An Empirical Evaluation of the Impact of Code Bad Smells on Software Testability Using Static Analysis Tool

Aasritha Thota, Devi Sree Dornala, Madhuri Bajjuri, Manoj Kumar Bochu, Meghana Reddy Ganugapenta
Lewis University

***Abstract*—This study empirically evaluates the impact of code bad smells on software testability across ten prominent open-source Java projects, including Apache Cassandra, Apache Kafka, and the Spring Framework, using static analysis tools. By analyzing code smells such as God Class, Long Method, and violations of the Law of Demeter, the research investigates their correlation with key software metrics from the Chidamber and Kemerer (CK) suite, specifically Weighted Methods per Class (WMC) and Coupling Between Objects (CBO). The study employs PMD for bad smell detection and a CK metrics analyzer to systematically assess code complexity, coupling, and their relationship with bad smells across diverse codebases. Findings reveal that higher levels of complexity (e.g., elevated WMC) and coupling (e.g., high CBO) are significantly correlated with an increase in code smells, which negatively impacts software testability, particularly in large and complex projects like CoreNLP and Apache Kafka. These results highlight the importance of managing complexity and coupling, and promoting modular design to enhance software quality. The study offers actionable insights by demonstrating the practical benefits of integrating static analysis tools into the development lifecycle, enabling proactive identification and mitigation of bad smells to improve maintainability and testability in large-scale software systems.**

I. Introduction

Software quality is a fundamental aspect of software engineering that directly influences development costs, system reliability, and overall project success. One critical dimension of software quality is testability, which refers to the ease with which software can be tested to detect defects and verify its functionality. High testability enables efficient testing processes, which not only reduce development time but also ensure greater confidence in software releases. However, several factors can impede testability, including code bad smells—poor coding practices or structural issues that indicate deeper design problems within the codebase. Examples of such bad smells include God Classes, which violate principles of modularity, Long Methods that obscure readability, and violations of the Law of Demeter, which suggest overly tight coupling between classes. While these bad smells may not always manifest as immediate bugs, they often complicate testing, increase maintenance costs, and degrade the overall quality of the software. This study is motivated by the need to empirically validate the theoretical claims that code bad smells significantly impact software testability. Despite widespread awareness of bad smells in the software engineering community, there is a need for quantitative evidence that links these smells to testability issues in real-world projects. By focusing on a diverse range of open-source Java projects, this research seeks to identify specific bad smells that correlate with reduced testability and to quantify this relationship using established software metrics from the Chidamber and Kemerer (CK) suite, such as Weighted Methods per Class (WMC) and Coupling Between Objects (CBO). This analysis provides a comprehensive view of how design complexity and coupling influence the occurrence of bad smells and subsequently affect testability. The structure of this paper begins with the methodological approach used to measure bad smells and analyze their impact on testability. The following section presents a detailed analysis of the results obtained from the ten selected projects. This is followed by a discussion that interprets the findings and their implications for software development practices, particularly in relation to modular design and testing strategies. The paper also addresses potential threats to the validity of the results to ensure a balanced and rigorous analysis. Finally, the conclusion summarizes the findings and discusses their broader implications for improving software quality through proactive design and testing practices.

II. Method or Approach

The study aimed to evaluate the impact of code bad smells on software testability using a systematic approach involving static analysis tools and the Chidamber and Kemerer (CK) metrics suite. The methodology was structured in several key phases: selection of software projects, data collection, and analysis using static analysis tools and CK metrics. This section outlines the approach in sufficient detail to enable replication of the study. Selection of Software Projects Ten open-source Java projects were selected for analysis based on their diversity in application domains, codebase size, and developer activity. The chosen projects included well-known frameworks and applications such as Apache Cassandra, Apache Kafka, CoreNLP, and Spring Framework. Each project varied in complexity, ranging from simple libraries to comprehensive frameworks, ensuring a diverse representation of different software development scenarios. This diversity provided a basis for generalizing the findings and understanding the relationship between code smells and testability across different types of software systems. Data Collection To maintain consistency, the source code for each project was cloned from its respective repository as of a specific snapshot. This ensured that the analysis was based on the same version of the codebase across all tools. Two primary tools were used for data collection: PMD (for detecting bad smells) and a CK metrics analyzer (for computing software metrics). The PMD tool was configured with a custom ruleset designed to identify bad smells known to affect testability, such as God Class, Long Method, and violations of the Law of Demeter. The CK metrics suite was used to compute object-oriented design metrics such as Weighted Methods per Class (WMC) and Coupling Between Objects (CBO), which are indicative of code complexity and coupling, respectively. Bad Smells Detection To detect code bad smells, PMD was executed against the entire codebase of each project using the following command: pmd.bat -d ”dir-path” -R ”path-to-customrules.xml” -f csv ¿ output.csv Here, ”dir-path” specifies the project directory, and ”path-to-customrules.xml” contains the custom ruleset targeting specific bad smells. This ruleset includes checks for God Class, Long Method, Law of Demeter violations, and other indicators known to complicate testing and maintenance. CK Metrics Computation CK metrics were collected using a custom CK metrics analyzer, which was executed on each project with the command: java -jar ck-0.7.1-SNAPSHOT-jar-with-dependencies.jar ”dir-path” false 0 false ”output-path” This command captures a wide range of software metrics, including WMC and CBO, for each class within the codebase. These metrics provided a quantitative basis for evaluating design quality and complexity. The data was collected for every class, allowing for detailed analysis and correlation with the detected bad smells. Data Analysis The collected data from PMD and the CK metrics analyzer were merged based on file and project identifiers. Aggregated counts of bad smells per file were computed, and these were matched with the corresponding CK metrics for each class. Scatter plots were generated to visualize the relationship between CK metrics (e.g., WMC and CBO) and the count of bad smells. Additionally, summary tables were created to present the mean, median, and maximum values for each metric across the ten projects.

III. Results and Discussion

The results of this study demonstrate the relationship between code bad smells and software testability, as analyzed across ten prominent open-source Java projects. By correlating software metrics from the Chidamber and Kemerer (CK) suite—such as Weighted Methods per Class (WMC) and Coupling Between Objects (CBO)—with bad smell counts, the study provides empirical evidence that higher complexity and coupling are associated with increased instances of code bad smells.

A. Correlation Between WMC and Bad Smells

The first scatter plot (Figure 1) shows the relationship between Weighted Methods per Class (WMC) and the number of bad smells detected in the analyzed projects. The plot reveals that files with low WMC values (typically below 100) generally maintain lower bad smell counts, indicating that these classes are likely well-structured and adhere to modularity principles. However, as WMC values increase beyond 100, there is a notable rise in the number of bad smells. Outliers with WMC values exceeding 600 often correspond to files with significantly higher bad smell counts, some reaching over 800. This trend suggests that files with high complexity are more prone to structural issues that impede testability. These results align with software engineering principles, where high complexity is expected to hinder testing efforts due to the challenge of understanding and managing the behavior of such classes. The presence of extreme outliers highlights critical areas within the projects that require immediate attention for refactoring. Reducing WMC through decomposition into smaller methods or classes could effectively minimize the bad smells in these files, thus enhancing the testability and maintainability of the software.

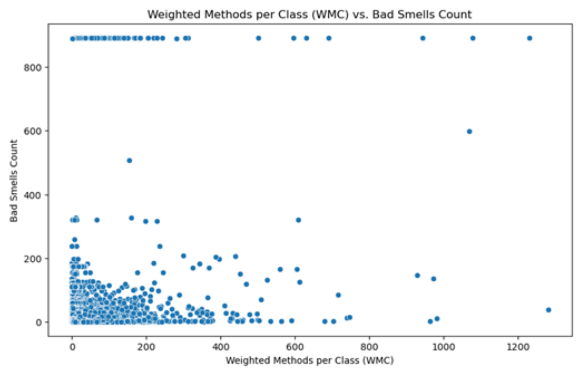


Fig. 1. Correlation Between WMC and Bad Smells

B. Relationship Between CBO and Bad Smells

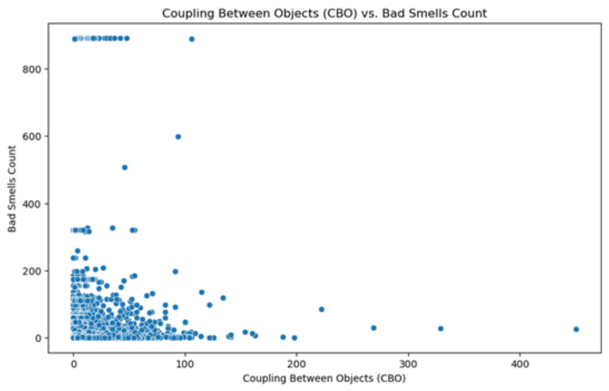
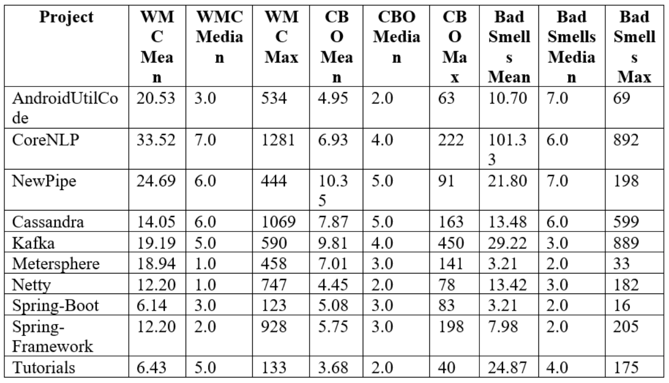


Fig. 2. Relationship Between CBO and Bad Smells

The second scatter plot (Figure 2) investigates the relationship between Coupling Between Objects (CBO) and the prevalence of bad smells. The data indicates that files with lower CBO values (below 100) generally maintain lower bad smell counts, suggesting that reduced coupling helps to minimize structural flaws. However, as CBO values increase—particularly beyond 150—there is a clear increase in the count of bad smells. This relationship is especially evident in projects such as CoreNLP and Apache Kafka, where high coupling values correspond to files with hundreds of detected bad smells. These findings confirm the theoretical expectation that tightly coupled components are more challenging to test. High CBO indicates that a class depends heavily on other classes, making it difficult to isolate and test its behavior independently. Files exhibiting high CBO values and corresponding high bad smell counts are likely to introduce risks in software testing, maintenance, and scalability. Addressing these issues requires refactoring efforts aimed at decoupling and modularizing these components.

C. Summary Table of Metrics Across Projects



Project	WMC Mean	WMC Median	WMC Max	CBO Mean	CBO Median	CBO Max	Bad Smells Mean	Bad Smells Median	Bad Smells Max
AndroidUtilCode	20.53	3.0	534	4.95	2.0	63	10.70	7.0	69
CoreNLP	33.52	7.0	1281	6.93	4.0	222	101.33	6.0	892
NewPipe	24.69	6.0	444	10.35	5.0	91	21.80	7.0	198
Cassandra	14.05	6.0	1069	7.87	5.0	163	13.48	6.0	599
Kafka	19.19	5.0	590	9.81	4.0	450	29.22	3.0	889
Metersphere	18.94	1.0	458	7.01	3.0	141	3.21	2.0	33
Netty	12.20	1.0	747	4.45	2.0	78	13.42	3.0	182
Spring-Boot	6.14	3.0	123	5.08	3.0	83	3.21	2.0	16
Spring-Framework	12.20	2.0	928	5.75	3.0	198	7.98	2.0	205
Tutorials	6.43	5.0	133	3.68	2.0	40	24.87	4.0	175

Fig. 3. Metrics Across Projects

Table provides a statistical summary of the key metrics across all ten projects, including mean, median, and maximum values for WMC, CBO, and bad smell counts. The table highlights the variation in complexity and coupling among the different projects:

- CoreNLP: Exhibits the highest maximum WMC value (1281) and high CBO values, correlating with the maximum number of bad smells (892). These extreme values indicate the need for significant refactoring to enhance testability.
- Apache Kafka: Shows similar trends with high WMC (590) and CBO (450) values, leading to a maximum bad smell count of 889. This suggests a need for reducing class complexity and decoupling components to improve testing processes.
- Spring Framework and Spring Boot: These projects generally maintain lower maximum values for WMC and CBO, and accordingly, they show lower bad smell counts. The adherence to modular design principles in these frameworks seems effective in reducing complexity and enhancing testability.
- AndroidUtilCode and NewPipe: These projects display moderate complexity and coupling, with WMC and CBO values reflecting a structured approach to code organization. However, certain files still present higher WMC and CBO values, indicating that some parts of these codebases may require refactoring to improve testability.

The table supports the findings from the scatter plots, emphasizing the relationship between high WMC and CBO values and the presence of bad smells. It also helps identify which projects are managing complexity and coupling effectively, as seen in Spring Framework and Spring Boot, versus those that require targeted intervention, such as CoreNLP and Apache Kafka.

D. Implications for Software Development

The analysis demonstrates that code complexity (as measured by WMC) and coupling (CBO) are key predictors of bad smells, which directly impact testability. Projects with higher metric values consistently show higher counts of bad smells, indicating that managing these aspects of software design is crucial for maintaining testability. These findings highlight the need for continuous quality assessment practices and iterative refactoring to manage the complexity in software systems. The study reinforces the importance of static analysis tools like PMD and CK metrics analyzers in identifying critical areas for improvement. Integrating these tools into the software development lifecycle allows developers to proactively address structural issues, thereby minimizing the accumulation of bad smells and enhancing the overall quality of the software. In practice, development teams should prioritize refactoring efforts on outlier files with high WMC and CBO values, using modularization and decoupling strategies to reduce complexity and improve testability.

E. Validation and Future Work

While the results offer valuable insights into the relationship between code complexity, coupling, and bad smells, further research could expand the sample size beyond ten projects to validate these findings across even broader software ecosystems. Additionally, exploring other quality attributes, such as maintainability or modifiability, in conjunction with testability could provide a more holistic view of how software design impacts various dimensions of software quality. By focusing on reducing WMC and CBO, development teams can improve not only testability but also other quality aspects, ensuring that software systems are robust, scalable, and maintainable over time. The integration of automated static analysis tools proves beneficial not only for current development practices but also as a proactive measure for future development cycles. Threats to Validity Ensuring the validity of this empirical study involved addressing internal and external factors that could influence the results. For internal validity, the study maintained consistent use of PMD and CK metrics tools across all projects, with data collected from specific code snapshots to ensure uniformity. A custom PMD ruleset was applied to detect testability-related bad smells accurately. However, recognizing the limitations of static analysis tools, manual code inspection was performed for a subset of files to validate and cross-check the results. External validity was considered by selecting a diverse range of ten open-source Java projects, varying in size, complexity, and application domain, to increase the generalizability of the findings. While the focus on Java might limit the applicability to other programming languages, this diversity aimed to represent different software types, with a recommendation for future studies to expand the scope to other languages like Python or C++. Construct and conclusion validity were also addressed by using well-established CK metrics, such as Weighted Methods per Class (WMC) and Coupling Between Objects (CBO), to assess the impact of code smells on testability. Although these metrics provide a reliable basis for evaluating complexity and coupling, they may not capture all aspects affecting testability. Scatter plots and statistical analyses were used to verify the correlations, while extreme outliers were isolated to prevent them from skewing the results. The methods and tools were thoroughly documented to ensure reliability, allowing for the reproducibility of the study in future research, thus strengthening the credibility of the findings.

IV. Conclusion

This study examines the environmental impact of blockchain technology, focusing on the energy consumption of Proof of Work (PoW) and Proof of Stake (PoS) mechanisms. While blockchain has revolutionized transactions and security, PoW, used in networks like Bitcoin, faces sustainability challenges due to its high energy demands. Using a hypothetical dataset from 2015 to 2021, the analysis shows that PoW energy usage increased from 100 TWh to 130 TWh, whereas PoS remained stable between 10 TWh and 12 TWh, highlighting its efficiency. These findings emphasize the need for the industry to transition to energy-efficient models like PoS to reduce carbon emissions and comply with global sustainability goals. PoS offers an efficient alternative by relying on validators based on holdings rather than computational power, minimizing environmental impact while ensuring security. The study recommends transitioning to PoS or hybrid models and incorporating renewable energy for existing PoW networks. Regulatory support through incentives for sustainable technologies and investment in efficient hardware and awareness initiatives are also suggested. Adopting energy-efficient consensus mechanisms is essential for blockchain's long-term growth and compliance with environmental regulations, ensuring sustainable expansion across industries.

References

[1] Apache, "Cassandra," [Online]. Available: https://github.com/apache/cassandra. [Accessed: Oct. 17, 2024].

[2] Blankj, "AndroidUtilCode," [Online]. Available: https://github.com/Blankj/AndroidUtilCode. [Accessed: Oct. 17, 2024].

[3] Stanford NLP Group, "CoreNLP," [Online]. Available: https://github.com/stanfordnlp/CoreNLP. [Accessed: Oct. 17, 2024].

[4] Apache, "Kafka," [Online]. Available: https://github.com/apache/kafka. [Accessed: Oct. 17, 2024].

[5] Metersphere Team, "Metersphere," [Online]. Available: https://github.com/metersphere/metersphere. [Accessed: Oct. 17, 2024].

[6] Netty Project, "Netty," [Online]. Available: https://github.com/netty/netty. [Accessed: Oct. 17, 2024].

[7] Team NewPipe, "NewPipe," [Online]. Available: https://github.com/TeamNewPipe/NewPipe. [Accessed: Oct. 17, 2024].

[8] Spring Projects, "Spring Boot," [Online]. Available: https://github.com/spring-projects/spring-boot. [Accessed: Oct. 17, 2024].

[9] Spring Projects, "Spring Framework," [Online]. Available: https://github.com/spring-projects/spring-framework. [Accessed: Oct. 17, 2024].

[10] Eugen P., "Tutorials," [Online]. Available: https://github.com/eugenp/tutorials. [Accessed: Oct. 17, 2024].

[11] M. Aniche, "CK Java Metrics," [Online]. Available: https://github.com/mauricioaniche/ck. [Accessed: Oct. 17, 2024].

[12] PMD, "PMD: Source Code Analyzer," [Online]. Available: https://pmd.github.io/. [Accessed: Oct. 17, 2024].