Study of Relationship between design patterns and code smells on open source java projects: an Empirical Evaluation

AkashSajja   
*Computer Science*  
*Lewis University*Romeoville  
akashsajja@lewisu.edu

*Abstract*—Open-Source projects has completely changed the extent of technology these days. It mainly tells and allows the users to know the working of the code and the development of the project which allows the code to be reused and recycled which reduces the burden on an individual who develops the whole project from scratch. The open-source projects are available for use, modification, and distribution with the original rights. Quality of the software is highly dependent on the well-structured code. Basically, code smells are a result of poor and misguided programming. These mistakes in the code might ultimately lead to the risk of failure of the project. We can rectify the errors if we get to know the code smells before the completion of the development process. The design patterns are a general repeatable solution to a commonly occurring problem in software design. These design patterns basically tell us how to solve a problem that can be used in various situations. This paper is a proof-of-concept for the analysis of the relationship between design patterns and code smells which is conducted on 30 different open-source projects which use the data repository called GitHub. The result of this research shows that there are some code smells in the part of a project. The results would contribute to the empirical evaluation of the open-source projects and the code smells.

Keywords—open-source, design patterns, code smells, empirical evaluations, software design

# Introduction

In object-oriented development, design pattern is a repeatable solution to the common occurring problem that can be occurred in various situations and parts of the code. This is not a standard design that we can directly use it as a code. The design patterns use certain terminology instead of standard explanations. There are various number of design patterns that we can use to a part of a code, these design patterns are generally categorized into three categories: creational design patterns which include abstract factory, builder, factory method, prototype, singleton. Behavioral design pattern which includes chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor. Structural design patterns which include adapter, bridge, composite, decorator, facade, flyweight, proxy. There are a total of 23 different design patterns generalized as GoF (Gang of Four). These design patterns consist of 4 essential elements: pattern name, problem, solution, consequences. We can solve design problems using design patterns by finding appropriate objects, determining object granularity, program to an interface which manipulate objects solely in terms of interfaces. The benefits of design patterns are that the clients remain unaware of the specific types of objects they use and unaware of the classes that implement the objects.

One of the common problems that most developers face, especially with the working of large java projects is code smells. These smells are certain structures in the code that indicate the violation of design principles which ultimately effect the quality of the project. Code smells are not a major issue in the code, these doesn’t affect the functioning of the code, but these smells are responsible for the complexity of the code. Code smells generally refer to some lines in the source code of a program that might have an issue in the design principles which may cause some issues to the maintainability of the software. We can find these code smells in the code by using various code smell detection tools that are available online, we have a numerous number of tools through which we can find duplicate code, God class, feature envy, type checking, long method. These are various types of code smells that may occur in the large projects.

The relationship between code smells and design patterns are analyzed in this paper which solely tells us about what type of code smells are generating if we select a particular design pattern in a part of the project. We did an empirical evaluation on the design patterns and how it impacts code smells. At the end of the evaluation, we observed that the classes which are involved in design patterns are consisting of code smells to the classes that are not involved in design patterns. We used a tool called JDeodorant [3], PMD[5] and design pattern detection tool [4] to find these results which will be discussed later in the paper.

# **Set of code smells**

There are around hundreds of code smells, but all those code smells are classified under two categories: code smells within classes [6], code smells between classes [6].

## CODE SMELLS WITHIN CLASSES

|  |  |  |
| --- | --- | --- |
| Type | Name of Code Smell | Solution |
| Type 1 | Comments | Make it as a complete another function |
| Type 2 | Long Method | Changing expression into sub-expression |
| Type 3 | Long Parameter List | Pass it as a function instead of a parameter |
| Type 4 | Large Classes | Refactoring techniques like extract class, duplicate observed data |
| Type 5 | Duplicate Code | Use a pull-up constructor body |
| Type 6 | Dead Code | Remove the parameters that are not needed in the method |

Table 1: Types of code smells(A) and respective solution

## **CODE SMELLS BETWEEN CLASSES**

|  |  |  |
| --- | --- | --- |
| Type | Name of Code Smell | Solution |
| Type 1 | Data Classes | Use global variables to refactor |
| Type 2 | Data Clumps | Introduce parameter objects, Preserve whole objects |
| Type 3 | Alternative class with different interfaces | Use refactoring techniques like Rename Method, Add Parameter |
| Type 4 | Refused Bequest | Replace Inheritance with the delegation and Extract superclass |
| Type 5 | Lazy Class or God Class | Inline class and collapsing hierarchy |
| Type 6 | Shotgun Surgery | Use of Move Methos and Move Field |

Table 1: Types of code smells(B) and respective solution

# **Literature Review**

Most of the researchers or developers studied design patterns and code smells by using the respective tools that are available online or by using custom code which they developed themselves. But the relationship between design patterns and code smells are studied very limited.

The design patterns dataset [7] to detect which type of design patterns are used in the code files. We studied nearly 30 open-source java projects from GitHub, including all the kinds of design patterns and the number of code smells in each design pattern were calculated.

The study of design patterns and code smells is done on all the projects to understand the maintainability, testability and coupling and we found out that the maintainability of the projects is higher with the less number of code smells detected and the testability of the project is higher with the higher number of code smells detected and coupling is not dependent on the code smells irrespective of the type of design pattern in a particular project.

The study of singleton design pattern on code smells is done on the projects [8][9][10][11][12] and studied the values of code smells. Some of the code smells are already treated before the deployment of the project. The values are discussed in the experiment section.

# **Experiments**

## **Experimenting God Class for singleton Projects for maintainability**

God Class is a class which do most of the work. It is usually a huge class that concentrates a lot of responsibilities, controls and oversees many different objects, and effectively does everything in the application. Code smells are patterns in object-oriented systems that are perceived to lead difficulties in the maintenance of such systems. It is held that to improve maintainability, code smells should be eliminated by refactoring. We experimented God class code smell for singleton projects [8][9][10][11][12], by conducting this experiment we got some values through which we understand how maintainable are the projects which we experimented. This is done by using Jdeodorant tool (an eclipse plug-in) which is used to identify bad smells in the code. God classes have more defects than any other class in the code. In the long run, it becomes more challenging and the cost of maintenance increases rapidly. The existence of these class is bad because, it makes the code fragile which means it is responsible for so much work and it gas to be frequently updated which becomes impossible most of the times. By experimenting we realized there are some code smells in the projects, and this can be rectified by using refactoring which tells us about the initial code which is written and the final code to which it is changed.

Chart, bar chart

Description automatically generated

q

Fig 1. Graph representing the code smells in singleton projects for desired values

## **Experimenting Feature Envy for adapter Projects for maintainability**

Feature Envy is defined as occurring when a method calls method on another class a greater number of times than the class which it is present. This is a type of code smell which we find in the projects which ultimately results in the low rate of maintainability. It is basically defined as the part of the code is present in the wrong class rather than the class which it must be in. In the projects [13][14][15][16][17] we experimented with feature envy and observed some code smells that the code is in the wrong class. To rectify these smells, we need to refactor the part of the code using Move Method and Extract Method, by doing this we are pushing the part of the code to its locality where it must be so that the code smells are eradicated. By this experiment we concluded that the projects which has the code smell feature envy and if it is not treated by refactoring it ultimately effects the maintainability of the project in the long run. We used Jdeodorant tool for extracting this code smell. We can either use Move Method or Extract Method for this, by using Extract Method we split the method into several parts to different classes which suits the code better.

## **Experimenting Long Method for Bridge Projects for Testability**

Long Method code smell is nothing but a method which consists of large lines of code i,e. large open source projects. We experimented with projects [18][19][20] to test the long method code smell because these are significantly larger projects, and we need to find if there are any duplicate lines or any of the lines that are repeated or the number of lines that are commented. These all types of things ultimately add up to the testability of the code which makes the tester to consume more amount of time and the code might mot be that reliable. We need to eradicate this kind of code smells by refactoring, by extracting a few sub-methods which will clean up some of the code which eventually reduces the lines of code that are used to build the project. We need to maintain short methods, if the significant number of lines of code in a method less than 50 LoC the projects are more reliable and testable. We used PMD tool to experiment with long Method.

## **D. Experimenting Switch Statements for Iterator Projects for Realiability**

Switch operators or Switch Statements are relatively rare in object-oriented development. But at a times we might end up with Switch Statement code smells, there are caused because the single switch might be in many places in the program though it might not be necessary to be present there. If we add a certain part of the code after a significant period, we need to modify the switch statements. To eradicate these code smells we might use Extract Method and Move Method. We experimented with projects [21][22] to identify switch statements for Iterator Projects and we concluded that the projects with switch statement code smells are less reliable when compared to the projects with different design pattern. We used PMD tool (an eclipse plug-in) to get the results for switch statements.

# **Conclusion**

In this paper, we performed empirical evaluation on the relationship between design patterns and code smells on open source java projects. We need different tools such as design pattern detection tool, PMD tool and Jdeodorant tool and choose projects with significant design patterns that are experimented with different types of code smells. The results show the relationship between particular code smells on particular design pattern. For future work, we can aim to study the relation between design pattern and code smells to different projects other than JAVA and use different mining tools and code smell detection tools to examine further regarding this topic and validate the results of this study. We only tested certain patterns in this paper, in future we can aim to work on all the design patterns to understand the relationship better between design patterns and code smells.

##### References

1. M. Aberdour. Achieving quality in open source software. IEEE Softw., 24(1):58–64, Jan. 2007.
2. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. J. Mach. Learn. Res., 3:993–1022, Mar. 2003.
3. https://users.encs.concordia.ca/~nikolaos/pattern\_detection.html
4. https://users.encs.concordia.ca/~nikolaos/jdeodorant/files\_JDeodorant/JDeodorant\_Installation\_Guide.pdf
5. https://pmd.github.io/
6. https://www.geeksforgeeks.org/code-smellq-a-general-introduction-and-its-type/
7. https://ediorg.github.io/ecocomDP/articles/model\_overview.html
8. https://github.com/OSSpk/Library-Management-System-JAVA
9. https://github.com/Marwa-Eltayeb/YoutubeDownloader
10. https://github.com/indrekru/design-patterns-spring-boot
11. https://github.com/apache/xerces2-j
12. https://github.com/apache/wicket
13. https://github.com/rick2785/JavaCode
14. https://github.com/bluedskim/javaDesignPatterns
15. https://github.com/melgenek/build\_systems
16. https://github.com/apache/incubator-weex
17. https://github.com/apache/qpid-jms
18. https://github.com/apache/nutch
19. https://github.com/apache/derby
20. https://github.com/marcusvnac/argouml-spl
21. https://github.com/Cyberster/Wipro-Training-Java-Fundamentals---Assignments-for-Flow-Control-Statements
22. https://github.com/bavatar/SwitchStmsRep
23. https://www.ieee.org/conferences/publishing/templates.html
24. http://ddasankaindrajith.blogspot.com/2012/03/how-to-detect-and-re-factor-bad-smells\_11.html
25. Cline, M. P. (1996). The pros and cons of adopting and applying design patterns in the real world. Communications of the ACM, 39(10), 47-49.
26. [Klemola2000] T. Klemola (2000), 'A Cognitive Model for Complexity Metrics', Proceedings of the 4th International Workshop on Quantitative Approaches in Object Oriented Software Engineering.
27. Min Zhang, Tracy Hall, Nathan Baddoo Code bad smells: a review of current knowledge J Softw Maint Evol Res Pract, 23 (2011), pp. 179-202
28. Jehad Al Dallal Identifying refactoring opportunities in object-oriented code: a systematic literature review Inf Softw Technol, 58 (2015), pp. 231-249
29. Chatzigeorgiou, A. and Manakos, A. (2010). “Investigating the Evolution of Bad Smells in Object-Oriented Code”, In: Seventh Intl Conf. Quality of Information and Communications Technology.
30. L. Prechelt and B. Unger, "A Series of Controlled Experiments on Design Patterns: Methodology and Results " in Proceedings of Software Technik, Software Technik Trends, 1998.
31. B. Cardoso and E. Figueiredo, ‘‘Co-occurrence of design patterns and bad smells in software systems: An exploratory study,’’ in Proc. Anais do Simpósio Brasileiro de Sistemas de Informação (SBSI), May 2015, pp. 347–354
32. https://www.techtarget.com/searchsoftwarequality/tip/Understanding-code-smells-and-how-refactoring-can-help#:~:text=Put%20simply%2C%20code%20smells%20are,in%20accordance%20with%20necessary%20standards.
33. https://www.sciencedirect.com/science/article/pii/S0950584916300210#:~:text=Abstract,maintenance%20of%20a%20software%20system.