Empirical Analysis of the Impact of Code Smells on Maintainabilityin Large-Scale Java Projects

Joel Sandeep Bethi, Pavan Kalyan Yerramsetti, Rishitha Musunuru, Samba Siva Rao Komerisetti, Vijesh Medikonda
Lewis University

***Abstract*—This study conducts an empirical evaluation to explore the impact of code smells on software quality attributes across ten substantial Java projects, including prominent open-source endeavors such as Apache Beam, Wildfly, and Netflix Conductor. By leveraging tools like PMD for the detection of code smells and CK metrics for assessing software quality, this research seeks to pinpoint how specific bad smells influence maintainability, a vital software quality attribute. Our approach involves a detailed analysis of each selected project, all of which exceed 5,000 lines of code, to systematically identify prevalent code smells and their correlation with various quality metrics derived from the CK suite. Expected outcomes aim to provide insightful conclusions regarding the relationship between code smells and software maintainability, thereby aiding developers and project managers in making informed decisions about refactoring priorities. This paper not only highlights the adverse effects of code smells on maintainability but also proposes a replicable methodological framework for ongoing research in software quality evaluation. Through a structured methodology for data collection and analysis, this study contributes to the broader understanding of software maintainability issues in the context of code smells, offering empirical evidence to support best practices in software development and maintenance.**

I. Introduction

Code smells, indicative of deeper issues within software design, pose significant challenges to maintaining software quality. These inefficiencies often complicate modifications and enhancements, leading to increased costs and delayed timelines. This research aims to empirically assess the impact of code smells on the maintainability of software within large Java projects. By focusing on ten diverse, open-source Java projects, each comprising over 5,000 lines of code, this study utilizes PMD for detecting code smells and the CK metric suite to evaluate maintainability. The objective is to systematically identify and quantify the relationship between specific bad smells and their effects on maintainability, providing a data-driven foundation for refactoring decisions. This analysis seeks not only to highlight the detrimental effects of code smells on maintainability but also to offer insights into mitigating these effects through targeted improvements.

This paper is structured to first outline the methods and approaches employed in collecting and analyzing data, followed by a detailed presentation and discussion of the results. We will then address potential threats to the validity of our findings and conclude with a summary of the implications and directions for future research. This study contributes to the broader discourse on software quality by offering empirical evidence and methodological rigor in the evaluation of code smells' impact on maintainability.

II. Method or Approach

A. Data Collection

This study utilizes data from ten large-scale Java projects, each exceeding 5,000 lines of code. These projects, selected from GitHub, provide a diverse basis for analysis. We collect data using the following tools:

1) Code Smell Detection: PMD is employed to detect a range of code smells within each project's source files.
2) Software Metrics: The CK metric suite gathers key software quality metrics such as Coupling Between Object Classes (CBO), Depth of Inheritance Tree (DIT), and Weighted Methods per Class (WMC).

B. Data Preparation

Data from PMD and CK metrics are merged based on filename and project attributes using Python scripts. This merged dataset ensures that each software metric is paired with its corresponding code smell data, forming the foundation for subsequent analyses.

C. Analytical Methods

The analysis proceeds through several structured steps:

1) Descriptive Statistics: We provide an overview of the datasets to establish a foundational understanding of the general distribution and trends of code smells and CK metrics across the selected projects.
2) Correlation Analysis: Pearson correlation coefficients are calculated to identify relationships between the frequency and types of code smells and various CK metrics. This step aims to highlight which specific bad smells correlate strongly with changes in software maintainability metrics.
3) Comparative Analysis:
   - Intra-Project Analysis: Within each project, files are grouped based on the intensity of specific bad smells. We then compare the average CK metrics across these groups to observe how varying levels of bad smells impact software metrics.
   - Inter-Project Analysis: Projects are compared against each other based on their average code smell intensity and corresponding CK metrics. This comparison helps to identify if projects with higher

instances of certain bad smells consistently show poorer maintainability metrics.

4) Visualization: Key findings from the correlation and comparative analyses are visualized using scatter plots, histograms, and box plots. These visualizations aid in illustrating the relationships and differences uncovered in the analytical processes.

D. Reproducibility

To ensure that the study can be replicated accurately, detailed command logs, configurations, and procedures are provided:

- PMD Command: To run PMD and detect code smells, use: pmd.bat -d [source_directory] -R rulesets/-java/quickstart.xml -f csv ¿ [output_file_path] Replace [source_directory] with the path to the Java project source files and [output_file_path] with the path where you want to save the output.
- CK Metrics Command: To gather CK metrics, execute: java -jar ck-x.x.x-SNAPSHOT-jar-with-dependencies.jar -p [project_directory] -o [output_directory] Here, replace [project_directory] with the path to the Java project directory and [output_directory] with the directory path where CK metrics should be saved.

E. PMD Rule Selection and Rationale

To assess maintainability, we use a targeted PMD ruleset focusing on complexity, structure, and resource management. This includes rules like Cyclomatic Complexity and NPath Complexity to evaluate method complexity, and GodClass and DataClass to identify classes that are overly complex or underutilized. LawOfDemeter promotes low coupling to enhance code modularity. Resource management is addressed via UseTryWithResources, and cleanliness through UnusedLocalVariable, ensuring our analysis captures essential aspects of software maintainability.

III. Results and Discussion

This section presents and discusses the results obtained from the empirical analysis of the impact of code smells on maintainability across ten large-scale Java projects. The analysis was carried out using CK metrics and code smell intensities, and the results are summarized through statistical tables, correlation matrices, and scatter plots.

A. Overview of Project Metrics

The table below provides a summary of the mean values for selected metrics across the ten projects analyzed. These metrics include Coupling Between Objects (CBO), Weighted Methods per Class (WMC), and the intensity of various code smells like God Class and Law of Demeter. The data offers an insight into the maintainability challenges associated with each project.

Project	cbo	wmc	CyclomaticComplexity	Data Class	God Class	LawOfDemeter	UnusedLocalVariable	UnusedPrivateMethod
LSPosed	5.522	11.101	1.844	1.000	1.182	17.345	1.000	1.167
SmartTubeNext	6.627	16.569	2.478	1.256	1.066	10.422	1.385	1.539
beam	7.384	6.924	1.369	1.100	1.077	3.491	3.171	1.477
conductor	7.498	13.090	1.605	1.095	1.000	5.795	1.143	1.000
dataease	4.823	7.600	2.854	1.031	1.000	2.724	1.913	1.000
iceberg	8.590	11.551	1.946	1.400	1.183	5.249	2.313	1.481
metersphere	6.533	12.653	1.406	1.012	1.000	3.163	2.333	1.222
spring-cloud-netflix	7.529	7.740	1.000	1.000	0.000	3.364	0.000	2.000
supertokens-core	9.598	16.741	1.777	1.143	1.024	11.483	2.000	4.000
wildfly	6.455	6.527	1.754	1.463	1.006	3.232	1.405	1.338

Fig. 1. Mean CK Metrics and Code Smell Intensities Across Projects

B. Correlation Analysis

The correlation matrix highlights the relationships between CK metrics and different code smells. Notable observations include a strong positive correlation between CBO (Coupling Between Objects) and UnusedPrivateMethod (0.74). This suggests that as coupling increases, there are likely more private methods that are not utilized, which can indicate poor modularity or unused code that could be removed for better maintenance. Another interesting correlation is between GodClass and UnusedLocalVariable (0.65), suggesting that classes that tend to accumulate too much responsibility also often contain unused variables. Law of Demeter shows a moderate positive correlation with WMC (0.56), indicating that loosely coupled code may still exhibit high complexity in terms of methods per class. This matrix is instrumental in identifying key areas where code smells directly impact maintainability metrics.

C. Comparative Analysis

Intra-Project Analysis: The scatter plot (Figure 3) shows the relationship between God Class intensity and WMC across projects. It highlights how projects like SmartTubeNext and beam show higher WMC as the God Class intensity increases, indicating these classes may be central hubs of complexity.

This scatter plot illustrates the relationship between the mean God Class intensity and the average WMC (Weighted Methods per Class) for each project. The use of mean values provides an overall indication of how projects manage complexity and responsibility distribution within their classes. Projects like SmartTubeNext and LSPosed show higher average WMC values corresponding with higher mean God Class intensity, suggesting that these projects have classes with a significant accumulation of responsibilities, which results in a higher number of methods. This pattern highlights the need for refactoring efforts to break down these classes into smaller, more focused units to reduce complexity and improve maintainability. Conversely, projects such as dataease and wildfly exhibit lower mean God Class intensities and

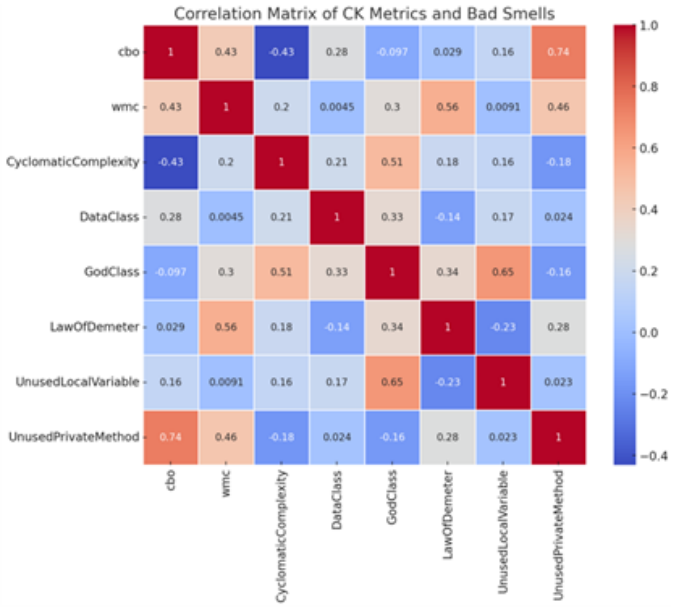

Fig. 2. Correlation Matrix of CK Metrics and Bad Smells

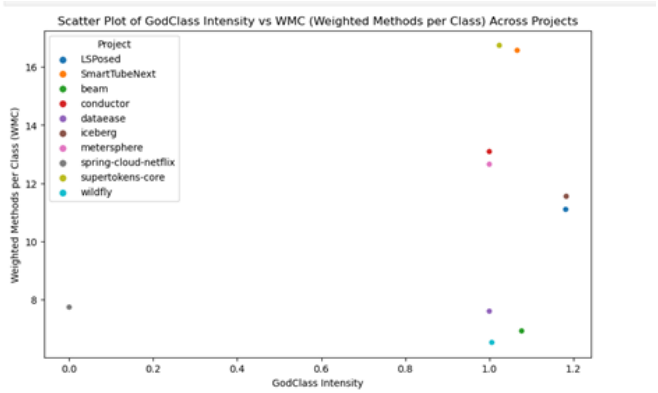

Fig. 3. Scatter Plot of God Class Intensity vs WMC

lower WMC, indicating a more modular and manageable approach to class design. This plot effectively emphasizes the benefits of avoiding God Classes to keep method complexity at a manageable level, promoting better software quality and maintainability. Inter-Project Analysis: The scatter plots below provide further comparisons: Figure 4 shows the relationship between CBO and WMC across all projects, illustrating how projects with higher coupling often display higher method complexity. This scatter plot demonstrates the relationship between CBO (Coupling Between Objects) and WMC across the selected projects. A pattern emerges where higher CBO values correlate with increased WMC, suggesting that projects with higher coupling also tend to have more complex classes in terms of methods. This is particularly evident in projects like metersphere and iceberg. The variation in the spread indicates that while some projects maintain low coupling and method complexity, others exhibit significantly higher values, pointing to potential maintainability issues. This plot

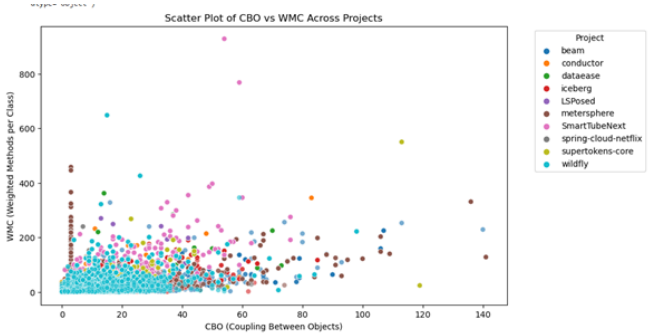

Fig. 4. Relationship Between CBO and WMC

emphasizes the importance of reducing dependencies between classes to manage method complexity better and enhance code modularity. Figure 5 visualizes the connection between LOC (Lines of Code) and WMC, emphasizing that as the size of classes increases, so does the complexity. The scatter plot

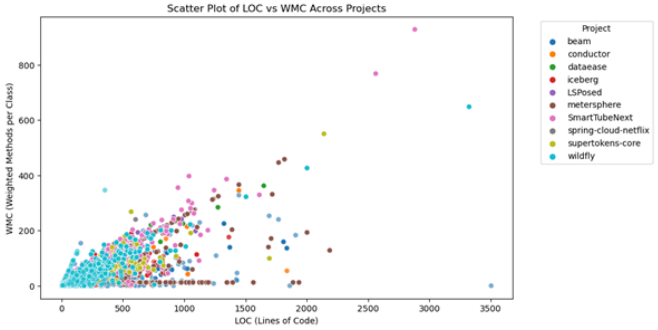

Fig. 5. Connection between LOC and WMC

illustrates the relationship between LOC (Lines of Code) and WMC across all projects. As expected, there is a general trend where an increase in the number of lines per class correlates with an increase in method complexity (WMC). Projects such as LSPosed and SmartTubeNext show extreme values where classes with over 3000 lines of code have high WMC values, indicating large, complex classes. Conversely, projects with smaller class sizes, such as spring-cloud-netflix and wildfly, demonstrate lower method complexity. This plot reinforces the necessity of maintaining smaller class sizes to manage complexity, making the codebase more maintainable and easier to modify or refactor.

D. Threats to Validity

In empirical studies, especially in software engineering, various factors can influence the results, and it is crucial to address these potential threats to ensure the reliability of the findings. In this study, we identified several threats to validity and took steps to minimize their impact:

1) Construct Validity: This threat concerns whether the metrics and methods used truly measure the maintainability aspect we intended to investigate. To mitigate this, we selected well-established tools like PMD for

detecting code smells and CK metrics, which are recognized for their reliability in assessing software quality attributes. The rules and metrics were carefully chosen to align with the focus on maintainability, ensuring that the findings directly relate to our research objective.

2) Internal Validity: The accuracy of the data collection and integration process could influence the validity of the results. We minimized this threat by automating the data collection process using scripts that consistently applied the PMD and CK metric tools across all projects. Additionally, the merging of data was handled programmatically, reducing the likelihood of manual errors.
3) External Validity: External validity pertains to the generalizability of the study findings to other contexts or projects. While the study involved a diverse selection of ten Java projects of significant size from different domains, it is possible that the results may not apply universally to all Java projects or those from other programming languages. We addressed this by ensuring our sample included projects of varying complexity and purpose (e.g., Apache Beam, Netflix Conductor), enhancing the diversity and thus the potential generalizability of our findings.

IV. Conclusion

This study provides empirical evidence on the impact of code smells on the maintainability of large-scale Java projects, focusing on a selection of ten prominent open-source projects including LSPosed, Apache Beam, Wildfly, and Netflix Conductor. By analyzing a wide range of CK metrics and code smell intensities, we identified significant correlations that highlight how specific bad smells like "God Class" and "Law of Demeter" directly influence maintainability attributes such as complexity (WMC) and coupling (CBO). Projects such as LSPosed and SmartTubeNext exhibited higher levels of method complexity (WMC) correlated with increased intensity of "God Class" code smells, suggesting that these projects may benefit from targeted refactoring efforts to improve modularity and reduce complexity. In contrast, projects like dataease and wildfly demonstrated lower levels of these smells, indicating better adherence to modularity principles, which corresponded to better overall maintainability metrics. Our findings underline the importance of monitoring and mitigating code smells in software development to maintain code quality and facilitate future enhancements. Developers and project managers are encouraged to prioritize refactoring efforts on identified hotspots, particularly focusing on classes that exhibit high complexity and coupling. Additionally, the methodological approach presented in this study provides a replicable framework for evaluating and improving software maintainability through code smell analysis and CK metric assessment. The diversity of projects analyzed and the correlations observed support the broader application of these findings across other Java projects, emphasizing that addressing code smells is not just beneficial but essential for maintaining the long-term quality and maintainability of software systems. This research contributes valuable insights into software maintenance practices, aiding the development community in making data-driven decisions for software quality improvement.

References

[1] DataEase, "DataEase," [Online]. Available: https://github.com/dataease/dataease. [Accessed: Oct. 17, 2024].
[2] LSPosed, "LSPosed," [Online]. Available: https://github.com/LSPosed/LSPosed. [Accessed: Oct. 17, 2024].
[3] Metersphere Team, "Metersphere," [Online]. Available: https://github.com/metersphere/metersphere. [Accessed: Oct. 17, 2024].
[4] Yuliskov, "SmartTubeNext," [Online]. Available: https://github.com/yuliskov/SmartTubeNext. [Accessed: Oct. 17, 2024].
[5] SuperTokens, "SuperTokens Core," [Online]. Available: https://github.com/supertokens/supertokens-core. [Accessed: Oct. 17, 2024].
[6] Apache, "Iceberg," [Online]. Available: https://github.com/apache/iceberg. [Accessed: Oct. 17, 2024].
[7] WildFly, "WildFly," [Online]. Available: https://github.com/wildfly/wildfly. [Accessed: Oct. 17, 2024].
[8] Apache, "Beam," [Online]. Available: https://github.com/apache/beam. [Accessed: Oct. 17, 2024.
[9] Spring Cloud, "Spring Cloud Netflix," [Online]. Available: https://github.com/spring-cloud/spring-cloud-netflix. [Accessed: Oct. 17, 2024.
[10] Netflix, "Conductor," [Online]. Available: https://github.com/Netflix/conductor. [Accessed: Oct. 17, 2024.
[11] PMD, "PMD: Source Code Analyzer," [Online]. Available: https://pmd.github.io/. [Accessed: Oct. 17, 2024.
[12] M. Aniche, "CK Java Metrics," [Online]. Available: https://github.com/mauricioaniche/ck. [Accessed: Oct. 17, 2024.