the kind of relationship it establishes between design patterns and bad smells: Refactoring or Structural. This table displays the title, authors, year of publication of the studies and the relationship. The studies are sorted by their publication year. Among the studies retrieved, six focus on refactoring opportunities, whereas four of them make a structural comparison design patterns and bad smells. We present in this section a short description of each of the 10 primary studies retrieved in this literature review.

Table 3.4. Summary of SLR Selected Studies

Title	Author	Year	Relation
SearchBased Determination of Refactorings for Improving the Class Structure of ObjectOriented Systems	Seng, Stammel and Burkhart	2006	Refactoring
Towards Automated Restructuring of Object Oriented Systems	Trifu and Reupke	2007	Refactoring
SQUAD: Software Quality Understanding through the Analysis of Design	Khomh	2009	Structural
Perspectives on Automated Correction of Bad Smells	Pérez and Crespo	2009	Refactoring
Sharing Bad Practices in Design to Improve the Use of Patterns	Bouhours et al.	2010	Structural
A Logic Based Approach to Locate Composite Refactoring Opportunities in Object-Oriented Code	Jebelean, Chiril and Cretu	2010	Refactoring
Impact of Refactoring on Quality Code Evaluation	Fontana and Spinelli	2011	Refactoring
Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems	von Detten and Becker	2011	Structural
Software Change in the Solo Iterative Process: An Experience Report	Dorman and Rajlich	2012	Refactoring
On Extended Similarity Scoring and Bitvector Algorithms for Design Smell Detection	Polášek et al.	2012	Structural

3.4.1 Structural Relationship

Four out of the ten studies establish a structural relationship between the terms design pattern and bad smell. During this sub-section we briefly analyze and explain these four studies, which are presented in the same order they appear in Table 3.4.

Khomh [2009] proposes a theory that helps to understand how the design phase of systems affects their quality. It also proposes to build quality models. In order to achieve this goal, the author analyzes the role of class in fragments that present design patterns and bad smells and, thus, establishes a parallel between the structures of these terms.

Bouhours et al. [2010] propose a study whose focus is on to promote a collaborative website. Its goal is to share bad practices used during the design phase of systems in order to improve the use of patterns. Throughout the paper, the authors define the concept of spoiled patterns as a pattern corresponding to a deterioration of the intrinsic qualities of a design pattern. Hence, the structural differences between a de-

sign pattern and spoiled pattern cause degradation in the efficiency of solving problems appropriately.

von Detten and Becker [2011] propose the combination of two strategies of reverse engineering to component-based systems: (1) clustering-based and (2) pattern-based. On one hand, clustering-based reverse engineering approaches try to recover a system architecture by grouping its basic elements, like classes. The authors explain that a clustering-based analysis can be carried out in short time frames, even for large systems. On the other hand, pattern-based reverse engineering approaches aim at the detection of pre-defined structural or behavioral patterns in the software, like design patterns [Gamma et al., 1994] and anti patterns [Brown et al., 1998]. The authors highlight that since pattern detection often takes much more detailed information like control and data flow into account than clustering-based approaches, it is generally much slower. The authors also state that, because bad smell occurrences in a system lead to strongly coupled classes, they will be clustered together in a clustering-based approach, different from design patterns, which for the most part strive to produce a modular, uncoupled architecture.

The study presented by Polasek et al. [2012] proposes to modify two tools for the detection of design patterns so that the algorithm can also find anti patterns. Therefore, this study highlights important differentiations between the structures of design patterns and anti patterns in order to modify and extend the algorithm. It also extend its application to detect anti patterns. In this study, the term design smell is employed as a synonym for anti patterns contained in the software model. The authors detach that some design patterns characteristics are also usable for flaw detection. The authors analyze structural features such as associations, generalizations, and class abstraction.

3.4.2 Refactoring Relationship

Six studies obtained as a result of this systematic review establishes a relationship of refactoring between bad smells and design patterns. During this sub-section, we briefly analyze and explain these six studies, which are presented in the same order they appear in Table 3.4.

Seng et al. [2006] work on the issue of degradation of the code of a system over time. They discuss that the structure of these systems must undergo constant maintenance in order to identify and eliminate fragments that negatively impact on the quality of software. These fragments are called bad smells. The authors describe an automatic search-based approach that proposes a list of possible refactorings based on

design patterns. However, the decision to apply or not the proposed refactorings is left to the user.

Trifu and Reupke [2007] highlight the practice of constant maintenance of software and the high costs involved. Such maintenance tasks may be caused mainly by two factors: poor quality of the original design and age of the system. In any case, refactorings are needed and this study introduces and evaluates a tool that proposes major refactorings to structural flaws generated in object-oriented systems. Among the proposed refactorings to eliminate bad smells, there are those that make use of design patterns.

In a similar way to Seng et al. [2006], Pérez and Crespo [2009] propose an approach to the issue of degradation of code over time. That is, the system undergoes constant changes and many of these changes diverge from the proposed during the design phase causing the emergence of bad smells. In addition to the historical review of tools that suggest refactorings, this study proposes a technique based on automatic refactorings in order to remove bad smells automatically, including refactoring through the introduction of design patterns.

Jebelean et al. [2010] propose a logical approach to automatically detect parts of the code that should be refactored. In this study, the authors proposed an approach that automatically detects code fragments that have bad smells and can be refactored by employing the Composite design pattern. This pattern intents to compose objects into tree structures to represent part-whole hierarchies.

Fontana and Spinelli [2011] also work with the theme of refactoring, making an analysis of the impact of software evolution when refactoring techniques are applied. The study cites the importance of two Gamma et al. [1994] pattern - Visitor and Strategy - to remove the bad smell Divergent Change. However, these patterns may also co-occur with the bad smell Feature Envy given their characteristics. Both Divergent Change and Feature Envy are described by Fowler et al. [1999].

Dorman and Rajlich [2012] aim at reporting the experience of a programmer who makes improvements to an open source system. This process is observed by the granularity of Software Change and Solo Iterative Process. Among the improvements, the authors discuss the elimination of bad smells and changes in design patterns. The study treats refactorings as a stage in the software change process.

3.5 Threats to Validity

By conducting a systematic review, one of the steps of information extraction is the resolution of disagreement between reviewers. The systematic literature review we conducted may have been threated by the solo view of the involved researchers. In order to minimize this threat, all the phases of the conduction of this review were detailed and documented. Among these phases, we record the definition of keywords and their variations, the search string, and the refinement of retrieval of studies through the protocol steps of this systematic review process.

Another possible threat is the fact that we used two repositories of research: IEEE Xplore and ACM Digital Library. If the search was expanded to a larger number of repositories, the conclusions made in this systematic review could have been broader. However, we made this decision due to time constraints. In addition, we focus on reputable repositories that are widely used in universities and research centers. This way, one can ensure that the returned studies are significant and relevant.

3.6 Final Remarks

This chapter presented some work related to design patterns and bad smells. In Section 3.1, we showed some papers focused on design patterns and software quality. This part was not made systematically, but we found interesting works analyzing the inadequate use of design patterns and its consequences. For instance, Wendorff [2001] identified some questionable use of design patterns, like Proxy, Observer, Bridge, and Command.

In the sequence, we narrowed our research to analyze how studies in software engineering field relate design patterns and bad smells. We performed a systematic literature review. As expected most of the studies focus on refactoring opportunities, six out of ten. On the other hand, four studies establish structural analysis concerning design patterns and bad smells. By accomplishing this systematic literature review we realized that the co-occurrence between design patterns and bad smells is not an unexplored theme, although it is mentioned in some papers.

The next chapter presents the exploratory study performed in this work in order to identify instances of co-occurrences of design patterns and bad smells. The study is tool-aided and it was performed over five software systems. The goal of this exploratory study is to discover some concrete instances of design patterns and bad smells co-occurrences.

Chapter 4

Exploratory Study

Design patterns are general solutions to recurrent design problems in object-oriented software design [Gamma et al., 1994]. Despite this definition, their inadequate use has been called in question [Wendorff, 2001] [Cline, 1996]. Design patterns definitely have their place, but their misuse or even their overuse may increase complexity and decrease understandability, effectiveness and other quality attributes [Bansiya and Davis, 2002]. Cline [1996] states that design patterns might be overly hyped in some cases. According to him, design patterns may not deliver the benefits that some have naively stated. Cline [1996] continues saying that almost everyone in the design patterns community has been concerned about this for some time, but unfortunately, some consultants do not understand the limits of the technology.

Following this idea, we believe that the indiscriminate employment of design patterns may even introduce bad smells in code. In the previous chapter, we presented some works on design patterns and software quality a systematic literature review. In the SLR we searched how primary studies associate design patterns and bad smells. We concluded that, in general, researches establish a refactoring or a structural parallel between design patterns and bad smells and only a few studies mention that they may co-occur [Seng et al., 2006] [Pérez and Crespo, 2009] [Fontana and Spinelli, 2011].

In order to identify concrete instances of design patterns and bad smells co-occurrences, this chapter presents an exploratory study using some of the detection tools presented in Chapter 2. Although the exploratory study uses automated tools, we also evaluated the results by analysing the source code in which the co-occurrences are detected. In Section 4.1, we present the systems in which we ran the detection tools to find design patterns and bad smells instances. In Section 4.2, we revisit the detection tools that were chosen to perform this study. Section 4.3 presents the statistical background that is necessary to understand the measures used to analyse the results of

the study. Section 4.4 details the procedures we followed to perform the study. Section 4.5 presents and analyses the results of the study. Section 4.6 discusses some threats to validity to this study and, at last, we conclude this chapter in Section 4.7.

4.1 Target Systems

We use a design pattern detection tool and two bad smells detection tools to analyze twelve systems available on the qualitas.corpus [1] dataset. In total, the dataset provide compiled Java projects for 111 systems. It also presents a large amount of measurement data about these systems [Terra et al., 2013]. It was desirable that the chosen systems have a large amount of design patterns and bad smells instances to search for valid co-occurrence instances. Yet these characteristics were not known in advance, then the systems were chosen randomly and trying to cover different system sizes and business fields. Seven out of the twelve chosen systems did not present relevant amount of design patterns or bad smells. Hence, they could not be used in the analysis of this study.

Table 4.1 lists and presents some information about the five remaining systems that were used in this study. The systems are ordered according to the number of lines of code. Besides the name and version of the systems, Table 4.1 presents some size metrics. As can be observed in this table, AspectJ is the system with the highest number of lines of code, while WebMail has the smallest values for all metrics. By observing these metrics, we consider them medium to large size software systems.

Table 4.1. Target Systems for the Exploratory Study

System	Version	# of classes	# of packages	# Lines of Code
AspectJ	1.6.9	3600	144	501,762
Hibernate	4.2.0	7119	856	431,693
JHotDraw	7.5.1	765	66	79,672
Velocity	1.6.4	444	25	26,854
WebMail	0.7.10	115	19	10,147

4.2 Detection Tools

In order to achieve the goals of this exploratory study, we used three detection tools: one for design patterns and two for bad smells. For design pattern, we selected the Design Pattern Detection using Similarity Scoring (DPDSS) tool [Tsantalis et al., 2006], which

[1] http://aserg.labsoft.dcc.ufmg.br/qualitas.class/

uses the algorithm called Similarity Scoring presented in Chapter 2. In Chapter 2 we discussed that the DPDSS tool is not able to distinguish Object Adapter and Command patterns and for this reason, Tsantalis et al. [2006] put the Command pattern in the group of the structural patterns. We have the same consideration about the Strategy and State patterns, whose instances cannot be distinguished by the tool.

Tsantalis et al. [2006] also explain that the DPDSS tool is able to detect not only the default Proxy pattern, but also a variant pattern called Proxy 2, which combines both Proxy and Decorator. To make the analysis simpler, when this tool finds an instance of the pattern Proxy 2, we considered that it is an instance of the default Proxy design pattern.

Considering these points, a total of eleven design patterns could be detected and analyzed in this exploratory study: Factory Method, Prototype, Singleton, Object Adapter/Command, Composite, Decorator, Proxy, Template Method, Observer, Strategy, State, and Visitor. We choose this tool because, besides the variety of design patterns that it is able to detect, it has a friendly user interface and it is very effective in terms of time and memory consumption. This effectiveness becomes even more relevant because, according to the authors, the tool algorithm seeks not only the basic structure of the design patterns, but it also seeks for modified pattern instances. Another relevant point that contributed for the choice of the DPDSS tool is that it outputs all classes that participate in an instance of a detected design pattern in a log file. In fact, it makes more than that, the tool makes explicit which role the class is playing in the pattern. This is especially interesting for this work because, besides the quantitative analysis, the final results of this study are based on qualitative analysis. To make this analysis, we need to know the participating classes.

For Bad Smells, we chose two tools in order to identify a larger amount of the bad smells proposed by Fowler et al. [1999]. The tools are: JDeodorant [Tsantalis et al., 2006] and PMD [Copeland, 2005]. These tools were presented in Chapter 2 and we showed that each of them is able to detect four bad smells. Since both JDeodorant and PMD are able to detect God Class and Long Method, the two tools can find a total of six different bad smells: Duplicated Code, Long Method, God Class, Long Parameter List, Feature Envy, and Switch Statements.

We chose JDeodorant and PMD because they are well-known both in industry and in the academia [Fontana et al., 2012] [Munro, 2005] [Moha et al., 2010]. They are also effective in terms of time and they are free for use. Besides that, they are both Eclipse plug-ins, which make them easy to install and configure. Another important characteristic that both JDeodorant and PMD present is that they detach the detected code, different from inFusion. This characteristic is especially important for the same

reason we explained for DPDSS: by detaching the detect code, the tools make the validation and the qualitative analysis easier. At last, we also choose two tools because since detection tools present a high rate of false positive [Marinescu, 2005], the bad smells that are detected by both tools, God Class and Long Method, are considered to exist in a system only if both tools detect the same bad smell instance. This decision makes the results for these two bad smells more reliable.

4.3 Statistical Background

The detection tools presented in the previous section outputs the classes that participate in the design pattern and bad smell instances. However, in order to make a consistent quantitative analysis, it is necessary to have a statistical background. Hence, an important concept that is used to analyze the results of this study is association rules [Agrawal et al., 1993][Brin et al., 1997]. Association rules are combinations of items that occur in a dataset. In the context of this study, these rules represent the co-occurrence of two kinds of items: design patterns and bad smells. Four measures were calculated in order to understand the results: *Support* [Agrawal et al., 1993], *Confidence* [Agrawal et al., 1993], *Lift* [Brin et al., 1997], and *Conviction* [Brin et al., 1997]. To calculate these measures, it is considered that, in each system, each class is a transaction. It is then verified if the transaction (i) contains an instance of a bad smell and (ii) contains an instance of a design pattern. Each bad smell and each design pattern analyzed is called itemset.

Support of an itemset is the proportion of transactions which contains this itemset, showing its importance and significance [Agrawal et al., 1993]. For instance, if a system has 100 classes and 10 of these classes present the bad smell Feature Envy, it means that, in this system, Support of Feature Envy is 10%. For instance, the Support of the association Factory Method and God Class shows the proportion of transactions which contains both Factory Method and God Class. Therefore, Support is a measure of the frequency of an item in an association.

In order to understand the concept of Confidence [Agrawal et al., 1993] it is necessary to know the naming conventions used in association rules, which are antecedent and consequent. In the context of this study, we consider Design Patterns the antecedent and Bad Smells the consequent. Confidence is the probability of seeing the rule's consequent under the condition that the transactions contain the antecedent. In other words, it is the ration between Support of the association and Support of the antecedent. Confidence can be calculated by Equation 1. The value of Confidence

tends to be higher if the consequent has a high support, because this way, it is more likely that Support of the association is also high.

$$Conf(A \rightarrow B) = Sup(A \cup B) \div Sup(A) \tag{4.1}$$

Lift measures how often a design pattern and a bad smell occur together considering that they are statistically independent [Brin et al., 1997]. This measure can be calculated by Equation 2. If the value of Lift for an association rule is 1, then the itemsets of this association are independent. On the other hand, the higher than 1 for this measure, the more interesting the rule is, because it means that the antecedent lifted the consequent in a higher rate. This means, in the scope of this study, that a bad smell is more frequent with a determined design pattern.

$$Lift(A \rightarrow B) = Conf(A \rightarrow B) \div Sup(B) \tag{4.2}$$

Conviction is an alternative to Confidence since the latter does not capture direction of associations adequately. Conviction is calculated by the formula below [Brin et al., 1997]. Conviction presents three interesting characteristics: (i) it considers both Support of the antecedent and Support of the consequent; (ii) it shows if there is a complete independence between the antecedent and the consequent when the result is 1, and (iii) when the antecedent never appears without the consequent (confidence of 100%) the value of Conviction is infinite.

$$Conv(A \rightarrow B) = \frac{Sup(A) \times (1 - Sup(B))}{Sup(A) - Sup(A \cup B)} \tag{4.3}$$

4.4 Study Procedures

This section aims to explain how the exploratory study was performed. Besides this, we explain the decisions made along the study. In order to perform this study, we followed a set of procedures, divided in three general phases. Figure 4.1 shows the flow of these phases: Selection and Tuning, Execution, and Analysis. After the Execution phase, we persisted data in a relational database in order to make our analysis simpler. We documented these procedures to make this study replicable.

The first phase is Selection and Tuning and it consists of two procedures: (i) Choose systems and (ii) Choose and set detection tools. This phase and its procedures can be observed in Figure 4.1. Selection and Tuning phase consists in choosing the systems to be analyzed within the study and setting the detection tools that bring us the

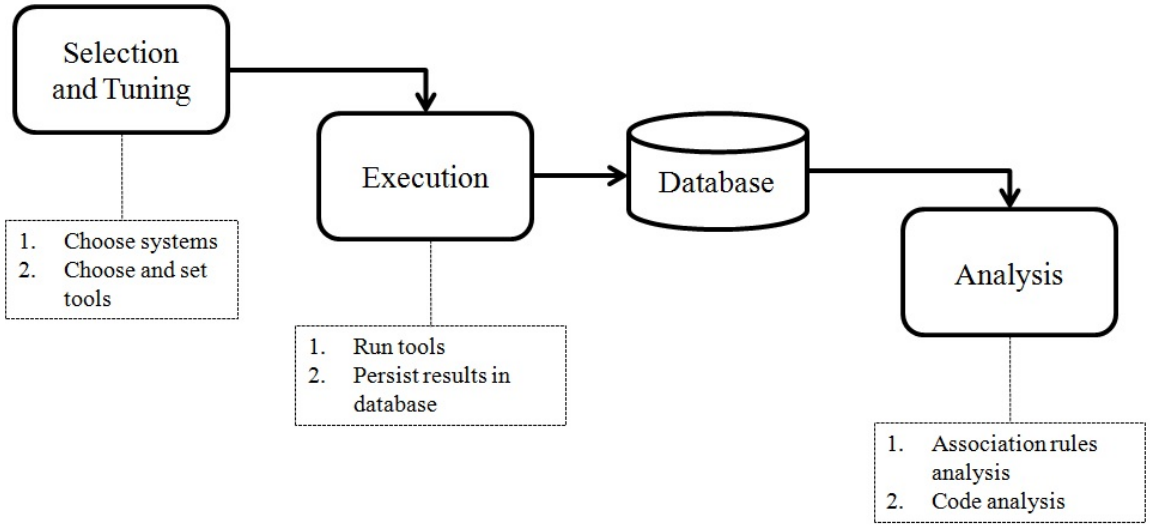

Figure 4.1. Procedures of the Exploratory Study

results. The chosen systems were presented in Section 4.1. The reasons why we chose DPDSS, JDeodorant and PMD were explained in Section 4.2. Besides choosing the tools, it was necessary to configure them. It was necessary to change two parameters in JDeodorant: Minimum Number of Statements and Minimum Number of Statements in Methods. We set the Minimum Number of Statements parameter to 3 and the Minimum Number of Statements in Methods to 2. For PMD it was also necessary to change two parameters: Parameter List and Method Length. We set the value of Parameter List to 10 parameters and Method Length to 100 LOC. These parameters were tuned in order to obtain a considerable amount of data that we were able to anylize. The other parameters of these tools remained with the default values. For DPDSS, it was not necessary to configure any parameter. This phase is very important since the final results of the study depend on these choices.

The second phase is Execution and it also consist of two procedures: (i) Run tools and (ii) Persist Results in Database (See Figure 4.1). The first procedure of Execution phase is the main part of this exploratory study. It consists in running each tool for each of the chosen systems and then collect the output of each detection tool for each system. At this point, we have many output files and each tool has an output format. Considering that PMD outputs two log files (there is a separate log file for Duplicated Code results), we have four log files for each system. Then, we had a total of 20 log files and we could not analyse the co-occurrence matchings manually. Therefore, it was necessary to develop a script that accepts the outputs of each detection tool as input and save the results in a database. This database schema was created with a

single table called "Class". It has 19 columns and they are all Booleans, except two: the class name and the system it belongs to. The other 17 columns indicate if a class has any of the eleven possible detected patterns or any of the six possible detected bad smells. By this moment, we have the information of which design patterns and which bad smells each class of the five systems has. This information makes it possible to calculate association rules for each of the 66 pairwise associations of design patterns and bad smells. The total number of associations is obtained by multiplying the 11 detectable design patterns by the 6 detectable bad smells.

The last phase is split in two steps: the association rule analysis and source code analysis. At first, we used the association rules as a starting point to establish relations between design patterns and bad smells. Based on interesting values, we focused our investigation on the analysis of the system source code. In this second step, we aim to understand deeply how and why design patterns and bad smells co-occur. This step is based on code review of each relevant co-occurrence.

4.5 Results

As already stated, five investigated systems present relevant results and were used in this exploratory study (Section 4.1). For the God Class and Long Method bad smells, it is considered an occurrence of them if both detection tools detected the same instance in order to make the results more reliable. This definition reduced significantly the amount of God Classes detected in the five systems and totally eliminated the occurrences of Long Methods because the tools found different results for this bad smell, since PMD and JDeodorant use different detection techniques [Fontana et al., 2012].

It is also important to detach that the Support value of any of these items is very high. This is expected since it is not common that a system would have many classes related to a great quantity of design patterns and bad smells. In fact, for some classes, Support of some design patterns and bad smells is zero.

Figure 4.2 shows the Support value for each design pattern for the five analyzed systems. It is easy to see by this figure that, indeed, Support value of any design pattern is high. The highest level is for the Strategy and State patterns (which cannot be differentiated by the DPDSS tool). JHotDraw has almost 25% of its classes participating in an instance of either Strategy or State, while Velocity has more than 10% of its classes participating in an instance of Strategy or State. Adapter and Command (which cannot be differentiated by the DPDSS tool either) also have detaching values, but they are only present in more than 10% of classes in the WebMail system.

Therefore, Figure 4.2 shows that the most common design patterns are Adapter, Command, Strategy, and State. However this result can be influenced by the fact that these four patterns appear in pairs (Section 2.2). Factory Method and Visitor participate in more than 5% of classes in JHotDraw and Velocity respectively, however these patterns are both absent in three out of the five systems. The other design patterns participate in less than 5% of classes of the systems in which they are present.

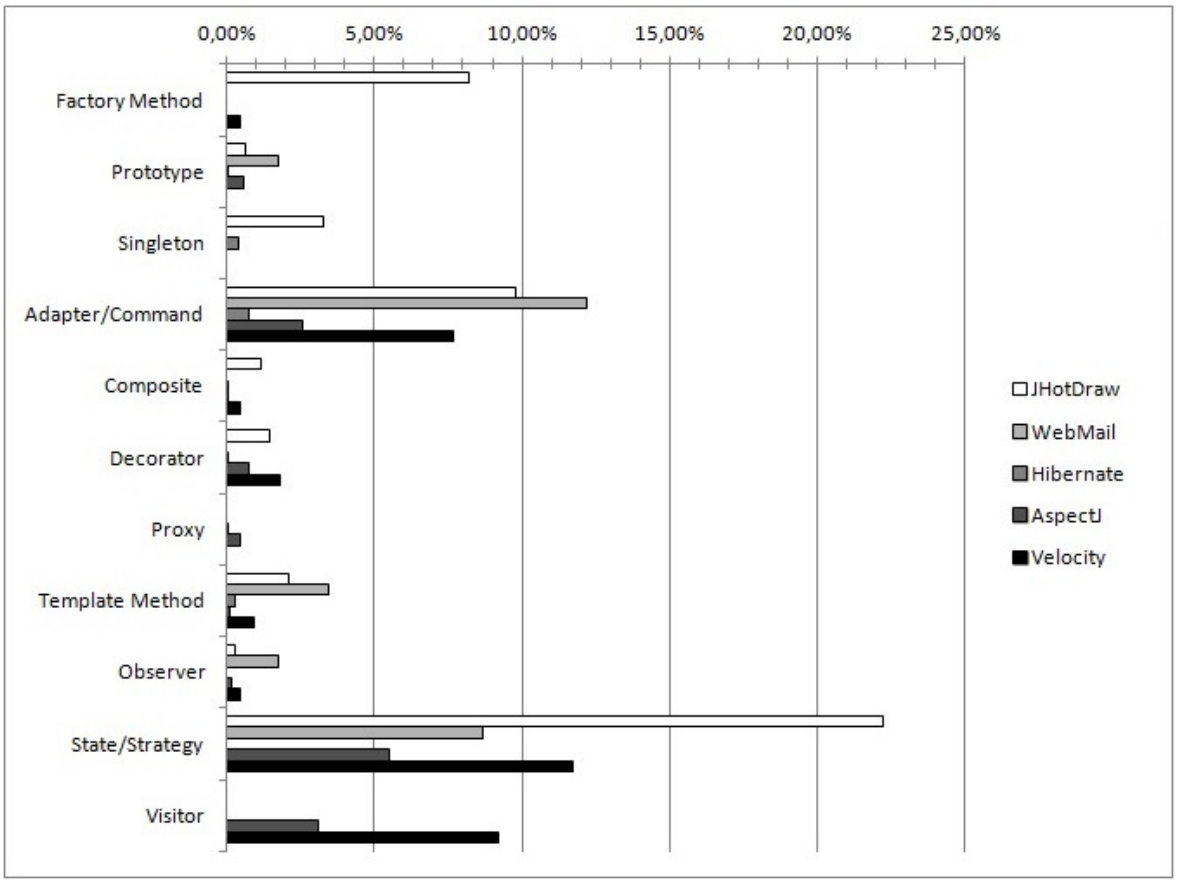

Figure 4.2. The Support Values of Each Design Pattern for the Five Systems

Figure 4.3 shows the Support value for each bad smell for the five analyzed systems. In this figure, we observe that over 15% of JHotDraw classes have Duplicated Code. In fact, Duplicated Code is the most common bad smell in overall, in fact, it is the only bad smell present in the five analyzed systems. Just the WebMail system has less than 5% of its classes with Duplicated Code. Feature Envy and God Class are the most frequent bad smell in the WebMail system. The first is present in almost 10% of classes in the WebMail system while the second is present in more than 5% of this system classes. The other bad smells participate in less than 5% of classes of the systems in which they are present.

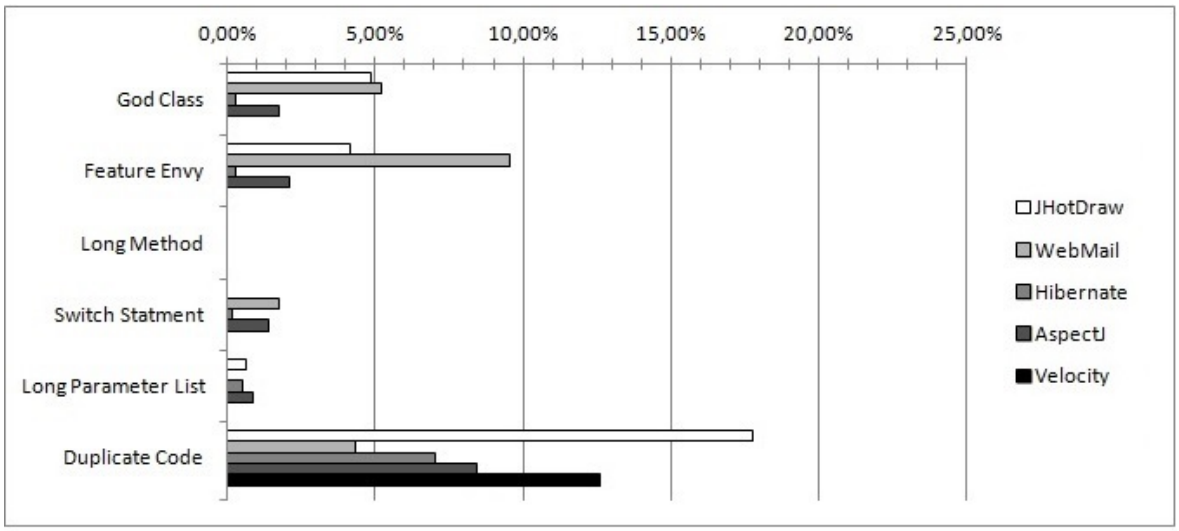

Figure 4.3. The Support Values of Each Bad Smell for the Five Systems

For this study, the more important results are based on the analysis of the Conviction of an association, and, of course, the understanding of the co-occurrence that we find. This way, after calculating Support, Confidence, Lift, and Conviction for all the possible associations, the next step of analysis is checking the associations that have the value for Conviction greater than 1.01. By considering just this threshold, a lot of associations were found. However, more important than these values is the reason why they happen. At this point, two co-occurrences called our attention: (i) the co-occurrence of the Command design pattern and the God Class bad smell and (ii) the co-occurrence of the Template Method design pattern and the Duplicated Code bad smell. These cases are discussed in the following subsections.

4.5.1 Command and God Class

At first glance, it was not possible to know if the Command pattern or the Adapter pattern or even if both patterns co-occur with the God Class bad smell, since the DPDSS tool cannot differentiate these two patterns. Therefore, we discuss these two patterns as a single design pattern. However, after we analyze the source code of the systems, we observe that the Command pattern co-occurs more frequently with God Class than Adapter does.

Table 4.2 shows the values for the Adapter/Command $\rightarrow$ God Class association rule in the analyzed systems. The five systems present instances of either Adapter or Command design pattern. Three out of these five systems present the God Class bad smell in one or more classes that participate of one of these two patterns. Just Velocity

does not present any God Class. Although Hibernate presents this smell, none of the God Classes co-occur with a class that participates of an Adapter or Command pattern instance. The other three systems have a value for the Conviction measure greater than 1.01. This way, we analyzed these co-occurrences in order to better understand them and to make possible to determine which of these patterns has strong correlation with God Class: Command or Adapter.

Table 4.2. Adapter/Command and God Class Association Rules Values

System	Support	Confidence	Lift	Conviction
AspectJ	1,00%	38,71%	22,12	1,60
Hibernate	0,00%	0,00%	0	0,99
JHotDraw	1,18%	12,00%	2,48	1,08
Velocity	0,00%	0,00%	0	1,00
WebMail	4,35%	35,71%	6,84	1,47

By analyzing the definitions of Command and God Class, it is not hard to conclude that a misused implementation of the pattern within the system evolution could cause God Class. For instance, Figure 4.4 shows a Class Diagram of an instance of the Command pattern identified in the WebMail system. The *Storage* class plays the role of a receiver in this pattern and it is associated with 14 concrete commands. The concrete commands are the classes that appear in the central part of the diagram in Figure 4.4. The base command is the *URLHandler* class and the abstract method is *handleURL()*.

By analyzing the diagram in Figure 4.4, we can observe that the *Storage* class has some getters and setters. In addition, it has many other methods that are responsible for too much work. Therefore, it is a God Class, as indicated by the JDeodorant and PMD tools. For instance, it is easy to see in Figure 4.4 that *Storage* class has many methods related with XML, debugging, configurations, authentication, user profile, and so on.

The *Storage* class probably became a God Class within the development of the WebMail system because it was necessary to support a large amount of commands. The names and code of these commands suggest that they do not belong to the same concern. Therefore, the best practice in this case would be the creation of different Command instances for different concerns; e.g., an instance of Command to deal only with XML. Instead of mixing up all concerns in a single design pattern instance, the creation of additional Command pattern instances would avoid that a class (i.e., Storage) plays the role of a God receiver. Therefore, methods of the *Storage* class should

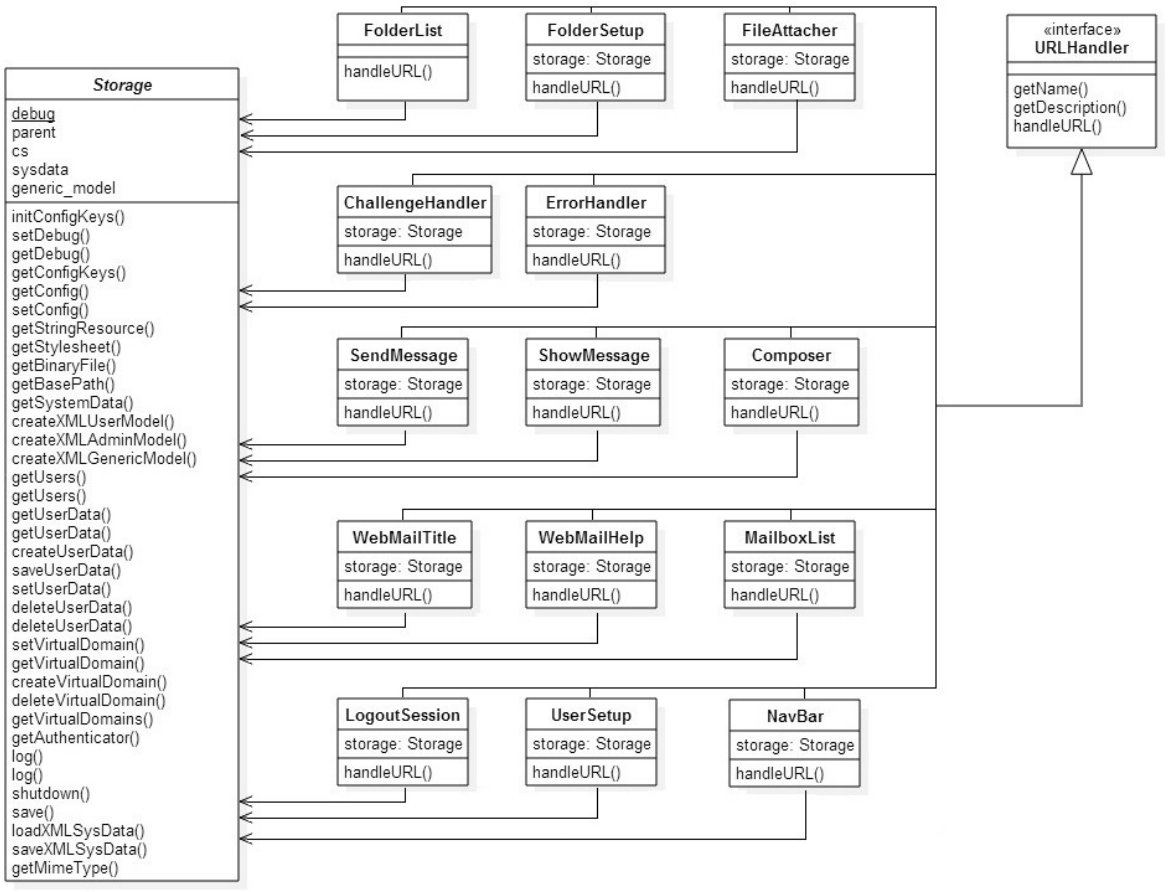

Figure 4.4. Example of Co-occurrence of Command and God Class

be extracted into other classes, which is a refactoring recommendation for most God Class instances [Fowler et al., 1999].

4.5.2 Template Method and Duplicated Code

Although we could not find many co-occurrences of Template Method and Duplicated Code, their existence called our attention because one of the goals of the Template Method design pattern is to avoid redundancy and, therefore, to avoid duplicated code. Table 4.3 shows the values for the association rule Template Method → Duplicated Code in the analyzed systems. The design pattern detection tool identified Template Method instances in JHotDraw and Hibernate. However, none of the systems presented many instances of this pattern, which made the values of the association rule relatively low. Except for WebMail, the other systems present many instances of Duplicated Code.

Table 4.3. Template Method and Duplicated Code Association Rules Values

System	Support	Confidence	Lift	Conviction
AspectJ	0,00%	0,00%	0	0,92
Hibernate	0,03%	9,09%	1,29	1,02
JHotDraw	0,39%	18,75%	1,05	1,02
Velocity	0,00%	0,00%	0	0,87
WebMail	0,00%	0,00%	0	0,96

We analyzed the source code of the systems to understand why Template Method and Duplicated Code co-occur. Figure 4.5 shows the Class Diagram of two instances of the Template Method pattern identified in the JHotDraw system. The two abstract classes in this figure have both a Template Method instance, which are called *drawFigure()* and *draw()*, respectively. By checking the source code of these two methods, we verified that they have the same implementation except for one line (copy-paste-modification). More interesting, these two classes have a super class in common and, therefore, they could have inherited this implementation from this common super class. However, the abstract classes are not the only ones in the hierarchy that present Duplicated Code. Subclasses in the same three, such as *ODGAttributedFigure* and *SVGAttributedFigure*, also present Duplicated Code. It is not the goal of this study to criticize the design, neither the development of any system, but the presence of this bad smell indicates that Template Method pattern seems misused over time since one of the achievements of this pattern is to eliminate code duplication.

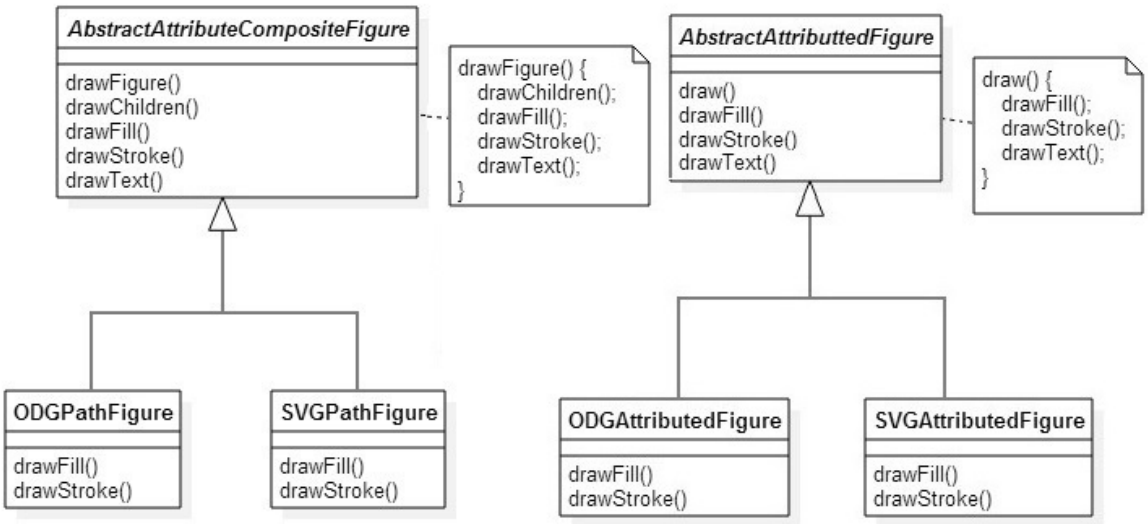

Figure 4.5. Example of Co-occurrence of Template Method and Duplicated Code

4.6 Threats to Validity

This exploratory study was performed using five systems and three detection tools and the results were analyzed by the authors. We analyzed five systems because seven out of the twelve systems we have chosen did not present relevant amount of design pattern or bad smell instances. However, after collecting data, we concluded that the number of systems is not so important, because we collected a large amount of information with the five ones. Besides that, we concluded that the size of the systems may not be a restriction, since, in some situations, we had more interesting results with the small systems.

Still concerning the systems used in this study, it is important to highlight that all of them are written in the Java language, which turns out to be a limitation. On the other hand, this is not a major problem since Java is one of the most employed programming language worldwide and it is one of the main references when it comes to design patterns. Therefore, a study which considers systems written in this language takes into account a great percentage of systems. Besides that, if we considered analysing systems written in other languages, it would add other variables into the study, since we had to check the language interference on the results. We also would have to find and analyze other detection tools for these languages, which may have different detection strategies, yielding different results.

Concerning the detection tools, we chose DPDSS, JDeodorant and PMD because they are easy to use and due to their acknowledgement in academia. As explained, bad smells that are detected by both JDeodorant and PMD were only accepted as a valid occurrence when both tools detected the same instance. We reinforce that the detection tools were used to determine the values for the association rules and these values are used as a starting point to guide our analysis and not to define the conclusions of this study. The final conclusions of this study are based on our analysis of the source code where the detection tools indicate co-occurrences between a design pattern and bad smell. Another consideration is on system versions and tools settings, since that choosing different system versions and by configuring different settings for the tools, other results may be achieved.

Although only two researchers analyzed the results of this exploratory study, all procedures of the study - including systems, tools information and how we derived the values for the association rules - are well documented. We also explain the reasons we confirmed the two co-occurrence instances: Command with God Class and Template Method with Duplicated Code. Therefore, this exploratory study is replicable.

4.7 Final Remarks

This chapter showed the development of an exploratory study which revealed interesting results. In this study, we identified two interesting co-occurrences between design patterns and bad smells. We found a co-occurrence instance between Command and God Class and also a co-occurrence between Template Method and Duplicated Code. Therefore, the results of this study supports that it is possible that the inappropriate employment of a design pattern leads to the arising of bad smells.

Unfortunately, this study is limited by the tools we use, since they do not cover all design patterns and bad smells. This way, the next chapter presents further co-occurrences cases that, based on the experience acquired during this work, we believe that are likely to happen in software systems.

Chapter 5

Lessons Learned

Design patterns represent knowledge that was already validated by the software engineering community and by software professionals in general. They are known solutions to specific problems and they can be reused in similar situations [Gamma et al., 1994]. The Factory Method design pattern [Gamma et al., 1994], for instance, defines an interface for creating objects. This way it allows a class to postpone instantiation to subclasses. This pattern is especially useful when the object creation is not straightforward and its creation may lead to a significant duplication of code as that type of object is necessary along the system code. Each of the GoF design patterns is concerned in solving a specific problem. Each one is detailed with its intent, applicability structure and so on.

However, by employing design patterns, two main issues arise: (1) designers have to identify the type of problem they encounter, before choosing a suitable design pattern. Moreover, not all problems have a suitable design pattern. In addition (2) designers have to ensure correct integration of the chosen pattern. A design pattern is not simply a design template and it requires adaptation in order to be well integrated into an existing context [Bouhours et al., 2014]. Furthermore, there are other points like pressure for fast development and developer inexperience, mainly when it comes to design patterns and their proper integration to the rest of the system code.

Considering these challenges, flaws may happen when using design patterns. For example, software professionals may employ a design pattern unnecessarily or the design pattern may not be properly integrated with the rest of the code. Besides these issues, due to design patterns complexity, not all professionals involved in the system development may understand their intent and misuse them over time as the software code evolves. Among these flaws, bad smells [Fowler et al., 1999] can be considered poor solutions to some specific implementation problems. In the previous chapter, we

presented an exploratory study in which we evidenced two instances of design patterns and bad smells co-occurrences. Such situations happen because of inadequate use of design patterns. In order to explore the inadequate design patterns usage, we present some lessons we learned during the development of this dissertation. We analyze some possible co-occurrence instances by exploring definitions of bad smells and design patterns.

In this chapter, we discuss six design patterns and bad smells possible co-occurrences. The examples are not real co-occurrence instances, but situations that may happen due to design patterns inadequate employment. Section 5.1 discusses the God Class smell and how it can co-occur with the Mediator and Facade patterns. In Section 5.2, we present and discuss the Middleman smell [Fowler et al., 1999] and its possible occurrence when the Mediator and Facade patterns are misapplied. Section 5.3 discuss the Divergent Change smell [Fowler et al., 1999] and how it may co-occur with the Strategy and State design patterns. Section 5.4 concludes this chapter.

5.1 Avoiding God Class

God Class is a common bad smell (Section 2.3). God classes perform too much work on its own, sometimes they know too much or control too much [Riel, 1996]. As previously cited, a God Class often arises when functionalities are incrementally added to an important class over the course of the system evolution. They are generally thought to be examples of bad code that should be detected and removed to ensure software quality [Vaucher et al., 2009].

God Class affects the evolution of design structures and considerably affects the use of inheritance [Deligiannis et al., 2004]. It centralizes the system functionality in one class, which contradicts the decomposition design principles. God Class clearly jeopardizes some quality attributes like reusability, maintainability, understandability, and modularity. Besides, if there are errors in the God Class, the error propagation tends to affect a big part of the system [Li and Shatnawi, 2007]. In fact, classes with God Class, Shotgun Surgery, and Long Method have a higher probability to be faulty than other classes [Li and Shatnawi, 2007].

Therefore, every class that centralizes data or functionalities are candidates to become a God Class. When it comes to design patterns, there are some cases, that, if the pattern is inadequately employed, a class that participates in this pattern may become a God Class. Some situations are discussed in the following sections.

5.1.1 Mediator

Mediator [Gamma et al., 1994] has the intent of defining an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly. This pattern lets the developer varies their interaction independently, which is desirable in systems code, since loose coupling promotes flexibility and reusability. The mediator is a good choice when the communication between objects is complicated, but well defined. It is desirable to have a central point of control when there are too many relationships between the objects in code to avoid tight coupling. As more classes are developed in a program, the problem of communication between these classes may become more complex and a Mediator may be necessary.

Figure 5.1 shows an object diagram of the Mediator pattern. This examples is presented by Gamma et al. [1994] and it illustrates how intense are the communication through and from the *Mediator*, which is represented by the *aFontDialogDirector* object in Figure 5.1. The *Mediator* defines an interface for communicating with *Colleague* objects. The *ConcreteMediator* class is the one which knows the colleague classes and it keeps a reference to the colleague objects. The *ConcreteMediator* is responsible for implementing the communication and transfer the messages between the colleague classes. The *Colleague* classes keep a reference to the Mediator object. Whenever it is necessary to communicate with each other, colleagues do not do it directly. They must communicate by using the mediator.

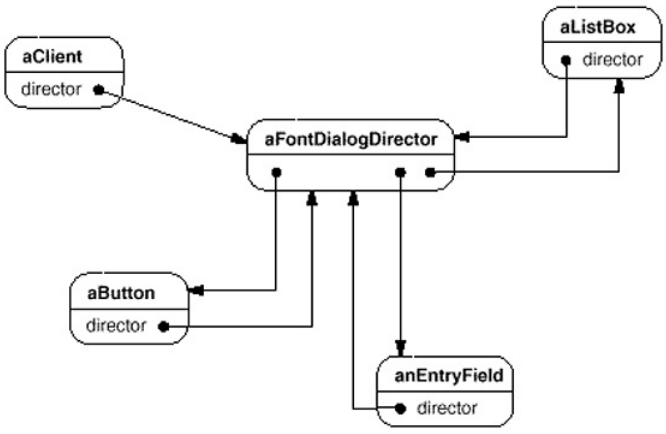

Figure 5.1. Mediator Pattern Object Diagram

The point is that it is necessary to pay especial attention to the *ConcreteMediator* class not only during design phase, but also during maintenance phase to avoid the *ConcreteMediator* to become a God Class. The chances of the *ConcreteMediator* class

becomes a God Class grows when it starts doing more than it should. For instance, the *ConcreteMediator* class should not be responsible for performing colleagues actions, but only coordinating actions across multiple colleagues. Another issue concerning the Mediator pattern is when the number of colleagues grows uncontrollably. As colleagues communicate back and forth using the *ConcreteMediator*, it tends to become larger and unwieldly and this may result in a too complex class, difficult both to test and maintain.

5.1.2 Facade

The Facade design pattern [Gamma et al., 1994] provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. The Facade pattern has some similarities with the Mediator pattern [Gamma et al., 1994]. Whereas Mediator centralizes communications of all colleagues, Facade is like a gateway to ease the clients access to sub-systems. Facade collects information or interacts with various sub-systems and passes the result back to client.

Facade aims at encapsulating a complex subsystem within a single interface. This reduces the learning curve necessary to appropriately use the subsystem. It also decouples the subsystem from its many clients. To illustrate these benefits, consider the arrangement for a trip. It is necessary to set several things, such as transportation, book the hotel, hostel or wherever the client might stay, check for tour options and restaurant. If there is a system for arranging trips, their clients would have to access and set all these issues individually.

Figure 5.2 shows the classes that compose some of the options a client have when arranging a trip. The goal of this diagram is not to fully contemplate all classes and attributes a trip system may contain, but just to illustrate some options the client may have and to show how difficult it is for a client to select these options. Figure 5.2 considers that the client can choose between a hostel and a hotel, but in a real system there might have other options. For transportation, three options are given: an airline, bus company or a train company. Besides, the client might check for restaurants and tour options on its own. At last, this diagram considers that the hotel might have partnership with restaurants, tour services and airlines.

In order to ease the clients choices, there are tour agencies. Considering the trip system, the tour agency plays the Facade role [Gamma et al., 1994], which provides a higher level interface and make the trip system, which in this case is the subsystem, easier to use. The role of the tour agency is to offer closed trip packages, like in a menu, so the client will not need to arrange so many details on its own.

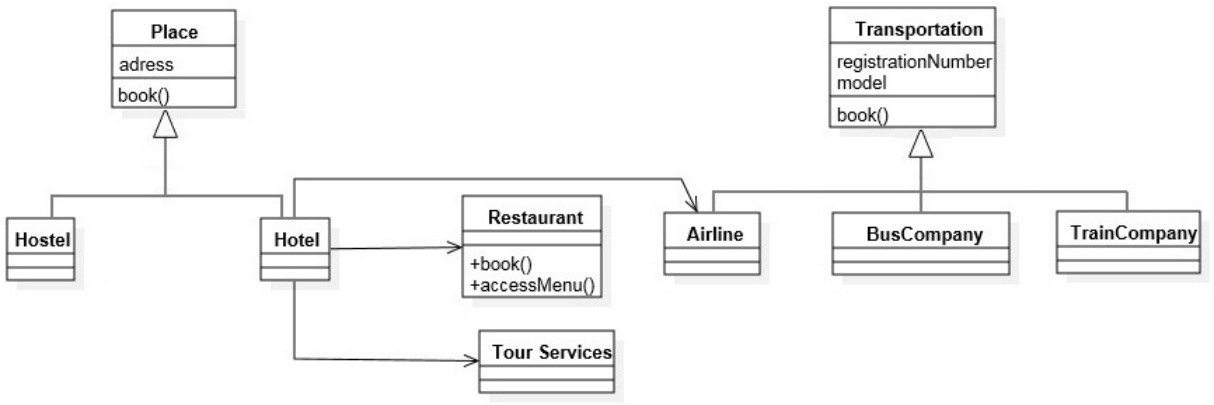

Figure 5.2. Class Diagram for Trip Arrangements

Figure 5.3.a shows an example on how the implementation of the Facade pattern may ease clients access to the subystem by providing a higher level interface. This interface should be furnished with functionalities that combines the subsystem classes methods. This way, the client is exempted from knowing the details of these classes in the subsystem.

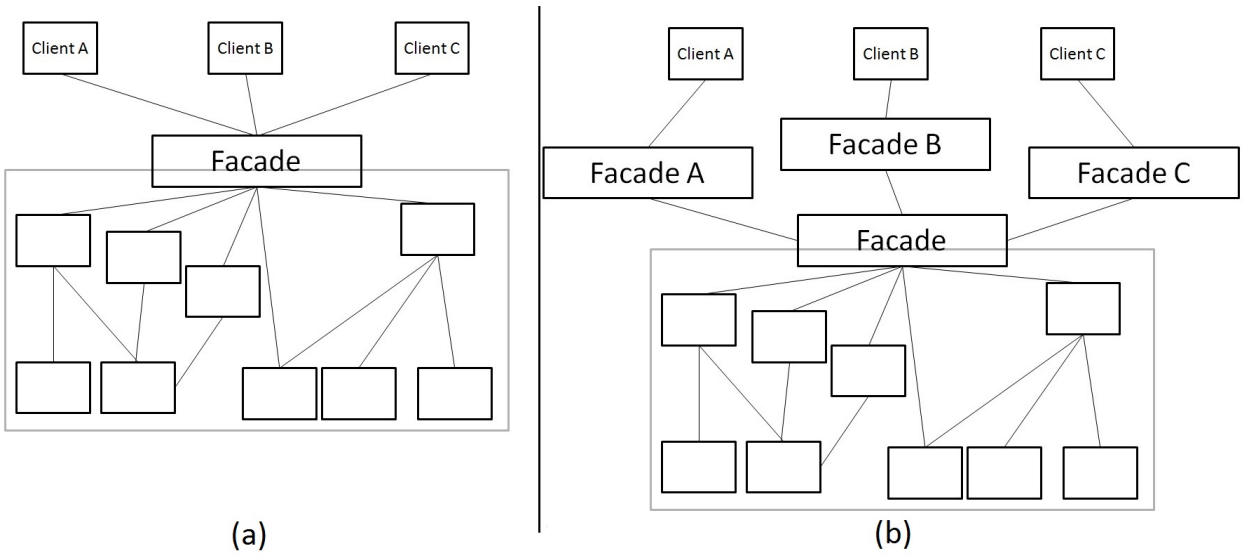

Figure 5.3. Two Examples of the Facade Pattern

However, it is necessary to be careful. Similarly to the Mediator pattern, depending on system, the *Facade* class is a potential candidate to become a God Class. The Facade should be a fairly simple facilitator, but in some contexts, like the tour agency system, the Facade is aware of so many options that its methods, and as a consequence the whole class may become true oracles and all decisions pass through the *Facade* class. Besides, if the Facade is the only access point for the subsystem, it

will limit the features and flexibility that the system may need. Another factor that might increase the *Facade* class complexity is the necessity of access for different kind of clients. Still considering the tour agency example, it happens if frequent customers have especial services, like a tour guide for free, or maybe especial sales offerings. If the *Facade* class has to handle it all, it is likely that it will become even larger and more complex, compromising its understandability, simplicity, and operability.

Figure 5.3.b exemplifies the situation in which may be necessary to split the *Facade* class in order to distribute the functionalities in different classes in a coherent way. Du Bois et al. [2006] states that when the Facade becomes to large, the best choice is to decompose the God Class, making a hierarchy of Facades. That is, more specialized Facades, such as Facade A, Facade B, and Facade C in Figure 5.3.b, interact with different clients. A general Facade offers a simple interface to specialized Facades.

5.2 Avoiding Middle Man

A Middle Man is characterized by a class that delegates most of its work to other classes [Fowler et al., 1999]. A Middleman is a class that is doing too much simple delegation instead of really contributing to the application. Of course, encapsulation is a great feature of object-oriented programming and encapsulation often comes with delegation, given that classes frequently delegate tasks to subsequent classes.

The Facade pattern, defined and exemplified in previous section, provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. However, what if the classes in the subsystem do not need an unified interface and the designer or maybe the developer thinks the Facade is necessary? Instead of implementing a Facade pattern instance, the developer is in fact creating a Middle Man. In this situation, most of the *Facade* class methods, if not all of them, are mere by passing operations that call a single method in a class in the subsystem. In fact, this is not a higher-level interface, but the same interface the subsystem provides, and it does not make it easier to use. This inadequate implementation of the Facade pattern misses its intention and is fact a Middle Man.

By implementing the Mediator pattern [Gamma et al., 1994], the software engineering professional must be equally careful because this pattern may also be misused and become a Middle Man. Mediator designs an intermediary to decouple many peers, but it is necessary to wary to the importance of this pattern so that this intermediary player becomes a simple Middle Man. According to Regulwar and Tugnayat [2012], the Middle Man smell is detected when many methods couple to one class with a low

cyclomatic complexity [McCabe, 1976]. In other words, the class that is a Middleman does not have many functionalities, concerning basically in delegating tasks to other classes.

For both misused design patterns that provoked the Middleman appearance, Facade and Mediator, Regulwar and Tugnayat [2012] state that the greatest problem is that every time it is necessary to create new methods or to modify the old ones, it is also necessary to add or modify the delegating methods, which is unnecessary and tiresome, for not saying error prone. Sometimes delegation can go too far and it is necessary to remove, or at least modify, a set of classes that are mere Middleman to simplify the system, since they do not do much on their own and just add needless complexity.

5.3 Avoiding Divergent Change

Divergent Change occurs when one class is commonly changed in different ways for different reasons [Fowler et al., 1999]. In other words, a class may be modified when different functionalities are implemented or evolved. Maybe, a class may be modified when lots of other classes have to suffer any change. Often these divergent modifications are due to poor program structure. Divergent Change may also be an indicator that the software code is tightly coupled and disorganized. This scenario turn maintenance tasks harder.

This section present two examples of design patterns that, when misused, may co-occur with the Divergent Change smell: Strategy and State [Gamma et al., 1994]. Although Strategy and State patterns are not so difficult to implement, they both promote a great increasing in number of classes. If Strategy and State are not well designed, their inadequate use may compromise the maintainability of classes that pertain to the pattern. Good object-oriented software design features a set of quality characteristics, such as maintainability, understandability, ease of evolution, and so on. However, even when developers are familiar with what has to be implemented, pressure, excessive focus on pure functionality, or just inexperience, mainly when it regards to design patterns implementation, may lead to flaws in system design.

5.3.1 Strategy

The Strategy design pattern [Gamma et al., 1994] defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. It is useful when there are multiple

algorithms for an specific task and it is necessary that the application to be flexible to choose any of the algorithms at runtime for an specific task. Its structure consists of a *Context* class, a base *Strategy* class and some concrete Strategies classes. The base *Strategy* class declares an interface common to all supported algorithms. The *Context* class uses this interface to call the algorithm defined by a *ConcreteStrategy*. The concrete strategies implement the algorithm using the *Strategy* interface. The *Context* class is configured with a *ConcreteStrategy* object; maintains a reference to a *Strategy* object and may define an interface that lets *Strategy* access its data.

For instance, considering the example of an online store, in which the system clients add some items to a cart. In the end, it is necessary to pay for the chosen items. Generally, online stores give the clients some payment options, such as credit card, back slip and Paypal, for instance. The payment options can be modeled as payment strategies. The strategies in the Strategy pattern are used in a *Context* class, which is the shopping cart in this example. Usually, each strategy needs some data from the *Context* class. Therefore, the payment strategy needs some information from the shopping cart. For instance, if the payment will be performed with credit card, it is necessary the card number, the date of expiry to name some.

In order to make use of the different strategies along the application, the *Context* class is used in a client and an application may have many clients. In the Strategy pattern's classic implementation, every client that uses the strategies should be aware of all the strategy classes. Hence, the clients have to pass information to the specific strategy while the client itself must aware of the all the strategies.

These characteristics unfolds in some situations that may provoke Divergent Change. First of all, an impaired implementation of this pattern compromises its scalability and, therefore, the addition of new strategies may cause changes in the client and maybe in the base strategy. Hence, the base *Strategy* class may be modified due to divergent changes in the concrete strategies. Considering a context in which the base *Strategy* class has some behavior common to all strategies, by adding a new strategy it is necessary to check if this behavior is still common to all subclasses and again, the base *Strategy* class may be modified. Moreover, modifications in existing strategies may also cause changes in the *Context* class, which also justifies this pattern limitations in terms of scalability and potential co-occurrence with Divergent Change.

5.3.2 State

The State pattern [Gamma et al., 1994] allows an object to alter its behavior when its internal state changes. The object will appear to change its class. It is the imple-

mentation of a state machine. The State pattern is a solution to the problem of how to make behavior depend on state. State and Strategy design patterns have similar structure, and both of them are based upon the open closed design principle [Martin, 1996], though they are totally different on there intent. Whereas the Strategy pattern is used to encapsulate a related set of algorithms, the State design pattern allows an object to behave differently at different state.

Figure 5.4 shows the State diagram. In the State structure, the base *State* class defines an interface for encapsulating the behavior associated with a particular state of the *Context*. The *Context* class has a *State* reference to one of the concrete implementations of the State and forwards the request to the state object for processing. Each of the *State* subclasses implement a behavior associated with a state of the *Context*.

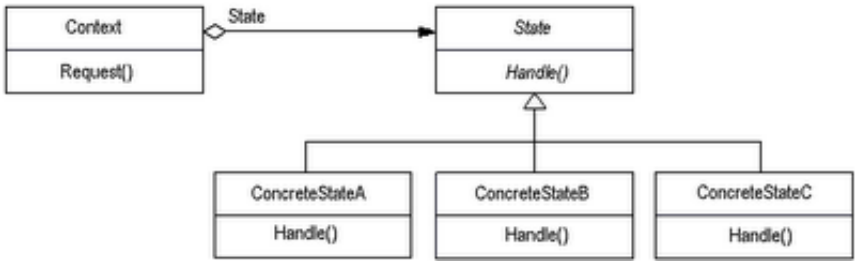

Figure 5.4. State Pattern Class Diagram

Since State and Strategy patterns are very similar, State suffers from the same weaknesses of Strategy and, then, it may co-occur with Divergent Change smell for the same reasons. However, State may be even more dangerous, since its intent may lead to an explosion of classes. Therefore, by applying this pattern, it is necessary to implement a class for each state. Even for situations without too many states, a large number of classes need to be implemented resulting in difficulty in reviewing the structure of the State pattern and undermining its maintainability and evolution once its implementation is smeared across multiple classes. This way, modification in the concrete states may affect the base *State* class and require changes in this class due to several reasons.

Another issue is that the State pattern does not specify where the state transitions will be defined. There are two options: the transitions are defined either in the *Context* class, or in each individual *State* derived class. If the criteria are fixed, the first option is preferable and, then, the transitions can be implemented entirely in the *Context* [Gamma et al., 1994], which limits the system evolution and changes in the states to affect the *Context* class. Then, in the State pattern, the *Context* class may also suffer from Divergent Change. The second option ease the addition of new states, but it

promotes a tighter coupling among the *ConcreteState* classes, this may lead to another bad smell: Shotgun Surgery [Fowler et al., 1999]. Since the states are aware of one another, a modification in a state may provoke changes in the other states.

5.4 Final Remarks

Based on the lessons learned from the systematic literature review (Chapter 3) and the exploratory study (Chapter 4), in this chapter, we analyze some design patterns and bad smells definitions and structures. The analysis involved four design patterns: Mediator, Facade, Strategy and State; and three bad smells: God Class, Middleman and Divergent Change. It is important to detach that the goal of this work is not to criticize the use of design patterns, but to alert that, due to their complexity, a wrong or unnecessary pattern usage may provoke the occurrence of bad smells in code.

Given their similarities, the Mediator and Facade patterns, when inadequately implemented, may co-occur with the same bad smells. In this chapter we discussed when and how they can co-occur with the God Class and Middle Man smells. The same happens to the Strategy and State patterns. Given their similarities, they have the same weaknesses. In this chapter, we exemplified how mistaken implementations of these patterns may provoke the Divergent Change smell.

The discussion presented in this chapter are just to illustrates possible issues that may raise from design patterns mistaken implementations. However, these examples demand further investigations and discussions on design patterns and bad smells co-occurrences. The next chapter summarizes the conclusions of this work and cite some possible directions for future work in order to investigate design patterns and bad smells co-occurrences deeper.

Chapter 6

Conclusion and Future Work

Within this dissertation, we analyzed the relationship between design patterns and bad smells. More specifically, we found and analyzed some co-occurrences instances between them as presented in Chapter 4. The main steps within this work involved the better understanding of the concepts related to it: design patterns and bad smells; the performance of a literature review; the accomplishment of an exploratory study in which we could find interesting results; and then we presented some lessons learned in order to aid professionals to use design patterns and avoid bad smells.

Most studies found in the systematic literature review (Chapter 3) involving the terms design patterns and bad smells focus on refactoring opportunities. Over half of the studies obtained, six out of ten, deal with this matter, and many propose tools for identifying fragments of code that can be refactored to patterns. It is well-marked that the main goal of current research regarding refactoring is the emergence of algorithms that attempt to identify fragments to be refactored and promote refactoring automatically. In some cases they aim at inserting or correcting design patterns no longer used or that were not used properly. Other four studies established a structural comparison between design patterns and bad smells.

The systematic literature review also showed the number of incipient primary studies researching and analyzing the co-occurrence of design patterns and bad smells. Of the ten papers, only three of them make reference to this research topic. For instance, Seng et al. [2006] mentioned that some design patterns deliberately violates existing design guidelines. The authors illustrate this argument through the Facade pattern, whose methods are often not cohesive since they only make requests to other classes that will meet these requests.

As previously mentioned, Strategy and Visitor design patterns can co-occur with the Feature Envy smell [Fontana and Spinelli, 2011].. Pérez and Crespo [2009] highlight

that while bad smells can be removed through the use of design patterns, the design patterns themselves may also be related to the emergence of bad smells, because some bad smells can be originated by the misuse of design patterns. Therefore, the detection of bad smells can be realized by identifying these misuses. The co-occurrence of design patterns and bad smells is cited in the software engineering literature, but it is still a little explored research field.

When performing the exploratory study (Chapter 4) we ran a design pattern detection tool and two bad smell detection tools in five systems. We identified some instances of design pattern misuse and we showed that this misuse may promote the arising of bad smells. The cases that called our attention the most were the co-occurrences of Command design pattern with God Class and Template Method pattern with Duplicated Code. We discussed how the overuse of a single receiver class in the Command pattern for different concerns turned this class into a God Class. We showed that two implementations of the Template Method pattern instead of eliminating unnecessary repetition, presented many duplicated code snippets.

We used the values for the association rules, especially Support and Conviction, as a starting point to derive our analysis. These values were used to analyze how strong a relation between a specific design pattern and a bad smell is. This way only after calculating these values, it was made an analysis considering the characteristics of the design patterns and the bad smells to check whether the association rules make sense. Despite we used association rules as a starting point, the final conclusions of this exploratory study were derived from our deep analysis of the source code.

At last, we discussed some lessons learned during the course of this work (Chapter 5). These lessons learned illustrate situations in which design patterns and bad smells may co-occur. These lessons aim at aiding software professionals to employ design patterns properly and avoid bad smells. These lessons may also help developers in the task of maintenance of classes that are part of a design pattern, since we believe that, in most cases, bad smells does not arise as a consequence of design patterns employment in the first moment.

The results of this work showed that it is possible that the inappropriate employment of a design pattern leads to the arising of bad smells, although this consequence seems totally aimless. Therefore, this work ease the way for future works and more interesting findings. In future work, it is important to analyze other concrete instances of design patterns that, due to their inadequate use, have classes that present bad smells. We foresee two possible studies that can be performed in order to achieve this goal.

The first possible future work is to replicate the exploratory study in an enterprise development context, which may provide more data and results with higher statistical significance. Besides changing the target systems, it is also possible to use other detection tools that may have better results and may detect more or maybe different design patterns and bad smells instances. Another option is to perform this study, - which does not exclude the first one, they can be both accomplished - without the aid of tools. For the success of this approach, it is better to make it in a totally controlled environment, which means that the systems should be well known and understood by the researchers. In this situation at least the design patterns employed have to be documented. It may be required that the system is not too large, which may ease the correct manual identification of bad smells. With data in hand, it is possible to make the analysis of co-occurrences between design patterns and bad smells.

As a future work, it is also interesting to undertake a survey involving preferably experienced professionals with the goal of analyzing code snippets in which contains different design patterns and bad smells. This survey may evaluate how these professionals behave when facing a co-occurrence of design patterns and bad smells.

Bibliography

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207--216.

Arcelli, F., Tosi, C., Zanoni, M., and Maggioni, S. (2008). The marple project: A tool for design pattern detection and software architecture reconstruction. In *1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*, pages 325--334.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4--17.

Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J. O., Dominick, L., and Paulisch, F. (1996). Industrial experience with design patterns. In *Proceedings of the 18th international conference on Software engineering*, pages 103--114.

Bieman, J. M., Straw, G., Wang, H., Munger, P. W., and Alexander, R. T. (2003). Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the 9th International Software Metrics Symposium*, pages 40--49.

Biolchini, A, J. C., Mian, P. G., Natali, A. C. C., Conte, T. U., and Travassos, G. H. (2007). Scientific research ontology to support systematic review in software engineering. *Advanced Engineering Informatics*, 21(2):133--151.

Blonski, H., Padilha, J., Barbosa, M., Santana, D., and Figueiredo, E. (2014). Concernmebs: Metrics-based detection of code smells. In *5th Brazilian Conference on Software, Tools Session.*

Bouhours, C., Leblanc, H., and Percebois, C. (2010). Sharing bad practices in design to improve the use of patterns. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, page 22.

Bouhours, C., Leblanc, H., and Percebois, C. (2014). Spoiled patterns: how to extend the gof. *Software Quality Journal*, pages 1--34.

Brin, S., Motwani, R., Ullman, J. D., and Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Record*, volume 26, pages 255--264.

Brown, W. H., Malveau, R. C., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley.

Buschmann, F., Henney, K., and Schimdt, D. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John wiley & sons.

Cardoso, B. and Figueiredo, E. (2014). Co-occurrence of design patterns and bad smells in software systems: A systematic literature review. In *Proceedings of the Workshop on Software Modularity (WMod), co-allocated with CBSoft*, pages 347--354.

Cardoso, B. and Figueiredo, E. (2015). Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In *Proceedings of the Annual Conference on Brazilian Symposium on Information Systems (SBSI)*, pages 347--354.

Carneiro, G. d. F., Silva, M., Mara, L., Figueiredo, E., Sant'Anna, C., Garcia, A., and Mendonca, M. (2010). Identifying code smells with multiple concern views. In *Proceedings of the Brazilian Symposium on Software Engineering*, pages 128--137.

Cline, M. P. (1996). The pros and cons of adopting and applying design patterns in the real world. *Communications of the ACM*, 39(10):47--49.

Copeland, T. (2005). *PMD applied*. Centennial Books San Francisco.

Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., and Shepperd, M. (2004). A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129--143.

Dodani, M. H. (2006). Patterns of anti-patterns. *Journal of Object Technology*, 5(6):29--33.

Dong, J., Zhao, Y., and Peng, T. (2009). A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823--855.

Dorman, C. and Rajlich, V. (2012). Software change in the solo iterative process: An experience report. In *Agile Conference (AGILE)*, pages 21--30.

Du Bois, B., Demeyer, S., Verelst, J., Mens, T., and Temmerman, M. (2006). Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346--355.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5--1.

Fontana, F. A. and Spinelli, S. (2011). Impact of refactoring on quality code evaluation. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 37--40.

Fontana, F. A. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information sciences*, 181(7):1306--1324.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: improving the design of existing code test*. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Gil, J. Y. and Maman, I. (2005). Micro patterns in java code. In *ACM SIGPLAN Notices*, volume 40, pages 97--116.

Guéhéneuc, Y.-G. (2005). Ptidej: Promoting patterns with patterns. In *European Conference on Object Oriented Programming, workshop on Building a System with Patterns, Glasgow, Scotland*.

Jebelean, C., Chirila, C.-B., and Cretu, V. (2010). A logic based approach to locate composite refactoring opportunities in object-oriented code. In *International Conference on Automation Quality and Testing Robotics (AQTR)*, volume 3, pages 1--6.

Kerievsky, J. (2005). *Refactoring to patterns*. Pearson Deutschland GmbH.

Khomh, F. (2009). Squad: Software quality understanding through the analysis of design. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 303--306.

Khomh, F. (2010). Patterns and quality of object-oriented software systems.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE Technical Report EBSE-2007-01.

Krein, J. L., Pratt, L. J., Swenson, A. B., MacLean, A. C., Knutson, C. D., and Eggett, D. L. (2011). Design patterns in software maintenance: An experiment replication at brigham young university. In *2nd International Workshop on Replication in Empirical Software Engineering Research (RESER)*, pages 25--34.

Lee, H., Youn, H., and Lee, E. (2008). A design pattern detection technique that aids reverse engineering. *International Journal of Security and its Applications*, 2(1):1--12.

Li, W. and Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120--1128.

Luo, Y., Hoss, A., and Carver, D. L. (2010). An ontological identification of relationships between anti-patterns and code smells. In *Aerospace Conference*, pages 1--10.

Maggioni, S. (2006). Design pattern clues for creational design patterns. In *Proceedings of the 1st International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE), Benevento, Italy.*

Marinescu, R. (2005). Measurement and quality in object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 701--704.

Marinescu, R. and Lanza, M. (2006). *Object-oriented metrics in practice.* Springer.

Martin, R. C. (1996). The open-closed principle. *More C++ gems*, pages 97--112.

Mc Smith, J. C. and Stotts, D. (2003). Spqr: Flexible automated design pattern extraction from source code. In *In Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 215--224.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308--320.

McNatt, W. B. and Bieman, J. M. (2001). Coupling of design patterns: Common practices and their benefits. In *25th Annual International Conference on Computer Software and Applications Conference (COMPSAC)*, pages 574--579.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20--36.

Munro, M. J. (2005). Product metrics for automatic identification of bad smell design problems in java source-code. In *International Symposium on Software Metrics*, pages 15--15. IEEE.

Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., and Welsh, J. (2002). Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering (CICSE)*, pages 338--348.

Oliveira, J. A., Fernandes, E. M., and Figueiredo, E. (2015). Evaluation of duplicated code detection tools in cross-project context. In *3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pages 49--56.

Paiva, T., Damasceno, A., Padilha, J., Figueiredo, E., and Sant'Anna, C. (2014). Experimental evaluation of code smell detection tools. In *2nd Workshop on Software Visualization, Evolution, and Maintenance (VEM)*.

Pérez, J. and Crespo, Y. (2009). Perspectives on automated correction of bad smells. In *Proceedings of the joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, pages 99--108.

Polasek, I., Liska, P., Kelemen, J., and Lang, J. (2012). On extended similarity scoring and bit-vector algorithms for design smell detection. In *16th International Conference on Intelligent Engineering Systems (INES)*, pages 115--120.

Prechelt, L., Unger, B., Tichy, W. F., Brossler, P., and Votta, L. G. (2001). A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134--1144.

Raibulet, C. and Arcelli, F. (2005). The role of design pattern decomposition in reverse engineering tools. *IEEE International Workshop on Software Technology and Engineering Practice (STEP)*, page 230.

Regulwar, G. B. and Tugnayat, R. M. (2012). Bad smelling concept in software refactoring. *Jawaharlal Darda Institute of Engineering & Technology, MIDC, Lohara, Yavaymal (MS), INDIA*.

Riel, A. J. (1996). *Object-oriented design heuristics*, volume 338. Addison-Wesley Reading.

Seng, O., Stammel, J., and Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of*

the 8th Annual Conference on Genetic and Evolutionary Computation, pages 1909--1916.

Shi, N. and Olsson, R. A. (2005). Reverse engineering of design patterns for high performance computing. In *Workshop on Patterns in High Performance Computing*.

Smith, J. and Stotts, D. (2002). An elemental design pattern catalog. Technical report, Technical Report TR-02-040, Univ. of North Carolina.

Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. (2013). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4.

Trifu, A. and Reupke, U. (2007). Towards automated restructuring of object oriented systems. In *11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 39--48.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329--331.

Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35:347--367.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896--909.

Vale, G., Figueiredo, E., Abilio, R., and Costa, H. (2014). Bad smells in software product lines: A systematic review. In *18th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 84--94.

Vaucher, S., Khomh, F., Moha, N., and Guéhéneuc, Y.-G. (2009). Tracking design smells: Lessons from a study of god classes. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 145--154.

Venners, B. (2005). How to use design patterns. a conversation with erich gamma, part i. *Leading-Edge Java*.

Vokáč, M., Tichy, W., Sjøberg, D. I., Arisholm, E., and Aldrin, M. (2004). A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Software Engineering*, 9(3):149--195.

von Detten, M. and Becker, S. (2011). Combining clustering and pattern detection for the reengineering of component-based software systems. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, pages 23--32.

Welling, R. (2011). A performance analysis on maximal common subgraph algorithms. In *15th Twente Student Conference on IT, Enschede, The Netherlands. University of Twente*, volume 14.

Wendorff, P. (2001). Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *5th European Conference on Software Maintenance and Reengineering*, pages 77--84. IEEE.

Wydaeghe, B., Verschaeve, K., Michiels, B., Van Bamme, I., Arckens, E., and Jonckers, V. (1998). Building an omt-editor using design patterns: An experience report. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 20--32.