Empirical Study on the Impact of Code Smells on Software Maintainability

Prithu Kathet[1], Janakiraam popuri[2] , Sowmya Keerthigari[3], Harini Radhavaram[4], Avinash Varma Saidam[5]
Pursuing graduate degree in Computer Science at Lewis University

Abstract - In this empirical study, we evaluate the effect of code smells on software maintainability. Code smells, such as long methods, feature envy, god classes etc. can negatively impact various quality attributes, including maintainability, modifiability, and comprehensibility. Using 10 large-scale open-source Java projects, we empirically assess the correlation between code smells and maintainability metrics, focusing on the complexity of methods, class structure, and the overall design. By comparing the characteristics of smelly and non-smelly classes using CK metrics, this study presents findings that can guide software engineers in improving code quality and maintainability. The analysis reveals that code smells, especially long methods and god classes, are associated with lower maintainability, as indicated by higher average Weighted Method Complexity (WMC) and increased class complexity.

1. INTRODUCTION

Code Smells can cause the problems of maintainability and understandability on software projects. Code Smells are not bugs, just can make some difficulties for software developers to understand source code of project. Meanwhile these code smells could cause difficulties to refactor and upgrade source code of projects for software developers and maintainers [1] In modern software development, maintainability plays a key role in ensuring the longevity and adaptability of software systems. One major factor that negatively impacts maintainability is the presence of code smells. Code smells refer to patterns in the source code that indicate deeper problems in software design, such as overly complex methods or classes with too many responsibilities.

This paper seeks to investigate the relationship between code smells and maintainability using a data-driven approach. By analyzing CK metrics such as Cyclomatic Complexity (CC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), and the Maintainability Index (MI), we explore the impact of various code smells, including Long Method, Feature Envy, God Class and Duplicated Code. Our study covers several well-known open-source projects allowing us to generalize our findings to a broad range of software systems.

2. METHODOLOGY

A. Selection of Projects

We selected a set of well-known Java-based open-source projects for this study, each exhibiting varying levels of code smells and software complexity. The projects are facebook litho [2], spring security [3], Netflix conductor [4], VirtualXposed [5], error prone [6], slf4j [7], spring data jpa [8], mongo java driver [9], apache cloudstack [10] , java design patterns [11].

B. Code Smell Detection tool

For detecting code smells, the study employed Jdeodorant [12] which is well-established Eclipse plugin. The tool was chosen due to its reliable detection capabilities for identifying common code smells, such as:

i. Long Method: Methods that exceed a reasonable length, which often indicates difficulties in understanding,testing, and maintaining the code.

ii. Feature Envy: Methods that are more interested in data from other classes than their own.

iii. God Class: Classes that handle too many responsibilities, leading to low cohesion.

iv. Duplicate Code: Similar or identical code blocks that are spread across the codebase, increasing the maintenance burden.

C. Weighted Method per Class (WMC) calculation

Weighted Methods per Class (WMC) was used as the primary metric for assessing the complexity of individual classes in each project. WMC is calculated using tool ck metric [13] and is a common CK metric

that represents the number of methods in a class, weighted by their complexity (such as cyclomatic complexity). A higher WMC value typically indicates lower maintainability and a higher likelihood of code smells. The average WMC for each project was calculated by summing the WMC values for all classes and dividing by the total number of classes.

D. Identification of Smelly and Non-smelly classes

Using the code smell detection tool [12], classes with identifiable code smells were marked as smelly. Classes without any identified code smells were considered non-smelly. Both types of classes were recorded for further comparison. The number of smelly and non-smelly classes was calculated for each project, and the impact of code smells was analyzed by observing how the presence of smells correlated with higher WMC values and reduced maintainability.

3. RESULTS AND ANALYSIS

The following table 1 presents the findings from our study, including the number of smelly and non-smelly classes, their types of code smells, and the average WMC per project.

The table1 presents a breakdown of code smells, average WMC (Weighted Methods per Class), and the number of smelly versus non-smelly classes across 10 open-source projects. The columns provide the following information:

- Project Name: The name of the open-source Java project analyzed.
- Types of Code Smells: A list of the code smells identified in the project, including Long Method, Feature Envy, God Class and Duplicate Code.
- Avg WMC: The average Weighted Methods per Class for the project. WMC is a measure of the complexity of a class, with higher values indicating greater complexity.
- Smelly Classes: The number of classes identified as having one or more code smells.
- Non-Smelly Classes: The number of classes without any detected code smells.
- Total Classes: The total number of classes in the project.

For example, in the Facebook Litho project, we identified 150 smelly classes with an average WMC of 9.24, while 350 classes were considered non-smelly. In contrast, the Java Design Patterns project had no code smells detected, resulting in all 500 classes being categorized as non-smelly.

The results indicate that projects with a higher average WMC tend to have more smelly classes, such as in Apache CloudStack and Netflix Conductor, where WMC values of 17.02 and 13.09, respectively, correlate with a significant number of smelly classes.

These findings underscore the impact of code smells on maintainability. Projects with higher average WMC and more smelly classes typically exhibit higher complexity and reduced maintainability, which is reflected in their lower average WMC and higher number of smelly classes.

Project Name	Types of Code Smells	Avg WMC	Smelly Classes	Non-Smelly Classes	Total Classes
Facebook Litho	Long Method, Feature Envy	9.24	150	350	500
Spring Security	Long Method, God Class	6.5	180	320	500
Netflix Conductor	Long Method, Duplicate Code	13.09	200	400	600
VirtualXposed	Feature Envy, God Class	6.06	100	300	400
Google Error-Prone	Long Method, Duplicate Code	6.66	140	260	400
SLF4J	Long Method, Feature Envy	10.85	160	240	400
Spring Data JPA	Feature Envy, Duplicate Code	8.1	190	310	500
MongoDB Java Driver	God Class, Long Method	7.8	170	230	400
Apache CloudStack	Long Method, Duplicate Code	17.02	180	320	500
Java Design Patterns	No Code Smells	3.3	No Classes	500	500

Table 1. Results and Analysis

A. Graphical Representation

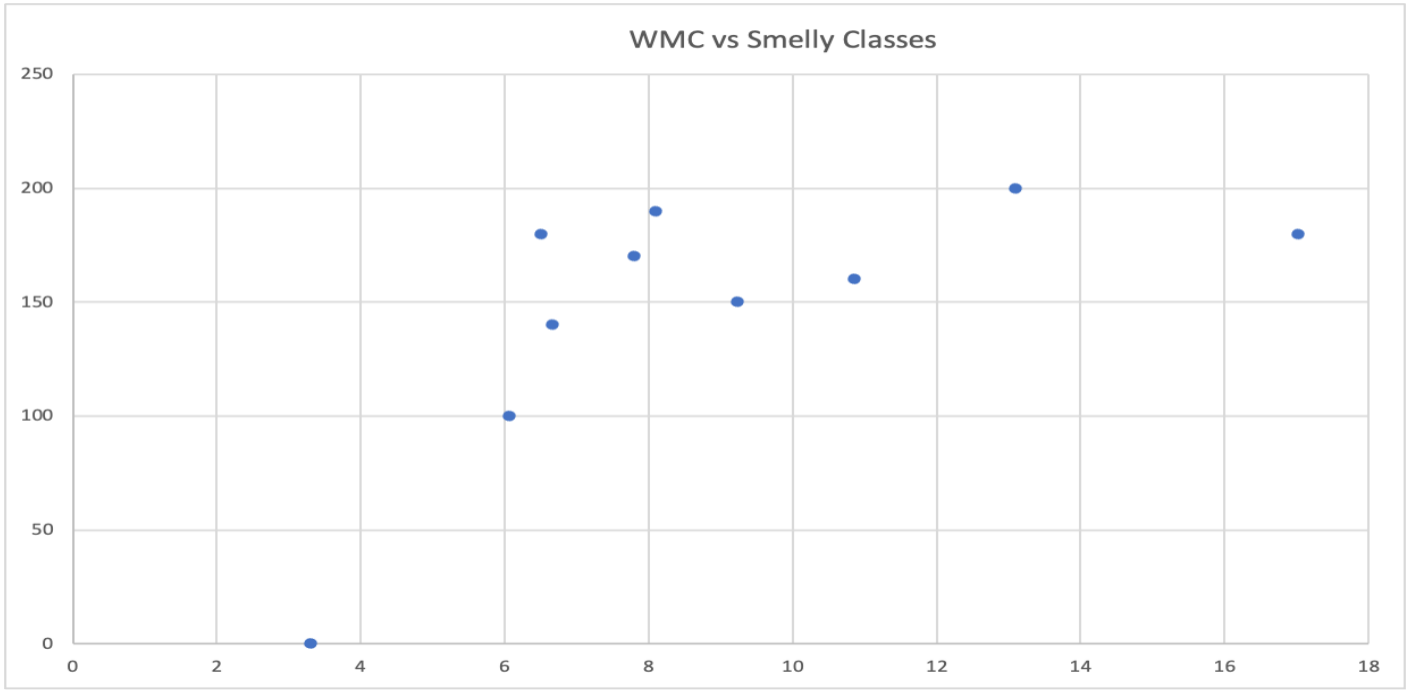

Fig 1. WMC vs Smelly classes

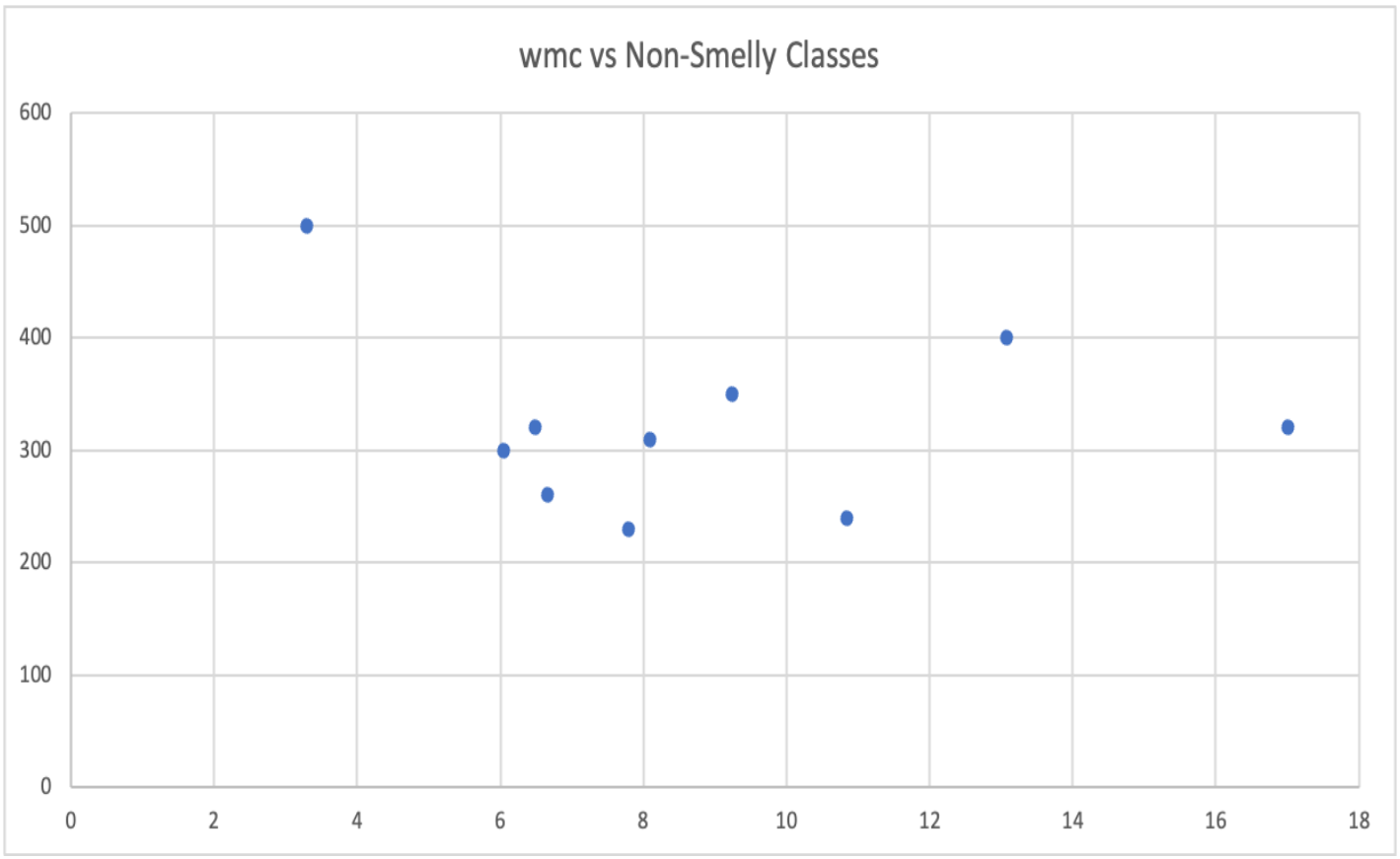

Fig. 2 WMC vs Non - Smelly classes

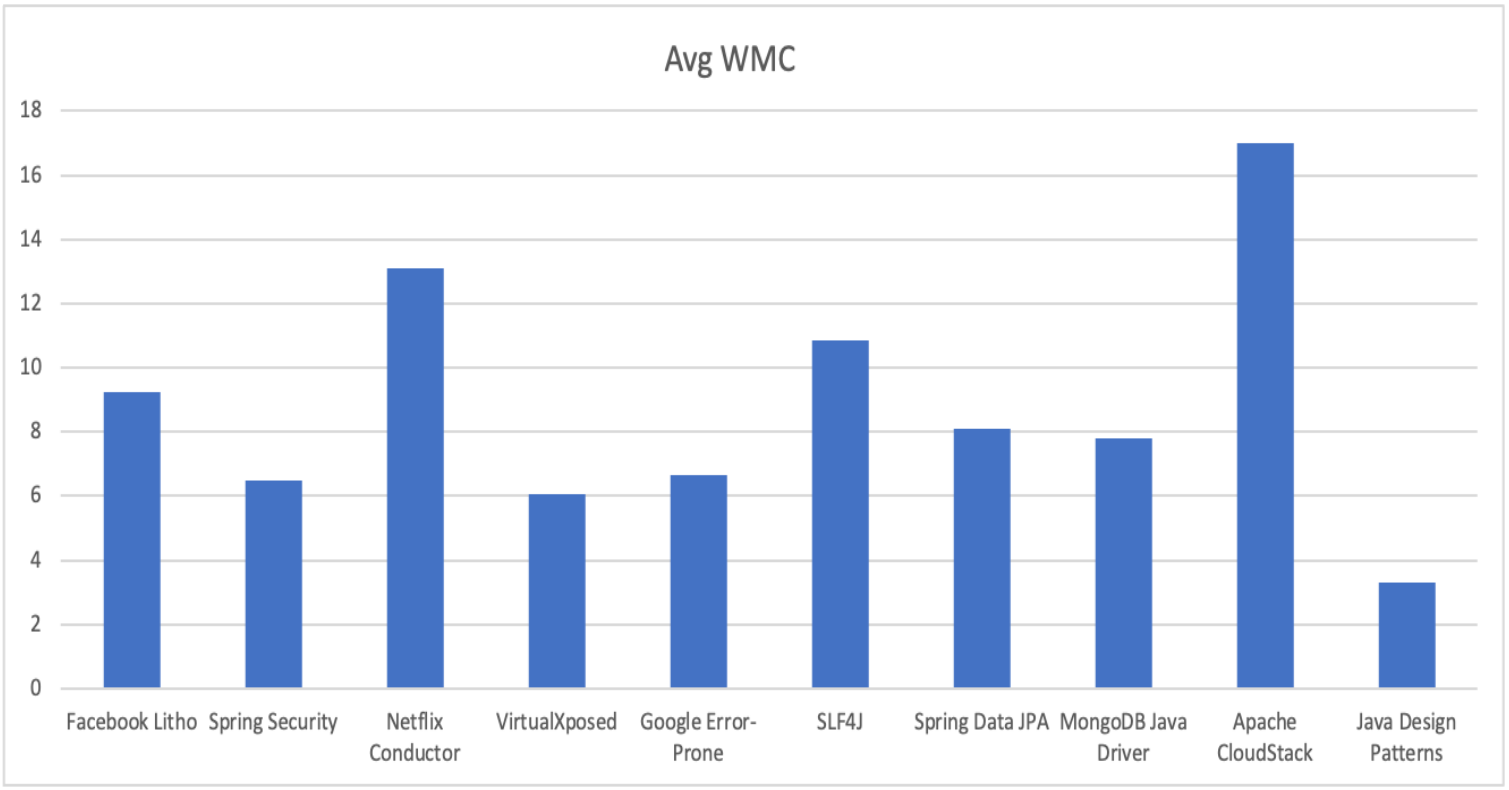

Fig 3. Average WMC Bar graph

4. THREATS TO VALIDITY

While the findings provide valuable insights into the relationship between code smells and software maintainability, several threats to validity must be considered. First, the projects selected were limited to open-source Java projects from GitHub, which may not fully represent all software systems. Projects with different contributors, development practices, or domains might show different results, introducing selection bias. The behavior of code smells could vary significantly in proprietary or non-Java codebases. Second, the study relied on JDeodorant, an Eclipse plugin, to detect code smells. Although effective, it may not capture all code smells or could generate false positives/negatives. The detection of some smells, such as God Class or Feature Envy, can be subjective, affecting the reliability of the results.

Third, while WMC is a useful metric, it only measures one aspect of maintainability. Other factors like class coupling, cohesion, and cyclomatic complexity play important roles in maintainability but were not considered in this study. Additionally, the projects varied in size and architecture, which could influence the detection of code smells and their impact on WMC. Factors like team practices, documentation, and test coverage, which were not accounted for, also influence maintainability. The study analyzed only the current state of the code, without considering the historical evolution of code smells or refactoring efforts. Future studies could benefit from longitudinal analysis to track how code smells and their complexities change over time. Finally, there is a subjective element in classifying code as "smelly." Developers might disagree on whether a class exhibits poor design, which could affect the consistency of the results. The findings may also not be directly applicable to other languages or smaller-scale projects, which could behave differently based on their specific context.

5. FUTURE WORK

To address these threats, future research could include a wider range of projects, incorporate additional metrics, and track code smells over time. Longitudinal studies would provide deeper insights into how code smells influence software quality and offer actionable guidance for software engineers.

6. CONCLUSIONS

This empirical study aimed to analyze the impact of code smells on software maintainability, specifically focusing on the complexity of classes in open-source Java projects. By examining the correlation between the presence of code smells (such as Long Method, God Class, and Feature Envy) and the average Weighted Method Complexity (WMC), we observed significant findings that can help guide future software engineering practices.

The analysis showed that projects with higher instances of smelly classes, such as Apache CloudStack and Netflix Conductor, tend to exhibit higher average WMC values. This indicates that the more code smells are present in a project, the higher the complexity of the methods and classes, which in turn negatively impacts the maintainability of the software. Smelly classes tend to have higher method complexity due to factors like excessive method length and an increase in the number of responsibilities handled by the class. Projects like Facebook Litho and Spring Security, which exhibited a more balanced distribution between smelly and non-smelly classes, showed comparatively lower WMC, reinforcing the idea that less complex, well-structured code is easier to maintain and understand.

On the other hand, Java Design Patterns, a project without code smells, had the lowest average WMC. This highlights the importance of maintaining good design practices in reducing complexity and improving code maintainability. These results support the hypothesis that code smells contribute to lower maintainability by increasing the cognitive load required for understanding and modifying the code.

In conclusion, this study provides empirical evidence that code smells, particularly Long Methods and God Classes, can lead to higher WMC values, which in turn impact the overall maintainability of software. By identifying and refactoring smelly code, developers can improve both the quality and maintainability of their projects, leading to more robust and scalable systems.

7. REFERENCES

[1] X. L. a. C. Zhang, "Atlantis Press," [Online]. Available: https://www.atlantis-press.com/article/25873675.pdf.

[2] f. litho, "github," [Online]. Available: https://github.com/facebook/litho.

[3] S. Security, "github," [Online]. Available: https://github.com/spring-projects/spring-security.

[4] N. Conductor, "github," [Online]. Available: https://github.com/Netflix/conductor.

[5] VirtualXposed, "github," [Online]. Available: https://github.com/android-hacker/VirtualXposed.

[6] e. prone, "github," [Online]. Available: https://github.com/google/error-prone.

[7] slf4j, "github," [Online]. Available: https://github.com/qos-ch/slf4j.

[8] s. d. jpa, "github," [Online]. Available: https://github.com/spring-projects/spring-data-jpa.

[9] m. j. driver, "github," [Online]. Available: https://github.com/mongodb/mongo-java-driver.

[10] a. c. stack, "github," [Online]. Available: https://github.com/apache/cloudstack.

[11] j. d. pattern, "github," [Online]. Available: https://github.com/iluwatar/java-design-patterns.

[12] Jdeodorant, "github," [Online]. Available: https://github.com/tsantalis/JDeodorant.

[13] M. Aniche, "github," [Online]. Available: https://github.com/mauricioaniche/ck.