Received: 24 May 2018 | Revised: 20 January 2019 | Accepted: 25 March 2019

DOI: 10.1002/spe.2697

RESEARCH ARTICLE

WILEY

An exploratory study on cooccurrence of design patterns and bad smells using software metrics

Bruno L. Sousa[1] | Mariza A. S. Bigonha[1] | Kecia A. M. Ferreira[2]

[1]Department of Computer Science, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

[2]Department of Computing, Federal Center for Technological Education of Minas Gerais (CEFET-MG), Belo Horizonte, Brazil

Correspondence
Bruno L. Sousa, Department of Computer Science, Universidade Federal de Minas Gerais, 31270-901 Belo Horizonte-MG, Brazil.
Email: bruno.luan.sousa@dcc.ufmg.br

Funding information
CAPES; UFMG Programming Language Research Group; UFMG Graduate Program in Computer Science; CEFET-MG

Summary

A design pattern is a general reusable solution to commonly recurring problems in software projects. Bad smells are symptoms existing in the source code that possibly indicate the presence of a structural problem that requires code refactoring. Although design pattern and bad smells be different concepts, literature has shown that they may be related and cooccur during the evolution of a software system. This paper presents an empirical study that investigates cooccurrences of design patterns and bad smells as well as identifies the main factors that contribute to the emergence of the relationship between them. We carried out a case study with five Java systems to: (1) investigate if the use of design pattern reduces bad smell occurrence, (2) identify cooccurrences of design patterns and bad smells, and (3) identify situations that contribute for the cooccurrence emergence. As the main result, we found that the application of design pattern not necessarily avoid bad smell occurrences. The results also show that some design patterns such as composite, factory method, and singleton, are intrinsically modular and might be useful in creating high-quality systems. However, other design patterns such as adapter-command, proxy, and state-strategy, have presented high cooccurrence frequency with bad smells; therefore, they require attention in their implementation. Finally, via manual inspection in the components with cooccurrence, we found that the identified cooccurrences appeared due to poor planning and inadequate application of design patterns.

KEYWORDS

bad smell, cooccurrence, design pattern, software metrics

1 | INTRODUCTION

Design patterns are general solutions applied to recurring problems in the context of software development.[1] Their main goal is to ensure the creation of flexible and extensible systems with high reusability and maintainability. In general, design patterns are recognized as good programming practice. Gamma et al[1] described a set of 23 design patterns widely referenced as "Gang of Four (GoF) design patterns."

Bad smells are structures in the source code that may indicate a more serious problem and may demand refactoring to remove them.[2] Code regions with these symptoms are not considered errors, but they impair the software quality and violate some quality concepts of software engineering such as modularity, readability, and reuse. Although design patterns have not been designed to remove occurrences of bad smells, they may contribute to reduce the emergence of

TABLE 1 Design patterns and bad smells used in this study

Bad Smell	Design Patterns		
Data class	Adapter	Bridge	Command
Feature envy	Composite	Decorator	Factory method
Large class	Observer	Prototype	Proxy
Long method	State	Strategy	Singleton
Refused bequest	Template method	Visitor	

these structures in the source code when correctly applied.[3,4] Previous studies have made use of design patterns to indicate refactoring suggestions and remove bad smells.[5-9] However, design patterns may also introduce complex structures in the source code, such as the bad smells occurrences, when they are not correctly applied.[10-14]

Previously, we carried out systematic literature mapping, and we concluded that these two structures have been studied under three aspects: impact on software quality, refactoring, and cooccurrences.[15] Moreover, we also found that, although there are papers associating design pattern with bad smells,[10-13] this topic is still little explored and there are some gaps not covered by the studies done, such as situations that culminate in the cooccurrence between design patterns and bad smells and situations in which design patterns are more likely to cooccur with bad smells. Therefore, an investigation about these subjects may serve to alert researchers and developers that the application of design patterns may result in problematic structures that increase the complexity, readability, and comprehension of a software system.

To cover these gaps, we carried out an empirical study to investigate the relationship between design patterns and bad smells via source code analysis of oriented-object systems. To do that, we verify the design pattern effectiveness in the bad smell reduction, identify the design patterns that presented cooccurrence with bad smells, and identify situations that contribute to the emergence of cooccurrences between design patterns and bad smells. We considered in this study five bad smells defined by Fowler and Beck[2] and 14 design patterns from the GoF catalog. Table 1 summarizes the design patterns and the bad smells used in the present work.

The remaining of this paper is organized as follows. Section 2 describes the methodology used in this study emphasizing its main steps. Section 3 presents a tool for identifying cooccurrences of design patterns and bad smells, called design pattern smell. Section 4 reports the case study carried out in this paper and details the main results by answering the research questions (RQs). Section 5 discusses the lessons learned with this study. Section 6 shows the threats to validity and discusses the main decisions to mitigate them. Section 7 presents some related work. Section 8 concludes this paper.

2 | METHODOLOGY

This section describes the methodology applied in this study. Its main steps are shown in Figure 1. Section 2.1 presents the RQs investigated in this work. Section 2.2 shows the detection strategies proposed by Souza[16] and used to detect bad smells in our analysis. Section 2.3 describes how we composed the data set to perform the study. Section 2.4 details the data collection process. Section 2.5 presents the association rules used for identifying cooccurrences. Lastly, Section 2.6 summarizes the method used for the analysis of the results.

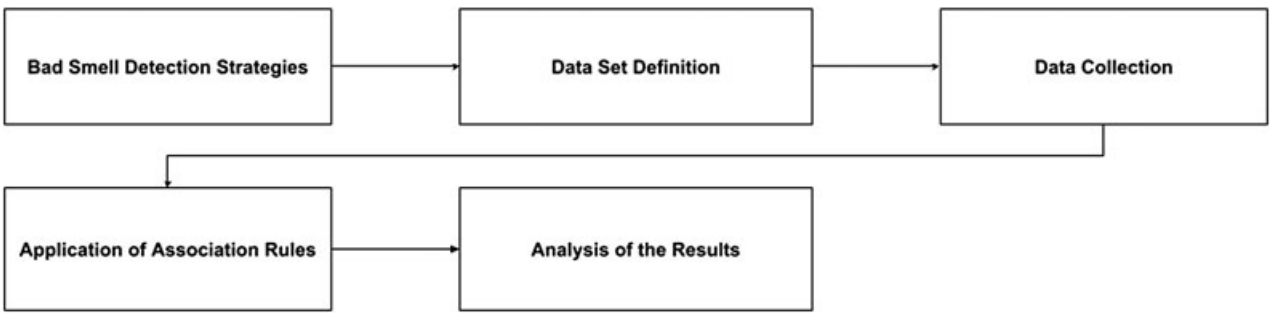

FIGURE 1 Steps of the case study

2.1 | Research questions

To evaluate the cooccurrence relationship between GoF design patterns and bad smells, we proposed three RQs. They are as follows:

- **RQ1:** Do the design patterns defined by the GOF catalog avoid bad smell occurrences in software systems?
- **RQ2:** Which design patterns of the GOF catalog have cooccurrence with bad smells?
- **RQ3:** What are the more common situations in which bad smells appear in software systems that apply GOF design patterns?

2.2 | Bad smell detection strategies

We considered five bad smells described by Fowler and Beck[2]: data class, feature envy, large class, long method, and refused bequest. We chose them because they are problematic for software maintenance and they are related to data problems, a large amount of information, complexity, and coupling. Moreover, previous studies[10-13] have used them to investigate their impact on the emergence of software faults and have analyzed the static relationships between them and design patterns.

One way to detect bad smells is done by using detection strategies. Detection strategy is a quantifiable expression of a rule that evaluates whether source code fragments have properties of a given bad smell.[17] Along with detection strategies, thresholds are used to determine the relationship between a metric and a bad smell. Although there are other detection strategies proposed and evaluated in the literature,[16,17] in the present work, we decided to use the detection strategies proposed by Souza[16] because they apply well-known software metrics, they were previously evaluated, and their results indicate that they have a good recall and accuracy to identify bad smells. The metric thresholds used in the detection strategies were proposed by Filó et al.[18] Filó et al presented an extraction method of thresholds for object-oriented software metrics and applied it to identify thresholds for 18 software metrics. The threshold catalog created by them classifies a metric in three ranges: good/common, regular/casual, and bad/uncommon. The good/common range corresponds to the high-frequency values, characterizing the most common metric values in practice. The bad/uncommon range corresponds to the low-frequency values, and the regular/casual range is intermediate, corresponding to values which are not very frequent or rare. Table 2 presents the Filó's catalog.

The literature reports that tools such as DECOR,[20] JDeodorant,[21] and JSpIRIT[22] have supported researchers and developers to automatically detect bad smells. However, we identified some problem to adopt them. For instance, DECOR does not use an approach based on metrics, and it is not currently available for download.[23] JSpIRIT and JDeodorant use an approach based on metrics,[23] but they do not allow the user to customize the detection strategies or the metric thresholds. Therefore, we opted to use another tool, ie, RAFTool,[24] which is based on software metrics that allow users to implement their detection strategies for detecting bad smells.

Each detection strategy is composed by a sequence of clauses connected by `AND` and `OR` logical operators. A clause consists of a composition of metrics with their respective thresholds. In the sequel, we describe the bad smells analyzed in this work and the corresponding software metrics applied in their detection strategies.

Data Class. Data class refers to classes that have only *gets* and *sets* methods and do not have many features over these data. A data class acts as a data container in a system, providing information for the use of other components. According to Yamashita and Moonen,[25] this kind of class negatively impacts the system maintainability because it often creates incoming dependencies from envy methods, and consequently incentivizes the emergence of the feature-envy bad smell in the system. The detection strategy regarding the data class is composed by the following metrics:

- Number of children (NSC): It measures the number of direct subclasses from a given class. As classes with a data class symptom tend to have few subclasses, it is considered that low values of NSC may help to identify this bad smell.
- Depth of inheritance tree (DIT): It indicates the level where a given class is in the inheritance tree. Data classes are self-contained and only provide access to their attributes. Moreover, they do not need to inherit features from another class. Therefore, it is considered that low values of DIT may be useful to identify data class.
- Number of fields (NOF): It measures the number of attributes of a class. Data classes may store a high quantity of data. So, it is considered that a regular or a bad number of attributes may indicate the presence of data class.

Feature envy. Feature envy occurs when methods from a given class are more interested in features from other components and use them in excess. This symptom indicates that there are methods in a class that should be removed. Feature

TABLE 2 Threshold catalog for object-oriented metrics[18,19]

Metric	Good/Common	Regular/Casual	Bad/Rare
CA	$m \leq 7$	$7 < m \leq 39$	$m > 39$
CE	$m \leq 6$	$6 < m \leq 16$	$m > 16$
DIT	$m \leq 2$	$2 < m \leq 4$	$m > 4$
LCOM	$m \leq 0.167$	$0.167 < m \leq 0.725$	$m > 0.725$
MLOC	$m \leq 10$	$10 < m \leq 30$	$m > 30$
NBD	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NOC	$m \leq 11$	$11 < m \leq 28$	$m > 28$
NOF	$m \leq 3$	$3 < m \leq 8$	$m > 8$
NOM	$m \leq 6$	$6 < m \leq 14$	$m > 14$
NORM	$m \leq 2$	$2 < m \leq 4$	$m > 4$
NSC	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NSF	$m \leq 1$	$1 < m \leq 5$	$m > 5$
NSM	$m \leq 1$	$1 < m \leq 3$	$m > 3$
PAR	$m \leq 2$	$2 < m \leq 4$	$m > 4$
RMD	$m \leq 0.467$	$0.467 < m \leq 0.750$	$m > 0.750$
SIX	$m \leq 0.019$	$0.019 < m \leq 1.333$	$m > 1.333$
VG	$m \leq 2$	$2 < m \leq 4$	$m > 4$
WMC	$m \leq 11$	$11 < m \leq 34$	$m > 34$

Abbreviations: CA, Afferent Coupling; CE, Efferent Coupling; DIT, depth of inheritance tree; LCOM, lack of cohesion of methods; MLOC, method lines of code; NBD, nested block depth; NOC, number of classes; NOF, number of fields; NOM, number of methods; NORM, number of overriden methods; NSC, number of children; NSF, Number of Static Attributes; NSM; PAR, Number of Parameters; RMD, Normalized Distance; SI, specialized index; VG, McCabe cyclomatic complexity; WMC, weighted methods per class.

envy negatively impacts the system maintainability because, as components with this symptom access data and methods from different areas of the system, it forces the developers to examine all the artifacts called by these envious components to understand their behavior within the system.[25] Detection strategies regarding feature envy may be proposed to detect both methods and classes. The detection strategy of Souza[16] aims to detect feature envy to the class level, and it is composed by only one metric:

- Lack of cohesion of methods (LCOM): It measures the internal cohesion of a class by computing the lack of cohesion between the methods of the class. The greater the value of this metric is, the less the internal cohesion of the class. Classes with envious feature tend to have low cohesion because they are interested in methods from other classes. Therefore, it is considered that a regular or bad value of LCOM may detect this feature.

Large class. Large class occurs in classes that perform a lot of tasks and have many responsibilities in the system. This anomaly may be characterized as an object that knows very much and has many instances of variables. According to Yamashita and Moonen,[25] a large class negatively impacts the systems maintainability because it is related to high values of size measures. Moreover, as it often uses a lot of different variables, there is a higher chance of the same temporary variable to be used for several different purposes. The detection strategy regarding large class bad smell is based in the following metrics:

- Number of fields (NOF): Souza[16] chose NOF because large classes have an excessive knowledge of the system; therefore, it tends to have a large number of attribute. So, this metric in the bad range may aid to detect classes that concentrate knowledge.
- Number of methods (NOM): Souza[16] chose NOM because besides concentrating knowledge, large classes also have high data processing and therefore tends to have a large NOM. So, this metric in the bad range may detect this high data processing in classes.
- Lack of cohesion of methods (LCOM): one of the main characteristics of the large class bad smell is the high overhead of tasks and responsibilities in the system. LCOM is a metric that may measure the rate of cohesion in a class. Therefore,

when a class has a high value of LCOM, ie, values into the bad range, may be an evidence that the class is performing excessive work.
- Weighted methods per class (WMC): It measures the complexity of a class taking into account the methods within that class. Large classes perform too much work, and they tend to present large and complex methods. Therefore, when a class has a high value of WMC, ie, a bad value, it may be performing excessive responsibilities.

Long method. Long method is a method that implements several features and is complex and difficult to understand. This kind of a method tends to centralize the functionality of the class. A long method negatively impacts the system maintainability because it is linked to high values of size metrics and it demands complex tasks for removing it from the system structure.[25] The detection strategy regarding the long method bad smell is composed by the following metrics:

- Method lines of code (MLOC): It measures the number of lines of code of a method. The main characteristic of the long method is a large number of lines of code. Therefore, when a method has a bad value of MLOC, it might be a long method.
- Nested block depth (NBD): It measures the depth of nested block existing in a method. Nested blocks occur when control structures, such as *IF*, *WHILE*, and *FOR*, are one inside the other. This metric is an indicator of complexity since increasing the number of nested blocks makes the code more difficult to understand. So, when a method has a high value of NBD, there is an evidence of excessive and complex processing; therefore, Souza[16] included NBD in the detection strategy.
- McCabe cyclomatic complexity (VG): It aims to measure the number of independent execution paths in the source code. Cyclomatic complexity indicates that the method has a high complexity. Therefore, when a method has a high value of VG, ie, a bad value, it might be an evidence of the long method.

Refused bequest. Refused bequest occurs when a subclass does not use features and attributes inherited from its superclass. The occurrence of this smell is an evidence that there is something wrong with the inheritance structure. According to Palomba et al,[26] refused bequest is a highly diffused smell, and it may negatively impair the system maintainability because its occurrence tends to increase the systems proneness to change. Moreover, as refused bequest consists of misuse of inheritance, its high occurrence makes the system comprehension complex and difficult. The detection strategy regarding the refused bequest bad smell is composed by only one metric:

- Specialization index (SIX): This metric indicates the level of specialization of a class. It is given by the ratio between the ratio of the number of overridden methods (NORM) weighted by the DIT and the NOM in the class, ie, $(NORM \times DIT)/NOM$. So, the greater the degree of overwriting in the subclass regarding its superclass, the greater the value of SIX and the greater the chances of the subclass deny the responsibilities provided through inheritance. As the main characteristic of the refused bequest is the high overwriting of methods, when a class has the SIX value in the regular or bad range, it might indicate the presence of this bad smell.

Souza[16] evaluated the detection strategies through three experiments, which were supported by a specialist in object-oriented programming and bad smells. The first experiment compared the detection strategies results with the JSpIRIT[22] results. Souza[16] collected the bad smell instances of two Java systems, namely, Squirrel SQL 3.1.2 and Webmail 0.7.10, and used them as a verification list to compute the recall and precision of the detection strategies. However, relying purely on the results of a tool may introduce bias in the results. So, the participation of the specialist was fundamental in this experiment because he manually analyzed and validated the results returned by JSpIRIT to remove the false positives. The results of recall and precision were recalculated based on the validated reference list. Table 3 summarizes the results of this first experiment.[16] The column 'NV' contains the recall and precision results regarding the JSpIRIT nonvalidated results. The column 'V' contains the recall and precision results regarding the JSpIRIT validated results. Finally, the column 'DIF' contains the difference of recall and precision metrics between the nonvalidated and validated data. A DIF positive value indicates that the specialist identified and removed false positives returned by JSpIRIT. In contrast, a DIF negative value indicates that the specialist identified false positives in the JSpIRIT results that were being recognized by the detection strategies as a bad smell. According to Souza[16], the data class was not evaluated in this first experiment because JSpIRIT did not find instances of this bad smell in the analyzed systems.

The second experiment carried out by Souza[16] aimed to compare the detection strategies with results returned by another tool, JDeodorant.[21] In this experiment, they collected bad smell instances of 12 Java systems and compared the

TABLE 3 Recall and precision of the detection strategies based on results of JSpIRIT[16]

System		Feature Envy			Large Class			Long Method			Refused Bequest		
		NV	V	DIF	NV	V	DIF	NV	V	DIF	NV	V	DIF
Squirrel	Recall	50%	60%	10%	100%	100%	0%	100%	100%	0%	-	-	-
	Precision	55%	30%	-25%	60%	40%	-20%	17%	17%	0%	-	-	-
Webmail	Recall	27%	33%	6%	15%	50%	35%	100%	100%	0%	57%	80%	23%
	Precision	47%	24%	-23%	100%	50%	-50%	56%	41%	-15%	14%	14%	0%

Abbreviations: Abbreviations: DIF, difference of recall and precision metrics between nonvalidated and validated data; NV, recall and precision results regarding the JSpIRIT nonvalidated results; V, recall and precision results regarding the JSpIRIT validated results.

TABLE 4 Recall and precision of the detection strategies based on results of JDeodorant[16]

Bad Smell	Recall	Precision
Data class	-	-
Feature envy	47.5%	24%
Large class	16%	91%
Long method	18.5%	51.5%
Refused bequest	-	-

TABLE 5 Recall and precision of the detection strategies based on the analysis of a specialist[16]

Bad Smell	Recall	Precision
Data class	47%	32%
Feature envy	-	-
Large class	70%	37%
Long method	73%	19%
Refused bequest	100%	2%

results of detection strategies with the data collected in each system. Table 4 summarizes the detection strategies results regarding to recall and precision obtained in this second experiment. The values reported refer to the median returned for these 12 systems. According to Souza,[16] data class and refused bequest were not evaluated in this experiment because JDeodorant did not support the detection of them.

The third experiment was carried out with the support of the specialist again. Here, the specialist manually analyzed a Java system, Apache Maven 3.0.5, to identify bad smell instances based on his knowledge and background. He identified instances of all bad smells, except feature envy. The specialist did not analyze feature envy because such analysis will demand a great deal of time, and it would overload the specialist who was a volunteer in this study. Table 5 summarizes the recall and precision results obtained in this experiment.

2.3 | Data set definition

The second step of this study was the definition of the sample of systems to be analyzed. We chose five Java systems to compose our data set. They are Hibernate 4.2.0 JHoDraw 7.5.1, Kolmafia 4.2.0, Webmail 0.7.10, and Weka 3.6.9. We used these systems because they are open source, belong to different domains, have different size, use GoF design patterns, and present the bad smells considered in this work. Hibernate 4.2.0, JHotDraw 7.5.1, and Webmail 0.7.10 are respectively large, medium and small systems that have been studied by previous work with focus in both design patterns and bad smells.[11]Kolmafia 4.2.0 is a medium system. Previous studies pointed out it has metric values considered problematic,[27] but they did not identify the correlation of these values with bad smells or design patterns. Weka 3.6.9 is a medium system with support to a collection of machine learning algorithms for data mining tasks. We identified both GoF design pattern and bad smell instances in Weka 3.6.9. As this project has not been evaluated in this manner in previous studies, we decided to analyze its internal structure quality and investigate the existing of cooccurrences between design patterns and bad smells on it.

The software systems chosen to compose the data set were extracted from *Qualitas.class Corpus*,[28] a compiled version of *Qualitas Corpus*.[29] *Qualitas Corpus* is formed by a collection of open-source systems developed in Java that are made available for empirical studies.[18] Moreover, *Qualitas.class Corpus* contains software metrics already collected from 112 systems.

2.4 | Data collection

The third step of this study was the data collection. *Qualitas.class Corpus* has XML files with metrics collected from the systems. As most of the systems considered in our analysis are from the *Qualitas.class Corpus*, we used these metrics already collected. Kolmafia 17.3 is the only system that does not belong to *Qualitas.class Corpus*. So, we had to download its source code and collect its metrics value. We used the IDE Eclipse 4.2 Juno[30] and Metrics 1.3.6[31] plugin to collect the Kolmafia 17.3 metrics. After collecting the metrics, we exported and saved them in an XML file.

To verify the existence of the design patterns in the systems, we used the Design Pattern Detection* tool.[32] According to Tsantalis et al,[32] this tool models all aspects of design patterns via directed graphs represented in quadratic matrices and applies an algorithm called *similarity scoring*. This algorithm uses a system and a design pattern graph as input to calculate the similarity scoring between the vertices. For Tsantalis et al,[32] the main advantage of this approach is the ability to detect not only design patterns in their base form, normally found in the literature, but also modified versions of them. This tool was tested by its authors in three systems, namely, JHotDraw 5.1, JRefactory 2.6.24, and JUnit 3.7, and false positives were not returned. False negatives were returned only for two design patterns: factory method and state. The results presented by this tool indicate that it is effective in identifying design pattern instances. Moreover, among the tools studied, Design Pattern Detection was identified in the literature as the one that recognizes a greater number of GOF catalog design patterns, ie, 14 design patterns in total. However, adapter, command, state, and strategy could not be separately identified by this tool. Instead, Design Pattern Detection identifies these design patterns as being an adapter-command type of instance, and state-strategy being another type of instance, totaling 12 design pattern instances identified by this tool. Also, Design Pattern Detection is one of the most used tool in researches that require design patterns detection.

Filó et al[24] developed a tool, RAFTool†, which performs the identification of methods, classes, and packages with anomalous measurements of object-oriented software metrics. RAFTool was used in this research to implement the detection strategies. The tool receives as input the XML file of the target system with its metrics and a detection strategy described by a logic expression in a given format. RAFTool reports as results the classes or methods whose metric values fit the detection strategy.

To collect the bad smell information from the systems, we transformed the detection strategies into filtering expressions so that they could be used by RAFTool. RAFTool uses the Filó's catalog showed in Table 2. The threshold of the Filó's catalog was divided into three types of range. To use RAFTool, we need to use the following keywords to attribute the threshold, in consonance with the values exhibited in Table 2 - to the metrics:

- **COMMON**: It corresponds to the GOOD/COMMON range of the thresholds proposed by Filó et al.[18] For instance, if we attribute the COMMON keyword to the LCOM metric (COMMON[LCOM]), components with values of LCOM less than or equal to 0.167 will be returned.
- **CASUAL**: It corresponds to the REGULAR/CASUAL range of the thresholds proposed by Filó et al.[18] For instance, if we attribute the CASUAL keyword to the LCOM metric (CASUAL[LCOM]), components with values of LCOM greater than 0.167 and less than or equal to 0.725 will be returned.
- **UNCOMMON**: It corresponds to the BAD/UNCOMMON range of the thresholds proposed by Filó et al.[18] For instance, if we attribute the UNCOMMON keyword to the LCOM metric (UNCOMMON[LCOM]), components with values of LCOM greater than 0.725 will be returned.

A metric with its respective threshold represents a specific characteristic of a bad smell and a little part of the detection strategy. In general, a detection strategy is composed by several characteristics connected by composition mechanisms. The composition mechanisms are logical operators used to connect the metrics and to establish the semantic between the characteristics defined by the bad smells.[17] Therefore, after connecting the metrics to their respective threshold, we

*https://users.encs.concordia\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{c\global\mathchardef\accent@spacefactor\spacefactor}\accent10c\egroup\spacefactor\accent@spacefactora/~nikolaos/pattern_detection.html
†http://homepages.dcc.ufmg.br/~tfilo/raftool/

used the AND and OR logical expressions to join different metrics and create the expressions according to the detection strategies proposed by Souza[16] The final settings regarding the expressions used to detect the bad smells are as follows:

Exp1 COMMON[NSC] AND COMMON[DIT] AND (UNCOMMON[NOF] OR CASUAL[NOF])
Exp2 UNCOMMON[LCOM]
Exp3 UNCOMMON[LCOM] AND UNCOMMON[WMC] AND UNCOMMON[NOF] AND UNCOMMON[NOM]
Exp4 UNCOMMON[MLOC] AND UNCOMMON[VG] AND UNCOMMON[NBD]
Exp5 UNCOMMON[SIX]

The first filtering expression refers to the data class bad smell. It is made up of a combination of good (COMMON) range for the NSC and DIT metrics, and regular (CASUAL) and Bad (UNCOMMON) ranges for the NOF metric.

The second filtering expression refers to the feature envy bad smell. It is made up only by the LCOM metric using the bad (UNCOMMON) range exhibited on Table 2.

The third filtering expression refers to the large class bad smell. It is made up of a combination of bad (UNCOMMON) range with the four metrics: LCOM, WMC, NOF, and NOM associated with this bad smell.

The fourth filtering expression refers to the long method bad smell. It is made up of a combination of bad (UNCOMMON) range with the three metrics: MLOC, VG, and NBD associated with this bad smell.

The fifth filtering expression refers to the refused bequest bad smell. It is made up of only one metric, ie, SIX, using the bad (UNCOMMON) range.

The design patterns' instances returned by Design Pattern Detection may be composed of one or more classes or methods. An example of this is the instance returned to the bridge design pattern. It has a class responsible for representing the implementation part and another with the role of serving the abstraction part. To solve this problem, we developed the Design Pattern Smell tool, described in Section 3, which counts the classes and methods in a design pattern instance as well as identifies the components with a given design pattern and a given bad smell. This tool receives as input the files in the XML format exported by Design Pattern Detection, containing the design patterns instances of a system, and the CSV files generated by RAFTool, containing the classes or methods with a given bad smell.

2.5 | Association rules

The fourth step of the study consisted of the association of design patterns and bad smell to identify the cooccurrences. In this process, we applied association rules based on the data mining concept.[33] We chose association rules because this method combines items from a data set to extract knowledge about the analyzed data. Reinforcing this choice, previous studies in the same context, Cardoso and Figueiredo[11] and Walter and Alkhaeir[13] also applied this method to identify the cooccurrences between bad smells and design patterns.

In general, association rules may be extracted based on three different metrics: support,[33] confidence,[33] and conviction.[34] All of them use the following concepts:

- **Transaction:** It is defined as a set of items.
- **Antecedent:** It is an item that appears on the left side of the association rule.
- **Consequent:** It is an item that appears on the right side of the association rule.

An association rule has the following form: Antecedent ⇒ Consequent. The support (sup) metric in an association rule, which indicates the frequency that an item occurs in a transaction (Equation (1)).

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

For instance, consider a shopping base in a supermarket. Suppose that there is a data set with the records from 1000 transactions, which are the set of purchased items. In it, the pasta and tomato items appear together in 100 of these records. Thus, the support of this relationship is 0.1, ie, 10%.

The confidence metric (conf) expresses the probability of a consequent occurring since the antecedent occurs. In other words, it indicates the chance of the right side of the rule to happen, given the occurrence of the left side (Equation (2)).

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

In the example mentioned above, let us consider that the `pasta` item is found alone in 200 of 1000 transactions of the data set. To compute the confidence of the `pasta` ⇒ `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, ie, 50%. Confidence is very sensitive to the frequency on the right side of the association rule, ie, a very high value in the right side of the association rule may generate a high confidence value, even if the items do not have any relation.

To solve this problem, Brin et al[34] proposed the conviction metric. This metric uses the support in both the antecedent and the consequent (Equation (3)).

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

In the given example, let us consider that the item `tomato` is found alone in 300 of 1000 transactions of the data set. Thus, the `tomato` support, sup(`tomato`), is 0.3 and the confidence, conf(`pasta` ⇒ `tomato`), is 0.5. Applying these values in the Equation (3), the conviction conv(`pasta` ⇒ `tomato`) is 1.4. When the conviction value is 1.0, it indicates that the antecedent and the consequent have no relationship at all. When the conviction value is less than 1.0, it indicates that if the antecedent occurs, the consequent tends not to occur. When the conviction value is greater than 1.0, it means that the antecedent and the consequent have a relationship; the greater the conviction value, the greater the relationship between the antecedent and the consequent. An infinite result indicates that the antecedent never appears in the transactions.

To apply the association rules, we considered antecedent, consequent, and transaction as follows:

- **Transaction:** It represents each class in the analyzed system.
- **Antecedent:** It represents each design pattern explored in this study and belongs to the GOF catalog.
- **Consequent:** It represents each bad smell explored in this study.

2.6 | Analysis of the results

Figure 2 illustrates the method used to analyze the data obtained in this study. The first step identifies presence of the design patterns in the systems. To distinguish them, we used the Design Pattern Detection[32] and stored the results obtained in a table.

The second step applies the **Exp1**, **Exp2**, **Exp3**, **Exp4**, and **Exp5** (Section 2.4) filtering expressions in RAFTool. The goal of this step is to identify classes and methods with problematic metric values and with the presence of the bad smells investigated.

The third step identifies the cooccurrences between the design patterns and bad smells and the application of the association rules. To accomplish these two tasks, we used the Design Pattern Smell tool, which is presented in Section 3.

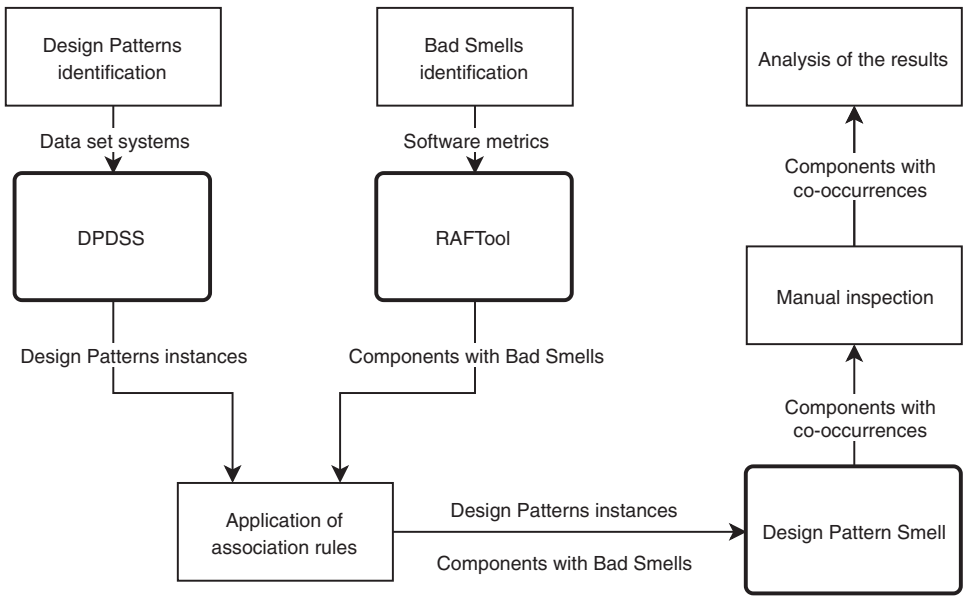

FIGURE 2 Method of the results analysis

The fourth step consists of the manual inspection of methods and classes with cooccurrence, to identify situations that favored the presence of these relationships in such components.

In the fifth step, we analyzed the data to answer the RQs.

- **RQ1:** Do the design patterns defined by the GOF catalog avoid bad smell occurrences in software systems?
- **RQ2:** Which design patterns of the GOF catalog have cooccurrence with bad smells?
- **RQ3:** What are the more common situations in which bad smells appear in software systems that apply GOF design patterns?

We consider cooccurrence as a class or method that is part of a design pattern instance and has the presence of a bad smell. These components are identified by Design Pattern Smell, which matches the design pattern components with the bad smell components. Design Pattern Smell detects the cooccurrences according to the bad smell level. For instance, large class is a bad smell at the class level. Therefore, to identify it, Design Pattern Smell inspects the classes listed in the XML files, in order to match them with the bad smells data reported by RAFTool in a CSV file. For bad smells at the method level, eg, long method, Design Pattern Smell performs the same process considering methods instead of classes. It is important to highlight the methods analyzed here are the ones that play some role within the design patterns, such as the notifier methods existing in the Observer design pattern instance. After identifying the cooccurrences and applying the association rules, we carried out a manual inspection on the components belonging to the highest intensity relationship to determine the situations in the systems source code that might have contributed to the emergence of the cooccurrences.

3 | DESIGN PATTERN SMELL

Design Pattern Smell is a static analysis tool proposed for identifying cooccurrences between GoF design patterns and bad smells based on information extracted from the source code. The main motivation to propose Design Pattern Smell was the lack of tools and techniques in the literature to support developers and researchers in the detection of cooccurrences between design patterns and bad smells. The main contributions provided by Design Pattern Smell are as follows: (1) It allows developers and researchers to identify the existing cooccurrences between GoF design patterns and bad smells, and (2) it supports to carry out an exploratory analysis to understand the reasons that lead to the cooccurrence emergence.

Therefore, the principal purpose of Design Pattern Smell is to support the evaluation of software quality by identifying code structures whose presence of bad smells may degenerate the application of GoF design patterns. Currently, Design Pattern Smell supports detection of 14 GoF design patterns with any bad smell in both class or method level. To do that, the bad smell instances must be previously computed and supplied to Design Pattern Smell via a CSV file.

3.1 | Main features

We described the main features of the Design Pattern Smell as follows.

Import computed design pattern instances. To identify the cooccurrences between GoF design patterns and bad smells, Design Pattern Smell requires the user to import XML files with GoF design pattern instances regarding a given system. This input file follows the same format exported by the Design Pattern Detection tool,[32] and we described it on the Design Pattern Smell's website.[35]

Import computed bad smell instances. Design Pattern Smell also requires the user to provide a CSV file containing classes/methods with the presence of bad smell. After importing this file, Design Pattern Smell extracts all components existing in the input files and performs the matching between the design pattern and bad smell components, by analyzing and comparing their names, signatures, and path to identify those that have the cooccurrence of these two structures. The user needs to write the CSV input file according to the format specified on the tool's website.[35]

Application of association rules. To identify the cooccurrence between design patterns and bad smells, we used an important concept of data mining, called association rules.[33,34] The association rule allows you to combine items from a data set through some metrics such as support,[33] confidence,[33] lift [34] and conviction,[34] and extract a knowledge about these data. To apply the association rules, one needs to know some important terms used by these rules, such as *transaction*, *antecedent*, and *consequent*, as we have explained in Section 2.5. In the context of cooccurrence in Design Pattern Smell, the *antecedent* is considered a GoF design pattern, the *consequent* is considered a bad smell, and a *transaction* is a class or method of a system according to the granularity of the bad smell used. For instance,

if the bad smell analyzed is in the class level, the transaction is the classes of the system. In contrast, if the bad smell analyzed is in the method level, the transaction is the methods of the system. The user should apply association rules in the data used, to analyze the intensity of cooccurrence between design patterns and bad smells. The *Association Rules Calculator* is a module of Design Pattern Smell that allows the user to apply the association rules automatically. Design Pattern Smell uses GoF design pattern and bad smell instances precomputed provided by the user; therefore, the tool does not recognize the total number of classes (NOC) or NOM existing in the system. Considering that this information is essential to calculate the association rules metrics, to use this feature, Design Pattern Smell requires the user to enter the NOC or NOM metrics via a field. These metrics may be easily computed with the support of some tools such as Metrics[‡], which analyzes a source code of the system and extracts these values. Moreover, we also make available a panel with four kinds of association rules'[33,34] so that the user may choose which metrics he/she wants the Design Pattern Smell calculates. To compute these metrics, Design Pattern Smell uses the metrics defined in Section 2.5, and applies to them the information extracted from the input files provided by the user, such as the NOC/NOM regarding the GoF design patterns, NOC/NOM regarding the bad smells, among others, and the NOC/NOF metrics provided by the user in this feature. These metrics point out the intensity of cooccurrences in the source code, and through them, it is possible to identify which GoF design patterns present cooccurrences with bad smells. By default, these metrics are preselected to be computed in this feature. However, the users may compute only those that are of their concern.

Result Generation, Visualization, and Export. After performing the matching of the components and identifying the cooccurrences, the user may consult the following reports: (1) the number of design pattern instances in the system, and (2) the amount and information about the components that presented cooccurrences between design patterns and bad smell. Moreover, after applying the association rules, the report with the results are displayed to the user. These reports are presented in a data grid view. Design Pattern Smell exports these results to a CSV file for further analysis and manipulation.

Data Management. This functionality allows the user to use data from GoF design pattern already stored in Design Pattern Smell to perform the matching of design pattern instances with other bad smell instances. In this case, the user must clear old information regarding to the bad smell under analysis and import other CSV files. This process may also be performed for design patterns when the user wants to change the system under review.

Help. For users unfamiliar with association rules, this functionality describes this method of analysis, the formulas used to identify the cooccurrences, an overview of how to analyze results, and the definition of technical terms used to calculate formulas. Moreover, this feature presents general information about the version of this tool and provides a link to a video tutorial that presents a running example of the tool.

3.2 | Architecture

The Design Pattern Smell's architecture consists of six internal modules, as shown in Figure 3.

Input Manager. This module is responsible for managing the input files as well as performing the verification and validation of the required format. It receives one or more XML files with precomputed GoF design pattern instances from a software and a CSV file with precomputed bad smell instances from the same system. The XML file follows the format exported by the Design Pattern Detection.[32] We decided to use this format because it shows the GoF design pattern instances and details the components that compose each instance. Both XML and CSV format files are described in Design Pattern Smell's website.[35] Moreover, this module allows cleaning the information of the GoF design patterns and bad smells kept in the tool and entering information about GoF design patterns and bad smells from another system.

Data Parser. This module is responsible for performing a static analysis of the input files to identify and extract the software components existing in them. The GoF design pattern instances presented in the XML files may be composed of several components that play a certain role within these instances. For instance, bridge instance contains two kinds of classes. One of them is responsible for representing the abstraction part, and the other class is responsible for representing the implementation part. Then, when this module identifies a bridge instance, it separates these components and extracts the following information about them: name, path, granularity, and the roles played by each component from this design pattern instance. The granularity of the components is divided into package, class,

[‡]http://metrics.sourceforge.net

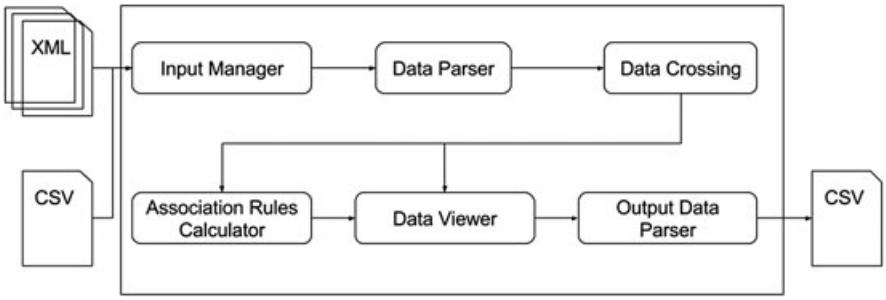

FIGURE 3 Design Pattern Smell's architecture

method, or attribute. Finally, the CSV file is also analyzed to extract the following information of each component: name, path, and granularity. Information extracted in this module is persisted in the memory and used by the other modules.

Data Crossing. This module identifies the cooccurrences between GoF design patterns and bad smells. Cooccurrence is a class or method that makes part of a GoF design pattern instance and contains some bad smell. So, this module analyzes the components extracted from the XML and CSV files. During this analysis, the name and the signature of each GoF design pattern component are checked and compared with the bad smell components to perform matching of them. After this process, the classes and the methods identified in both GoF design pattern and bad smell instances are considered cooccurrences and returned as a list of components.

Association Rules Calculator. This module implements the application of the association rules to identify the intensity of cooccurrences between GoF design patterns and bad smell. Through the data provided by the Design Pattern Smell, this module computes some quantitative information, such as total NOC/NOM regarding to the bad smells, the total NOC/NOM regarding to the design patterns, NOC/NOM with cooccurrence, among others. To compute the rules, Design Pattern Smell uses the information computed by this module to calculate the metrics support,[33] confidence,[33] lift[34] and conviction,[34] and therefore, identify the intensity of the cooccurrences. Finally, as the user provides to the Design Pattern Smell the bad smells and GoF design patterns components precomputed, it does not have access to the total NOC/NOM existing in the analyzed system. This number is important because the association rules metrics uses it. So, to solve this problem, this module requires the user enters the total NOC/NOM existing in the system so that the metrics may be computed and the cooccurrences may be discovered.

Data Viewer. This module allows reporting the results in a data grid view format. From there, the user can navigate in the list of affected components identified with cooccurrence. Also, it is possible to generate other types of reports with the information computed by both the Data Crossing module and the Association Rules Calculator, such as design pattern instances in a system, rate of artifacts affected by cooccurrence, and intensity of cooccurrence identified in each pattern existing in the system.

Output Data Parser. This module parses the reports emitted by the Data Viewer and generates an output CSV file stored in a user-defined location on the user machine. This file contains the same information of the data grid view, and it may be useful to analyze cooccurrences.

3.3 | Implementation

Design Pattern Smell was developed in Java programming language with support of JDK 1.7 and the Java Swing API to create the graphical user interface. We chose Java due to its portability and because it is a popular language both in academia and industry. To parse the input XML files, we used the JDOM API[§] to interpret and manipulate XML data from Java source code. To compute the association rules, we implemented the metrics of association rules: *support*, *confidence*, *lift*, and *conviction*, in Design Pattern Smell and considered only rules from a design pattern to a bad smell. We did not consider rules with more than one design pattern. After calculating the metrics, the tool displays the metrics values for cooccurrence analysis. To present the metrics of association rules, in the Help option, we used the JLaTeXMath 1.0.3 API[¶]

[§]http://www.jdom.org/
[¶]https://forge.scilab.org/index.php/p/jlatexmath/

to show the mathematical formulas used in each of its metrics. Design Pattern Smell was constructed using the Netbeans IDE 8.0.2[#] , which provides drag-and-drop functionality for constructing user interfaces. Design Pattern Smell is available in version 1.0 on the tool's website.[35]

4 | RESULTS

This section presents the study performed in this research and its results. We organized it as follows. Section 4.1 describes the case study carried out in the paper. Section 4.2 presents the results and answers the RQs.

4.1 | Case study

To perform an exploratory analysis, we designed a case study to analyze the presence of cooccurrences between a design pattern and bad smell in oriented-object systems. This case study considered a data set composed of five open-source Java systems. Initially, we collected the design pattern instances by using Design Pattern Detection.[32] Next, we created five filtering expressions based on the detection strategies proposed by Souza[16] and implemented them in RAFTool to collect the bad smell instances.

We manually validated the bad smell instances returned by the detection strategies to remove the inconsistencies and possible imprecisions. We based our analysis on the implementation of the components returned as bad smell instances and made the appropriate corrections in the own CSV file with these components provided by the RAFTool. Moreover, our results could be affected by the presence of false negatives. However, the sample of the bad smell instances identified is large enough to support our conclusions. We summarized the final result of bad smell instances collected in the appendix of this paper (see Section 1).

After collecting the design pattern and bad smell instances, we used Design Pattern Smell to identify the cooccurrences between these two structures. We summarized the number of cooccurrences detected in the appendix of this paper (see Section 2). To identify the intensity of the cooccurrences between the design patterns and bad smells, we applied the association rules in the data summarized in Section 2. We based our analysis on the conviction metric because it measures how important and accurate one given rule is in a data set. Moreover, this metric has been used by previous studies[11,13] to evaluate cooccurrences between design patterns and bad smells. Figures 4, 5, 6, 7 and 8 show the results of conviction metric for the *data class*, *feature envy*, *large class*, *long method*, and *refused bequest* bad smells, respectively. These results are discussed in details in Sections 4.2.1 to 4.2.3. The charts in Figures 4 to 8 aim to show which design patterns had the highest cooccurrences with each one of the bad smells. We created them as ranking charts, where the design patterns are organized from the highest to the lowest cooccurrences. Therefore, the order which design pattern appears in the figures may change.

4.2 | Experimental results

This section aims to answer the RQs. Section 4.2.1 presents the analysis and discusses the arguments used to answer RQ1. Section 4.2.2 presents the answer to RQ2 and summarizes the design patterns that presented cooccurrences with each of the five bad smells used in this study. Section 4.2.3 presents the analysis that has led to the answer of RQ3.

4.2.1 | Research question 1

RQ1: "Do the design patterns defined by the GOF catalog avoid bad smell occurrences in software systems?"

The GoF design patterns investigated in this paper were proposed and validated by Gamma et al.[1] They have a modular structure, and its usage is highly encouraged, since they help to design software with low internal coupling, and they favor the creation of flexible and reusable software.

These solutions are highly encouraged by developers and researchers, and they bring several benefits to the internal quality of software. However, it is necessary to have attention to apply them. They are available in the literature as a general template. To be used, the developers need to adapt them so that these solutions fit in the problem context to be solved. Sometimes, when the adaptation process of these solutions is performed, complex structures may be inserted in the internal structure of the software, degrading the design patterns and resulting in bad smells.

[#] https://netbeans.org/

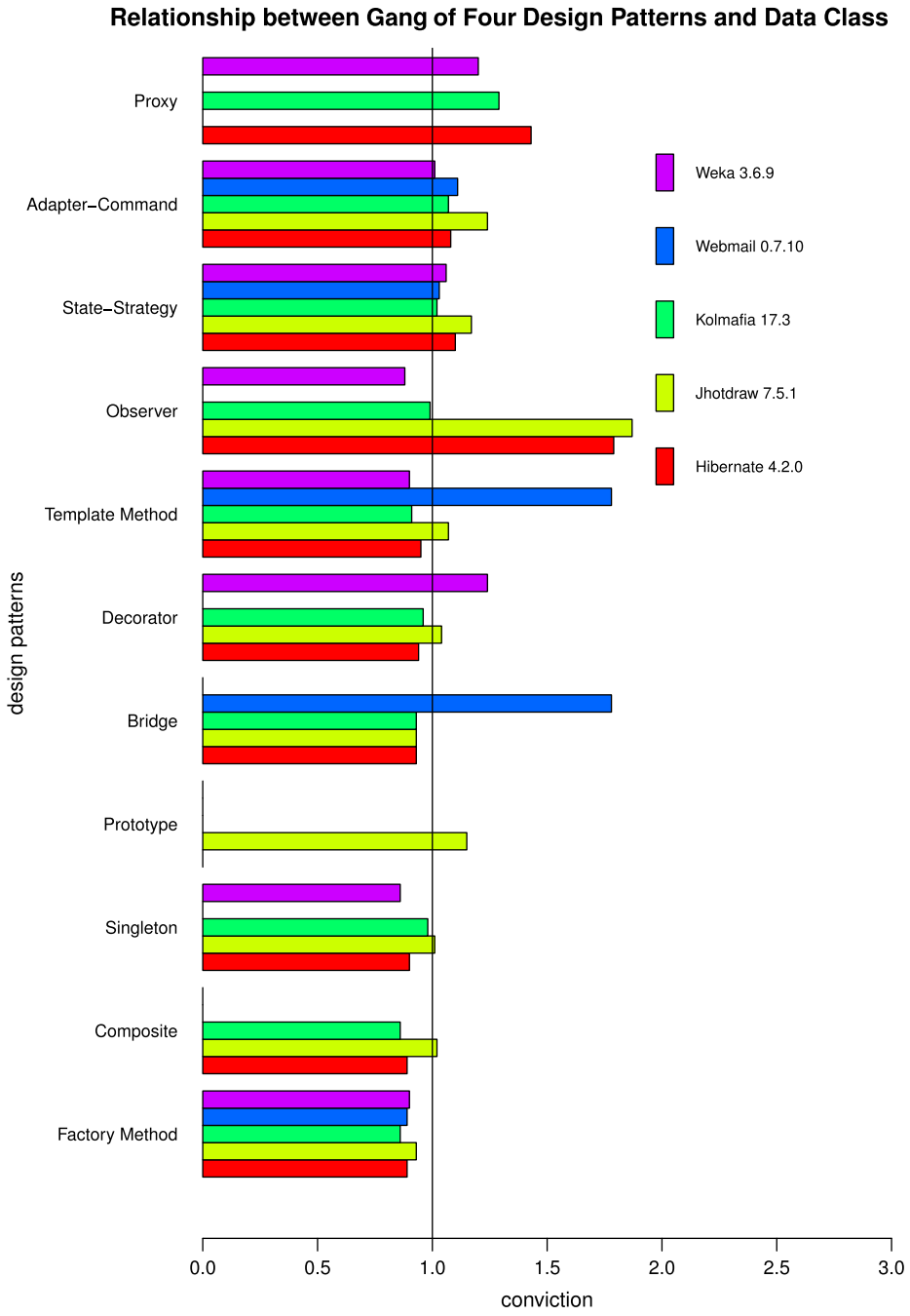

FIGURE 4 Result of the association (Design Pattern ⇒ Data Class) [Colour figure can be viewed at wileyonlinelibrary.com]

As we described in Section 2.5, we used the association rules to evaluate the intensity of the cooccurrences between the GoF design patterns with bad smells and focused our analysis on the conviction metric. When the value of the conviction metric is less than 1.0, it indicates that if the antecedent occurs, the consequent tends not to happen, ie, conviction values less than 1.0 suggest that the use of a given design pattern tends to avoid the occurrence of a given bad smell. In contrast, when the value of the conviction metric is higher than 1.0, it indicates that there is a positive relationship between antecedent and consequent, and the higher the conviction value, the higher the relation between antecedent and consequent. Therefore, the number of conviction metric greater than 1.0 indicates that there is a relationship between a given design pattern and a given bad smell. Finally, the conviction value equal to 1.0 means that the antecedent and the consequent are independent.

Analyzing Figures 4 to 8, we identified two design patterns that presented low cooccurrence with the bad smells considered in this study: composite and factory method. We concluded this because for most of the systems, each bad smell

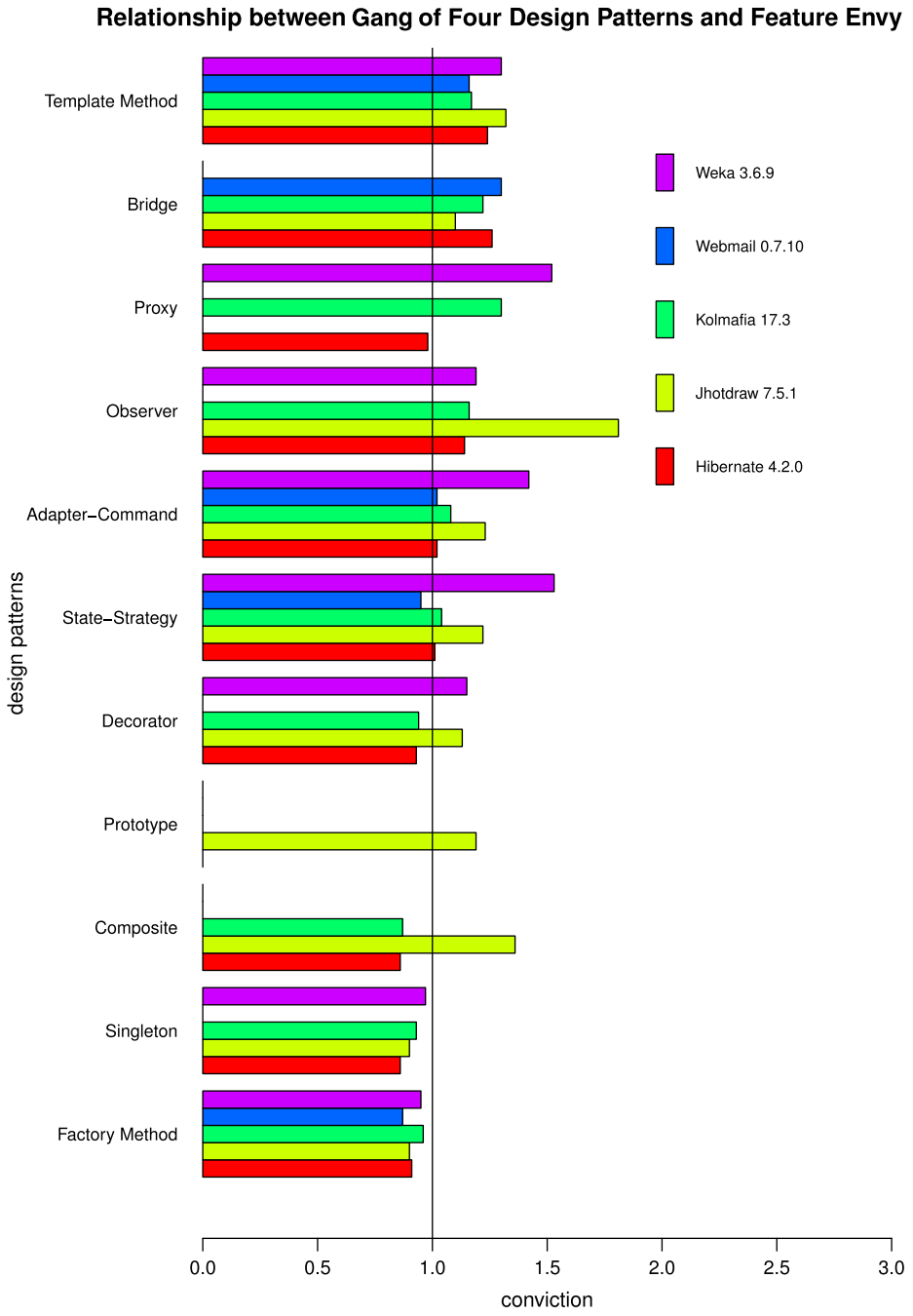

FIGURE 5 Results of the association (Design Pattern ⇒ Feature Envy) [Colour figure can be viewed at wileyonlinelibrary.com]

presented a conviction value less than 1.0. We inspected the Composite and Factory Method instances returned in this study to identify the reasons why these two design patterns presented low cooccurrences. The idea of the composite design pattern is to build complex objects via simpler objects. These simpler objects are defined in modules so that the intelligence of the object is divided between them, reducing the complexity of classes. The Factory Method design pattern simulates the idea of a factory where there is an interface for creating objects, but the creation itself is performed by the subclass that implements such interface. Thus, it is possible to create several modules, each one responsible for creating and managing the information of a set of objects in the system, removing the workload from a single class. The composite and factory method design patterns did not present high cooccurrence with bad smells because they allow creating complex objects from other smaller and simpler objects. In this way, they reduce the coupling and complexity of the system.

Among the five bad smells evaluated, singleton presented low cooccurrence with three of them: data class, feature envy, and large class. This result suggests that singleton may be a good choice when it is desired to avoid occurrences of bad

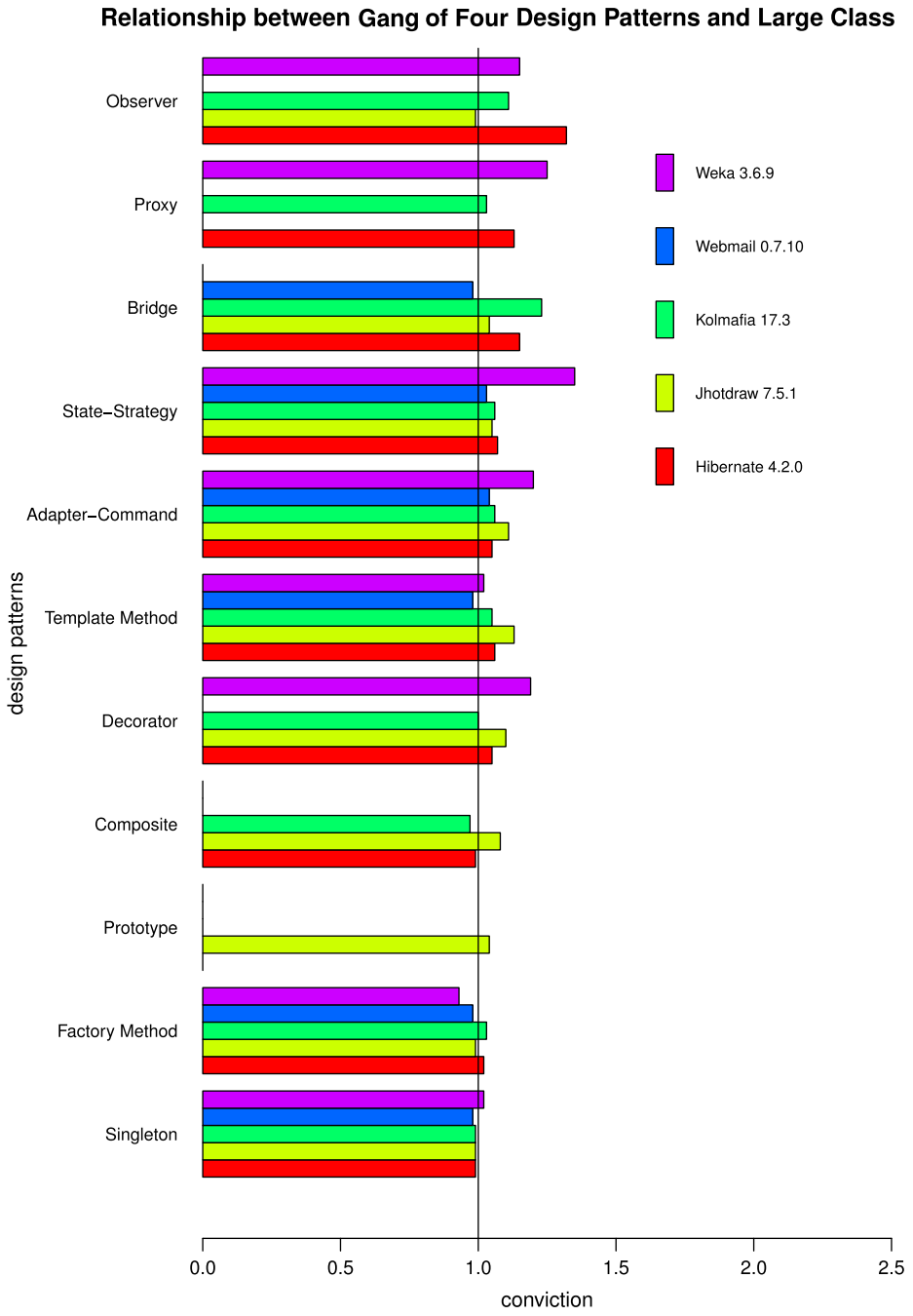

FIGURE 6 Result of the association (Design Pattern ⇒ Large Class) [Colour figure can be viewed at wileyonlinelibrary.com]

smells referring to the class-level complexity in software systems. In relation to long method bad smell, singleton is a particular case. Although the result displayed in Figure 7 shows that there is a low cooccurrence of this design pattern with long method, it is considered a false negative, because the instances of this design pattern identified by Design Pattern Detection are based only on the static attribute presented in the class. This tool does not consider any method as characteristic of this design pattern. For this reason, when we performed the matching of the design pattern and the bad smell information, it returned zero. However, when singleton classes were manually inspected, we found some instances of long method within their classes.

Summary of RQ1. Although the factory method, composite and singleton design patterns have presented low cooccurrence with the bad smells studied, we identified that most of GoF design patterns are associated with the bad smells considered in this study. Therefore, in response to RQ1, we conclude that the GoF design patterns studied in this work do not necessarily avoid bad smell occurrences.

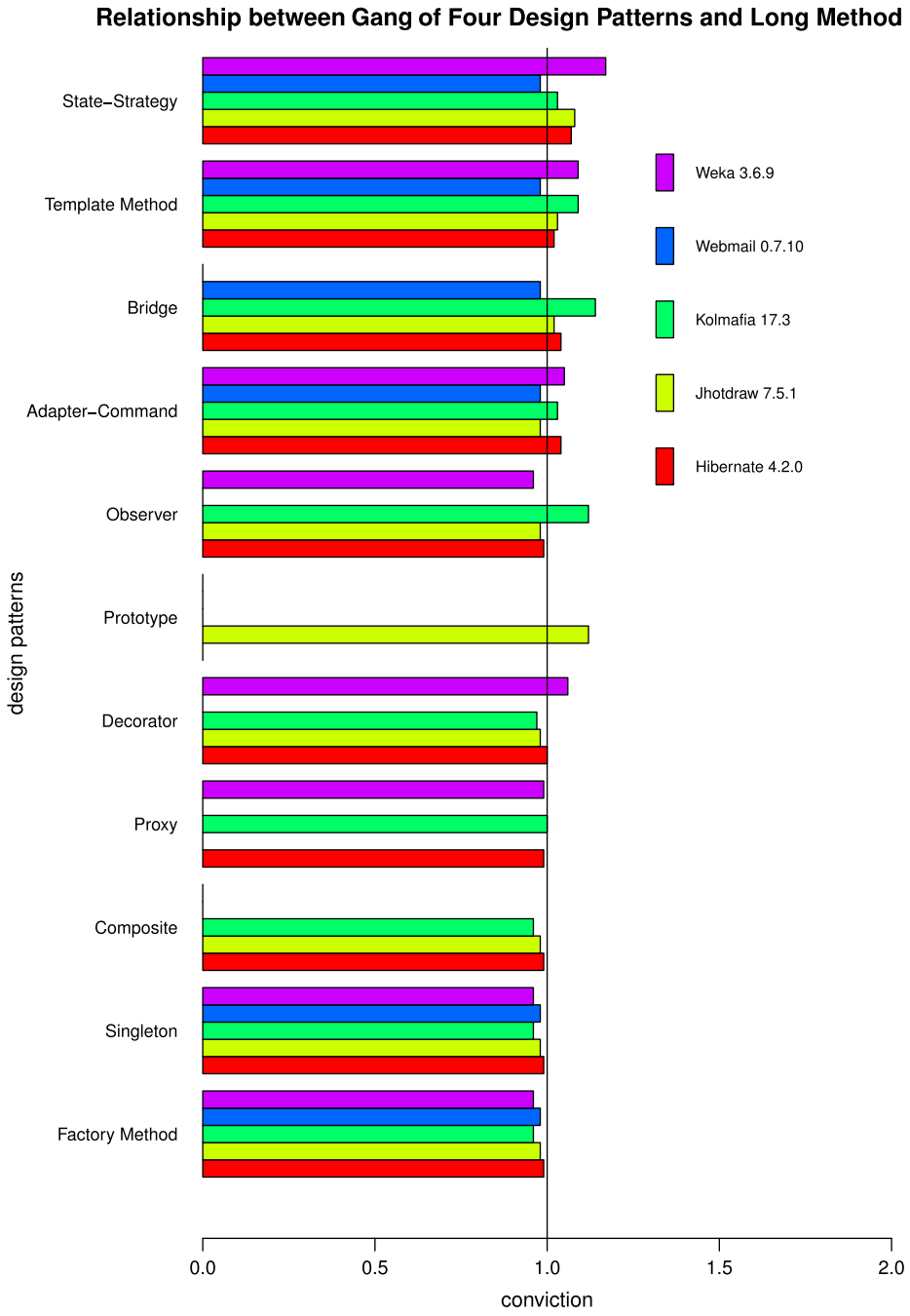

FIGURE 7 Result of the association (Design Pattern ⇒ Long Method) [Colour figure can be viewed at wileyonlinelibrary.com]

4.2.2 | Research question 2

The analysis performed in this section aims to answer the RQ2: "Which design patterns of the GOF catalog have cooccurrence with bad smells?"

To analyze the associations between design patterns and bad smells, we considered the conviction values. The choice of this metric occurred because this metric was able to establish a relationship between the support and confidence metrics. In addition, conviction has the best sensibility in identifying the cooccurrences relationship between antecedent and consequent. Thus, to identify the cooccurrence relationship between design patterns and bad smells, we considered the thresholds of the conviction metric, mentioned in Section 2.5. The criterion used to identify cooccurrences consists of observing prevalent cases, that is, cases in which design patterns presented a conviction value greater than 1.0 for most of the systems that had instances of the respective design patterns.

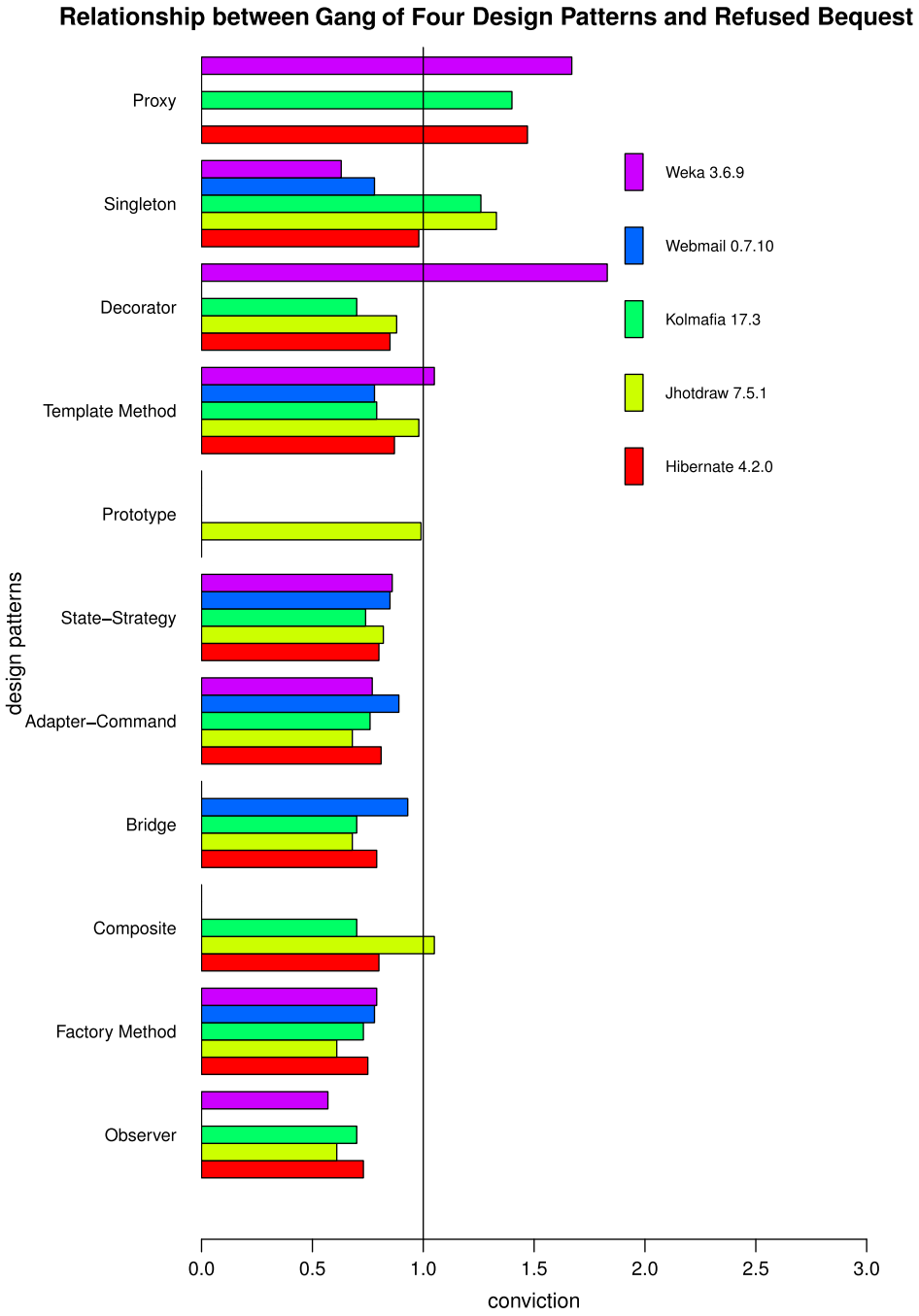

FIGURE 8 Result of the association (Design Pattern ⇒ Refused Bequest) [Colour figure can be viewed at wileyonlinelibrary.com]

Analyzing the results of Figure 4, we noticed that proxy, adapter-command, and state-strategy design patterns were the ones that presented the highest cooccurrence relationship with the data class bad smell. The observer design pattern had a high cooccurrence rate in only two systems, JHotDraw and Hibernate. However, by the criteria used to identify cooccurrences, it is not possible to state that the observer design pattern presented cooccurrence with data class. The template method and bridge also had high cooccurrence in a single system, Webmail, but in general, the cooccurrence relationship was not very intense like for proxy, adapter-command, and state-strategy. Therefore, these results suggest that the proxy, adapter-command, and state-strategy were those that presented the highest cooccurrence relationship with the data class bad smell.

Figure 5 shows a chart with the results of the conviction metric calculated for the feature envy bad smell. Analyzing this chart, we noticed that several design patterns have a cooccurrence relationship with this bad smell. Among these relationships, it is possible to highlight six design patterns with high cooccurrence: template method, bridge, proxy, observer,

TABLE 6 Summarization of identified cooccurrences

Bad Smell	Cooccurrence Identified with the Design Patterns
Data class	Proxy Adapter-comand State-strategy
Feature envy	Template method Bridge Proxy Observer Adapter-command State-strategy
Large class	Observer Proxy State-strategy Adapter-command Bridge Template method Decorator
Long method	State-strategy Template method Adapter-command Bridge
Refused bequest	Proxy Singleton

adapter-command, and state-strategy. However, the `Template Method ⇒ Feature Envy` relationship was the one that presented a higher intensity in this study, since for all the systems of the data set used, it was the design pattern that presented higher conviction Values.

Figure 6 displays a chart with the results of the conviction metric calculated for the large class bad smel. Analyzing this chart, we noticed that, as well as for the feature envy bad smell, there are several design patterns with strong cooccurrence. Among the relationships displayed, it is possible to list seven design patterns that presented cooccurrences: observer, proxy, state-strategy, adapter-command, bridge, template method, and decorator. However, the `Observer ⇒ Large Class` relationship was the one that presented a greater intensity compared with the other cooccurrences.

Figure 7 presents the results obtained the long method bad smell, via a chart with the conviction metric. In this chart, we may see that the design patterns did not present a relationship as high as for the other bad smells. The design patterns that presented the highest cooccurrence with the long method bad smells were: state-strategy, template method, bridge, and adapter-command. However, when comparing the cooccurrences intensity, we observed that the `State-Strategy ⇒ Long Method` was the one that presented the highest intensity.

Finally, the results obtained for the refused bequest bad smell (see Figure 8) suggest that two design patterns, ie, proxy and singleton, presented the highest cooccurrence with this bad smell. The decorator design pattern had a high conviction value for a single system, Weka. However, for the other systems, this value remained low, and in general, the cooccurrence of this design pattern was not as intense as the cooccurrence obtained for proxy and singleton.

Summary of RQ2. The GOF design patterns that presented cooccurrence with bad smells are adapter-command, bridge, decorator, observer, proxy, singleton, state-strategy, and template method. The cooccurrences identified in this study are summarized in Table 6.

4.2.3 | Research question 3

The analysis performed in this section aims to answer RQ3: "What are the more common situations in which bad smells appear in software systems that apply GOF design patterns?" For each bad smell, we discuss an example of cooccurrence and present a class diagram to support our discussion. The class diagrams are presented in Section 3 of the Appendix. The following examples are explained.

Data class. The results found in this paper indicate that proxy, adapter-command, and state-strategy design patterns were those that presented the highest cooccurrence relationship with the data class bad smell. To identify the reason

for such cooccurrences, we performed a manual inspection into the classes that had the `Proxy ⇒ Data Class` relationship.

The proxy design pattern is a solution whose purpose is to provide a substitute or marker for other object's location to control access to itself. This solution is composed by a set of classes in which the class that plays the *Subject* role is responsible for standardizing the access interface for the *RealSubject* and *Proxy* classes. The *RealSubject* class represents the real object over which the *Proxy* class will exercise an access control. The *Proxy* class is the one responsible for controlling access to an object, which in this case is *RealSubject*.

When analyzing the classes that presented `Proxy ⇒ Data Class` relationship, we noticed that all classes with cooccurrence played the *RealSubject* role. Moreover, we performed a manual inspection in these classes to identify situations that led to the emergence of this relationship. We found that these classes have a large number of attributes and many occurrences of gets and sets methods for assigning and accessing the values stored in these classes, configuring the symbolic role of a database within the systems.

Figure C1 shows a class diagram extracted from Hibernate, containing a class that presented the `Proxy ⇒ Data Class` cooccurrence. The design pattern has one class, *FromReferenceNode*, which plays the *Proxy* role, and another responsible for playing the *RealSubject* role, *FromElement*. In this diagram, the *FromElement* class has a large amount of data, characterizing the occurrence of data class bad smell. Based on this, we may conclude that the implementation of *RealSubject* class within the proxy design pattern was the factor that contributed to the emergence of `Proxy ⇒ Data Class` cooccurrence.

Summary. The main situation that contributes to the emergence of `Proxy ⇒ Data Class` cooccurrence is the implementation of the *RealSubject* class within the `Proxy` design pattern, maintaining a large amount of data and few functionalities over the data

Feature envy. We found six design patterns that presented cooccurrence with feature envy. To identify the reason for this relationship, we performed a manual inspection in the classes with the `Template Method ⇒ Feature Envy` cooccurrence, since it was the highest intensity relationship for this bad smell.

The template method is a solution that defines the skeleton of an algorithm via an operation, transferring some steps to the subclasses, which have the power to redefine the characteristics of this algorithm without changing its structure. This design pattern consists of a set of classes. One of them is the *AbstractClass* class, which defines the primitive and generic operations for all subclasses. In the *AbstractClass*, a *template* method is defined to implement the skeleton of the desired algorithm. The other classes in this design pattern are named *ConcreteClass*. They are subclasses of the *AbstractClass* and are responsible for redefining the characteristics of an object, using the template method defined in the *AbstractClass*. This design pattern uses a modular structure, in which the behavior of an object is modeled in subclasses and assigned to it via polymorphism. The advantage of this implementation is the reduction of the complexity in the superclass, since the definitions of conditional structures like `if` and `switch` may be replaced by polymorphism.

When analyzing the classes that presented the `Template Method ⇒ Feature Envy`, we noticed that the classes with this cooccurrence play the *AbstractClass* role within this design pattern. Besides, some *template* methods in these classes contain a high amount of access to methods allocated to other classes, increasing the coupling level of *AbstractClass* and reducing its cohesion. Therefore, we may conclude that the implementation of these methods in inappropriate classes was the central situation that contributed to the emergence of this characteristic. A solution to this problem is to apply methods extraction refactoring in the template methods and to reallocate the implementation of an "envious" characteristic to its respective class. Thus, it seems that these cooccurrences could be mitigated in the systems.

Figure C2 shows a class diagram extracted from Webmail, containing a class that presented the `Template Method ⇒ Feature Envy` cooccurrence. In this diagram, the *WebmailServer* class is responsible for playing the *AbstractClass* role and making the *template* methods with a predefined algorithm. This class has three kinds of *template* methods: *doInit*, *restart*, and *shutdown*. The *WebmailServlet* class plays the *ConcreteClass* role, which is free to define a specific feature for the algorithm defined in the *AbstractClasstemplate* methods. However, during the manual inspection, we observed that two out of three *template* methods, *restart* and *shutdown* methods, make use only of features related to the storage object instantiated in this same class. Due to these facts, it seems that the implementation of these methods contributed to the emergence of feature envy bad smell in this class, since they should be allocated within the class referring to the used object, *Storage*.

Summary. The main situation that contributed to the emergence of `Template Method ⇒ Feature Envy` was the implementation of methods accessed by the *template* method outside of the *AbstractClass* class. Such methods

implement the feature of the same concern of the *AbstractClass* class and, therefore, should have been implemented within that class.

Large class. Regarding large class, seven design patterns presented cooccurrence with this bad smell: observer, proxy, bridge, state-strategy, adapter-command, template method, and decorator. To identify the situation that caused this relationship, we performed a manual inspection in the classes that presented the `Observer ⇒ Large Class` cooccurrence.

The observer design pattern is a solution that defines a one-to-many dependency between objects. This design pattern is composed of a set of classes, where each one is responsible for playing the *Subject* role and storing information that is used by the *Observer* classes. The *Subject* class of this design pattern has a list of all *Observer* classes that use its data. When some information is changed by any *Observer* class, the *Subject* class is triggered, changing the other *Observer* objects. The purpose of this design pattern is to synchronize data and update objects in real time. This updating occurs via polymorphism of inclusion, avoiding the increase in the complexity that usually occurs with the use of conditional structures.

When analyzing the classes that presented the `Observer ⇒ Large Class`, we noticed that all components with cooccurrence played the *Subject* role within the design pattern. Moreover, during the manual inspection, it was observed that the inappropriate implementation of this class increased its complexity and, consequently, implied the emergence of this cooccurrence.

Figure C3 shows a class diagram extracted from Weka, containing a class that presented the `Observer ⇒ Large Class`. Analyzing the class diagram in Figure C3, we observed that the class responsible for playing the *Subject* role, represented by the *Classifier* class, within the `Observer` design pattern, has an inappropriate implementation. Moreover, due to a large amount of data, we identified through the class diagram that it works with several kinds of observers which are likely to have different responsibilities. Therefore, the best practice, in this case, would be the creation of several *Subject* components, one for each concern, and connect them with their respective observers. This action would avoid several concerns from being dealt with by a single *Subject* and would reduce the complexity of these classes. For instance, a *Subject* should be implemented to interact with *GraphListener*, another to interact with *IncrementerClassifierListener*, and so on. The notification methods and the attributes and methods used by each of the observers must be extracted to others classes, implying the application of a class extraction refactoring.

Summary. The main situation that contributed to the emergence of `Observer ⇒ Large Class` was the inappropriate implementation of the *Subject* component within the `Observer` design pattern, which increased the complexity of the *Subject* class and generated the occurrence of the Large Class bad smell with the observer design pattern.

Long method. For the long method, we identified two design patterns, state-strategy and template method, which presented a high cooccurrence rate. Therefore, we manually inspected the `State-Strategy ⇒ Long Method` relationship to identify the situations that caused these cooccurrences.

Considering that Design Pattern Detection does not separate the strategy and state design patterns, initially, it was not possible to know whether one of them or both cooccurred with the long method bad smell. However, when we performed the manual inspection in the systems' source code, we identified that the strategy design pattern appears much more frequently than state design pattern; hence, it has a higher cooccurrence rate with the long method bad smell.

Gamma et al[1] propose the strategy design pattern with the purpose of defining a family of algorithms, encapsulating each one and making them interchangeable. This design pattern allows the algorithm to vary regardless the clients that use it. Strategy is composed of a set of classes in such a way that each one is responsible for playing the strategy role and defining a common interface for the family of algorithms so that the *Context* class may use them. The *ConcreteStrategy* class is responsible for implementing each of the algorithms defined as "strategy." The *Context* class is configured as a *ConcreteStrategy* object, and it is responsible for passing requests from its clients to the configured "strategy." The advantage of this solution is the flexibility provided by the partitioning of the business rules of a system into small components. Moreover, in the case of inclusion of new rules or algorithms, this solution allows them to be added easily without changing the source code.

When analyzing the methods that presented the `Strategy ⇒ Long Method` relationship, we found that all of them were located inside the *Context* classes, in the same place the strategies to be used by the clients were defined. These methods have an excess of code because the strategies were determined by conditional structures rather than polymorphism. This practice generated high-complexity methods and made the code difficult to read and understand.

Figure C4 shows a class diagram extracted from Hibernate, containing an instance of the Strategy design pattern with long method cooccurrence. The *Mappings* and *ExtendedMappings* interfaces play the *Strategy* role and define a common

service interface for all the strategies. The *MappingsImpl* class plays the *ConcreteStrategy* role and implements the services defined by the *Strategy* interfaces, according to the strategy assigned to this class. The *SimpleValueBinder* class plays the *Context* role and implements the strategies defined by the client. However, *SimpleValueBinder* has a long method bad smell. The *setType* method of the *SimpleValueBinder* class, responsible for implementing a strategy requested by the client, is an instance of this occurrence. This method has a large amount of code and uses conditional structures to define implementation choices. The best practice, in this case, would be to create several smaller components, one for each conditional structure defined within the method, with a default interface, and instantiate them within the method as needed. This practice would make the code less complex and improve its readability and understanding.

Summary. The main situation that contributes to the emergence of `Strategy ⇒ Long Method` cooccurrence is the code excess implemented in the methods that define the strategies. This excess is due to the excessive use of conditional structures used to define the steps and behaviors performed by each of the strategy algorithms. This code excess impaired the readability and understanding of the source code.

Refused bequest. We identified two design patterns, proxy and singleton, that presented the highest cooccurrence rate with refused bequest. So, we manually inspected the `Proxy ⇒ Refused Bequest` relationship to identify some situations that caused these cooccurrences.

When analyzing the classes that presented the `Proxy ⇒ Refused Bequest` relationship, we noticed that most of the classes that introduced cooccurrence played the *Proxy* role in the design pattern. Furthermore, we identified two main situations that caused the emergence of this relationship. The first one was the use of inheritance as a reuse mechanism. Inheritance is a mechanism that should only be used when the superclass and the subclass have an "is a" relationship. However, several classes of the systems analyzed apply inheritance to reuse codes already implemented in other classes, without an "is a" relationship between the classes. The second situation identified was the inappropriate use of inheritance between classes. When this resource is not used correctly, the class in the lowest level of the hierarchy begins to overwrite a large number of services from its parent classes, something that should not happen when using this mechanism.

Based on this, the main solution to remove cooccurrences in these classes would be the application of refactoring, changing the inheritance relationship by the composition relationship. This action would be a good practice for reducing method overriding and rejection of features defined in the superclass.

Figure C5 shows a class diagram extracted from Kolmafia, containing an instance of proxy design pattern that presented refused bequest bad smell. In this diagram, the *Value* class plays the *RealSubject* role inside the design pattern. The *ForEachLoop* class plays the *Proxy* role. By analyzing the class diagram, we may see that the *ForEachLoop* class is positioned at the lowest level of the inheritance hierarchy. During the manual inspection, we noticed that this class is positioned at the wrong level. This class overrides a method already implemented by its parent class, *Loop*, thus rejecting an already defined feature. This class should be refactored and moved to the same level of the *Loop* class in this hierarchy, ie, it should have a direct inheritance relationship with the *ParseTreeNode* class, since it uses features only of that class and implements a method that is only defined as abstract by this class.

Summary. The main situation that contributed to the emergence of `Proxy ⇒ Refused Bequest` cooccurrence is the inappropriate use of the inheritance, especially when components are located at an inappropriate level in the inheritance hierarchy. Such situations have contributed to components rejecting features provided by their superclasses.

5 | LESSONS LEARNED

Analyzing the results, we verified that the application of design pattern in systems does not necessarily avoid bad smell cooccurrences. However, some design patterns had a low relationship with the bad smells evaluated in this study. They are factory method, composite, and singleton. Factory method and composite design patterns are expected to exhibit low cooccurrence, since they are intrinsically modular design patterns. However, it was a surprise to see the results indicating that the *singleton* design pattern presents low cooccurring with bad smells when dealing with large complex data. Singleto's main purpose is to provide a unique global point of access for one class. According to the previous work, this design pattern tends to centralize the intelligence of the system and consequently to increase the complexity of its internal components generating bad smell occurrences.[36] The results obtained in the present study contradict the results of Vokac.[36]

Moreover, we extracted a lot of cooccurrences between the GoF design patterns and bad smells. Analyzing these relationships (see Table 6), we observed that proxy, adapter-command, and state-strategy were those that were most frequent in the relationship identified for each bad smell. They presented cooccurrences with four out of five bad smells considered in this study, and for data class and refused bequest, the design pattern that presented the highest cooccurrence with them was proxy. Bridge and TEMPLATE METHOD APPEAR IN THE SECOND PLACE, presenting cooccurrence with three out of five bad smells.

Finally, each relationship of higher cooccurrence with each of the bad smells was manually inspected to identify situations that impacted on the emergence of this relationship. Although these situations are case-specific, we concluded they arose due to misuse and inappropriate implementation of the design patterns in the analyzed systems. Therefore, using design patterns requires special attention and proper planning of them to avoid bad smells occurrences.

6 | THREATS TO VALIDITY

This section presents the threats to validity according to the guidelines proposed by Wohlin et al.[37] We discuss threats to external, internal, and construct validity.

External validity. According to Wohlin et al,[37] external validity is "concerned with to what extent it is possible to generalize the findings and to what extent the findings are of interest to other people outside the investigated case." We presented a study carried out with a data set composed of five open-source Java systems. From these five systems, four of them were extracted from a large data set, called Qualitas Corpus. Our sample of systems has small, medium, and large systems. However, due to the small size of the sample, we are not able to generalize the results found in this study. Nevertheless, the results obtained are important because they show that the use of design patterns do not necessarily avoid bad smells in object-oriented systems.

Internal validity. This kind of validity is of concern when causal relations are examined, ie, whether there is a risk of some factor to affect in the investigation of a causal relation between two variable of the experiment.[37] Our data collection was carried out by tools. To identify the methods and classes that compose the design patterns instances, we used Design Pattern Detection. To identify the methods and classes that have bad smell occurrences, we used detection strategies and implemented them in RAFTool. Finally, to identify the cooccurrences, we used the Design Pattern Smell.[38] Although all these tools have been evaluated and presented good results, we are not able to ensure that their results are error free. However, to mitigate these threats, we chose tools that have good accuracy in detecting design pattern and bad smells. Moreover, we manually analyzed the results obtained by these tools and removed the false positives based on our knowledge of bad smells and design patterns.

Construct validity. It refers to the extent to which the experiment setting reflects the theory that the researcher has in mind.[37] To identify the situations that caused the emergence of cooccurrence between design patterns and bad smells, we manually inspected the classes and methods involved in such cooccurrences. This inspection was carried out by one of the authors of this paper, and the results were discussed among the other authors. Although the inspector has a 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 number of systems in this work.

7 | RELATED WORK

Several studies have been developed to investigate the relationship between design patterns and bad smells. We previously conducted a systematic literature mapping[15] and found that the literature has approached this topic in three different ways: impact on software quality, refactoring, and cooccurrence. Moreover, we concluded that cooccurrence between design patterns and bad smells is a current topic and have been little explored. In this section, we provide an overview of the studies identified in this systematic literature mapping, highlighting the main differences between them and the work we presented in this paper.

7.1 | Empirical studies on design patterns

Since design patterns were proposed, several studies have investigated the effectiveness of these solutions regarding the software quality. For instance, Wendorff[39] analyzed a commercial system developed in C++ language to find negative

impacts provided by the design pattern in the development context. He concluded that the lack of logic comprehension of the design patterns implementation is one of the main factors that lead developers to misuse them. Moreover, he considered that the application of unneeded design pattern and requirements portability impact the internal software structure in a negative manner.

In the same line, McNatt and Bieman[40] and Izurieta and Bieman[41] also identified some other factors that negatively impact on the design patterns quality. McNatt and Bieman found that occurrences of design patterns coupling, ie, design patterns with the same component in their instances hinder the modularization of these solutions and make the system hard to modify since they increase the coupling in the system. Izurieta and Bieman investigated how design patterns behaved during the evolution of systems and analyzed whether they maintain the software structure flexible and easy to maintain. They concluded that the design pattern quality tends to deteriorate during the software evolution and pointed out the accumulation of components unrelated to classes that play roles in design patterns as the main reason that impairs the design pattern quality.

Khomh and Gueheneuce[42] carried out a study more specific than Wendorff,[39] McNatt and Bieman,[40] and Izurieta and Bieman.[41] They evaluated the impact of the GoF design patterns on some external quality attributes, such as reuse, comprehension, and modularity, among others. Through this study, they identified that the GoF design pattern do not always improve the software quality and pointed out that Flyweight negatively affects all attributes, except scalability. Wagey et al[43] also investigated the impact of design patterns on quality external attributes, but they focused only on the maintainability. They proposed a quality model based on design patterns and, through it, they identified that these solutions positively affect the software maintainability and the higher the use of the design pattern in software, the better its internal structure and the greater its maintainability, reducing part of the costs in the software maintenance phase.

Vokac[36] analyzed the corrective maintenance of a large commercial software system to identify possible defects generated in software due to the design pattern application. The author discovered that the observer and singleton design patterns are the ones that are correlated with the highest defect rate. In contrast, the factory method design pattern was identified as the least prone to errors because the classes that play the factory role are more compact and less coupled. Finally, the template method design pattern did not present any clear trend for the occurrence of the defects in the study of Vokac.

The previous studies presented in this section investigated the impact of design patterns on software quality. In summary, Werdoff,[39] McNatt and Bieman,[40] and Izurieta and Bieman[41] analyzed negative impacts caused by these solution both in software quality and software evolution. Khomh and Gueheneuce,[42] and Wagey et al[43] investigated whether design patterns improve the external attributes of the software systems and what is improved. Finally, Vokac[36] analyzed whether GoF design patterns may cause defects and what of them is the most prone to present defects. The present study also evaluated the impact of design pattern on software quality. However, the main goal of this study was to analyze whether design patterns might cooccur with bad smells, and what situations might contribute to the cooccurrence of these two structures.

7.2 | Cooccurrence

The investigation of cooccurrence between design patterns and bad smell is a current research topic on software engineering, and it has been explored since 2013.[15] As it is a recent topic, few studies in the literature have approached this theme. In our previous study, we identified only four studies available on the research on this topic. Therefore, in this section, we present these papers and discuss the main differences between them and the study we have presented in this paper.

Cardoso and Figueiredo[11] performed an exploratory study to identify bad smells cooccurrences in systems that apply design patterns. The authors considered the god class and duplicate code bad smells and 11 of 23 of GOF design patterns. As results, Cardoso and Figueiredo[11] identified the cooccurrence of command design pattern with the god class bad smell and the template method design pattern with the duplicate code bad smell. They analyzed the components that presented cooccurrence and identified that in the case of the command design pattern, the use of a single receiving class for different concerns caused the God Class emergence. For the template method design pattern, the several duplications of implementation within the design pattern were responsible for the appearance of duplicate code bad smell.

Jaafar et al[10] investigated the evolution of three open-source Java systems to identify the existence of the static relationship between a design pattern and antipatterns and to evaluate the impact and the behavior of these relationships during the evolution of the systems. They discovered that the relationship existing between these two structures are not casual since they are continually growing during the software evolution. The command design pattern was identified as those that presented the highest static relationship with antipatterns. Finally, Jaafar et al[10] concluded that classes representing

a static relationship between design patterns and antipatterns are more likely to change and less likely to fail than classes that do not participate in such relationships.

Jaafar et al[12] analyzed the impact of static and cochange dependency in classes with design patterns and bad smells, and verify the relationship of these dependencies with software failures. Cochange dependency consists of changes made in one class that directly impact another. Jaafar et al[12] observed the evolution of three open-source Java systems and concluded that classes that have a static and cochange relationship with design patterns or bad smells have significantly more failures and, consequently, are more likely to structural changes and code addition.

Walter and Alkhaeir[13] performed an exploratory study to determine and investigate whether the presence of design patterns is related to the presence of bad smells and whether these relationships change during the code evolution. They evaluated the evolution of two open-source Java systems and identified that the presence of the design pattern is related to the absence of bad smells in the same classes. Moreover, some design patterns as adapter, command, factory method, state, strategy, and singleton were pointed out as those that are more likely to not present bad smell occurrence. On the other hand, the composite design pattern was pointed out as prone to bad smell occurrences.

In a preliminary study conducted with the god class and long method bad smells,[14] we identified indications that GoF design patterns are not able to avoid bad smells. In the present study, we expanded our sample of bad smells to determine other types of existing cooccurrences. This paper differs from the research of Cardoso and Figueiredo[11] and of Jaafar et al[10,12] because we investigated cooccurrence between GoF design patterns with other kinds of bad smells. Moreover, we applied a different approach regarding the reported studies to detect the bad smells. We used detection strategies to identify bad smells that were previously proposed and evaluated by Souza.[16] Regarding the study carried out by Walter and Alkhaeir,[13] the present work differs from it because we investigated some different kind of bad smells; moreover, we achieved insights not obtained by them.

8 | CONCLUSION

In this study, we performed an evaluation of object-oriented systems with the purpose of: (1) investigating whether the use of GOF design pattern avoid cooccurrences of bad smells, (2) identifying possible design patterns that present cooccurrences with bad smell, and (3) identifying situations present in the source code that led to the emergence of these cooccurrences. This study considered a sample of five open-source Java systems, 14 GoF catalog design patterns, and five bad smells described by Fowler and Beck.[2]

We applied the detection strategies proposed by Souza[16] to identify the following bad smells: data class, feature envy, large class, long method, and refused bequest. The detection strategies contain well-known software metrics and were previously evaluated.

The results of this study indicate that the use of GoF design pattern does not necessarily avoid bad smells occurrence. However, we identified that some design patterns as composite, factory method, and singleton presented a low cooccurrence rate with the bad smells because they have a modular structure that allows dividing the intelligence of the systems in several classes. For this reason, these design patterns may be a good choice for the creation of systems with good internal quality and flexible structure. In contrast, several design patterns presented cooccurrence with the bad smells considered in this work. The cooccurrences identified in this study are summarized in Table 6. Analyzing these relationships, we perceived that the adapter-command, proxy and state-strategy design patterns are those that have a higher frequency of cooccurrences with bad smells. This insight indicates that these solutions need special attention to avoid the occurrence of bad smells.

We collected the data of design patterns, bad smells, and cooccurrences from the systems to analyze the intensity of cooccurrences between the GoF design patterns and bad smells in the software, via association rules. Finally, we performed a manual inspection of the components that presented these cooccurrences and identified the main situations that led to the emergence of these relationships within the systems source code. Although the situations analyzed in this manual inspection are specific to each case, we noted some similarities between them, such as classes with a lot of responsibilities and tasks, complex methods, and excessive repetition of code. Thus, it is possible to point out that these cooccurrences are due to poor planning and inadequate implementation of these design patterns. Other possible reasons are the lack of experience of the developers with these solutions and excessive system maintenance. These results indicate that special attention in the implementation of design pattern and better planning of the software design are needed to avoid the occurrence of bad smells.

Based on the main results, we may summarize the findings as follows:

- The use of design patterns does not avoid the bad smell occurrences.
- The composite, factory method, and singleton design patterns are intrinsically modular; thus, they presented a low rate of cooccurrences with bad smells.
- The adapter-command, proxy and state-strategy design patterns appeared most frequently related to the five explored bad smells, and for this reason, the application of these design patterns demand special attention.
- Poor planning and inadequate implementation of design patterns were the main situations identified that contributed to the emergence of cooccurrences with bad smells.

This study should help the software engineering community to comprehend better the internal structure of software systems that apply design patterns. As future work, we suggest to extend this research to a more significant amount of software sample, investigate design patterns cooccurrences with other bad smells, and examine the relationships of design patterns, bad smells, and software failures.

ACKNOWLEDGEMENTS

This work was supported in part by CAPES, UFMG Programming Language Research Group, UFMG Graduate Program in Computer Science, and CEFET-MG.

ORCID

Bruno L. Sousa https://orcid.org/0000-0002-8217-3524

REFERENCES

1. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley Longman Publishing Co; 1995.
2. Fowler M, Beck K. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley; 1999.
3. Speicher D. Code quality cultivation. In: *Knowledge Discovery, Knowledge Engineering and Knowledge Management: Third International Joint Conference, IC3K 2011, Paris, France, October 26-29, 2011. Revised Selected Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2013:334-349. *Communications in Computer and Information Science*; vol. 348.
4. Kerievsky J. *Refactoring to Patterns*. Munich, Germany: Pearson Higher Education; 2004.
5. Christopoulou A, Giakoumakis EA, Zafeiris VE, Soukara V. Automated refactoring to the strategy design pattern. *Inf Softw Technol*. 2012;54(11):1202-1214.
6. Liu W, Hu Z, Liu H, Yang L. Automated pattern-directed refactoring for complex conditional statements. *J Cent South Univ*. 2014;21(5):1935-1945.
7. Nahar N, Sakib K. Automatic recommendation of software design patterns using anti-patterns in the design phase: a case study on abstract factory. Paper presented at: 3rd International Workshop on Quantitative Approaches to Software Quality (QuASoQ); 2015; New Delhi, India.
8. Nahar N, Sakib K. ACDPR: a recommendation system for the creational design patterns using anti-patterns. Paper presented at: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER); 2016; Suita, Japan.
9. Zafeiris VE, Poulias SH, Diamantidis NA, Giakoumakis EA. Automated refactoring of super-class method invocations to the template method design pattern. *Inf Softw Technol*. 2017;82:19-35.
10. Jaafar F, Guéhéneuc Y, Hamel S, Khomh F. Analysing anti-patterns static relationships with design patterns. *Electron Commun EASST*. 2013;59.
11. Cardoso B, Figueiredo E. Co-occurrence of design patterns and bad smells in software systems: an exploratory study. In: Proceedings of the Annual Conference on Brazilian Symposium on Information Systems (SBSI); 2015; Goiânia, Brazil.
12. Jaafar F, Guéhéneuc Y-G, Hamel S, Khomh F, Zulkernine M. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empir Softw Eng*. 2016;21(3):896-931.
13. Walter B, Alkhaeir T. The relationship between design patterns and code smells: an exploratory study. *Inf Softw Technol*. 2016;74:127-142.
14. Sousa BL, Bigonha MAS, Ferreira KAM. 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; 2017; Lavras, Brazil.
15. Sousa BL, Bigonha MAS, Ferreira KAM. A systematic literature mapping on the relationship between design patterns and bad smells. In: Proceedings of 33rd Annual ACM Symposium on Applied (SAC); 2018; Pau, France.
16. Souza P. *The Use of Metrics Threshold on the Evaluation of Oriented-Object Software Quality* [master's thesis] [in Portuguese]. Belo Horizonte, Brazil: Federal University of Minas Gerais; 2016.

17. Marinescu R. *Em Measurement and Quality in Object-Oriented Design* [PhD thesis]. Timisoara, Romania: Politehnica University of Timisoara; 2002.
18. Filó TG, Bigonha M, Ferreira K. A catalogue of thresholds for object-oriented software metrics. In: Proceedings of the First International Conference on Advances and Trends in Software Engineering (SOFTENG); 2015; Barcelona, Spain.
19. Filó TGS. *Identification of Thresholds for Metrics of Oriented-Object Software* [master's thesis] [in Portuguese]. Belo Horizonte, Brazil: Federal University of Minas Gerais; 2014.
20. Moha N, Gueheneuc Y-G, Duchien L, Le Meur AF. Decor: a method for the specification and detection of code and design smells. *IEEE Trans Softw Eng*. 2010;36(1):20-36.
21. Tsantalis N, Chaikalis T, Chatzigeorgiou A. JDeodorant: identification and removal of type-checking bad smells. Paper presented at: 2008 12th European Conference on Software Maintenance and Reengineering (CSMR); 2008; Athens, Greece.
22. Vidal SA, Marcos C, Díaz-Pace JA. An approach to prioritize code smells for refactoring. *Autom Softw Eng*. 2014;23:501-532.
23. Fernandes E, Oliveira J, Vale G, Paiva T, Figueiredo E. A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE); 2016; Limerick, Ireland.
24. Filó TGS, Bigonha MAS, Ferreira KAM. Raftool - filtering tool of methods, classes and packages with uncommon measures of software metrics [in Portuguese]. In: Proceedings of the X Workshop Anual do MPS (WAMPS); 2014; Campinas, Brazil.
25. Yamashita A, Moonen L. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE); 2013; San Francisco, CA.
26. Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empir Softw Eng*. 2018;23(3):1188-1221.
27. Ferreira KAM, Bigonha MAS, Bigonha RS, Mendes LFO, Almeida HC. Identifying thresholds for object-oriented software metrics. *J Syst Softw*. 2012;85:244-257.
28. Terra R, Miranda LF, Valente MT, Bigonha RS. Qualitas.class corpus: a compiled version of the qualitas corpus. *ACM SIGSOFT Softw Eng Notes*. 2013;38(5):1-4.
29. Tempero E, Anslow C, Dietrich J, et al. The qualitas corpus: a curated collection of java code for empirical studies. Paper presented at: 2010 Asia Pacific Software Engineering Conference; 2010; Sydney, Australia.
30. Eclipse. Eclipse 4.2 Juno. 2016. http://www.eclipse.org/downloads/packages/release/Juno/SR2. Accessed June 2016.
31. Metrics. Metrics. 2016. http://metrics.sourceforge.net. Accessed June 2016.
32. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST. Design pattern detection using similarity scoring. *IEEE Trans Softw Eng*. 2006;32(11):896-909.
33. Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. *ACM SIGMOD Rec*. 1993;22(2):207-216.
34. Brin S, Motwani R, Ullman JD, Tsur S. Dynamic itemset counting and implication rules for market basket data. *ACM SIGMOD Rec*. 1997;26(2):255-264.
35. Sousa B, Bigonha M, Ferreira K. Design pattern smell. 2017. http://llp.dcc.ufmg.br/Products/indexProducts.html. Accessed September 25, 2018.
36. Vokac M. Defect frequency and design patterns: an empirical study of industrial code. *IEEE Trans Softw Eng*. 2004;30(12):904-917.
37. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2012.
38. Sousa BL, Bigonha MAS, Ferreira KAM. *A Tool for Detection of Co-Occurrences Between Design Patterns and Bad Smells*. Technical report. Programming Language Lab (UFMG); 2017. LLP 001-2017.
39. Wendorff P. Assessment of design patterns during software reengineering: lessons learned from a large commercial project. In: Proceedings Fifth European Conference on Software Maintenance and Reengineering (CSMR); 2001; Lisbon, Portugal.
40. McNatt WB, Bieman JM. Coupling of design patterns: common practices and their benefits. In: Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC); 2001; Chicago, IL.
41. Izurieta C, Bieman JM. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Softw Qual J*. 2013;21(2):289-323.
42. Khomh F, Gueheneuce Y-G. Do design patterns impact software quality positively? In: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR); 2008; Athens, Greece.
43. Wagey BC, Hendradjaya B, Mardiyanto MS. A proposal of software maintainability model using code smell measurement. Paper presented at: 2015 International Conference on Data and Software Engineering (ICoDSE); 2015; Yogyakarta, Indonesia.

How to cite this article: Sousa BL, Bigonha MAS, Ferreira KAM. An exploratory study on cooccurrence of design patterns and bad smells using software metrics. *Softw: Pract Exper*. 2019;49:1079–1113. https://doi.org/10.1002/spe.2697

APPENDIX A

RESULTS OF THE BAD SMELL COLLECTION

Here, we present the results obtained during the data collection step, and the classes diagram on the cooccurrence examples.

This section presents the results obtained after executing the detection strategies with the RAFTool Support. We summarized the results in Tables A1 to A5. Each table brings the results concerning to a bad smell and obtained in each

TABLE A1 Results for data class

Software	# Classes with Data Class	# Total of Classes	% Classes with Data Class
Hibernate	825	7711	10.70%
JHotDraw	70	1061	6.60%
Kolmafia	441	3225	13.67%
Webmail	14	129	10.85%
Weka	348	2401	14.49%

TABLE A2 Results for feature envy

Software	# Classes with Feature Envy	# Total of Classes	% Classes with Feature Envy
Hibernate	1099	7711	14.25%
JHotDraw	101	1061	9.52%
Kolmafia	422	3225	13.09%
Webmail	17	129	13.18%
Weka	421	2401	17.53%

TABLE A3 Results for large class

Software	# Classes with Large Class	# Total of Classes	% Classes with Large Class
Hibernate	79	7711	1.02%
JHotDraw	15	1061	1.41%
Kolmafia	103	3225	3.19%
Webmail	2	129	1.55%
Weka	175	2401	7.29%

TABLE A4 Results for long method

Software	# Methods with Long Method	# Total of Methods	% Methods with Long Method
Hibernate	331	48 234	0.69%
JHotDraw	133	7633	1.74%
Kolmafia	1015	28 214	3.60%
Webmail	24	1091	2.20%
Weka	860	20 871	4.12%

TABLE A5 Results for refused bequest

Software	# Classes with Refused Bequest	# Total of Classes	% Classes with Refused Bequest
Hibernate	2050	7711	26.59%
JHotDraw	411	1061	38.74%
Kolmafia	960	3225	29.77%
Webmail	29	129	22.48%
Weka	1025	2401	42.69%

system. The first column indicates the NOC/NOM affected by bad smell. The second column shows the total NOC/NOM existing in the analyzed system. Finally, the third column shows the percentage of the system that was affected by the bad smell.

APPENDIX B

RESULTS OF THE COOCCURRENCES

This section presents the results obtained after identifying the cooccurrences between GoF design patterns and bad smells with the Design Pattern Smell support. We summarized the results in Tables B1 to B5. We organized these tables as the GoF design patterns being the lines and the systems being the columns. For each system, there are two different information. The "T" column indicates the total NOC/NOM that make part of the design pattern instances identified in this study. The "DP&BS" column indicates the NOC/NOM that presented cooccurrence between GoF design patterns and bad smells.

TABLE B1 Total number of classes with design pattern and amount of classes with cooccurrences between design pattern and data 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	40	53	13	386	75	40	8	152	23
Bridge	56	2	40	0	14	1	6	3	0	0
Composite	12	0	12	1	8	0	0	0	0	0
Decorator	37	2	10	1	67	7	0	0	32	10
Factory method	37	0	5	0	31	0	2	0	22	1
Observer	4	2	2	1	8	1	0	0	36	1
Prototype	0	0	21	4	0	0	0	0	0	0
Proxy	8	3	0	0	18	6	0	0	35	10
Singleton	232	3	13	1	77	9	1	1	34	0
State-strategy	271	51	121	24	334	52	23	3	93	18
Template method	87	5	16	2	54	3	4	2	22	1

Abbreviation: DP&BS, design pattern and bad smells.

TABLE B2 Total number of classes with design pattern and amount of classes with cooccurrence between design pattern and feature envy

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	36	53	14	386	74	40	6	152	64
Bridge	56	18	40	7	14	4	6	2	0	0
Composite	12	0	12	4	8	0	0	0	0	0
Decorator	37	3	10	2	67	5	0	0	32	9
Factory method	37	2	5	0	31	3	2	0	22	3
Observer	4	1	2	1	8	2	0	0	36	11
Prototype	0	0	21	5	0	0	0	0	0	0
Proxy	8	1	0	0	18	6	0	0	35	16
Singleton	232	2	13	0	77	5	1	1	34	5
State-strategy	271	41	121	31	334	54	23	2	93	43
Template method	87	27	16	5	54	14	4	1	22	8

Abbreviation: DP&BS, design pattern and bad smells.

TABLE B3 Total number of classes with design pattern and amount of classes with cooccurrence between design pattern and large 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	13	53	6	386	33	40	2	152	35
Bridge	56	8	40	2	14	3	6	0	0	0
Composite	12	0	12	1	8	0	0	0	0	0
Decorator	37	2	10	1	67	2	0	0	32	7
Factory method	37	1	5	0	31	2	2	0	22	0
Observer	4	1	2	0	8	1	0	0	36	7
Prototype	0	0	21	1	0	0	0	0	0	0
Proxy	8	1	0	0	18	1	0	0	35	9
Singleton	232	0	13	0	77	2	1	0	34	3
State-strategy	271	21	121	7	334	30	23	1	93	29
Template method	87	6	16	2	54	4	4	0	22	2

Abbreviation: DP&BS, design pattern and bad smells.

TABLE B4 Total number of methods with design pattern and amount of methods with cooccurrence between design pattern and 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	12	73	0	703	44	50	0	222	20
Bridge	61	3	51	2	19	3	8	0	0	0
Composite	8	0	29	0	37	0	0	0	0	0
Decorator	115	1	31	0	255	2	0	0	61	6
Factory method	58	0	23	0	45	0	2	0	27	0
Observer	8	0	2	0	7	1	0	0	24	0
Prototype	0	0	16	2	0	0	0	0	0	0
Proxy	6	0	0	0	31	1	0	0	37	1
Singleton	340	0	15	0	672	0	1	0	83	0
State-strategy	343	24	227	21	974	61	19	0	173	31
Template method	275	8	47	2	161	19	14	0	34	4

Abbreviation: DP&BS, design pattern and bad smells.

TABLE B5 Total number of classes with design pattern and amount of classes with cooccurrence between design pattern and refused bequest

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	22	53	5	386	30	40	5	152	39
Bridge	56	4	40	4	14	0	6	1	0	0
Composite	12	1	12	5	8	0	0	0	0	0
Decorator	37	5	10	3	67	0	0	0	32	22
Factory method	37	1	5	0	31	1	2	0	22	6
Observer	4	0	2	0	8	0	0	0	36	0
Prototype	0	0	21	8	0	0	0	0	0	0
Proxy	8	4	0	0	18	9	0	0	35	23
Singleton	232	59	13	7	77	34	1	0	34	3
State-strategy	271	23	121	31	334	19	23	2	93	31
Template method	87	14	16	6	54	6	4	0	22	10

Abbreviation: DP&BS, design pattern and bad smells.

APPENDIX C

CLASS DIAGRAMS OF THE COOCCURRENCES

This section presents the visual representation regarding the cases discussed in Section 4.2.3.

FIGURE C1 Class diagram that represents the `Proxy` ⇒ `Data Class` relationship

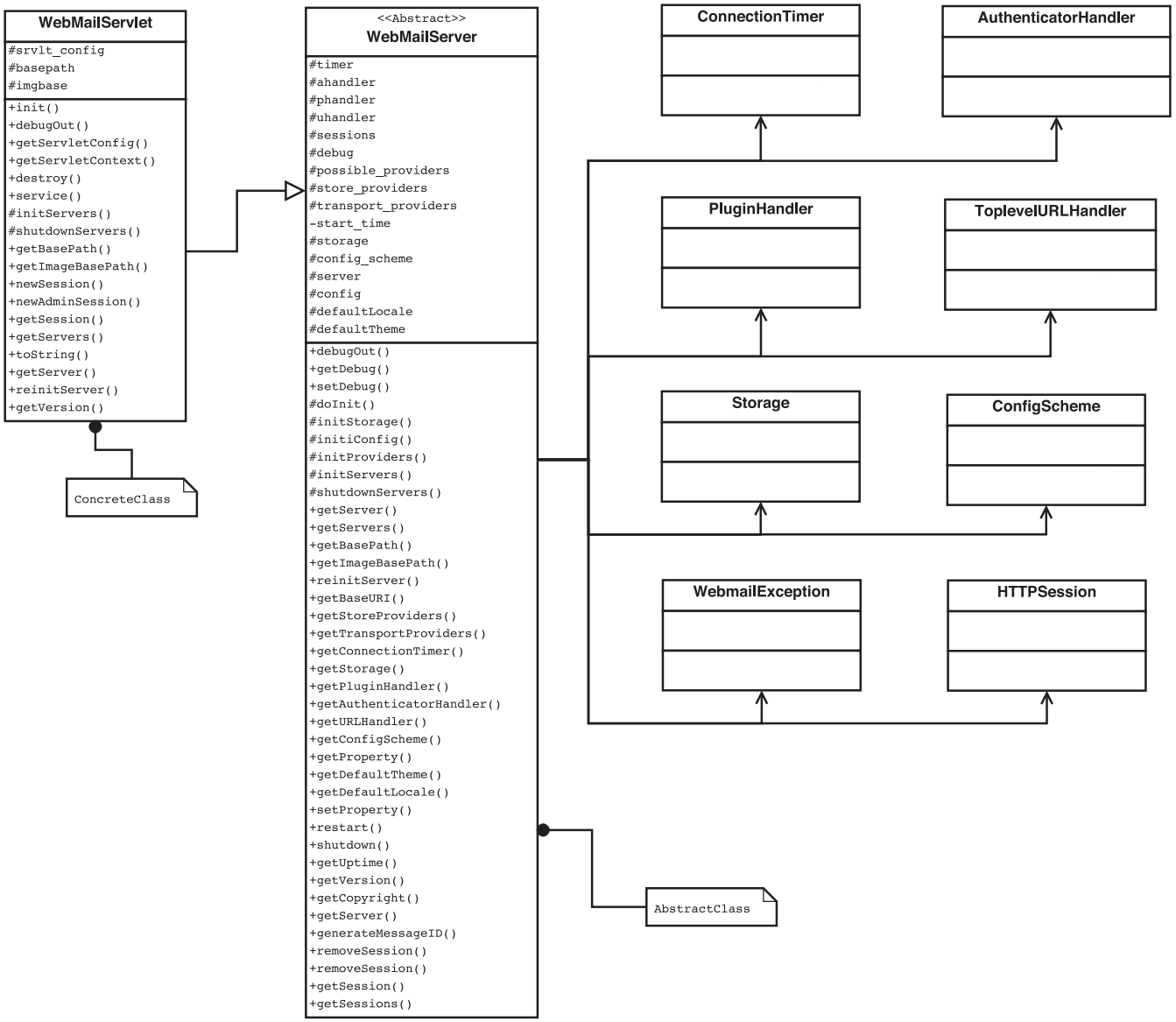

FIGURE C2 Class diagram that represents the `Template Method` ⇒ `Feature Envy` relationship

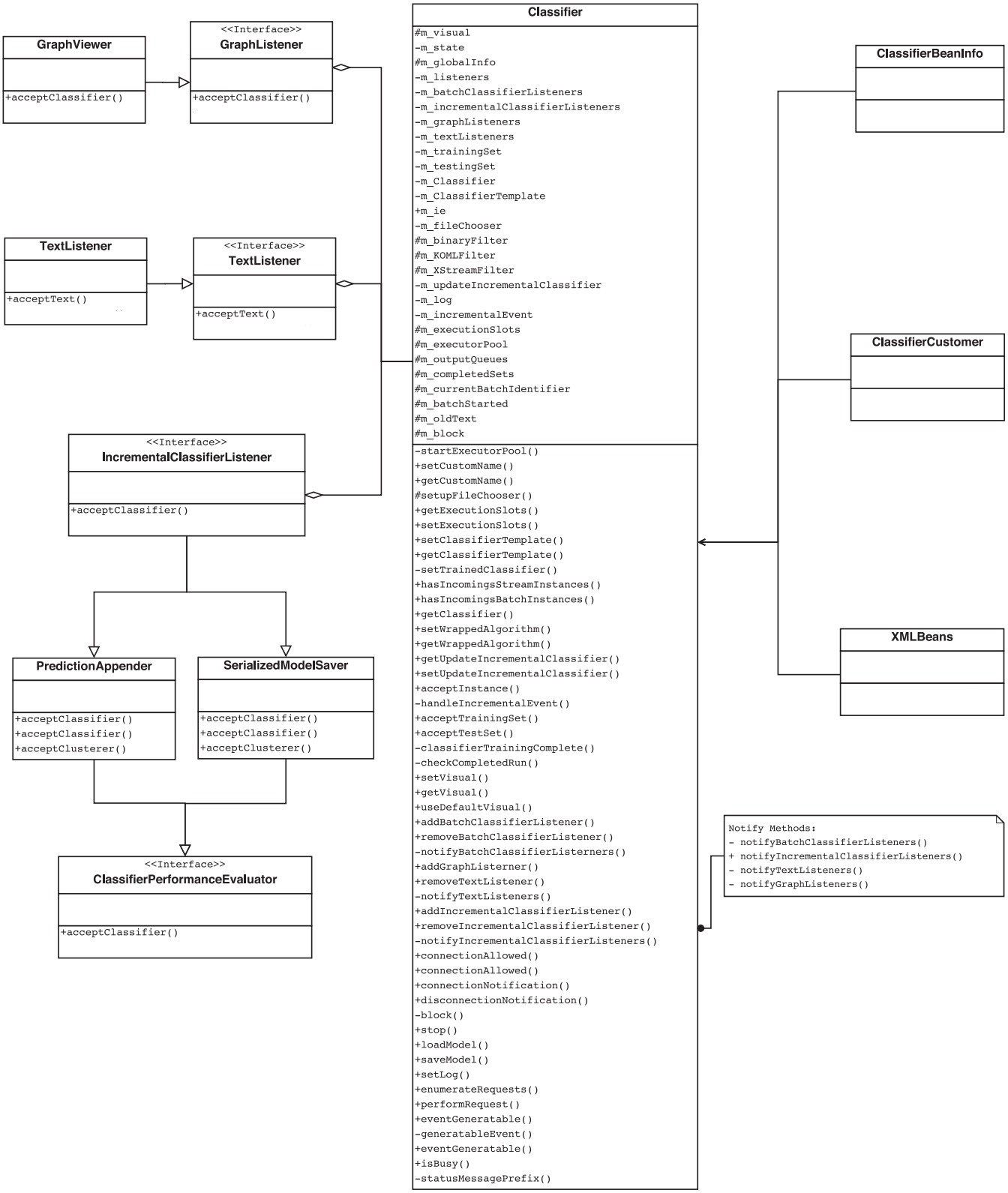

FIGURE C3 Class diagram that represents the `Observer ⇒ Large Class` relationship

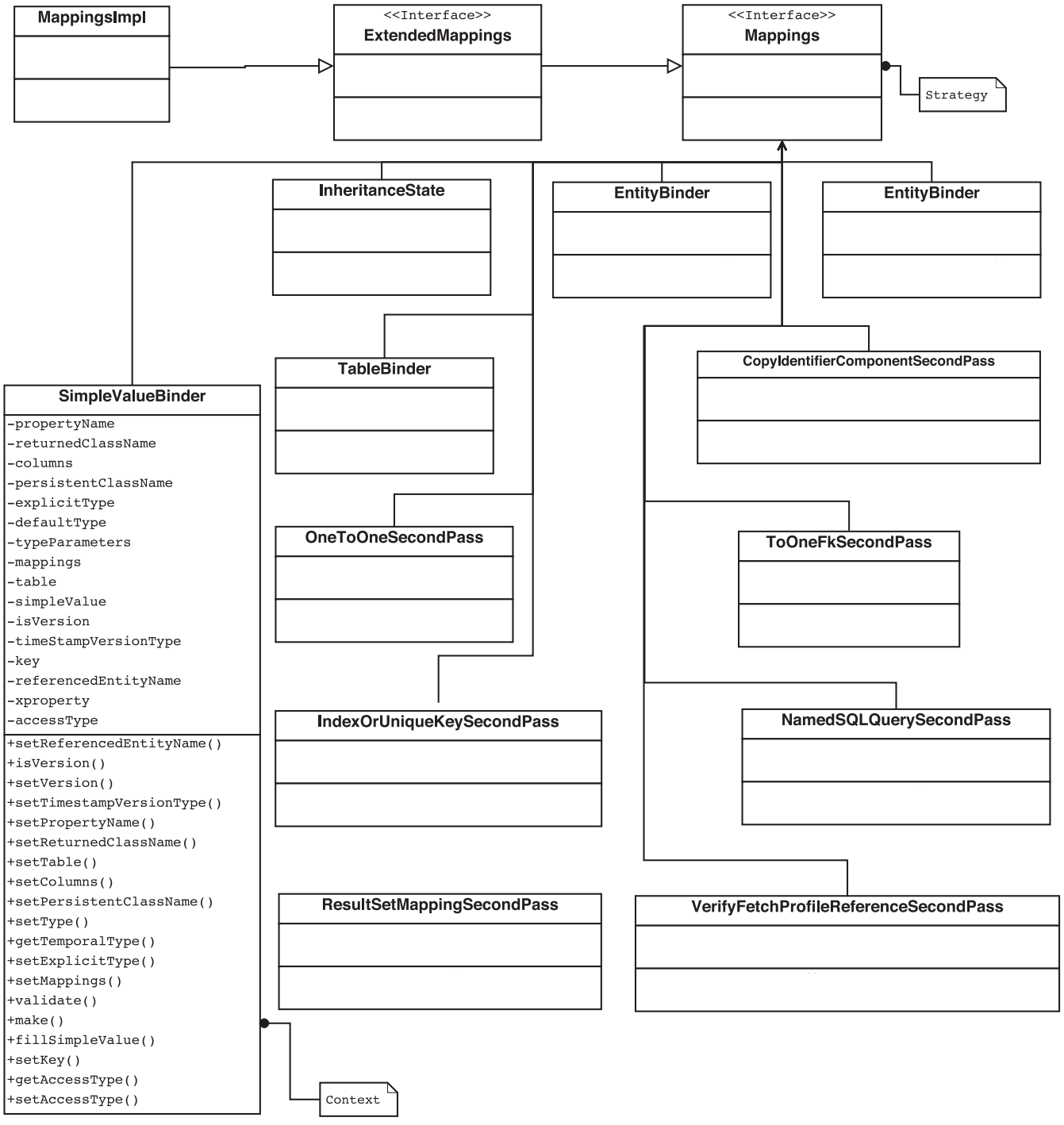

FIGURE C4 Class diagram that represents the `Strategy` ⇒ `Long Method` relationship

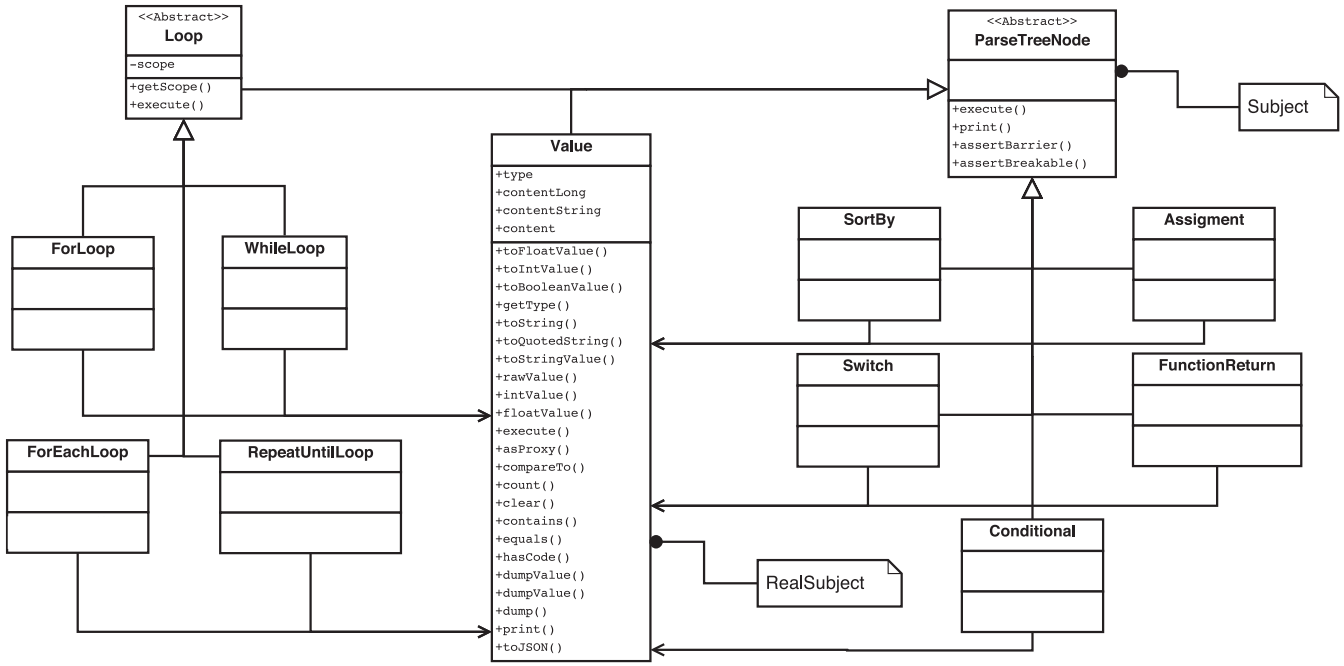

FIGURE C5 Class diagram that represents the `Proxy` ⇒ `Refused Bequest` relationship