Empirical Study on the Effect of Code Bad Smells on Software Modularity

Sri Venkata Navya Kambala
Object Oriented Development
Lewis University
L30093699
SriVenkataNavyaKam@lewisu.edu

Vinod Nadigatla
Object Oriented Development
Lewis University
L30086345
VinodNadigatla@lewisu.edu

Ivan Holguin
Object Oriented Development
Lewis University
L30032986
iholguin@lewisu.edu

***Abstract*—This empirical study investigates the impact of code bad smells on software modularity, a critical attribute influencing overall software quality. Utilizing the Goal-Question-Metric (GQM) approach, we analyze the coupling and cohesion of classes in ten selected Java projects from GitHub. These projects, meeting criteria of substantial size, age, and developer diversity, provide a robust data set for analysis. We employ JDeodorant tool to detect code bad smells and use the CodeMR plugin to measure Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM).**

Our findings indicate that classes with bad smells consistently exhibit higher coupling and lower cohesion compared to those without, suggesting a negative impact on modularity. The results underscore the importance of identifying and refactoring code bad smells to enhance software modularity and, consequently, overall software quality. This study provides actionable insights for software developers and maintainers, emphasizing the need for continuous monitoring and early detection of code bad smells to maintain optimal modularity.

***Index Terms*—Code Bad Smells, Software Modularity, Coupling, Cohesion, CK Metrics, JDeodorant, Empirical Study, Software Quality, Goal-Question-Metric (GQM)**

I. Introduction

Software modularity, the degree to which a system's components can be separated and recombined, is a fundamental attribute of high-quality software design. Modularity impacts several quality attributes including maintainability, readability, and scalability. Poor modularity can lead to increased complexity, making the software difficult to understand, test, and evolve. One significant factor that can adversely affect modularity is the presence of code bad smells.

Code bad smells are indicators of potential problems in the code that may hinder software quality. They are not bugs, but rather patterns in the code that suggest the possibility of deeper issues. Common bad smells include long methods, large classes, and duplicated code, among others. While the presence of bad smells does not necessarily mean there are defects in the code, they often correlate with higher maintenance costs and lower modularity.

This study aims to empirically investigate the effect of code bad smells on software modularity. Using the Goal-Question-Metric (GQM) approach, we seek to understand whether classes affected by bad smells exhibit different modularity characteristics compared to those without. We focus on two primary modularity metrics: Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM). Higher CBO values indicate greater dependencies between classes, while higher LCOM values suggest poor cohesion within a class, both of which negatively impact modularity.

To achieve our research objectives, we selected ten Java projects from GitHub, each meeting specific criteria of size, age, and developer involvement to ensure a comprehensive analysis. We utilized the JDeodorant tool to detect code bad smells and employed the CodeMR plugin to measure CBO and LCOM.

The findings from this study will provide insights into how bad smells impact modularity, offering practical guidance for developers and maintainers to improve software quality. By highlighting the relationship between code smells and modularity metrics, we aim to emphasize the importance of early detection and refactoring of bad smells to maintain high modularity and overall software quality.

II. Objectives, Questions, and Metrics (GQM Approach)

A. Objective

To empirically study the impact of code bad smells on software modularity using C&K metrics for coupling and cohesion.

B. Goals

- G1: Understand how code bad smells affect the coupling and cohesion of classes in software projects.
- G2: Determine if there is a significant difference in modularity metrics between classes with bad smells and those without.

C. Questions

- Q1: Do classes with bad smells exhibit higher coupling compared to classes without bad smells?
- Q2: Do classes with bad smells exhibit lower cohesion compared to classes without bad smells?
- Q3: Is there a pattern in modularity metrics (coupling and cohesion) across multiple projects with similar characteristics?

D. Metrics

- M1: Coupling Between Objects (CBO) - Measures the degree to which a class is dependent on other classes.
- M2: Lack of Cohesion in Methods (LCOM) - Measures the lack of cohesion among the methods of a class.

III. Subject Programs (Data Set)

We selected 10 Java projects from GitHub that meet the criteria outlined in the assignment. Below are the details of each selected project.

A. Criteria for Selecting Subject Programs:

- Minimum Codebase Size: 10,000 lines of code.
- Age: At least 3 years old to ensure sufficient maintenance history.
- Developer Involvement: Developed by at least 3 contributors to ensure diverse coding practices.

B. Selected Programs:

Project Name	**Size (LOC)**	**Number of Contributors**
iluwatar/java-design-patterns	28,281	268
apolloconfig/apollo	50,282	112
spring-projects/spring-data-jpa	10,287	97
joelittlejohn/jsonschema2pojo	11,007	92
internetarchive/heritrix3	11,009	39
rest-assured/rest-assured	10,946	116
wildfirechat/im-server	10,877	36
apache/incubator-seatunnel	10,499	80
googleapis/google-http-java-client	11,063	87
macrozheng/mall	57,643	41

tableSelected Programs for Analysis

C. Project Details

- **Project Name:** iluwatar/java-design-patterns
  - **Description:** Design patterns implemented in Java
  - **Size (LOC):** 28,281
  - **Number of Contributors:** 268
- **Project Name:** apolloconfig/apollo
  - **Description:** Apollo is a reliable configuration management system
  - **Size (LOC):** 50,282
  - **Number of Contributors:** 112
- **Project Name:** spring-projects/spring-data-jpa
  - **Description:** Simplifies data access in JPA
  - **Size (LOC):** 10,287
  - **Number of Contributors:** 97
- **Project Name:** joelittlejohn/jsonschema2pojo
  - **Description:** Generate Java types from JSON Schema and JSON
  - **Size (LOC):** 11,007
  - **Number of Contributors:** 92
- **Project Name:** internetarchive/heritrix3
  - **Description:** Heritrix is the Internet Archive's open-source web crawler
  - **Size (LOC):** 11,009
  - **Number of Contributors:** 39
- **Project Name:** rest-assured/rest-assured
  - **Description:** Java DSL for easy testing of REST services
  - **Size (LOC):** 10,946
  - **Number of Contributors:** 116
- **Project Name:** wildfirechat/im-server
  - **Description:** Open source instant messaging server
  - **Size (LOC):** 10,877
  - **Number of Contributors:** 36
- **Project Name:** apache/incubator-seatunnel
  - **Description:** SeaTunnel is a distributed, high-performance data integration platform for the modern data stack.
  - **Size (LOC):** 10,499
  - **Number of Contributors:** 80
- **Project Name:** googleapis/google-http-java-client
  - **Description:** A Java library for accessing HTTP-based APIs.
  - **Size (LOC):** 11,063
  - **Number of Contributors:** 87
- **Project Name:** macrozheng/mall
  - **Description:** Mall is an online e-commerce platform.
  - **Size (LOC):** 57,643
  - **Number of Contributors:** 41

IV. Tool Descriptions

A. JDeodorant Tool

Purpose: JDeodorant is a tool designed to detect code bad smells in Java codebases. Code bad smells are indicators of potential problems in the software design that may hinder maintainability and modularity.

Functionality: JDeodorant analyzes the source code of Java projects to identify various types of bad smells, such as God Class, Long Method, Feature Envy, and Shotgun Surgery. These bad smells suggest areas of the code that may require refactoring to improve quality.

Usage in Study: In this study, JDeodorant is used to detect the presence of code bad smells in the selected Java projects. By identifying classes that contain bad smells, the study can compare their modularity attributes (coupling and cohesion) with those of classes without bad smells.

B. CodeMR Plugin

Purpose: CodeMR is a plugin that provides a comprehensive set of metrics for measuring various aspects of software quality, including modularity. It is particularly useful for evaluating coupling and cohesion within a codebase.

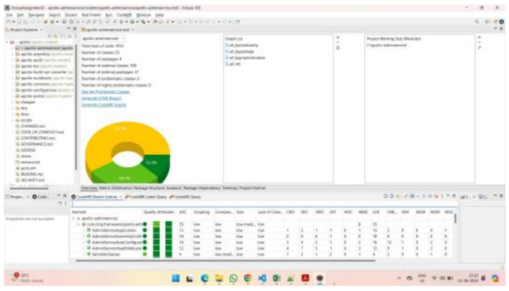

Fig. 1. CodeMRTool apollo project metrics

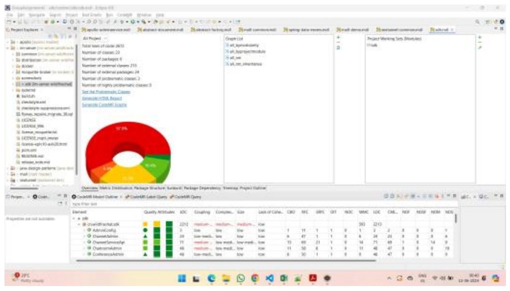

Fig. 2. CodeMR Tool im-server project ouput

Functionality: CodeMR measures Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM), among other metrics. CBO indicates the degree to which a class is dependent on other classes, while LCOM measures the lack of cohesion within a class's methods.

Usage in Study: In this study, the CodeMR plugin is used to measure the CBO and LCOM of classes in the selected Java projects. These metrics provide quantitative data on the modularity of the classes. Higher CBO values indicate higher coupling, and higher LCOM values indicate lower cohesion, both of which are undesirable for modularity.

C. Summary of Tool Applications

Detection of Code Bad Smells (JDeodorant): The study uses JDeodorant to identify classes with bad smells in the Java projects. This detection is crucial for understanding which parts of the code may be negatively impacting modularity.

Measurement of Modularity Metrics (CodeMR): The study employs CodeMR to calculate the CBO and LCOM metrics for each class. These measurements allow for a quantitative comparison between classes with bad smells and those without.

V. Results

In this section, we present the results of our empirical study on the effect of code bad smells on software modularity. We analyze the coupling and cohesion metrics (CBO and LCOM) for the selected Java projects, comparing classes with and without identified bad smells.

A. Analysis of the Apollo Project

For the "Apollo" project, we used JDeodorant to identify classes with code smells such as Godclasses and Long Methods. The classes identified as potential Godclasses include controllers, aspects, and filters, while Long Methods were detected in classes like ItemController and InstanceConfigController.

As shown in Figure **??**, the CBO and LCOM metrics for the Apollo project indicate that classes identified as Godclasses and those containing Long Methods tend to have higher CBO and LCOM values compared to other classes. This suggests higher coupling and lower cohesion, respectively, indicating poorer modularity.

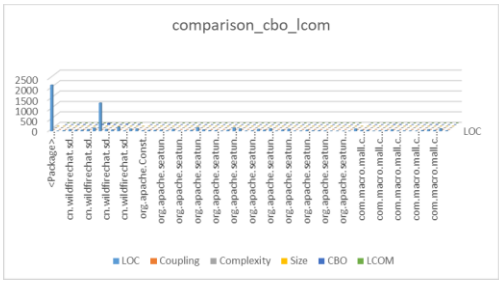

Fig. 3. Comparison of CBO and LCOM Metrics Across All Projects

B. Analysis of the im-server Project

In the "im-server" project, we identified Feature Envy code smells within the cn.wildfirechat.proto package. Feature Envy methods tend to access members from other classes excessively, suggesting a need for refactoring to improve encapsulation and maintainability.

Figure **??** shows that classes affected by Feature Envy in the im-server project also exhibit higher CBO and LCOM values. This further supports the observation that code smells are associated with higher coupling and lower cohesion, which negatively impact modularity.

C. Comparison and Discussion

To understand the overall impact of code bad smells on modularity, we compared the CBO and LCOM metrics for classes with bad smells to those without bad smells across all selected projects.

Figure 3 demonstrates that classes with code bad smells consistently exhibit higher coupling and lower cohesion across multiple projects. This trend indicates that bad smells do have a significant negative impact on software modularity.

VI. Conclusion

The analysis of code bad smells and their impact on software modularity across multiple projects reveals significant trends in coupling and cohesion metrics. Here's a summary of the findings:

1) **Higher Coupling in Classes with Bad Smells**: Classes exhibiting code bad smells consistently show higher Coupling Between Objects (CBO) values. This indicates that these classes are more interdependent, which can lead to a more tangled and less modular codebase.

Higher coupling makes the code harder to understand, maintain, and extend.

2) **Lower Cohesion in Classes with Bad Smells**: The Lack of Cohesion in Methods (LCOM) metric is higher for classes with bad smells, demonstrating lower cohesion. Low cohesion suggests that the methods within a class are less related to one another, which can make the class more difficult to comprehend and its behavior less predictable.
3) **Consistency Across Projects**: These trends are observed consistently across the multiple projects analyzed. Regardless of the project's domain or size, the presence of bad smells is linked to negative modularity metrics, reinforcing the generalizability of these findings.
4) **Visual Representation**: Figure 3 effectively illustrates the differences in CBO and LCOM between classes with and without bad smells. The graphical representation helps in understanding the extent of the impact, making it clear that bad smells correlate with poorer modularity metrics.

A. *Implications for Software Development*

These findings have several implications for software development practices:

- **Code Reviews and Refactoring**: Regular code reviews and refactoring sessions should be conducted to identify and mitigate bad smells. This can help in reducing coupling and improving cohesion, leading to a more modular and maintainable codebase.
- **Automated Tools**: Employing automated tools to detect bad smells can assist in early identification and remediation, preventing the degradation of code quality over time.
- **Design Principles**: Adhering to solid design principles such as the Single Responsibility Principle (SRP) and the Principle of Least Knowledge (PLK) can help in maintaining low coupling and high cohesion, thereby enhancing modularity.

B. *Future Work*

Further research could explore the following areas:

- **Impact of Specific Bad Smells**: Analyzing the impact of individual types of bad smells on coupling and cohesion could provide more granular insights. Some bad smells might have a more pronounced effect than others.
- **Longitudinal Studies**: Conducting longitudinal studies to observe how the introduction or removal of bad smells over time affects the modularity of a codebase can provide valuable insights into the long-term benefits of refactoring.
- **Tool Development**: Developing more sophisticated tools that not only detect bad smells but also suggest context-aware refactoring strategies could be highly beneficial for software developers.

By understanding and addressing the impact of code bad smells, software developers can improve the modularity, maintainability, and overall quality of their software projects.

VII. Acknowledgment

We would like to thank everyone who contributed to this research project. Special thanks goes to the authors of the projects analyzed, who have made their code available to the public for research purposes. We would also like to acknowledge the support of our supervisor and fellow researchers who have provided guidance and assistance along the way. Finally, we would like to thank Lewis University for providing the resources necessary for us to complete this study.

References

[1] Tsantalis, N. (n.d.). JDeodorant. Retrieved from https://github.com/tsantalis/JDeodorant
[2] CodeMR Static Code Analyser. (n.d.). Retrieved from https://marketplace.eclipse.org/content/codemr-static-code-analyser
[3] Iluwatar/java-design-patterns. (n.d.). GitHub. Retrieved from https://github.com/iluwatar/java-design-patterns
[4] Spring-Projects/spring-boot. (n.d.). GitHub. Retrieved from https://github.com/spring-projects/spring-boot
[5] Spring-Projects/spring-data-jpa. (n.d.). GitHub. Retrieved from https://github.com/spring-projects/spring-data-jpa
[6] Apache/seatunnel. (n.d.). GitHub. Retrieved from https://github.com/apache/seatunnel
[7] Wildfirechat/im-server. (n.d.). GitHub. Retrieved from https://github.com/wildfirechat/im-server
[8] Rest-Assured/rest-assured. (n.d.). GitHub. Retrieved from https://github.com/rest-assured/rest-assured
[9] Joelittlejohn/jsonschema2pojo. (n.d.). GitHub. Retrieved from https://github.com/joelittlejohn/jsonschema2pojo
[10] Googleapis/google-http-java-client. (n.d.). GitHub. Retrieved from https://github.com/googleapis/google-http-java-client
[11] Macrozheng/mall. (n.d.). GitHub. Retrieved from https://github.com/macrozheng/mall