View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIS Electronic Library (AISeL)

Association for Information Systems
AIS Electronic Library (AISeL)

Proceedings of the XI Brazilian Symposium on Information Systems (SBSI 2015)

Brazilian Symposium on Information Systems (SBIS)

5-2015

Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study

Bruno dos Santos Azevedo Cardoso
Federal University of Minas Gerais (UFMG), brunosac@dcc.ufmg.br

Eduardo Figueiredo
Federal University of Minas Gerais (UFMG), figueiredo@dcc.ufmg.br

Follow this and additional works at: http://aisel.aisnet.org/sbis2015

Recommended Citation
Cardoso, Bruno dos Santos Azevedo and Figueiredo, Eduardo, "Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study" (2015). *Proceedings of the XI Brazilian Symposium on Information Systems (SBSI 2015)*. 55.
http://aisel.aisnet.org/sbis2015/55

This material is brought to you by the Brazilian Symposium on Information Systems (SBIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in Proceedings of the XI Brazilian Symposium on Information Systems (SBSI 2015) by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study

Bruno Cardoso
Software Engineering Lab (LabSoft)
Federal University of Minas Gerais (UFMG)
Belo Horizonte, MG - Brazil
brunosac@dcc.ufmg.br

Eduardo Figueiredo
Software Engineering Lab (LabSoft)
Federal University of Minas Gerais (UFMG)
Belo Horizonte, MG - Brazil
figueiredo@dcc.ufmg.br

ABSTRACT
A design pattern is a general reusable solution to a recurring problem in software design. Bad smells are symptoms that may indicate something wrong in the system design or code. Therefore, design patterns and bad smells represent antagonistic structures. They are subject of recurring research and typically appear in software systems. Although design patterns represent good design, their use is often inadequate because their implementation is not always trivial or they may be unnecessarily employed. The inadequate use of design patterns may lead to a bad smell. Therefore, this paper performs an exploratory study in order to identify instances of co-occurrences of design patterns and bad smells. This study is performed over five systems and discovers some co-occurrences between design patterns and bad smells. For instance, we observed the co-occurrences of Command with God Class and Template Method with Duplicated Code. The results of this study make it possible to understand in which situations design patterns are misused or overused and establish guidelines for their better use.

Categories and Subject Descriptors
[**Software systems design**]: Design of software systems.

General Terms
Design.

Keywords
Design Patterns, Bad Smells.

1. INTRODUCTION
A design pattern is a general reusable solution to a recurring problem in software design [4]. 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 [4] 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.

On the other hand, 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 [14]. One of the main references on this topic is the Fowler's book [14] which has catalogued 22 bad smells. In this book [14], bad smells are defined as code fragments that need refactoring. Other authors have also contributed to expand the set of bad smells [9] [15]. Kerievsky [9] emphasizes the use of design patterns as a refactoring technique to remove bad smells. Lanza and Marinescu [15] presented a catalog of bad smells called "disharmonies".

In a previous work [1], we performed a systematic literature review in order to understand how studies investigate these two topics, design patterns and bad smells, together. Our results showed that, in general, studies have a narrow view concerning the relation between these concepts. Most of them focus on refactoring opportunities. Among these, some tools [3][7][15][17] have been proposed to identify code fragments that can be refactored to patterns. Other studies [2][6] establish a structural relationship between the terms design pattern and bad smell. 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 [19] [25]. For instance, McNatt and Bieman [25] assessed the positive and negative effects of design patterns on maintainability, factorability, and reusability. Wendorff [19] performed a study and identified some questionable use of design patterns, like Proxy, Observer, Bridge, and Command.

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, we perform in this paper an exploratory study in order to identify instances of co-occurrences between design patterns and bad smells. In this study, we run 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, this co-occurrence has not been deeply investigated in the software engineering literature and there are only a few references concerning this topic [5][18].

Results of this study indicate some correlations between design patterns and bad smells. We rely on association rules [20][21] to indicate how strong a relation 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. Of course, it is necessary to analyze in which situations these associations are due to misuse of Factory Method. Based on detaching values of association rules, we analyzed two situations that called our 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 design patterns and bad smells may co-occur. We aim to establish guidelines for better employment of the GoF patterns.

While this section introduced this study, the rest of this paper is organized as follows. Section 2 details the main concepts of this study and presents the detection tools used. Section 3 explains the settings of this exploratory study by detailing the decisions we made during its execution. Section 4 presents the main results obtained and aims at explaining such results. Section 5 discuss some related work. Section 6 presents some threats to the study

validity while Section 7 summarizes the conclusions obtained and suggests directions for future work.

2. BACKGROUND

This section revisits the concepts of design patterns (Section 2.1) and bad smells (Section 2.3). In addition, it presents the tools used for detect them.

2.1 Design Patterns

Design patterns were presented by Gamma et. al. [4] in 1994 as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" and, 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 proven correct solution.

Design patterns aim at reducing coupling and improving flexibility within software systems. As an example of that, many of these patterns postpone decisions and actions until runtime [12], which makes code more flexible and this way it is easier to add new functionality without deep changing in existing code. They also make it easier to think clearly about a design and encourage the use of "best practices" [12]. Also, by providing a common terminology, design patterns improve communication among designers and developers [11]. By using well-known and well-understood concepts, it eases code readability and the software system design in general becomes better understood and easier to maintain. For instance, a designer is better understood by saying that the Decorator pattern was employed in the system [4] than saying that some extra responsibilities need to be dynamically added to an object. To illustrate these ideas, we briefly explain two design patterns that are common in software systems and used in this study: 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 [4]. In order 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 sets an object of the invoker class, which queues the requests for this command. By the time, this command is executed it is done by an object of the receiver class.

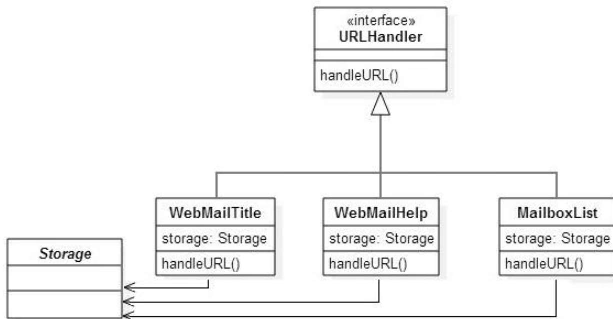

Figure 1. Partial class diagram of the Command pattern in WebMail

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 [4]. It is useful when supporting activities that require the execution of a series of commands, as the orders of customers in a restaurant. This 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 1 shows an example of a real instance of this pattern detected in the WebMail system[2]. In this example, the `URLHandler` is the abstract command and the three classes that inherit from it are the concrete commands. The concrete commands are linked to the `Storage` class, which plays the role of the receiver in 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 pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm structure [4]. 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.

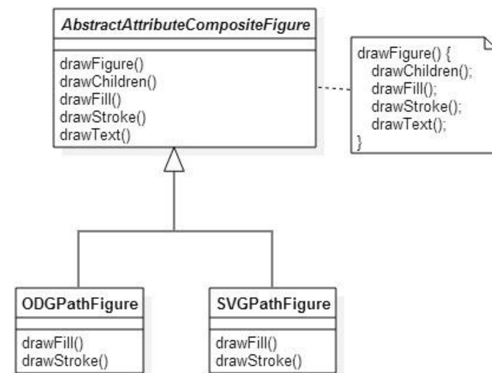

Figure 2. Partial class diagram example of Template Method in JHotDraw

Figure 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[3]. The abstract class `AbstractAttibuteCompositeFigure` defines a drawing template method, called `drawFigure()` that can be modified by the specialized classes. As shown in Figure 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.

[2] http://webmail-beta.locaweb.com.br/

[3] http://www.jhotdraw.org/

2.2 Design Pattern Detection Tool

In order to achieve the goals of this exploratory study, we used the Design Pattern Detection using Similarity Scoring[4] (DPDSS) tool [16], which uses the algorithm called Similarity Scoring. This tool applies the algorithm Similarity Scoring, in which the modeling is performed by using directed graphs that are mapped into square matrices [16].

Table 1 lists the design patterns that can be detected by the used tool, following the same classification used by GoF: by purpose and scope. In Table 1, the Command design pattern is shown in the second column, which lists the structural patterns, despite it is a behavioral one. This 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.

Another relevant issue is the fact that Table 1 displays not only the default structural Proxy design pattern, but also a variant pattern called Proxy 2 [16]. 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 it is an instance of the default Proxy pattern.

Considering the DPDSS tool limitations and also this simplification about the Proxy pattern variation, a total of eleven design patterns could be detected and analyzed in this exploratory study. 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 consuming. 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.

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 [14]. They are defined as code fragments that need refactoring [14]. Although bad smells can harm the development, maintenance, and evolution of the software system [26], but 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 [14]. 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. Some authors treat these classifications interchangeably, while others establish minor differences among them. Another related concept is anti pattern which is understood and defined in different ways in the literature [23] [24]. For example, Dodani [13] considers that if a design pattern has the best solution to solve a problem, then the anti pattern presents the lesson learned.

Fowler [14] catalogued 22 bad smells and indicated appropriate refactoring techniques for each of them. Though this is the main reference of bad smells, there are other catalogues of bad smells in the literature [9] [15]. For instance, Kerievsky [9] 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 [14]. Other authors have proposed a new set of bad smells, such as Lanza and Marinescu [15], 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 [14].

Two common bad smells investigated in this study are God Class and Duplicated Code [14]. God class [15] 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 has a negative impact on the reusability and the understandability of this part of the system. God Class is similar to the Fowler's Large Class bad smell [14]. In this study these terms are used interchangeably. Therefore, a 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 classes the ones that control too much. Often a God Class arises by accident as functionalities are incrementally added to a central class over the course of the system evolution.

Duplicated code [14] is the most pervasive and pungent bad smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet essentially the same.

2.4 Bad Smell Detection Tools

We used two tools for detecting bad smells, in order to identify a larger amount of the bad smells proposed by Fowler. The tools are: JDeodorant[5], which detects four bad smells; and PMD[6], which also detects four bad smells. JDeodorant [17] 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. This tool applies automatic refactoring, since it provides suggestions to solve the bad smells that it detects. PMD [22] is also a plug-in installed on the Eclipse IDE. It performs detection through the use of simple metrics such as number of

Table 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

[4] http://java.uom.gr/~nikos/pattern-detection.html

[5] http://www.jdeodorant.com/

[6] http://pmd.sourceforge.net/

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.

Table 2 lists the bad smells detected by JDeodorant and PMD. In this table, only the bad smells that are detected by at least one of the tools are shown. This way, as it can be observed in Table 2, a total of six bad smells, out of 22 listed by Fowler, are detected combining both tools.

We choose these tools because they are both Eclipse plug-ins, which make them easy to install and configure. Besides that, they are well-known both in industry and in the academia and they are effective in terms of time. We also choose two tools because since detection tools present a high rate of failure, 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 definition makes the results for these two bad smells more reliable.

Table 2. Bad Smells detected by each tool

	Tool	
Bad Smell	***JDeodorant***	***PMD***
Duplicatedd Code	-	X
Long Method	X	X
Large/God Class	X	X
Long Parameter List	-	X
Feature Envy	X	-
Switch Statements	X	-
Total	**4**	**4**

3. STUDY SETTINGS

This section presents the settings of the exploratory study, which investigates the co-occurrence of design patterns and bad smells. Here we present the systems we used to collect the data to be analyzed, the association rules, which are used as a starting point to understand the results and the general procedures of this study.

3.1 Research Questions

In this study we aim at answering the following research questions:

RQ1: Do design patterns may co-occur with bad smells? If so, which patterns are these?
RQ2: If there are co-occurrences between design patterns and bad smells, do they happen due to the overuse or the misuse of design patterns?

3.2 Target Systems

We use a design pattern detection tool and two bad smells detection tools to analyze twelve systems available on the qualitas.corpus[7] dataset. However, seven out of these twelve systems did not present relevant amount of design patterns or bad smells. Hence, they could not be used in the analysis of this study. The five remaining systems that were used are listed in Table 3.

Table 3 presents some information about the five systems used in this study. They are ordered according to the number of lines of code. Besides the name and version of the systems, Table 3 presents some size metrics. As can be seen in this table, AspectJ is the one 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 size software systems.

Table 3. Target-systems

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

3.3 Association Rules

An important concept that is used to analyze the results of this study is association rules [20][21]. These 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. This way, four measures were calculated in order to understand the results: Support [20], Confidence [20], Lift [21] and Conviction [21]. 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.

The Support of an itemset is the proportion of transactions which contains this itemset, showing its importance and significance [20]. 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, the 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, the Support is a measure of the frequency of an item in an association.

In order to understand the concept of Confidence [20] 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 the Support of the association and the Support of the antecedent. The 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 the Support of the association is also high.

$$(1) \qquad \mathrm{Conf}\,(\mathrm{A} \rightarrow \mathrm{B}) = \mathrm{Sup}\,(\mathrm{A} \cup \mathrm{B}) \div \mathrm{Sup}(\mathrm{A})$$

Lift measures how many times more often a design pattern and a bad smell occur together than expected if they where statistically independent [21]. 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.

$$(2) \qquad \mathrm{Lift}\,(\mathrm{A} \rightarrow \mathrm{B}) = \mathrm{Conf}\,(\mathrm{A} \rightarrow \mathrm{B}) \div \mathrm{Sup}(\mathrm{B})$$

Conviction is an alternative to Confidence since the latter was found to not capture direction of associations adequately. Conviction is calculated by the formula below [21]. Conviction presents three interesting characteristics: (i) it considers both the Support of the antecedent and the Support of the consequent; (ii)

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

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.

$$\text{Conv}\,(A \rightarrow B) = \frac{\text{Sup}(A) * (1 - \text{Sup}(B))}{\text{Sup}(A) - \text{Sup}(A \cup B)} \quad (3)$$

3.4 Procedures

In order to perform this exploratory study, we followed a set of procedures, divided in three general phases. Figure 3 shows the flow of these phases, which are Selection and Tuning, Execution and Analysis. After the Execution phase, we persisted data in a relational database in order to make our analysis simpler. The first phase is the Selection and Tuning. It consists in choosing the systems to be analyzed within the study and choosing the detection tools that bring us relevant results. Besides choosing the tools it was necessary to configure them. Therefore, in JDeodorant, we set the Minimum Number of Statements parameter to 3 and, in PMD, we set the value of Parameter List to 10 and Method Length to 100. The other parameters of these tools remained with the default values. This phase is very important since the final results of the study depend of these choices.

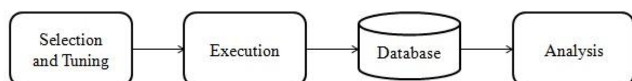

Figure 3. Procedures flow

The second phase is Execution. The first procedure of this phase is to run 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. Therefore, it was necessary to develop a routine that accept the outputs of each tool as input and save the results in the 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. So far, we have the information of which design patterns and which bad smells each class of the five systems have. This information makes it possible to calculate association rules for each of the 66 associations. This 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 a 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.

4. RESULTS

As already stated, five investigated systems present relevant results and were used in this exploratory study. As explained before, for the bad smells God Class and Long Method, it is considered an occurrence of them if both detection tools detected them in order to make the results more reliable. This definition reduced significantly the quantity of God Classes detected in the five systems and totally eliminated the occurrences of Long Methods, since the tools find different results for this bad smell.

It is also important to detach that the Support value of any of these items is not 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. Therefore, 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. We focus on results for Conviction measure between 1.01 and 5, since values that are higher than 5 are usually obvious. However, considering the context of this exploratory study, the upper threshold is not really relevant, since an association between a design pattern and a bad smell is rare.

This way, after calculating the 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.

4.1 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 shows the values for the {Adapter/Command → God Class} association rule in the analyzed systems. The five systems present instances of either Adapter or Command. Three out of these five systems present the God Class bad smell in one or more classes that participate in one of these two patterns. Just Velocity does not present any God Class. Although Hibernate presents this smell, no God Class co-occurs with a class that participates of an Adapter or Command pattern. 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. 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 this pattern within the system evolution could cause God Class. For instance, Figure 4 shows a Class Diagram of an instance of the Command pattern identified in the WebMail system. Part of this diagram is shown in Figure 1 (Section 2.1). The `Storage` class plays the role of a receiver in this pattern and it is associated to 14 concrete commands. The concrete commands are the classes that appear in

the central part of the diagram in Figure 4. The base command is the `URLHandler` class and the abstract method is `handleURL()`.

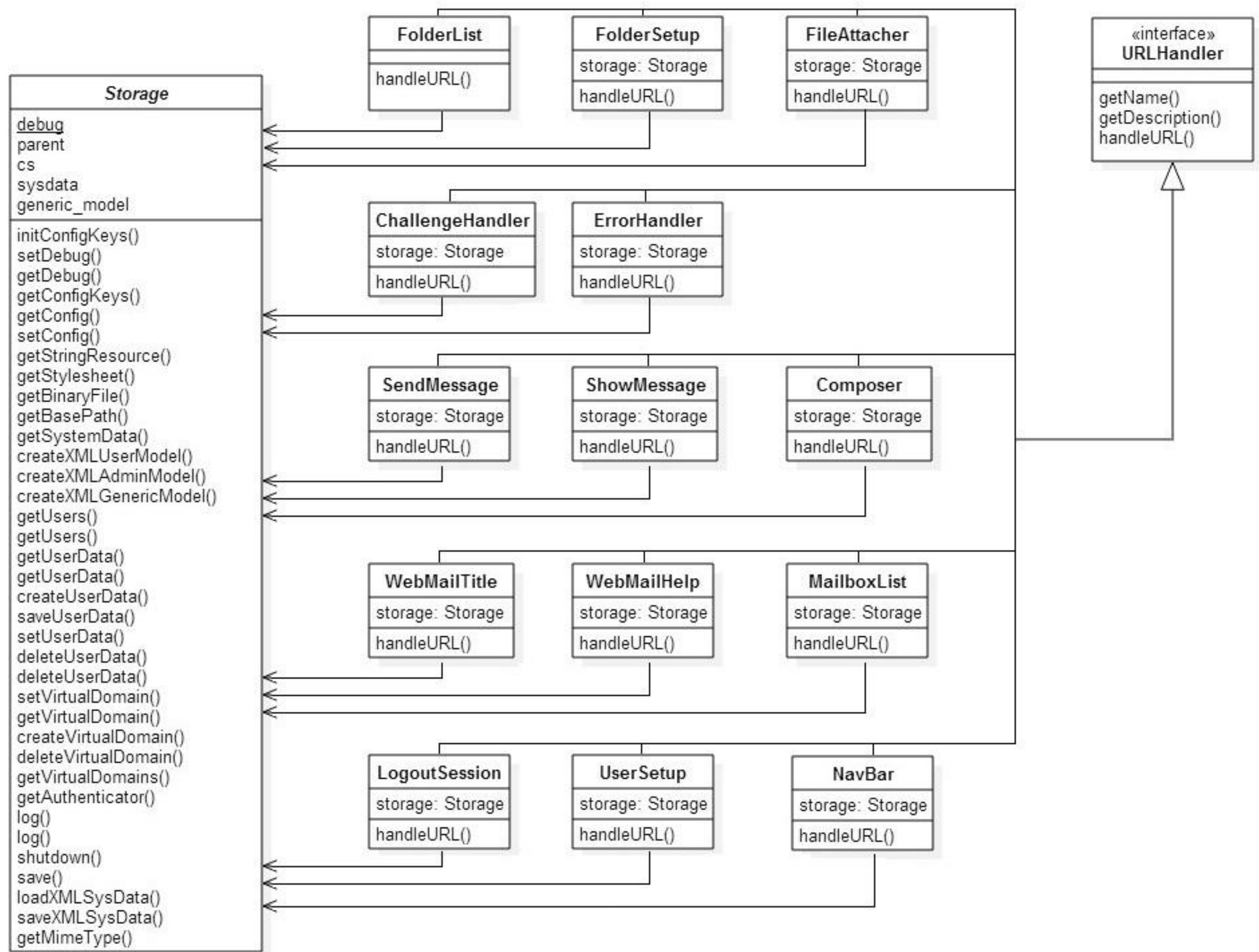

Figure 4. Example of co-occurrence of Command and God Class

By analyzing the diagram in Figure 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, there are many methods related with XML, debugging, configurations, authentication, user profile, etc.

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 the 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, many pattern instances would avoid that a class (i.e., `Storage`) plays the role of a God receiver does too much. Therefore, methods of the `Storage` class should be extracted into other classes, which is a refactoring recommendation for most God Class [14].

4.2 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 goal of the Template Method design pattern is to avoid redundancy and, therefore, to avoid duplicated code. Table 5 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 5. 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 5 shows the Class Diagram of one instance of the Template Method pattern

identified in the JHotDraw system. Figure 2 (Section 2.1) shows another instance of the same pattern. The two abstract classes in Figure 2 and 5 have both a template method, 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.

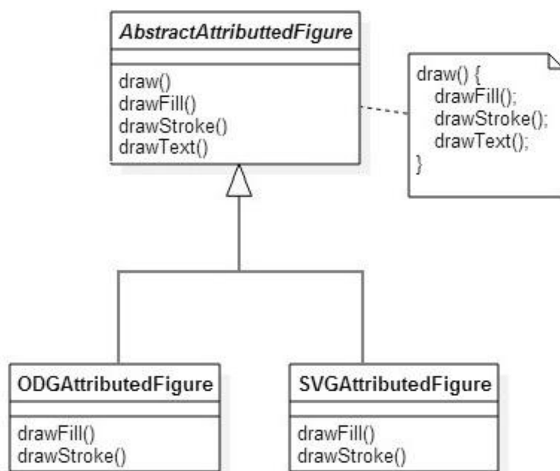

Figure 5. Instance of Template Method and Duplicated Code co-occurrence

5. RELATED WORK

Although design patterns [4] and bad smells [14] are recurrently target of research studies in software engineering [7] [8], a few studies portraits the idea of co-occurrence of these concepts. In fact, the most well-known and exploited relationship between these topics is the use of refactoring techniques in order to change a bad situation (i.e., bad smells) into a nice design solution (i.e., design patterns) [9]. In a previous study, we performed a systematic literature review [1] in order to understand how studies relate design patterns and bad smells. Our results showed that, in general, studies have a narrow view concerning the relation between these concepts. This review found a total of ten studies. As expected, most of these studies are related to refactoring to design patterns, whereas others establish a structural comparison concerning these topics. Only three of the studies found in the systematic review mention the co-occurrence of design patterns and bad smells [5][10][18], bu they do not focus on this topic.

Although this systematic review did not find studies focused on the co-occurrence of design patterns and bad smells, there are reported cases in the software engineering literature where the use of design patterns may not always be the best option [19][25]. In general, we tend to think that if developers employ cataloged design patterns [4], then hypothetically the best solution to solve the problem has been used [2] and the best practices are been employed. However, this thought is not necessarily true since a design pattern may be misused or even overused. This wrong employment of a design pattern can even introduce bad smells in code. For instance, McNatt and Bieman [25] qualitatively assess the coupling of pattern in terms of their effects on maintainability, factorability, and reusability. The authors point out that this coupling may provide benefits, but also costs to the system.

Wendorff [19] presents a paper in which design patterns are assessed during the reengineering phase. The author shows examples of the questionable use of some patterns, like Proxy, Observer, Bridge, and Command. The Proxy pattern is considered a simple one and, therefore, it is widely employed by beginners, who tend to use them freely and inadvertently [19]. One of the problems noticed by the author is that Proxy pattern is frequently used based on the expectation of future needs for flexibility, access control, and performance that never materialize. The author [19] also states that some instances of Proxy make the interaction of objects complicated and that this pattern usage naturally leads to a substantial increase in number of classes and consequently, the size and complexity of the software grow considerably.

The Observer pattern use is justified when it is necessary to achieve flexibility and reusability. Wendorff [19] presents situations in which it was not necessary to employ this pattern to achieve these qualities since the analyzed software system was already too simple. Therefore, in this situation, the developer implements a complex and expensive functionality in order to achieve nonexistent requirements.

We believe that these questionable uses of design patterns happen due to two main reasons. First, excess of engineering, when design patterns are employed unnecessarily, making the code bigger, more complex and more expensive and also decreasing reusability, maintainability and flexibility, which are exactly the opposite intent of the GoF patterns. Second, the complexity of employing a design pattern properly, since they are not trivial structures and Gamma *et.* al. [4] only provides sample code, which are much simpler than the industrial software systems.

6. THREATS TO VALIDITY

The fact that this study was not performed in a full controlled environment may introduce the main threats to external validity. The study was performed using five systems, three detection tools and the results were analyzed by authors.

We analyzed five systems because seven out of the twelve systems we chose do 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 have already a lot of information with the five ones we investigated. Besides that, we concluded that the size of the systems may not be a restriction since, in some cases, we had more interesting results with the small systems.

Concerning the detection tools, we chose DPDSS, JDeodorant and PMD mainly because of their facility to use and their acknowledgment in academia. As already explained, the bad smells that are detected by both JDeodorant and PMD - God Class and Long Method - were only accepted when both tools detected the same instances. 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 that the detection tools indicate co-occurrences between a design pattern and bad smell. Although only two researchers analyzed the results of this study, all settings of the study - including systems, tools

information and how we derived the values for the association rules - are well documented, which turns this study replicable.

7. CONCLUSIONS AND FUTURE WORK

Within this study, we ran a design patterns detection tool and two bad smells detection tools in five systems. We identified some samples 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 the design pattern Command with the bad smell God Class and the pattern Template Method with the bad smell Duplicated Code. We showed 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 Template Method instead of eliminating unnecessary repetition, presented many duplications.

We used the values for association rules, especially Support and Conviction, as a starting point to derive our analysis. These association rules were used to analyze how strong a relation between a specific design pattern and a bad smell is. This way only after calculating these measures, 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 the used association rules, the final conclusions of this study were derived from our deep analysis of the source code.

The results of this study showed that it is completely possible that the inappropriate employment of a design pattern leads to the arising of bad smells, although this consequence seems totally aimless. The next step of this work is to analyze other design patterns that, due to its misuse, have classes that present bad smells. In future work, we intend to replicate this study in an enterprise development context, which may provide more data and results with higher statistical significance. We also aim at defining guidelines that would help developers in the task of maintenance of classes that are part of a design pattern, since we believe that bad smells does not arises as a consequence of design patterns misuse in the first moment.

8. ACKNOWLEDGEMENTS

This work was partially supported by CNPq (grant Universal 485907/2013-5) and FAPEMIG (grant PPM-00382-14).

9. REFERENCES

[1] B. Cardoso and E. Figueiredo. Co-Occurrence of Design Patterns and Bad Smells in Software Systems: A Systematic Literature Review. In proc. of the Workshop on Software Modularity, 2014

[2] C. Bouhours, H. Leblanc, and C. Percebois. Sharing bad practices in design to improve the use of patterns. In proc. of the Conference on Pattern Languages of Programs, 2010.

[3] C. Jebelean, C. Chirila, and V. Cretu. A Logic Based Approach to Locate Composite Refactoring Opportunities in Object-Oriented Code. In Int'l Conf. on Automation Quality and Testing Robotics, vol. 3, p. 1–6, 2010

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, 1994.

[5] F. Fontana and S. Spinelli. Impact of Refactoring on Quality Code Evaluation. In proc. of the 4th Workshop on Refactoring Tools, p. 37–40, 2011.

[6] F. Khomh. Squad: Software Quality Understanding Through the Analysis of Design. In 16th Working Conference on Reverse Engineering, p. 303– 306, 2009.

[7] G. Carneiro et al. Identifying Code Smells with Multiple Concern Views. In Brazilian Symposium on Software Engineering, p. 128-137, 2010.

[8] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic. Identifying Architectural Bad Smells. European Conf. on Software Maintenance and Reeng, 2009.

[9] J. Kerievsky. Refactoring to Patterns. Pearson, 2005.

[10] J. Perez and Y. Crespo. Perspectives on Automated Correction of Bad Smells. In proc. of the Int'l Workshop on Principles of Software Evolution, p. 99–108, 2009.

[11] K. Beck, et al. Industrial Experience with Design Patterns, In Proc. Conf. Software Eng. p. 103-114, 1996.

[12] L. Prechelt, B. Unger, W. Tichy, P. Brössler, L. Votta. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions, In IEEE Transactions on Software Eng., vol. 27, no. 12, p. 1134-1144, 2001

[13] M. Dodani. Patterns of Anti-Patterns. Journal of Object Technology, p. 29–33, 2006.

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

[15] M. Lanza and R. Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

[16] N. Tsantsalis, A. Chatzigeorgiou, G. Stephanides, S. Halkidis, Design Pattern Detection Using Similarity Scoring. IEEE Trans. on Soft. Engineering, vol. 32, p. 896-909, 2006.

[17] N. Tsantalis and A. Chatzigeorgiou, Identification of Move Method Refactoring Opportunities. IEEE Transactions on Software Engineering, pp 347–367, 2009.

[18] O. Seng, J. Stammel, and D. Burkhart. Search-based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. In proc. of the Conf. on Genetic and Evolutionary Computation, 2006.

[19] P. Wendorff. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In European Conference on Software Maintenance and Reengineering, p. 77–84, 2001.

[20] R. Agrawal, I. Tomasz, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. ACM SIGMOD Record. Vol. 22. No. 2. 1993.

[21] S. Brin et al. Dynamic Itemset Counting and Implication Rules for Market Basket Data. ACM SIGMOD Record. vol. 26, 1997.

[22] T. Copeland, PMD Applied: Centennial Books, 2005.

[23] Y. Luo, A. Hoss, and D. Carver. An Ontological Identification of Relationships between Anti-patterns and Code Smells. In Aerospace Conference, p. 1–10, 2010.

[24] W. Brown, et al. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis, 1998.

[25] W. McNatt, and J. Bieman. Coupling of design patterns: Common practices and their benefits. In Annual International Computer Software and Applications Conference, p. 574–579, 200.

[26] J. Padilha et al. On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study. In Int'l Conf. on Advanced Information Systems Engineering (CAiSE), 2014.