An Empirical Study on the Evolution of Design smells in Code and Maintainability

By

Priyanka Bandi

Sharath Kolapalli

Lewis University

12/16/2022

Abstract

Design smells is a part of the programming. Design smells are the structures , present in the design that will indicate the opposing the principles of fundamental design and the negatively impacted design quality. Design smell can be identified in many ways such as manual way , semi automatical way , automatical way . In this research paper , we will determine the effect of Design smells in code in terms of maintainability . Also we determine the design smells , the occurrence of design smells in the code , and we determine the impact of maintainability factors on the design smells in code . We will also study about some recommendations to the code users or the developers who are already in the field of design module . We’ll study about the code refactoring when there occurred any bad smells in the code that is developed or under development .

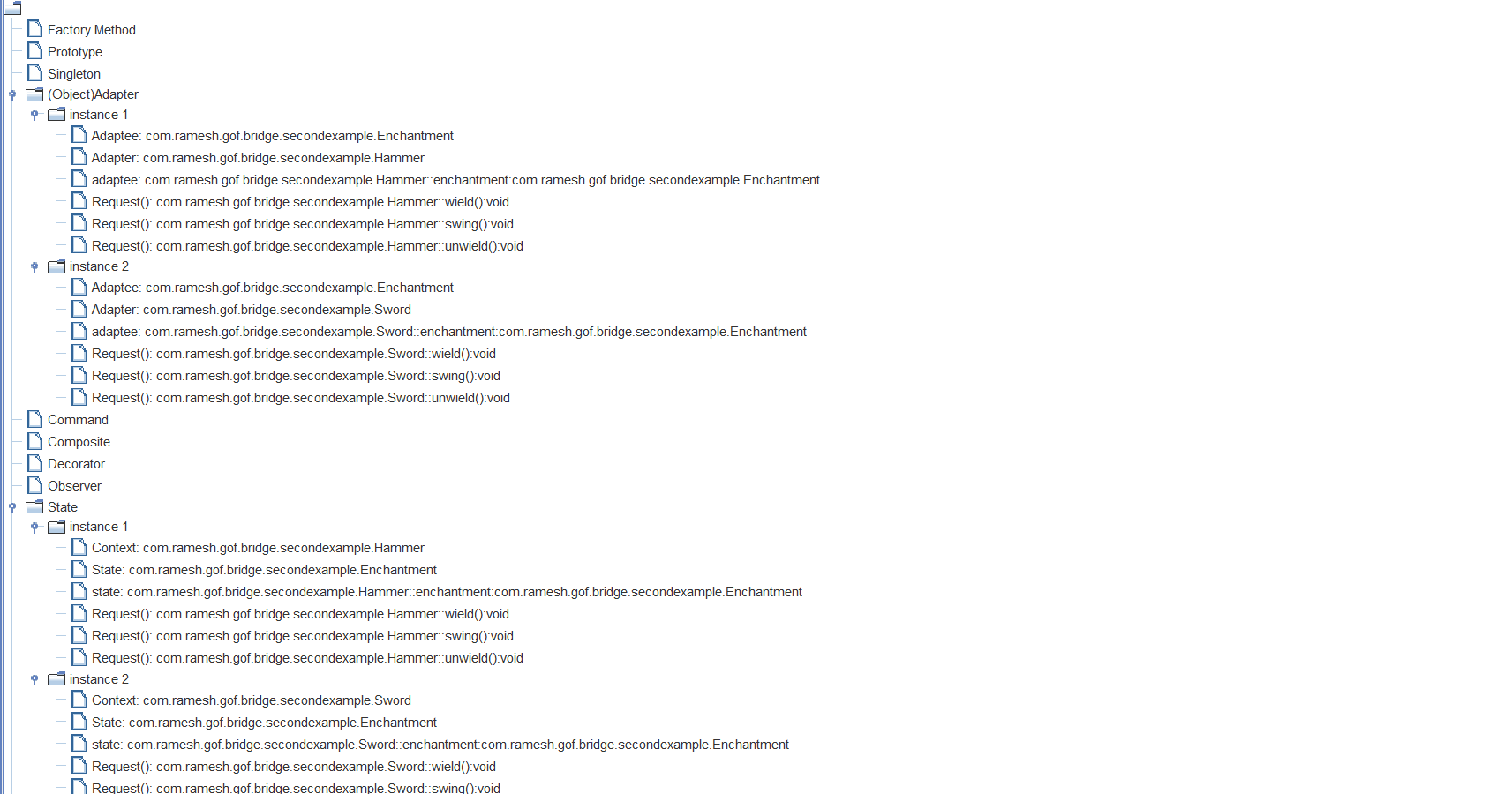
**Introduction**

Design smells will indicate the accumulated form of design debt , it is one of the important dimensions of the technical debt . Any bugs or the features that are not implemented are unaccounted as the design smells . Design smells will occur due to the poor design decisions that will make the design weak and the difficulty in its maintenance . It needs a good practice to indicate the design smells in a software system and will apply the suitable refactoring if the elimination requires and to avoid the accumulated rate of the application rate of technical debt . The context will play an prominent role to choose whether a certain structure or a decision that should be considered as a design smells . Normally it is suitable to be with design smells due to the constraints that is imposed by the context . The design smells has to be tracked and managed as the technical debt[1]

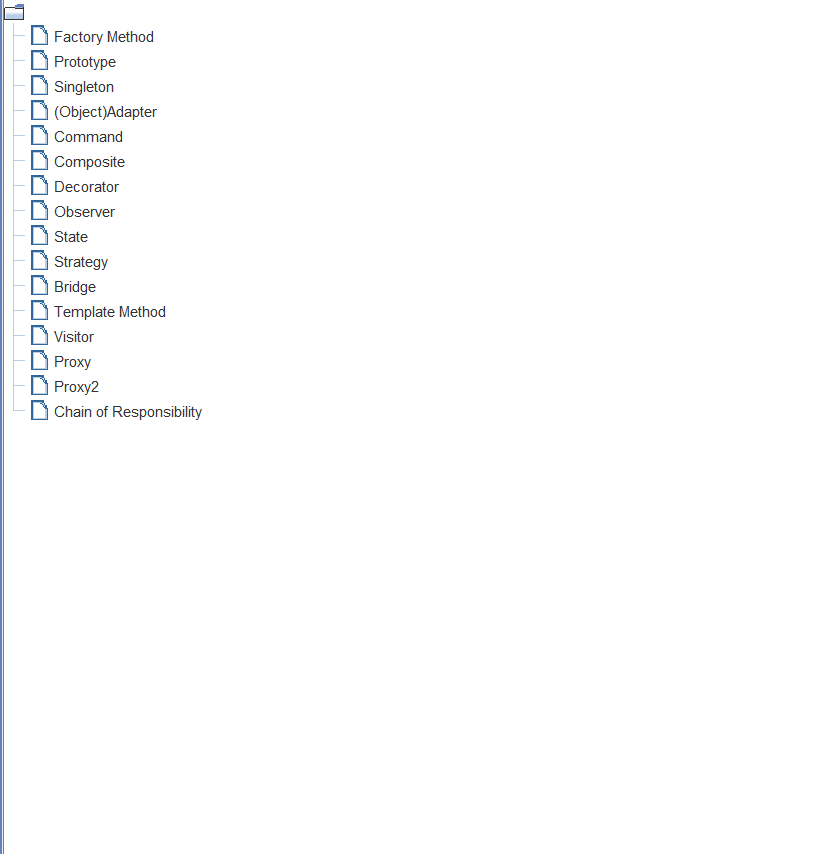
**Design smells :**

Design smells are the part of computer programming . Design smells are the structures in the design that will indicate the violation of the principles of fundamental design and negatively impacted design quality. The design smells are primarily classified into hierarchy , abstraction , modularity , encapsulation .The design smell features are the type of smell , the level of smell , design smell scope , design smell property . It is further featured into the type of smell as the architecture smell , bad pattern , anti patterns , disharmony , code smell and the level of the smell as the low level smell and the high level smell , the smell scope as the method , system , class , subsystem , package and the smell property as the structural , measurable and lexical . Design smells are the design aspects that will violates the fundamental design principles and also violates the Design quality . The bad smells are the certain structures in the code that will negatively has an impact on the software quality . When we analyze the smells we realize that the smells are at the different levels of granularity and are in the same way and refactoring may have different scope . Design smells include the examples such as refused bequest , it is a method in the derived class rejects the method will be inherited from its foundation classes and the classes with multiple responsibilities and cyclically dependable classes [2]

1 Bridge Design Pattern



2 Builder Design Pattern



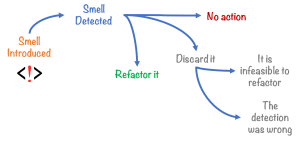
3 Chain Of Responsibility

**Occurance of design smells in code :**

Having design smells not certainly means that the software will not work, and it would still give an output, but it may slow down processing a little , an increased risk of failure and the errors while making the program vulnerable to the errors in the future. Bad smell code will contribute to the poor code quality and hence there is increasing the technical debt . Bad smells are symptoms that indicates there is something wrong in the system design or in the code. Therefore, the design patterns and the bad smells will represent an antagonistic structures. They is a subject of recurring research and that typically appears in the software systems. Design smells can be detected or indicated in different ways .They are : manually , semi-automatically ,automatically .In a manual way , we analyze the code and its design . In a semi-automatically way , we calculate metrics first and then inferring the design smells by combining a few metrics . The other one is automatically way , by which we employ a design smell detection tool. The Slow down processing a little , an increased risk of failure and the errors while making the program vulnerable to the errors in the future. Bad smell code will contribute to the poor code quality and hence there is increasing the technical debt . Bad smells are symptoms that indicates there is something wrong in the system design or in the code. Therefore, the design patterns and the bad smells will represent an antagonistic structures. They is a subject of recurring research and that typically appears in the software systems. Design smells can be detected or indicated in different ways .They are : manually , semi-automatically ,automatically .In a manual way , we analyses the code and its design[7]

**The effect of design smells in code in terms of maintainability**

The code smells are manifestation of the flaws of design that will decrease the maintainability of code . So that the existence of code smell looks like an ideal indicator for the maintainability of assessments . In order to achieve the comprehensive or the descriptive and the exact depending on the code smells we need to know the much about the reflecting factors affecting the maintainability . After the process of identifying which the maintainability factors are reflected by the design smells and which will not then we can use the complimentary means to be allow the factors that are not addressed by this smells . Different code characteristics are suggested that will affect the maintainability . Some of them have attempted to combine them into a solo value that are called as maintainability index . The measures for inheritance , Coupling and the cohesion were suggested to cope up with the object oriented program analysis . the occurrence of design smells in the code , and we determine the impact of maintainability factors on the design smells in code . We will also study about some recommendations to the code users or the developers who are already in the field of design module . We’ll study about the code refactoring when there occurred any bad smells in the code that is developed or under developments . Design smells will indicate the accumulated form of design debt , it is one of the important dimensions of the technical debt . Any bugs or the features that are not implemented are unaccounted as the design smells [6]



Design smells will occur due to the poor design decisions that will make the design weak and the difficulty in its maintenance . It needs a good practice to indicate the design smells in a software system and will apply the suitable refactoring if the elimination requires and to avoid the accumulated rate of the application rate of technical debt . The context will play an prominent role to choose whether a certain structure or a decision that should be considered as a design smells . Normally it is suitable to be with design smells due to the constraints that is imposed by the context . The design smells has to be tracked and managed as the technical debt . Design smells can be indicated manually when the developers uses their experiences and their core knowledge so that they will understand the existing design and can identify the potential refactoring candidate . How ever for a large code base , the manual design smells detection is a time consuming process and is not feasible . There are a few tools that can accurately developed to detect the design smells and to improve the quality of software designs few steps are to be followed to know the procedure of identifying the designing smells for the project . We can also detect the design smells by the semi automatical way . It means by intially calculating the metrics and then we infer the design smells by combining a few metrics . We can also identify the design smells in an automatic way . Automatic way means by employing a small design smell detection tool . This tool is employed and used to detect the design smell [5]

**Overcoming the code refactoring**

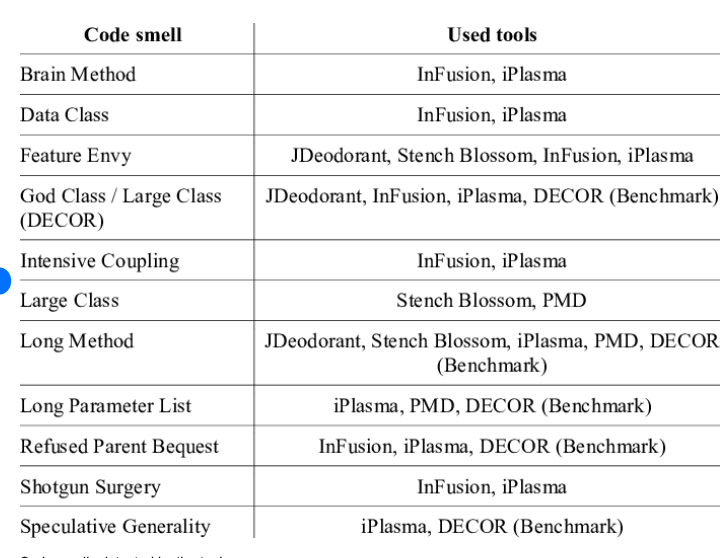
The software developers often rely on a quickly approach towards coding to facilitate fast shipping. However, when it comes to modifying the source code for enhancements, the complexity of the code makes it challenging to reiterate through its program is evolved, its complexity increases unless work is done to maintain or reduce it. It implies minimizing the difference between the current source code and how it should actually be Reduce technical debt: means to reduce messy and unrefined code, which may otherwise lead to additional rework later As a unit tests are executed once before the refactoring and once after refactoring. This helps in validating that the external behavior of the software system is not interrupted along the process. The Thousands of lines of code ) and messy code structures make it non-understandable, leading to an increase in software maintenance costs. To manage the internal quality of the code, its restructuring is essential code .review is a part of the software development process which involves testing the source code to identify bugs at an early stage. A code review process is typically conducted before merging with the codebase . The .Affective code review prevents bugs and errors from getting into your project by improving code quality at an early stage of the software development process. In this post, we’ll explain what code review is and explore popular code review tools that help organizations with the code review process. The primary goal of the code review process is to assess any new code for bugs, errors, and quality standards set by the organization. The code review process should not just consist of one-sided feedback. Therefore, an intangible benefit of the code review process is the collective team’s improved coding skills. If you would like to initiate a code review process in your organization, you should first decide who would review the code. If you belong to a small team, you may assign team leads to review all code. In a larger team size with multiple reviewers, you could enable a process in which every code review is The standard practice for this damage control is called code refactoring [4[.



The code refactoring approach revolves around changing the design and structure of the code so that the software’s external behavior does not get affected. The objective of refactoring is to improve the design of existing code to make it clean and understandable. Here’s everything you should know about code refactoring. Code refactoring is the process of modifying the code’s internal structure so that its external behavior does not get affected. It helps eliminate the poor design choices (anti-patterns) that make a code challenging to understand and maintain. Improve code quality: implies cleaning up code and improving its structure to make it less complex and understandable . Maintain a good software architecture: means to enhance the design of the source code to accelerate the delivery of the new product features . The implies minimizing the difference between the current source code and how it should actually be Reduce technical debt: means to reduce messy and unrefined code, which may otherwise lead to additional rework later As a n th tests are executed once before the refactoring and once after refactoring. This helps in validating that the external behavior of the software system is not interrupted along the process[3]

**Recommendations to code users or developers**

Structural degradation is the process in which quality attributes of a system are negatively impacted. When due attention is not paid to structural degradation Moreover, recent studies show that developers most often apply composite refactoring’s – i.e., sequences of two or more refactoring – for removing code smells. Despite showing the importance of performing composite refactoring, most studies do not provide information on which composite refactoring patterns are recurrent in practice. In this context, a previous study identified 35 smell removal patterns that are frequent across multiple open source systems. However, such study has not explored how the removal patterns could help developers to apply effective composite refactoring’s. Thus, in this work, we propose a suite of new recommendation heuristics to help developers in applying effective composite refactorings.



Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. Manifestation of the flaws of design that will decrease the maintainability of code . So that the existence of code smell looks like an ideal indicator for the maintainability of assessments . In order to achieve the comprehensive or the descriptive and the exact depending on the code smells we need to know the much about the reflecting factors affecting the maintainability . After the process of identifying which the maintainability factors are reflected by the design smells and which will not then we can use the complimentary They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review. A coding standard gives a uniform appearance to the codes written by different engineers. It improves readability, and maintainability of the code and it reduces complexity also. It helps in code reuse and helps to detect error easily. It promotes sound programming practices and increases efficiency of the programmers[2]



These heuristics are intended to remove three code smell types, namely Complex Class, Feature Envy, and God Class. After designing the heuristics, we evaluated their effectiveness through a quasi-experiment. This evaluation was conducted with 12 software developers and 9 smelly Java classes. Results indicate that developers considered our heuristics effective or partially effective in more than 93% of the cases. In addition, the evaluation helped us to identify multiple factors that contribute to the acceptance or rejection of the refactoring recommendations. Based on these factors, we defined new guidelines for the effective recommendation of smell-removal composite refactoring’s , the source code may also become difficult to change. Code smells are recurring structures in the source code that may represent structural degradation. Hence, there are many catalogs and techniques for supporting the removal of code smells through refactoring recommendations, which usually consist of single refactoring such as a Move Method or an Extract Method. However, single refactoring’s are often not enough for completely removing certain smell occurrences . A software development guideline is a piece of information intended to advise developers. In particular, it offers advise on how various aspects of the software development process should be done or what they should be. Furthermore, the advise it offers is non-specific and focuses on rules and principles[1]

References:

[1]. .Hochstein, L.; Lindvall, M. Combating architectural degeneration: A survey. Inf. Softw. Technol.2005,47,643–656. [CrossRef]

2.Garcia, J.; Popescu, D.; Edwards, G.; Medvidovic, N. Toward a Catalogue of Architectural Bad Smells.In Computer Vision; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2009; Volume5581, pp. 146–162.

3.Le, D.; Medvidovic, N. Architectural-based speculative analysis to predict bugs in a software system.In Proceedings of the Proceedings of the 38th International Conference on Software Engineering Companion;Association for Computing Machinery (ACM): New York, NY, USA, 2016; pp. 807–810.

4.Fowler, M. Refactoring: Improving the Design of Existing Code; Springer Science and Business Media LLC:Berlin/Heidelberg, Germany, 2002; Volume 2418, p. 256.

5.Alkharabsheh, K.; Crespo, Y.; Manso, E.; Taboada, J.A. Software Design Smell Detection: A systematicmapping study. Softw. Qual. J. 2018,27, 1069–1148. [CrossRef]

6.Sharma, T.; Mishra, P.; Tiwari, R. Designite. In Proceedings of the 1st International Workshop on Agents andCyberSecurity-ACySE ’14; Association for Computing Machinery (ACM): New York, NY, USA, 2016; pp. 1–4.

7. Raymond, E. The cathedral and the bazaar. Knowl. Soc. 1999,12, 23–49. [CrossRef]

8.Bar, M.; Fogel, K. Open Source Development with CVS, 3rd ed.; Paraglyph Press: Norwich, NY, USA, 2003;ISBN 1-932111-81-6

.9.Sharma, S.; Sugumaran, V.; Rajagopalan, B. A framework for creating hybrid-open source softwarecommunities. Inf. Syst. J. 2002,12, 7–25. [CrossRef]

10.Kumar, L.; Sureka, A. Thirteen Years of Mining Software Repositories (MSR) Conference-What is theBibliography Data Telling Us? arXiv 2016, arXiv:1609.06