An Empirical Analysis of When GOF Design Patterns Occur with God Class and Long Method Bad Smells

Abstract:

Design patterns are all-purpose, repeatable answers to regularly occurring issues in software development. When properly implemented, these techniques are meant to improve software's modular and flexible architecture. This research aims to investigate the incidence of God Class and Long Method bad smells in design pattern-based software systems. To accomplish this, we conducted an exploratory study using five Java systems to (I) determine whether design patterns prevent bad smell, (ii) recognize design patterns that may co-occur with bad smells, and (iii) extract the key factors influencing such co-occurrence. Eleven of the twenty-three GOF design patterns are taken into consideration in our analysis. We also take into account the God Class and Long Method as bad smells.

Index words: design pattern, bad smell, software metric, threshold.

1. Introduction:

A design pattern is a generic response to a recurrent issue in a particular software design setting. Its primary objective is to develop software systems that are adaptable, extendable, have reusable components, and are simple to maintain. They are accepted as sound programming techniques. They haven't been suggested for this purpose, but when used properly, they might assist reduce software smells. Bad smells are indications found in a program's source code that may point to a more significant issue requiring code refactoring.

These symptoms are not considered mistakes, but they degrade program quality and go against fundamental principles of software engineering including modularity, readability, and reuse. Bad smells may be eliminated using design patterns. On the other hand, several research have found a correlation between bad smells and design patterns. Although design patterns are meant to increase software quality, they don't always keep bad smell at bay. In order to evaluate object-oriented software that employs the design patterns specified by Gamma et al., this paper presents an exploratory study. The major goals of this study are to: (I) determine whether design patterns prevent bad smells; (ii) pinpoint design patterns that may co-occur with bad smell; and (iii) identify the key factors that contribute to the avoidance of bad smell.

We conduct a case study using thirty Java software systems to accomplish these goals. These programs were taken from GitHub and are open source. In this case study, we analyze the correlation between bad smell Class and Long Method —and eleven design patterns identified by Gamma et al. We carefully reviewed the source code in the current version where a bad smell and a design pattern were found to co-occur. We describe the architecture of two instances of this in detail. We discuss the key factor that resulted in the design pattern and the bad smells co-occurrence considered in this analysis' lessons learnt.

1. Research Methodology

Six stages were taken to complete this study: (1) definition of the research questions (ii) defining the data set that consists of the software systems taken into consideration in the study, (iii) identifying poor smell detection techniques, (iv) data collection, (v) putting association rules to use, and (vi) method of data analysis.

* 1. Research Questions

The following research questions (RQn) are looked into in this paper:

* RQ1: Do the design patterns listed in the GOF catalogue prevent the formation of software smells like God Classes and Long Methods?
* RQ2: Which GOF catalogue design patterns co-occurred with the God Class and Long Method bad smells?
* RQ3: What are some of the more frequent instances where software systems using GOF design patterns exhibit the God Class and Long Method bad smells?
  1. Identification of Bad Smells Detection Strategies

Marinescu claims that a detection method is a formal rule that characterizes a particular bad smell. Software metric thresholds can be utilized in addition to detection methodologies to determine the association between a metric and a bad smell and, as a result, spot anomalous entities.

We explore the God Class and Long Method bad smell in this study. God Class is a class that assigns small details to other classes and conducts an excessive amount of work. A long method is one with a lot of lines, temporary variables, and parameters that does too much work. We selected these bad smells because they pose particular difficulties for software maintenance. Additionally, they are linked to a lot of information, which makes it harder to understand software and tightens the bonds between classes and methods in the system.

Filó et al. suggested the detection methods employed in this work. These tactics were picked because they include well-known software metrics. In addition, they have already been analyzed, and no false negative results were obtained. False positive results could be reported, however the likelihood of this happening is low. These findings imply that these detection techniques are reliable for picking up bad smell. A software metric's thresholds are divided into three categories: Good, Regular, and Bad. This demonstrated that a low incidence of bad smell is associated with the Good range. As a result, the Regular and Bad ranges are used in the detection strategies. The detection methods for the God Class and Long Method are then described.

The metrics Weighted Methods per Class (WMC), Number of Methods (NOM), Number of Attributes (NOF), and Lack of Cohesion of Methods (LCOM) are used in the God Class Detection Strategy.

The method for detecting bad smell in the Long Method is Method Lines of Code (MLOC), Nested Block Depth (NBD), and McCabe Cyclomatic Complexity (VG) are the metrics it employs.

* 1. Data Set

The open-source software systems whose metrics were used in this study were taken from GitHub, a data set that includes Java-developed open-source software systems. It provides bytecodes for the software systems as well as 23 software metrics. This data set was chosen because it contains a sizable number of open-source Java programs that are frequently utilized in empirical research on software artefacts. We used a sample of five software systems for this study's manual inspection: Hibernate 4.2.0, JHotDraw 7.5.1, Kolmafia 17.3, Webmail 0.7.10, and Weka 3.6.9.

The selection of these systems was primarily based on two factors: (I) the utilization of design patterns from the GOF catalogue, and (ii) the presentation of the bad smells taken into consideration in this study.

* 1. Data Collection

The third phase included gathering the data that will be examined. We used the Design Pattern Detection using Similarity Scoring 4 (DPDSS) tool to confirm the presence of design patterns in the software. Using quadratic matrices to describe directed graphs, it uses the similarity scoring algorithm to model all facets of design patterns. This program determines the similarity scores between the graph's vertices after receiving as inputs the system and the design pattern's graph. The primary benefit of this method is its capacity to identify patterns in both their modified and original forms, which are typically seen in the literature. JHotDraw 5.1, JRefactory 2.6.24, and JUnit 3.7 were the three systems used to test DPDSS, and no false positive instances of design patterns were found. Only the Factory Method and State design patterns produced false negatives. The tool's results were quite good, demonstrating its effectiveness in locating design pattern occurrences.

To put detection algorithms into practice, we employed RAFTool. The XML file containing software measurements for the target system and a detection method that is defined by a logical expression in a predetermined manner are inputs to the tool. When a class or method's metric values match the detection strategy, the tool reports those classes or methods.

The design patterns instances that DPDSS returns could include a single class or several methods. For instance, the Bridge design pattern's returned instance has two classes that are responsible for expressing the implementation and abstraction parts, respectively. We used the Design Pattern Smell3 to count the classes and methods in the design pattern instances and to identify the components that have a certain design pattern and a specific bad smell in order to address this issue. This program accepts two types of input: (1) XML files output by DPDSS that contain instances of a system's design patterns, and (2) CSV files produced by RafTool that contain the components with a specific smell.

* 1. Application of Association Rules

We used association criteria, which are based on data mining principles, to find instances of the design pattern and bad smell together.

* 1. Method of the Data Analysis

There are five steps in this process. We start by locating design patterns in the software systems. To locate them, we used the DPDSS program, and we tabled the outcomes. The second section seeks to pinpoint classes and methods that smell like the God Class or the Long Method. We used RAFTool's logical expressions to locate them. After locating the negative smells associated with God Class and Long Method and design pattern occurrences, we pre-processed the data by using the Design Pattern Smell tool.

In the third section, we apply association rules to the pre-processed data to detect co-occurrences that already exist in the systems. In the fourth section, we manually examined the co-occurring classes to find the circumstances that led to the development of this relation in those classes. Finally, we conducted an analysis of the data to address the given research questions.

3 Results

The outcomes of the study that is being discussed in this part are presented first. In this part, we also respond to the suggested research questions. The number of classes and methods in the software systems with the God Class and Long Method bad smells, respectively, are shown in Tables 1 and 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Software | # Classes with God Class | # Classes | % Classes with God Class |
| Hibernate | 427 | 7118 | 16.66% |
| JHotDraw | 152 | 1561 | 10.26% |
| Kolmafia | 415 | 3300 | 7.95% |
| Webmail | 25 | 140 | 5.60% |
| Weka | 505 | 2510 | 4.97% |

Table 1: Number of classes with God Class.

|  |  |  |  |
| --- | --- | --- | --- |
| Software | # Methods with Long Method | # Methods | % Methods with Long Method |
| Hibernate | 2,900 | 47775 | 16.47% |
| JHotDraw | 1000 | 7377 | 7.37% |
| Kolmafia | 4500 | 27000 | 6.00% |
| Webmail | 113 | 2100 | 18.58% |
| Weka | 3288 | 18881 | 5.74% |

Table2: Number of methods with long method

After data preparation, the findings are displayed in tables 3 and 4. The "T" column in both tables indicates how many classes or methods utilize a certain design pattern, while the "DP&BS" column lists how many classes include specific design pattern (DP) instances as well as instances of the corresponding bad smell (BS).

3.1 Analysis of the Results

For the purpose of addressing the research issues outlined in this work, the classes with the God Class or Long Method bad smells were physically examined.

RQ1. Do the design patterns listed in the GOF catalogue prevent software from smelling like God Classes and Long Methods?

The Composite and Factory Method design patterns have minimal God Class occurrences, according to the results provided in Table 3. They feature a modular structure and divide the tasks into different classes, according to the manual inspection. The Composite design pattern encourages the construction of complex objects from simpler ones. These fewer complex objects are defined as modules, which allows intelligence to be distributed among them and lowers class complexity. The Factory Method design pattern simulates the idea of a factory in which there is an interface to create objects, but the object creation itself occurs in the class that implements this interface. Thus, it is possible to develop multiple modules that are each in charge of producing and maintaining the data for a certain set of objects, relieving one class of the burden. As a result, the design patterns Composite and Factory Method are essentially modular.

As seen in Table 4, we noticed a similar pattern of behavior with the bad smell from the Long Method. With the exception of Composite and Factory Method, which only seldom exhibit this bad smell, most design patterns have a high percentage of Long Method occurrences. A unique situation is the Singleton design pattern. It appears to be a false negative case, despite the fact that its occurrences do not smell like the Long Method. Singleton was flagged as a false negative since DPDSS only used the static attribute that was presented in the class to identify its instances. Methods are not regarded as being a feature of this design pattern. Therefore, this design pattern returned 0 when it intersected with methods that had bad smells.

Although the God Class and Long Method bad smells and the Factory Method and Composite design patterns seldom co-occur, the general conclusion of this analysis is that these two bad smells are connected to the majority of the design patterns examined in this work. Therefore, "No, design patterns GOF not necessarily avoid occurrences of the God Class and Long Method bad smells" is the correct response to RQ1.

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Design Pattern | Hibernate | | JHotDraw | | Kolmafia | | Webmail | | Weka | |
|  | T | DP&BS | T | DP&BS | T | DP&BS | T | DP&BS | T | DP&BS |
| Adapter-Command | 229 | 40 | 52 | 20 | 380 | 80 | 45 | 8 | 155 | 70 |
| Bridge | 55 | 15 | 39 | 10 | 16 | 6 | 5 | 2 | 1 | 1 |
| Composite | 12 | 0 | 12 | 4 | 8 | 0 | 0 | 0 | 0 | 0 |
| Decorator | 37 | 3 | 10 | 2 | 6 | 7 | 0 | 0 | 32 | 17 |
| Factory Method | 37 | 3 | 5 | 0 | 31 | 2 | 3 | 0 | 3 | 22 |
| Observer | 4 | 2 | 2 | 1 | 8 | 1 | 0 | 0 | 36 | 12 |
| Prototype | 0 | 0 | 21 | 9 | 0 | 0 | 0 | 0 | 0 | 0 |
| Proxy | 8 | 2 | 0 | 0 | 18 | 9 | 0 | 0 | 35 | 18 |
| Singleton | 232 | 3 | 13 | 1 | 77 | 9 | 1 | 1 | 34 | 5 |
| State-Strategy | 271 | 47 | 120 | 40 | 300 | 60 | 20 | 3 | 90 | 45 |
| Template Method | 80 | 20 | 15 | 7 | 53 | 12 | 4 | 2 | 20 | 8 |

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Design Pattern | Hibernate | | JHotDraw | | Kolmafia | | Webmail | | Weka | |
|  | T | DP&BS | T | DP&BS | T | DP&BS | T | DP&BS | T | DP&BS |
| Adapter-Command | 270 | 52 | 72 | 20 | 703 | 123 | 45 | 8 | 222 | 70 |
| Bridge | 52 | 23 | 55 | 21 | 19 | 2 | 3 | 8 | 0 | 1 |
| Composite | 9 | 0 | 29 | 3 | 38 | 0 | 0 | 0 | 1 | 0 |
| Decorator | 137 | 2 | 30 | 1 | 256 | 12 | 0 | 0 | 61 | 24 |
| Factory Method | 37 | 3 | 5 | 0 | 31 | 2 | 3 | 0 | 3 | 22 |
| Observer | 8 | 3 | 21 | 1 | 7 | 3 | 0 | 0 | 24 | 4 |
| Prototype | 0 | 0 | 17 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| Proxy | 5 | 3 | 0 | 0 | 30 | 10 | 0 | 0 | 35 | 12 |
| Singleton | 343 | 3 | 13 | 1 | 677 | 0 | 1 | 1 | 53 | 5 |
| State-Strategy | 340 | 121 | 227 | 90 | 975 | 154 | 20 | 3 | 175 | 98 |
| Template Method | 250 | 80 | 47 | 13 | 167 | 48 | 15 | 2 | 34 | 15 |

RQ2. Which GOF catalogue design patterns showed a correlation between the God Class and Long Method bad smells?

As a result, in response to RQ2, we determined that the Template Method, Observer, and Proxy presented the main co-occurrences with the God Class smell. The design pattern known as the Template Method was the one that showed the highest correlation with this bad smell.  We also determined that the main co-occurrences with the long Method bad smell were offered by the design patterns Observer, State-Strategy, and Template Method. However, Observer and Long Method had the highest co-occurrence.

RQ3. What are some of the more frequent instances where software systems using GOF design patterns exhibit the God Class and Long Method bad smell?

The findings show that God Class bad smell is more frequently associated with the Template Method design pattern and Long Method bad smell is more frequently associated with the Observer design pattern. We conducted a manual inspection of the classes with the Template Method => God Class and Observer => Long Method co-occurrences in order to determine the sources of such co-occurrences.

3.1.1 Template Method ⇒ God Class

By delegating some phases to subclasses, which have the authority to redefine the algorithm's properties without altering the algorithm's structure, the Template Method tries to define the skeleton of an algorithm through an operation. In other words, this pattern makes use of a modular structure, where the different behaviors of an object are described in the subclasses and assigned to the object via polymorphism. The benefit of this implementation is the decrease in complexity in the super class because polymorphism is used to define variables instead of conditional constructs like if, else, and switches.

To avoid giving the templates and the super class too many responsibilities, this approach must be used with caution. The manual inspection of the classes that displayed the co-occurrence of the template method and the god class showed that these classes were given a lot of responsibility.

We noticed that the templates methods make reference to the subclasses' implementations of the object behavior description. The implementation of the behavior is very difficult in several classes. This results in task overload in the template methods, which occasionally contributes to the occurrence of long methods. Additionally, it has been noted that super classes that implement Template Methods have a large number of dependencies. Such implementations increase the coupling of these classes by creating many instances and passing them small tasks.

3.1.2 Observer ⇒ Long Method

A solution that creates a one-to-many dependency between objects is called an observer. The subject class maintains a list of all observers’ classes that consume its data, and this list is used by the observer structure. When one of the subject's observers modifies certain data, the subject is activated by altering the other observers. The real-time updating of objects and the synchronization of data are the goals of this design pattern. By using polymorphism, this update avoids the complexity increase that normally results from the use of conditional structures. When utilizing this design pattern, it is crucial to appropriately execute the operation that alerts the subject's observers, as improper planning for this operation may lead to complex methods.

The overuse of object orientation, which results in the concentration of duties in the classes that implement the Template Method, is the primary cause of the co-occurrence of God Class bad smell and Template Method. Similar circumstances were also discovered in the case of the Long Method bad smell co-occurrences with Observer; in this instance, incorrect object orientation causes scattering and crosscutting issues.

4. Conclusion

Since design patterns are modular solutions, it is reasonable to assume that using them will help keep unwanted smells at bay. Design patterns weren't, however, particularly suggested for this purpose.

Making sure that object orientation and modularization are effectively used is crucial when using design patterns to create object-oriented software systems. Design patterns are frequently used in classes and methods to make software more extensible and flexible. Careless application of design patterns can lead to design deviations such responsibility concentration in classes and methods, scattering, and cross-cutting issues. Such deviations may cause software systems to bad smell and make the code complex and challenging to comprehend.

References

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

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

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

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

[5] Filó, T. G. S., Bigonha, M. A. S., and Ferreira, K. A. M. A catalogue of thresholds for objectoriented software metrics. In SOFTENG, pages 48–55, 2015.

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

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Objectoriented Software. Addison-Wesley, 1994.