When GOF Design Patterns occur with God Class and Long Method Bad Smells? - An Empirical Analysis

[1]BRUNO L. SOUSA
[1]MARIZA A. S. BIGONHA
[2]KECIA A. M. FERREIRA

[1]UFMG - Federal University of Minas Gerais – Computer Science Department
Antônio Carlos Avenue, 6627, 31270-010 - Belo Horizonte (MG) - Brazil
[2]CEFET-MG - Federal Center for Technological Education of Minas Gerais – Department of Computing
Amazonas Avenue, 5253, 30421-169 – Belo Horizonte (MG) - Brazil
[1] (bruno.luan.sousa,mariza)@dcc.ufmg.br
[2] kecia@decom.cefetmg.br

Abstract. Design patterns are general reusable solutions to common recurring problems in software projects. These solutions, when correctly applied, are supposed to enhance modular and flexible structures in software. The aim of this work is to study the occurrences of God Class and Long Method bad smells in software systems developed with design patterns. To achieve this aim, we carried out an exploratory study with five Java systems to: (i) investigate if design patterns avoid bad smells; (ii) identify design patterns that may have co-occurrence with bad smells; and (iii) extract the main reasons that impact on such co-occurrence. We consider in our analysis eleven of the twenty-three GOF design patterns. We also consider the God Class and Long Method bad smells. The results obtained suggest that Composite and Factory Method have a low co-occurrence with these bad smells, and Template Method and Observer have a high co-occurrence with God Class and Long Method, respectively. In addition, we have identified that the misuse of design patterns and the scattering and crosscutting concerns has contributed to such co-occurrences.

Keywords: design pattern, bad smell, software metric, threshold.

(Received September 15th, 2017 / Accepted March 29th, 2018)

1 Introduction

Design pattern is a general solution to a recurring problem in a given context in the software design [10]. Its main goal is to create flexible and extensible software systems, with a reusable structure and easy maintenance. They are recognized as good programming practice. When applied correctly, they may help reducing bad smells in software, although they have not been proposed for this purpose [20].

Bad smells are symptoms existing in the source code of a program that possibly indicate a more serious problem that requires code refactoring [9]. Code regions that exhibit these symptoms are not considered errors, but they impair software quality and violate Software Engineering principles such as modularity, readability and reuse. Design patterns may be used to remove bad smells [4, 14, 16, 17, 25]. On the other hand, there are studies that identify co-occurrence of design patterns and bad smells [3, 11, 12, 23]. Although design patterns are intended to improve software quality, they do not necessarily avoid bad smells.

This paper presents an exploratory study in order to investigate object-oriented software that apply the design patterns defined by Gamma et al. (GOF catalog) [10]. The main purpose of this study are: (i) to investigate if design patterns avoid bad smells; (ii) to identify

design patterns that may have co-occurrence with bad smells; and (iii) to extract the main reasons that impact on such co-occurrence.

To achieve these goals, we carry out a case study with five Java software systems. These systems are open source and they were extracted from Qualitas.class Corpus [21]. In this case study, we investigate in our analysis the co-occurrences relationship of eleven design patterns described by Gamma et al. [10] with two bad smells: (i) God Class [13] and (ii) Long Method [9].

This paper is an extended version of our previous work [18]. In the current version, we manually inspected source code in which bad smell and design pattern co-occurrence were detected. We present in details the architecture of two examples of such occurrence. From this analysis, we describe the lessons learned, discussing the main reason that lead to design pattern and the bad smells co-occurrence considered.

2 Research Methodology

This study was carried out in six steps: (i) definition of research questions (ii) identification of bad smells detection strategies, (iii) definition of the data set that comprises the software systems considered in the study, (iv) data collection, (v) application of the association rules and (vi) method of the data analysis.

2.1 Rearch Questions

The research questions (RQn) investigated in this paper are the following:

- **RQ1:** Do the design patterns defined in the GOF catalog avoid the occurrence of God Class and Long Method bad smells in software?
- **RQ2:** Which design patterns of GOF catalog presented co-occurrence with the God Class and Long Method bad smells?
- **RQ3:** What are the more common situations in which the God Class and Long Method bad smells appear in software systems that apply GOF design patterns?

2.2 Identification of Bad Smells Detection Strategies

According to Marinescu [15], detection strategy is a formal rule that charatecterizes a specific bad smell. In addition to detection strategies, software metric thresholds can be used to determinate the relationship of a metric with a bad smell and, thus, identify anomalous entities.

In this study, we consider the God Class and Long Method bad smells. God Class is a class that executes too much work and delegate minor details to other classes [13]. Long Method is a method that performs too much work, having many lines, temporary variables and parameters [9]. We choose these bad smells because they are especially problematic to the software maintenance. Besides, they are related to a large amount of information that may turn the software comprehension hard and increase coupling among the methods and among the classes of the system.

The detection strategies used in this study were proposed by Filó et al. [8]. We choose these strategies because they are composed of well known software metrics. Besides that, they were previously evaluated by Filó [6], and no false negative was returned. False positive may be returned, but with a low probability of occurrence. These results suggest that these detection strategies are effective in bad smells detection. The thresholds defined by Filó et al. [8] for a software metric is classified in three ranges: Good, Regular, and Bad. The authors indicated that the Good range is related to low occurrences of bad smells. Therefore, the detection strategies rely on the Regular and Bad ranges. Following, we describe the detection strategies of Filó et al. [8] for God Class and Long Method.

The God Class detection strategy, Figure 1, uses the metrics: Weighted Methods per Class (WMC), Number of Methods (NOM), Number of Attributes (NOF), and Lack of Cohesion of Methods (LCOM).

Figure 2 shows the detection strategy used for Long Method bad smell. It uses the following metrics: Method Lines of Code (MLOC), Nested Block Depth (NBD) and McCabe Cyclomatic Complexity (VG).

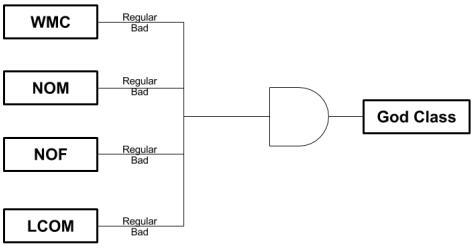

Figure 1: Detection strategy for God Class extracted from [8].

2.3 Data Set

The software sample used in this study are from Qualitas.class Corpus [21], a data set which comprises software metrics of 112 open source software systems developed in Java. Qualitas.class Corpus provides 23 software metrics and the bytecodes of the software systems.

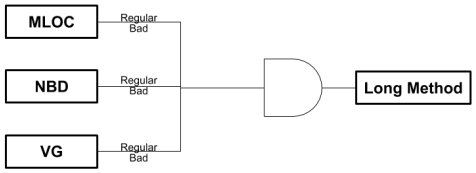

Figure 2: Detection strategy for Long Method extracted from [8].

This data set was chosen because it has a large collection of open source software developed in Java that are widely used in empirical studies of software artifacts.

As this study involves manual inspection, we considered a sample of five software systems: Hibernate 4.2.0, JHotDraw 7.5.1, Kolmafia 17.3, Webmail 0.7.10 and Weka 3.6.9. All of them, except Kolmafia, are from Qualitas Corpus. The main criteria for the selection of these systems was based in two points: (i) they use design patterns from GOF catalog; and (ii) they present the bad smells considered in this study. We decided to include Kolmafia 17.3 in this study because previous studies to point out metrics values considered problematic in this software [5], however, without any correlation of these values with bad smell or design patterns.

2.4 Data Collection

The third step involved the data collection to be analyzed. Qualitas.class Corpus comprises files in XML format with the metrics of the software systems. As most of the software used in this study are from Qualitas.class Corpus, we used these files. Since Kolmafia 17.3 data are not in this corpus. So, we downloaded its code and collected its metrics. We used the Eclipse 4.2 Juno IDE[1] and Metrics 1.3.6 plugin[2] for this purpose.

To verify the design patterns existence in the software, we used Design Pattern Detection using Similarity Scoring 4 (DPDSS) tool [22]. It models all aspects of design patterns by means of directed graphs, represented by quadratic matrices, and applies an algorithm called Similarity Scoring. This algorithm receives as input the system and the graph of the design pattern, and then, calculates the similarity scores between the vertices of the graph. The main advantage of this approach is the ability to detect not only the patterns in their base form, which is normally found in the literature, but also the modified versions of it [22]. We previously tested DPDSS with three systems: JHotDraw 5.1, JRefactory 2.6.24 and JUnit 3.7, and no false positive occurrence of design patterns was returned. False negatives were returned only for two design patterns: Factory Method and State. The results presented by this tool were very satisfactory, showing that it is effective in identifying instances of design patterns.

Filó et al. [7] developed a tool, RAFTool, which performs the identification of methods, classes and packages with anomalous measurements of object-oriented software metrics. We used RAFTool with the purpose of implementing detection strategies. The tool receives as entry the XML file with software metrics of the target system and a detection strategy that is described by a logical expression in a given format. The tool reports the classes or the methods whose metric values fit to the detection strategy.

In RAFTool, the metrics' thresholds that comprise the detection strategy are represented by the following keywords: COMMON, which corresponds to the Good/Frequent ranges of the metrics, CASUAL, which corresponds to the REGULAR/OCCASIONAL ranges of the metrics, and UNCOMMON, which corresponds the Bad/Rare ranges of the metrics. The God Class and Long Method logical expression are defined as follows.

Exp1 (UNCOMMON[WMC] OR CASUAL[WMC]) AND (UNCOMMON[NOF] OR CASUAL[NOF]) AND (UNCOMMON[NOM] OR CASUAL[NOM]) AND (UNCOMMON[LCOM] OR CASUAL[LCOM])

Exp2 (UNCOMMON[MLOC] OR CASUAL[MLOC]) AND (UNCOMMON[NBD] OR CASUAL[NBD]) AND (UNCOMMON[VG] OR CASUAL[VG])

The design patterns instances returned by DPDSS may be composed of one or more classes or methods. For instance, the returned instance to the Bridge design pattern has a responsible class for representing the implementation part and a responsible class for representing the abstraction part. To solve this problem, we developed the Design Pattern Smell[3] [19] to count the classes and methods in the design pattern instances, as well as to identify the components that have a given design pattern and a given bad smell. This tool receives as entry (1) the XML files exported by DPDSS, containing the design patterns instances of a system, and (2) the CSV files generated by RafTool, containing the components with a given bad smell.

2.5 Application of Association Rules

To identify the design pattern and bad smells co-occurrences, we applied association rules, based on

[1] http://www.eclipse.org/downloads/packages/release/Juno/SR2

[2] http://metrics.sourceforge.net

[3] http://www2.dcc.ufmg.br/laboratorios/llp/Products/indexProducts.html

concepts of data mining [1, 2]. We decided to use the association rules because they combine items from a data set to extract knowledge about the data. Moreover, previous works in the same context of this one have applied association rules [3, 23].

To apply the association rules, three metrics are used: Support [1], Confidence [1], and Conviction [2]. These metrics are based in the following main concepts: *Transaction*, defined as a set of items; *Antecedent* that is an item that appears on the left side of a association rule; and *Consequent*, an item that appears on the right side of a association rule. Therefore, a basic association rule has the following form: *Antecedent* $\Rightarrow$ *Consequent*.

Support (sup) on an association rule corresponds to the frequency that an item occurs in a transaction (Equation 1). For instance, let us consider a shopping base in a supermarket. Suppose that there is a data set with 1,000 transactions, which are the set of items that were purchased. In this data set, the items `pasta` and `tomato` appear together in 100 records. So, Support for this relationship is 0.1, i.e., 10.0%.

$$sup(X \Rightarrow Y) = P(X, Y) \tag{1}$$

Confidence (conf) expresses the probability of a Consequent occurs since Antecedent has occurred (Equation 2).

$$conf(X \Rightarrow Y) = \frac{sup(X \Rightarrow Y)}{sup(X)} \tag{2}$$

In the aforementioned example, let us consider that the item `pasta` is found alone in 200 of 1,000 transactions of the data set. To compute the Confidence of the `pasta` $\Rightarrow$ `tomato` association rule, it is necessary to divide the Support of this rule, 0.1, by the Support of `pasta` – Antecedent in the association rule –, 0.2, resulting in a confidence of 0.5, i.e., 50.0%.

Brin et al. [2] proposed the metric Conviction. This metric uses the Support in both the Antecedent and the Consequent (Equation 3).

$$conv(X \Rightarrow Y) = \frac{sup(X) * (1 - sup(Y))}{sup(X) - sup(X \Rightarrow Y)} \tag{3}$$

In the given example, let us consider that the item `tomato` is found alone in 300 of 1,000 transactions of the data set. Thus, the support `tomato`, $sup(tomato)$ is 0.3 and the confidence conf(`pasta` $\Rightarrow$ `tomato`) is 0.5. Applying these values in the Equation 3, the Conviction conv(`pasta` $\Rightarrow$ `tomato`) is 1.4. When the value of Conviction is 1.0, it indicates that the antecedent and the consequent have no relation at all. When the value of Conviction value is less than 1.0, it indicates that if the antecedent occurs, the consequent tends to not occur. When the value of the Conviction is greater than 1.0, it means that the antecedent and the consequent have relation; the greater the value of Conviction, the greater the relation between the antecedent and the consequent. An infinite result indicates that the antecedent never appears in the transactions.

In this study, for the association rules application, we consider a transaction being each class in the analyzed system; antecedent being a design pattern; consequent being a bad smell, in particular, the God Class and Long Method bad smells.

2.6 Method of the Data Analysis

Figure 3 illustrates, via diagram, the method used to analyze the data obtained in the study.

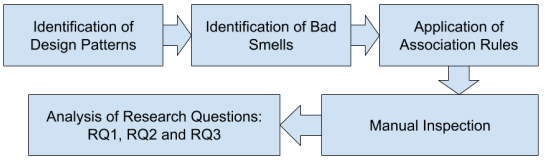

Figure 3: Method of the data analysis.

This step is made in five parts. Initially, we identify the design patterns in the software systems. We used DPDSS tool to identify them and stored the results obtained in a table.

The second part aims to identify classes and methods that have the God Class or Long Method bad smells. To find them, we applied the logical expressions **Exp1** and **Exp2** in RAFTool. After to identify the design pattern instances and God Class and Long Method bad smells, in sequel, we performed the pre-processing of these data by running the Design Pattern Smell tool.

In the third part, we apply the association rules on the data resulting from the pre-processing, and then, we identify the co-occurrences existing in the systems.

In the fourth part, we manually inspected the classes with co-occurrence to identify the situations that contributed the emergence of this relation in such classes.

Finally, we analyzed the data to answer the research questions proposed.

3 Results

This section presents the results followed by the discussion of the presented study. We also answer, in this section, the proposed research questions.

Tables 1 and 2 show the amount of classes and methods of the software systems with the God Class and Long Method bad smells, respectively.

Table 1: Amount of classes with God Class.

Software	# Classes with God Class	# Classes	% Classes with God Class
Hibernate	527	7,711	6.83%
JHotDraw	122	1,061	11.50%
Kolmafia	385	3,225	11.94%
Webmail	15	129	11.63%
Weka	467	2,401	19.45%

Table 2: Amount of methods with Long Method.

Software	# Methods with Long Method	# Methods	% Methods with Long Method
Hibernate	2,883	48,234	5.98%
JhotDraw	995	7,633	13.04%
Kolmafia	5,400	2,8078	19.23%
Webmail	131	1,091	12.01%
Weka	3,822	20,871	18.31%

Tables 3 and 4 show the results after the data pre-processing. In both tables, the "T" column indicates the total number of classes or method that uses a design pattern, and the "DP&BS" column indicates the number of classes that have instances of some design pattern (DP) and occurrence of the respective bad smell (BS).

After the data pre-processing, we performed the association rule application to identify the co-occurrences. For this purpose, we calculated the metrics described in Section 2.5 using the Tables 1, 2, 3, and 4 results. We used the Conviction metric result following the thresholds shown in Section 2.5. Figure 4 exhibits the Conviction metric results for the God Class bad smell, considering the *Design Pattern* $\Rightarrow$ *God Class* association rule. Figure 5 shows the results of the same metric for Long Method bad smell, according to the *Design Pattern* $\Rightarrow$ *Long Method* association rule.

3.1 Analysis of the Results

The classes with the God Class or Long Method bad smells were inspected manually to answer the research questions defined in this paper.

RQ1. Do the design patterns defined in the GOF catalog avoid the occurrence of God Class and Long Method bad smells in software?

The results reported in Table 3 indicate two design patterns with a low God Class occurrence: Composite and Factory Method. The manual inspection reveals that they have a modular structure and divides the tasks between several classes. The idea of the Composite design pattern is to build complex objects via simpler objects. These simpler objects are defined in modules, in

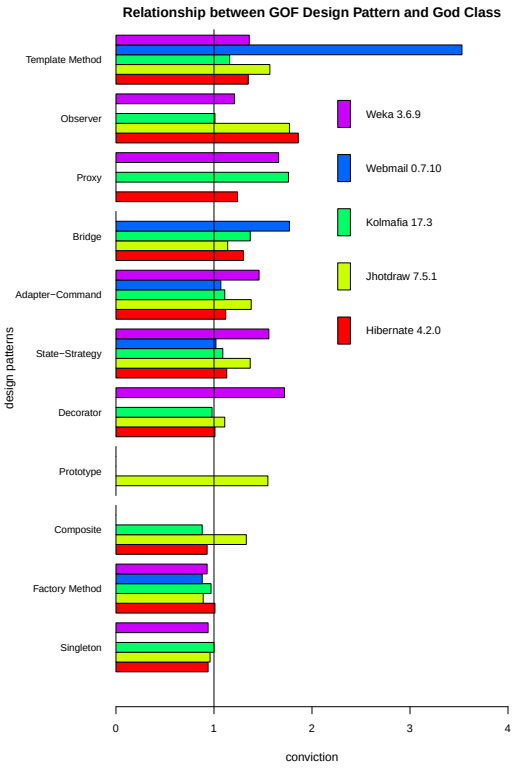

Figure 4: Results of the association rule *Design Pattern* $\Rightarrow$ *God Class*.

such a way intelligence is divided between them, reducing the complexity of the classes. The Factory Method design pattern simulates the idea of a factory in which there is an interface to create objects, but the object creation itself occurs in the class that implements this interface. Thus, it is possible to create several modules, each one responsible for creating and managing the information of a objects set, removing the workload from a single class. Therefore, both Composite and Factory Method are design patterns intrinsically modular.

We observed a similar behavior for the Long Method bad smell, as shown in Table 4. Most design patterns present a high amount of Long Method occurrences, except Composite and Factory Method, which present just a few occurrences of this bad smell.

The Singleton design pattern is a special case. Although its instances do not have the Long Method bad smell, it seems a false negative case. Singleton was indicated as false negative because its instances identified by DPDSS are based only on the static attribute presented in the class. It does not consider methods as characteristic of this design pattern. Thus, when this design pattern was intersected with methods that had bad smells, it returned 0. However, when classes containing

Table 3: Amount of classes that comprise each design pattern and amount of classes that contain both design pattern and the bad smell God Class.

Design Pattern	Hibernate 4.2.0		JHotDraw 7.5.1		Kolmafia 17.3		Webmail 0.7.10		Weka 3.6.9	
	T	DP&BS	T	DP&BS	T	DP&BS	T	DP&BS	T	DP&BS
Adapter-Command	228	39	53	19	386	81	40	7	152	68
Bridge	56	16	40	9	14	5	6	3	0	0
Composite	12	0	12	4	8	0	0	0	0	0
Decorator	37	3	10	2	67	7	0	0	32	17
Factory Method	37	3	5	0	31	3	2	0	22	3
Observer	4	2	2	1	8	1	0	0	36	12
Prototype	0	0	21	9	0	0	0	0	0	0
Proxy	8	2	0	0	18	9	0	0	35	18
Singleton	232	3	13	1	77	9	1	1	34	5
State-Strategy	271	47	121	43	334	64	23	3	93	45
Template Method	87	27	16	7	54	13	4	3	22	9

Table 4: Amount of methods that comprise each design pattern and amount of methods that contain both design pattern and the bad smell Long Method.

Design Pattern	Hibernate 4.2.0		JHotDraw 7.5.1		Kolmafia 17.3		Webmail 0.7.10		Weka 3.6.9	
	T	DP&BS	T	DP&BS	T	DP&BS	T	DP&BS	T	DP&BS
Adapter-Command	271	52	73	19	703	123	50	5	222	71
Bridge	61	13	51	11	19	3	8	2	0	0
Composite	8	0	29	0	37	0	0	0	0	0
Decorator	115	2	31	1	255	12	0	0	61	25
Factory Method	58	0	23	0	45	0	2	0	27	0
Observer	8	3	2	1	7	3	0	0	24	4
Prototype	0	0	16	6	0	0	0	0	0	0
Proxy	6	1	0	0	31	9	0	0	37	12
Singleton	340	0	15	0	672	0	1	0	83	0
State-Strategy	343	121	227	90	974	154	19	0	173	97
Template Method	275	90	47	13	161	48	14	3	34	16

Singleton were inspected, we identified methods with the Long Method bad smell inside them.

Summary. Although Factory Method and Composite design patterns present few co-occurrences with the God Class and Long Method bad smells, the general conclusion of this analysis is that most of the design patterns studied in this paper are associated with this two bad smells. Therefore, the answer of RQ1 is "No, design patterns GOF not necessarily avoid occurrences of the God Class and Long Method bad smells."

RQ2. Which design patterns of GOF catalog presented co-occurrence with the God Class and Long Method bad smells?

To analyze the association between design patterns and bad smells, we considered the values of Conviction. The choice of this metric is due to the fact that Conviction is able to establish a relationship between Support and Confidence metrics. Moreover, Conviction has a better sensitivity to the direction between the antecedent and the consequent. In order to define which design patterns and bad smells co-occurrences have the highest relation, we considered the Conviction thresholds, mentioned in Section 2.5.

The chart in Figure 4 indicates a strong relationship of several design patterns with the bad smell God Class. The design patterns with the highest Conviction values are: Template Method, Observer, and Proxy. We also observed that the results for those design patterns in four systems are very close: Hibernate 4.2.0, JHotdraw 7.5.1, Kolmafia 17.3, and Weka 3.6 .9. In Webmail 0.7.10, the association rule *Template Method* $\Rightarrow$ *God Class* prevailed over the others, showing that the Template Method was the design pattern that pre-

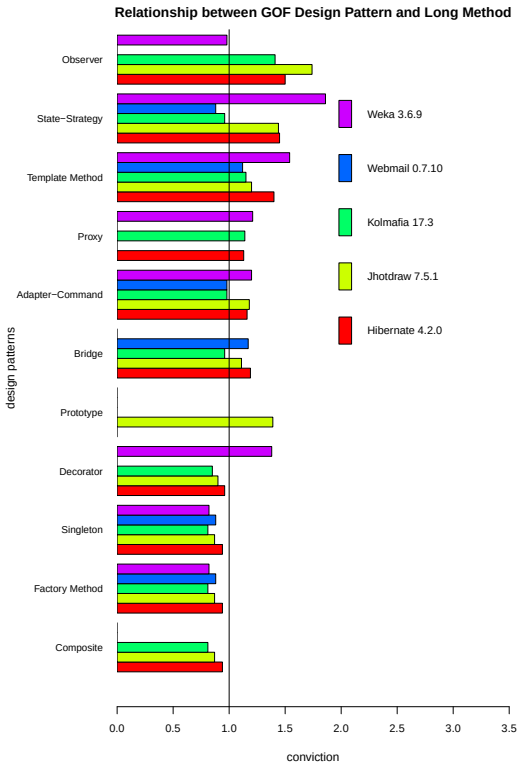

Figure 5: Results of the association rule *Design Pattern* ⇒ *Long Method*.

sented more co-occurrences with God Class. These results, then, indicate that Template Method, Observer, and Proxy presented more occurrences of the God Class bad smell than the other design patterns.

Long Method has a similar behavior. The chart in Figure 5 shows that many design patterns have a strong relation with Long Method. Three design patterns present a higher co-occurrence with Long Method: Observer, State-Strategy, and Template Method. Conviction is higher in 3 out of 5 systems: Hibernate 4.2.0, JHotDraw 7.5.1, and Kolmafia 17.3. The strongest association rule in these systems is *Observer* ⇒ *Long Method*. Webmail 0.7.10 does not present any Observer instance, therefore the *Observer* ⇒ *Long Method* association rule was not found in this project. In Weka 3.6.9, *Observer* ⇒ *Long Method* has a low occurrence. In conclusion, the *Observer* ⇒ *Long Method* association rule, despite having a low occurrence in two cases, was the one that presented the strongest Conviction in this paper. Nevertheless, State-Strategy and Template Method also present a high Conviction value. Then, the results indicate that Observer, State-Strategy, and Template Method presented more occurrences of the Long Method bad smell than the other design patterns.

Summary. So, answering RQ2, we identified Template Method, Observer and Proxy as those which presented the main co-occurrences with the God Class bad smell. Among them, Template Method design pattern was those which presented the highest co-occurrence with this bad smell. We also identified the Observer, State-Strategy and Template Method design patterns as those that presented the main co-occurrences with the Long Method bad smell. Nevertheless, Observer presented the highest co-occurrence with Long Method.

RQ3. What are the more common situations in which the God Class and Long Method bad smells appear in software systems that apply GOF design patterns?

The results indicate that God Class bad smell has more co-occurrences with the Template Method design pattern, and the Long Method bad smell has more co-occurrences with the Observer design pattern. To identify the causes of such co-occurrences, we performed a manual inspection in the classes with the *Template Method* ⇒ *God Class* and *Observer* ⇒ *Long Method* co-occurrences. Tables 5 and 6 show the computed metrics results for these two co-occurrences.

Table 5: Metrics results for the *Template Method* ⇒ *God Class*.

Software	Support (%)	Confidence (%)	Conviction
Hibernate	0.35	31.03	1,35
JHotDraw	0.66	43.75	1,57
Komafia	0.40	24.07	1,16
Webmail	2.33	75.00	3,53
Weka	0.37	40.91	1,36

Table 6: Metrics results for the *Observer* ⇒ *Long Method*.

Software	Support (%)	Confidence (%)	Conviction
Hibernate	0.01	37.50	1.50
JHotDraw	0.01	50.00	1.74
Komafia	0.01	42.86	1.41
Webmail	0.00	∞	∞
Weka	0.02	16.67	0.98

3.1.1 Template Method ⇒ God Class

Template Method aims to define the skeleton of an algorithm via an operation, transferring some steps to the subclasses, which have the power to redefine the characteristics of the algorithm without changing the algorithm structure [10]. That is, this pattern uses a modular structure, in which the various behaviors of an object are modeled in the subclasses and assigned to the

object via polymorphism. The advantage of this implementation is the reduction of complexity in the super class, since definitions via conditional structures such as if, else, and switches are replaced by polymorphism. However, using this pattern requires careful attention to avoid assigning many responsibilities to the templates and also to the super class.

The manual inspection in the classes that presented *Template Method* ⇒ *God Class* co-occurrence indicated that a great amount of responsibilities were assigned to the classes containing the template method. We observed that the templates methods refer to the definition of the object behavior, implemented in the subclasses. In some classes, the behavior implementation has a high complexity. This generates task overload in the templates methods, contributing with the Long Method occurrence in some cases. In addition, a high number of dependencies has been observed in super classes that implements a Template Method. Several objects are instantiated in such implementations, elevating the coupling of these classes, and small tasks are passed to them. For instance, Figure 6 shows a class diagram of an Template Method instance identified in the Webmail system. The *Storage* class plays the *AbstractClass* role on the design pattern and contains the template methods. The *initConfigKeys()*, *setConfig()*, *getUserData()*, *setUserData()*, *deleteUserData()*, *setVirtualDomain()* and *save()* methods are the template methods existing within the *Storage* class.

Analyzing the diagram in Figure 6, we can observe the characteristics discussed above. The *Storage* class contains the template method of this Template Method instance. This class has an intense amount of getters and setters methods that make use of few attributes. In addition, we observe that this class has too much services provided for others classes with different concerns. It contains services regarding to user authentication, log handling, data transport, among others. This task overloading at the *Storage* class, besides centralizing the intelligence of the system in a single component, impairs the maintainability of component and system.

Based on this analysis, we identify some reasons that contributed to the *Template Method* ⇒ *God Class* co-occurrence emergence. The Template Method design pattern allows the classes extension and the addition of new features. However, poor planning and misapplication of this design pattern contributes to increase the amount of super-class responsibilities, concentrating a large part of the system intelligence in the superclass, generating the God Class emergence. To eliminate these co-occurrences, it is necessary to extract, from the overloaded classes, methods and attributes, adding them to other classes to divide the amount of work and effort performed by the Template classes.

3.1.2 Observer ⇒ Long Method

Observer is a solution that establishes a one-to-many dependency between objects. Observer uses a structure where the subject class has a list of all observers classes that use its data. When some information is changed in the subject by one of its observers, the subject is triggered by changing the other observers. The aim of this design pattern is the synchronization of data and the updating of objects in real time. This update occurs via polymorphism, avoiding the increase of complexity that generally occurs with the use of conditional structures. However, when using this design pattern, it is important to implement the operation that notifies the observers in the subject properly, since a poor planning of this operation can result in complex methods.

We manually inspected the methods involved in the *Observer* ⇒ *Long Method* co-occurrence. Figure 7 shows a class diagram of an Observer instance identified in the Hibernate system.

In the diagram, the *ConnectionObserver* class represents the Abstract Observer and the *ConnectionObserverAdapter*, *ConnectionObserverStatsBridge* and *JournalingConnectionObserver* classes represent the Concrete Observers which actually use the data maintained by the Subject. The *LogicalConnectionImpl* plays the Subject role and is primarily responsible for maintaining and managing data that is essential for the proper observers operation. When the existing data in the Subject is changed for some of observers, the others observers may be notified by one of following methods: *close()*, *obtainConnection()*, *releaseConnection()* and *notifyObserversStatementPrepared()*.

Analyzing the structure showed by the diagram in Figure 7, we observed that three of four notify methods presented the Long Method bad smell: *close()*, *obtainConnection()* and *releaseConnection()*. During the manual inspection we noticed that the methods implemented in the subject class, responsible for notifying observers, perform a lot of work. Such methods are big and complex, what make the code difficult to read and to understand. In some case, there is also scattering in the code, involving log records and observer update, among others, that influence the high complexity and the large size of these methods.

The aforementioned analysis revealed some main situations that may lead to the presence of the Long Method bad smell in methods implemented in Observer. The high amount of code repetitions and responsibilities assigned to the notifying observers

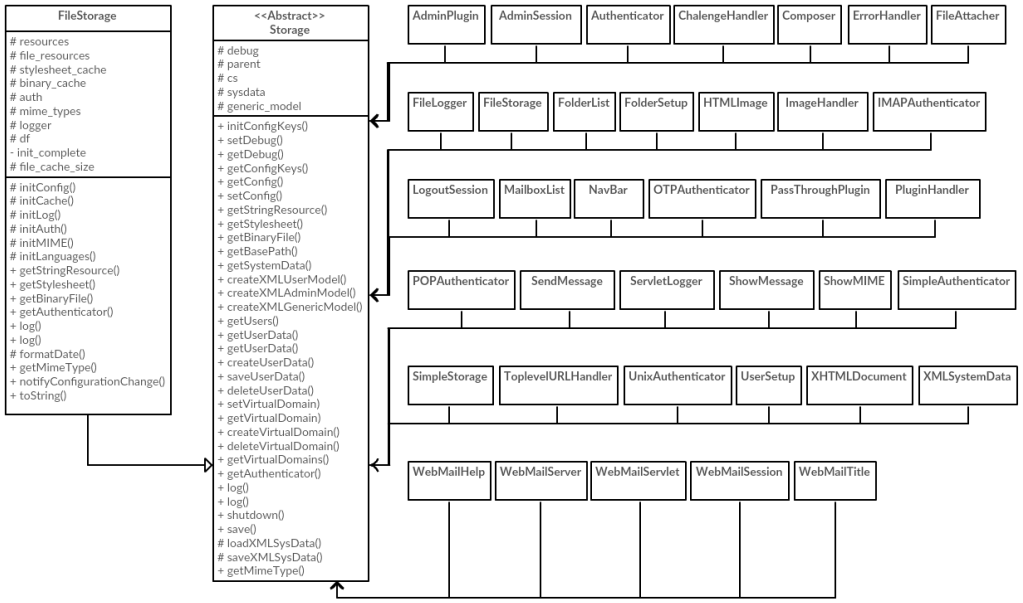

Figure 6: Class Diagram that represent the *Template Method* ⇒ *God Class* co-occurrence.

methods are the main situation we have found. Scattering and crosscutting concerns also have appeared in the implementation of such methods. A concern is a part of a problem that one needs to deal with in a software system. The registration of the operation log is an example of concern. When concerns are not modularized in a program, it leads to scattering and to crosscutting concerns in the code. In the methods implemented in Observer, we have detected scattering and crosscutting concerns in subject classes, specially due to repetition of code that implements register of log, aiming to notify observers. To eliminate such co-occurrences, it is necessary to eliminate these code repetitions and to modularize them so that they are implemented in a single entity and can be reused by other entities. With respect to scattering and crosscutting concerns, in object-oriented programming, these occurrences are difficult to control. However, they can be mitigated through the Singleton design pattern. Developers could create a Singleton class, responsible for managing log concern for example, that would provide a specific method for logging, requesting only the information that should be written.

Summary. The main reason of co-occurrences of God Class bad smell with Template Method is the misuse of object orientation, leading to concentration of responsibilities in the classes that implements the Template Method. A similar situation was also found in the case of the co-occurrences of Long Method bad smell with Observer; in this case, the misuse of object orientation leads to scattering and crosscutting concerns.

4 Lessons Learned

Design patterns are solutions with modular structure, so it is possible to infer think that their application avoid bad smells. Nevertheless, design patterns were not proposed specifically to this aim.

When applying design patterns in the construction of object-oriented software systems, it is important to ensure that object orientation and modularization are being properly applied. Design patterns are usually applied in methods and classes to allow software more flexible and extensible. When design patterns are not implemented carefully, it may result in design deviances, such as concentration of responsibilities in classes and methods, scattering and crosscutting concerns. Such deviances may lead to the presence of bad smell in software systems, and make the code complex and difficult to understand.

The main co-occurrences of design patterns and bad smells found in this work are *Template Method* ⇒ *God Class* and *Observer* ⇒ *Long Method*. In the manual inspection we found classes with many responsibilities, complex methods and code repetition. A better planning of the software design and its manutenability during the evolution phase could avoid the design patterns and bad smells co-occurrences. In particular, the application of refactoring techniques during the software

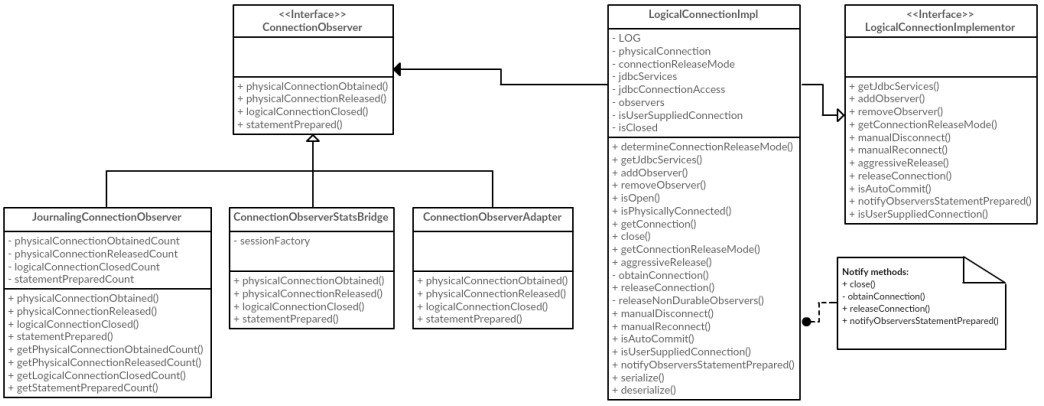

Figure 7: Class Diagram that represent the *Observer* ⇒ *Long Method* co-occurrence.

evolution could improve or keep the internal quality of the system, providing modularity, reducing the code complexity and improving the code readability.

Another important information found in this study is that most of the software systems analyzed presents both bad smells, God Class and Long Method in the implementation of the design pattern. The characteristics of these bad smells might be the reason of the co-occurrence of them with the design patterns. God Class is a class that performs a lot of work in a software system. To accomplish all the tasks a God Class has to do, there are two not excluding possibilities: the class has many methods and/or the methods of the class implement many tasks, i.e., some or all of them are Long Methods.

5 Threats to Validity

This section presents threats to validity according to the guidelines proposed by Wohlin et al. [24]. They are organized in external, internal and construct validity.

External Validity. We considered a sample composed of five systems. We mainly analyzed systems from a large data set, called Qualitas Corpus. The sample has small, medium, and large systems. Nevertheless, due to the small size of the sample, we are not able to generalize the results found in this study. Even though, the results found are still important because they show that the use of design patterns not necessarily avoid bad smells in object oriented software system.

Internal Validity. Our data collection was carried out by tools. The identification of design patterns was performed by DPDSS, and the identification of bad smells by RAFTool. We are not able to ensure that the results of those tools are totally right. However, we chose tools already used in previous work.

Construction Validity. To identify the main causes of the main co-occurrences of design pattern and the bad smells identified in this study, we performed a manual inspection in the classes and in the methods involved in such co-occurrences. This inspection was carried out by one of the authors of this work. Although the inspector has high level of knowledge of all the concepts involved in the analysis, the manual inspection might be error-prone. To overcome this threat, we decided to analyze a small quantity of software systems in this work.

6 Related Work

This section presents the main previous related work regarding the identification of co-occurrence between design patterns and bad smells.

Jaafar et al. [11] investigated the existence and the impact of the static relationship between anti-patterns and design patterns in software systems. They analyzed the behavior of these relationships during software evolution. In their work, a case study was performed with open source Java systems. The authors identified that design pattern have relationship with anti-patterns and that this relationship is in constant growth as the system evolves. In addition, the Command design pattern was identified as the one that presented the highest relation with the investigated anti-patterns.

Cardoso and Figueiredo [3] performed an exploratory analysis to investigate the co-occurrence of bad smells in software systems that use design patterns. Their study considered the God Class and Duplicate Code bad smells and eleven of twenty-three design patterns described by [10]. The authors extracted

the information of design patterns and bad smells instances, and through of association rules they found the co-occurrence between Command with God Class as well as Template Method with Duplicate Code.

Jaafar et al. [12] present a study on the impact of static and co-changes dependencies in classes with design patterns and bad smells, and verify the relation of these dependencies to occurrences of software failures. In their study, the authors observed the evolution of three open-source Java software projects and concluded that classes having static dependencies with anti-patterns as well as classes having static and co-change dependencies with anti-patterns and design patterns tend to have more flaws.

Walter and Alkhaeir [23] investigated the relationship between design patterns and bad smells, and examined how the presence of one interacts with the presence of the other in a class. The authors carried out an empirical study with seven bad smells and nine design patterns, identified in two applications. They concluded that the presence of design patterns is linked with a small number of cases of bad smells.

In this paper, we identified other kind of co-occurrences to the God Class and Long Method bad smells and discussed these cases. In addition we identified design patterns with low co-occurrence with the bad smells. Our study applied a different approach to detect bad smells. Our data rely in detection strategies to detect bad smells based in software metrics and their thresholds. These detection strategies were previously proposed and evaluated by Filó et al. [8].

7 Conclusion

In this paper we carried out an exploratory study with object-oriented systems that applies design patterns to (i) investigate if design patterns avoid the bad smells emergence in software; (ii) identify design patterns that may to present co-occurrence with bad smells; and (iii) extract the main reasons that impact on the co-occurrence emergence. The study considered a sample of five systems, of varying sizes. Our main contribution is that their findings may help the software engineering community in the comprehension of the internal structure of the software systems that apply design patterns.

To identify the bad smells in the software systems, we used detection strategies that are based in software metrics and that were previously proposed and evaluated in the literature. The detection strategies are based in metrics consistent with the characteristics of God Class and Long Method bad smells. Moreover, the thresholds of the metrics used in the strategies were also proposed and evaluated in previous work.

The results of this study show that the use of design patterns not necessarily avoid God Class and Long Method. We found that Composite and Factory Method are the design patterns less associated with the bad smells considered in this study. The main co-occurrences of design patterns and bad smells found in this paper are *Template Method* $\Rightarrow$ *God Class* and *Observer* $\Rightarrow$ *Long Method*. In the manual inspection of the artifacts that presented these relations, we found classes with many responsibilities, complex methods and code repetition. A better planning of the software design and its evolution could avoid the occurrences of bad smells in the implementation of design patterns. In particular, such problems would be avoided by a better planning of the software design, as well as by the application of refactoring techniques during the evolution of the software system.

Another important information found is that most of the analyzed systems presents both bad smells, God Class and Long Method, in the implementation of the design pattern. The characteristics of these bad smells might be the reason of the co-occurrence of them. God Class is a class that performs a lot of work in a software system, and to accomplish all the tasks there are two not excluding possibilities: the class has many methods and/or the methods of the class implement many tasks, i.e., some or all of them are Long Methods.

As future works, it is important (i) to extend this research to a greater amount of sample in order the results can be generalized; (ii) to investigate co-occurrences of design patterns with other bad smells; (iii) to conduct an analysis of a larger sample considering type and size of the software systems would be also of help to improve the comprehension of software systems that apply design patterns.

Acknowledgments

We would like to thank CAPES that supported to conduction of this work and the XIII Brazilian Symposium on Information Systems (SBSI 2017) where part of this study was published.

References

[1] Agrawal, R., Imieliński, T., and Swami, A. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[2] Brin, S., Motwani, R., Ullman, J. D., and Tsur, S. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2):255–264, June 1997.

[3] Cardoso, B. and Figueiredo, E. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In *Brazilian Symposium on Information Systems*, pages 347–354, 2015.

[4] Christopoulou, A., Giakoumakis, E. A., Zafeiris, V. E., and Soukara, V. Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202–1214, 2012.

[5] Ferreira, K. A. M., Bigonha, M. A., Bigonha, R. S., Mendes, L. F. O., and Almeida, H. C. Identifying thresholds for object-oriented software metrics. *The Journal of Systems and Software*, 85:244–257, 2012.

[6] Filó, T. G. S. Identifying reference values for object-oriented software metrics. Master's thesis, UFMG, Computer Science, 2014.

[7] Filó, T. G. S., Bigonha, M. A. S., and Ferreira, K. A. M. Raftool - filtering tools for methods, classes and packages with uncommon measurements of software metrics. In *WAMPS*, pages 1–6, 2014.

[8] Filó, T. G. S., Bigonha, M. A. S., and Ferreira, K. A. M. A catalogue of thresholds for object-oriented software metrics. In *SOFTENG*, pages 48–55, 2015.

[9] Fowler, M. and Beck, K. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1994.

[11] Jaafar, F., Guéhéneuc, Y., Hamel, S., and Khomh, F. Analysing anti-patterns static relationships with design patterns. *ECEASST*, 59, 2013.

[12] Jaafar, F., Gueheneuc, Y.-G., Hamel, S., Khomh, F., and Zulkernine, M. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 21(3):896–931, 2016.

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

[14] Liu, W., Hu, Z.-G. b., Liu, H.-T., and Yang, L. Automated pattern-directed refactoring for complex conditional statements. *Journal of Central South University*, 21(5):1935–1945, 2014.

[15] Marinescu, R. *Em Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.

[16] Nahar, N. and Sakib, K. Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory. In *CEUR Workshop Proc.*, pages 9–16, 2015.

[17] Nahar, N. and Sakib, K. Acdpr: A recommendation system for the creational design patterns using anti-patterns. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 4*, pages 4–7, 2016.

[18] Sousa, B. L., Bigonha, M. A. S., and Ferreira, K. A. M. Evaluating co-occurrence of gof design patterns with god class and long method bad smells. In *Proceedings of the XIII Brazilian Symposium on Information Systems*, pages 396–403, 2017.

[19] Sousa, B. L., Bigonha, M. A. S., and Ferreira, K. A. M. A tool for detection of co-occurrences between design patterns and bad smells. Technical report, Programming Language Lab (UFMG), llp 001-2017, 2017.

[20] Speicher, D. Code quality cultivation. *Communications in Computer and Information Science*, 348:334–349, 2013.

[21] Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. Qualitas. class corpus: A compiled version of the qualitas corpus. *ACM SIGSOFT Software Engineering Notes*, 38(5):1–4, 2013.

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

[23] Walter, B. and Alkhaeir, T. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 74:127–142, 2016.

[24] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. *Experimentation in Software Engineering*. Springer, 2012.

[25] Zafeiris, V. E., Poulias, S. H., Diamantidis, N., and Giakoumakis, E. Automated refactoring of super-class method invocations to the template method design pattern. *Information and Software Technology*, pages 19–35, 2017.