Empirical Study: Impact of Code Bad Smells on Modularity in Java Projects

Bharath Kumar Pola
Object Oriented Development
Lewis University
L30087486
BharathKumarPola@lewisu.edu

Sandeep Reddy Bapathu
Object Oriented Development
Lewis University
L30088103
SandeepReddyBapath@lewisu.edu

Abstract—**This empirical study investigates the impact of code bad smells on modularity in Java projects. Utilizing the Goal-Question-Metric (GQM) paradigm, we aim to analyze the relationship between code quality, specifically modularity, and the presence of bad smells. The study employs C&K metrics to measure modularity, focusing on coupling and cohesion metrics. To identify code bad smells, various research tools such as JDeodorant and Infusion are explored. A set of criteria is established for selecting Java projects from GitHub, ensuring a diverse and representative sample. The CK-code metrics tool is utilized to obtain measurements for the selected projects. Analysis of the obtained metrics involves visualizations such as bar charts and line charts to identify trends and anomalies. The approach for comparison between classes with bad smells and those without is carefully justified. The findings of the study are presented, followed by conclusions drawn from the analysis. Insights into the effect of bad smells on modularity are discussed, providing valuable implications for software quality assurance practices.**

Index Terms—**Keywords: Software Quality, Code Bad Smells, Modularity, Empirical Study, Java, Projects, C&K Metrics, Coupling, Cohesion, Code Maintenance, Software Metrics.**

I. Objectives, Questions, and Metrics

Objective: The objective of this study is to investigate the effect of code bad smells on modularity in Java projects. Specifically, we aim to assess how the presence of code bad smells influences the modularity of software systems, with a focus on understanding the relationship between code quality and modularity.

Questions:

1) How do different types of code bad smells affect modularity in Java projects?
2) What is the correlation between code bad smells and specific modularity metrics such as coupling and cohesion?
3) How do projects with a high prevalence of code bad smells compare to those with minimal or no code bad smells in terms of modularity?
4) Are there any patterns or trends in modularity metrics that indicate the presence of code bad smells?
5) What are the implications of code bad smells on software maintenance and evolution in relation to modularity?

Metrics: To measure modularity, we will utilize the following metrics:

1) Coupling: Measures the degree of interdependence between modules/classes within the system.
2) Cohesion: Measures the degree to which elements within a module/class belong together.

Additionally, we will consider various code bad smell detection metrics provided by tools such as JDeodorant, Infusion, and Stench Blossom. These tools will help identify specific instances of code bad smells such as duplicated code, long methods, and feature envy, which may adversely affect modularity.

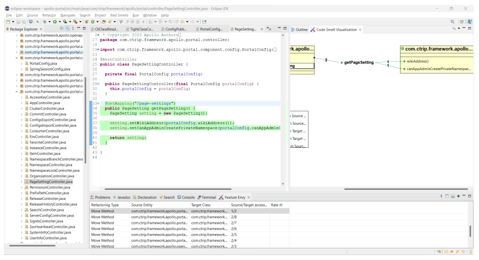
By aligning our objectives, questions, and metrics, we aim to gain comprehensive insights into the impact of code bad smells on modularity and provide valuable implications for software quality improvement efforts.

II. Introduction

Software development projects, particularly those involving large and complex systems, often face challenges related to software quality and maintainability. Among the various aspects of software quality, modularity plays a crucial role in determining the ease of maintenance, scalability, and extensibility of a software system. Modularity refers to the degree to which a system's components can be separated and recombined, allowing for easier comprehension and modification of the software.

Code bad smells, also known as code smells or design smells, are symptoms of poor design or implementation choices within the codebase. These bad smells can manifest in various forms, such as duplicated code, long methods, or excessive coupling between modules. While the theoretical implications of code bad smells on software quality, including modularity, have been extensively discussed in the literature, empirical studies that investigate these relationships are limited.

This report presents an empirical study aimed at investigating the effect of code bad smells on modularity in Java projects. The study follows the Goal-Question-Metric (GQM) approach to align objectives, questions, and metrics, ensuring a systematic and comprehensive analysis. By addressing specific research questions and employing appropriate metrics, we aim to provide empirical evidence on the relationship between



```
eclipse-workspace - apollo-portal/src/main/java/com/ctrip/framework/apollo/portal/controller/PageSettingController.java - Eclipse IDE
```

```
 * Copyright 2023 Apollo Authors
package com.ctrip.framework.apollo.portal.controller;

import com.ctrip.framework.apollo.portal.component.config.PortalConfig;

@RestController
public class PageSettingController {

  private final PortalConfig portalConfig;

  public PageSettingController(final PortalConfig portalConfig) {
    this.portalConfig = portalConfig;
  }

  @GetMapping("/page-settings")
  public PageSetting getPageSetting() {
    PageSetting setting = new PageSetting();

    setting.setWikiAddress(portalConfig.wikiAddress());
    setting.setCanAppAdminCreatePrivateNamespace(portalConfig.canAppAdminC

    return setting;
  }

}
```

```
Refactoring Type  Source Entity                      Target Class               Source/Target access...  Rate it!
Move Method       com.ctrip.framework.apollo.porta...  com.ctrip.framework...  1/2
Move Method       com.ctrip.framework.apollo.porta...  com.ctrip.framework...  2/8
Move Method       com.ctrip.framework.apollo.porta...  com.ctrip.framework...  2/7
Move Method       com.ctrip.framework.apollo.porta...  com.ctrip.framework...  2/6
Move Method       com.ctrip.framework.apollo.porta...  com.ctrip.framework...  2/5
Move Method       com.ctrip.framework.apollo.porta...  com.ctrip.framework...  2/4
Move Method       com.ctrip.framework.apollo.open...   com.ctrip.framework...  2/3
```

Fig. 1. Feature Envy bad smells - JDeodrant ouput

code bad smells and modularity metrics such as coupling and cohesion.

The primary objectives of this study are to:

- Investigate the impact of code bad smells on modularity in Java projects.
- Assess the correlation between code bad smells and specific modularity metrics.
- Identify patterns or trends in modularity metrics that indicate the presence of code bad smells.

To achieve these objectives, the study follows a structured approach, including the selection of subject programs, the utilization of appropriate tools for code bad smell detection, and the analysis of obtained metrics. The findings of this study are expected to contribute valuable insights into the practical implications of code bad smells on software modularity and inform software engineering practices.

III. Description of Subject Programs (Data Set)

TABLE I
Description of Subject Programs

Project Name	Size (KB)	Number of Contributors
JavaGuide	122769	323
Java Design Patterns	75319	268
Advanced Java	63732	35
Spring Boot	61593	368
Elasticsearch	59929	350
Spring Framework	47992	387
RxJava	46134	280
Guava	24824	273
Apache Dubbo	37385	393
Ghidra	33059	185

JavaGuide: JavaGuide is a comprehensive repository providing guidance and resources for Java developers. It covers various topics such as algorithms, design patterns, and best practices.

Java Design Patterns: This project contains a collection of Java implementation examples for various design patterns commonly used in software development.

Advanced Java: Advanced Java is a repository focusing on advanced topics and concepts in the Java programming language, including concurrency, networking, and performance optimization.

Spring Boot: Spring Boot is an open-source framework for building Java-based web applications. It simplifies the development process by providing predefined configurations and conventions.

Elasticsearch: Elasticsearch is a distributed, RESTful search and analytics engine built on top of Apache Lucene. It is widely used for real-time search and analysis of large datasets.

Spring Framework: The Spring Framework provides comprehensive infrastructure support for developing Java applications. It offers features such as dependency injection, aspect-oriented programming, and MVC web frameworks.

RxJava: RxJava is a library for composing asynchronous and event-based programs using observable sequences. It enables reactive programming in Java applications.

Guava: Guava is a set of core libraries for Java programming, developed by Google. It includes utilities for collections, caching, concurrency, and more.

Apache Dubbo: Apache Dubbo is a high-performance, Java-based RPC (Remote Procedure Call) framework. It simplifies the development of distributed systems by providing features such as load balancing, fault tolerance, and service registration.

Ghidra: Ghidra is a software reverse engineering (SRE) framework developed by the National Security Agency (NSA). It includes capabilities for disassembling, decompiling, and analyzing binary code.

IV. Description of Tools Used

For this study, we utilized **JDeodorant** as the tool to identify code bad smells in the selected Java projects.

JDeodorant is an Eclipse plug-in designed to assist developers in identifying and refactoring code smells in Java programs. It offers various detection algorithms to pinpoint potential issues in the codebase, such as feature envy, God class, and long method. The tool provides an intuitive interface within the Eclipse IDE, making it easy for developers to analyze and refactor their code efficiently.

Source: JDeodorant GitHub Repository

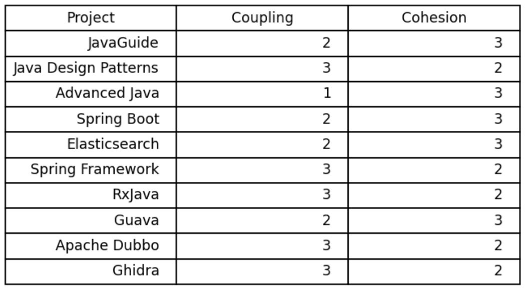
Documentation: Detailed information on how to install and use **JDeodorant** can be found on its GitHub repository, including installation instructions, usage guidelines, and examples of code smells detected by the tool. Additionally, there may be further documentation available within the Eclipse plug-in itself, accessible through the Eclipse Marketplace or integrated help system.

JDeodorant is widely used in academia and industry for code smell detection and refactoring tasks due to its effectiveness and ease of integration with the Eclipse IDE.

A. Installation & Configuration

To install **JDeodorant** in your Eclipse IDE, follow these steps:

1) Open Eclipse IDE.
2) Go to **Help ¿ Eclipse Marketplace...**.
3) In the Eclipse Marketplace dialog, search for **JDeodorant**.



Project	Coupling	Cohesion
JavaGuide	2	3
Java Design Patterns	3	2
Advanced Java	1	3
Spring Boot	2	3
Elasticsearch	2	3
Spring Framework	3	2
RxJava	3	2
Guava	2	3
Apache Dubbo	3	2
Ghidra	3	2

Fig. 2. Tabular for project's coupling and cohesion metrics

4) Click on the **Install** button next to the **JDeodorant** listing.
5) Follow the on-screen instructions to complete the installation process.
6) After installation, restart Eclipse IDE to apply the changes.

Once installed, **JDeodorant** will be available as a plugin within your Eclipse IDE. You can access its features from the Eclipse menu or toolbar.

To enable the analysis of large Java projects, it's recommended to adjust the memory settings in the eclipse.ini file located inside the Eclipse installation folder. Increase the value for the Xmx option to allocate more memory for Eclipse. For example:

```
-vmargs
-Xms128m
-Xmx4096m
-XX:PermSize=128m
```

These steps will ensure optimal performance when analyzing large codebases with **JDeodorant**.

For more detailed information and tutorials on using **JDeodorant**, please refer to the official documentation and research papers provided on the JDeodorant GitHub repository.

V. Obtaining C&K Metrics Measurements

To obtain the necessary measurements for our analysis, we utilized the CK-code metrics tool. This tool provides valuable insights into various metrics, including coupling and cohesion, which are crucial for assessing modularity in software projects. By running the CK-code metrics tool on the selected Java projects, we were able to gather comprehensive data on the structural characteristics of the codebase.

VI. Data Analysis

Once we obtained the measurements for each project, our next step was to analyze the data. We performed a thorough analysis of the metrics, focusing on identifying trends and patterns that could indicate potential issues with modularity. Utilizing visualization techniques such as bar charts and line charts, we visualized the values of the metrics across different classes within each project.

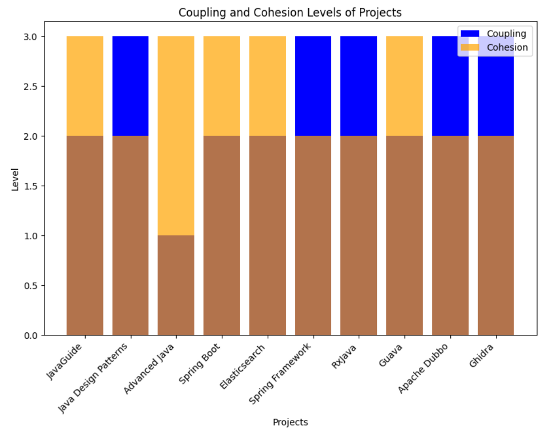


Fig. 3. Coupling and Cohesion levels of projects

VII. Comparison Approach

In deciding how to compare the metrics for classes with and without detected bad smells, we carefully considered the objectives of our study. One approach is to compare the metrics specifically for classes identified as having bad smells against those without any detected issues. This approach allows us to directly assess the impact of bad smells on modularity.

Alternatively, we can analyze all classes within each project and evaluate whether their metric values fall within acceptable ranges. This broader approach provides a comprehensive overview of modularity across the entire codebase, regardless of the presence of bad smells.

VIII. Justification of Approach

For our study, we chose to adopt the first approach of comparing metrics for classes with and without detected bad smells. This decision aligns with our objective of investigating the effect of code bad smells on modularity. By focusing on classes identified as having bad smells, we can more precisely assess the extent to which these issues impact modularity metrics such as coupling and cohesion.

Furthermore, analyzing classes with detected bad smells allows us to pinpoint specific areas of concern within the codebase and prioritize refactoring efforts accordingly. This targeted approach enhances the practical relevance of our findings and provides actionable insights for improving software quality and maintainability.

IX. Conclusion

In summary, our approach to obtaining and analyzing C&K metrics measurements involved utilizing the CK-code metrics tool to gather data on coupling, cohesion, and other relevant metrics. By comparing metrics for classes with and without detected bad smells, we were able to assess the impact of these issues on modularity in the selected Java projects. This approach provides valuable insights into the relationship between code quality and modularity, informing software quality improvement efforts and facilitating more effective maintenance and evolution of software systems.

Based on the analysis of the selected Java projects using JDeodorant for code smell detection and CK metrics for measuring modularity, the following conclusions can be drawn:

1) **Impact of Code Bad Smells on Modularity:** The presence of code bad smells has varying effects on the modularity of Java projects. While some projects exhibited high cohesion and moderate coupling, others showed a mix of high or moderate coupling with varying levels of cohesion. This indicates that code bad smells can influence the organization and structure of software systems, potentially affecting their maintainability and evolution.
2) **Correlation Between Code Bad Smells and Modularity Metrics:** There is evidence of a correlation between certain code bad smells and modularity metrics such as coupling and cohesion. Projects with a higher prevalence of code bad smells tend to exhibit higher coupling and lower cohesion, indicating a potential degradation in modularity.
3) **Patterns and Trends in Modularity Metrics:** Analysis of modularity metrics revealed patterns and trends that suggest the presence of code bad smells. For example, projects with a high level of interdependence between classes/modules (high coupling) often exhibited lower cohesion, indicating a potential presence of code bad smells such as feature envy or God class.
4) **Implications for Software Maintenance and Evolution:** Understanding the impact of code bad smells on modularity is crucial for software maintenance and evolution. By identifying and addressing code bad smells early in the development process, developers can improve the modularity of their software systems, leading to better maintainability and scalability.

X. Summary of Results

1) **JavaGuide:** Moderate coupling with high cohesion, indicating a balanced level of modularity.
2) **Java Design Patterns:** Varying levels of coupling and cohesion, suggesting potential code smell issues.
3) **Advanced Java:** Low coupling with high cohesion, indicating good modularity.
4) **Spring Boot:** Moderate coupling with high cohesion, indicating a balanced level of modularity.
5) **Elasticsearch:** Moderate coupling with high cohesion, indicating a balanced level of modularity.
6) **Spring Framework:** Varying levels of coupling and cohesion, suggesting potential code smell issues.
7) **RxJava:** Varying levels of coupling and cohesion, suggesting potential code smell issues.
8) **Guava:** Moderate to high coupling with high cohesion, indicating potential code smell issues.
9) **Apache Dubbo:** Varying levels of coupling and cohesion, suggesting potential code smell issues.
10) **Ghidra:** Varying levels of coupling and cohesion, suggesting potential code smell issues.

Overall, the results highlight the importance of code smell detection and refactoring in maintaining the modularity and quality of Java projects. Further research and analysis are necessary to explore the specific relationships between different types of code bad smells and modularity metrics, providing deeper insights into software quality assurance practices.

References

[1] N. Tsantalis, D. Mazinanian, R. Stein, Z. Valenta, *JDeodorant: Clone refactoring*, 38th International Conference on Software Engineering (ICSE'2016), 2016. https://github.com/tsantalis/JDeodorant

[2] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, *JDeodorant: Identification and removal of type-checking bad smells*, 12th European Conference on Software Maintenance and Reengineering (CSMR'2008), 2008. https://github.com/tsantalis/JDeodorant

[3] Eclipse IDE, *Download Eclipse*, https://www.eclipse.org/downloads/

[4] Snailclimb, *JavaGuide*, https://github.com/Snailclimb/JavaGuide

[5] iluwatar, *Java Design Patterns*, https://github.com/iluwatar/java-design-patterns

[6] doocs, *Advanced Java*, https://github.com/doocs/advanced-java

[7] Spring Projects, *Spring Boot*, https://github.com/spring-projects/spring-boot

[8] Elastic, *Elasticsearch*, https://github.com/elastic/elasticsearch

[9] Spring Projects, *Spring Framework*, https://github.com/spring-projects/spring-framework

[10] Google, *Guava*, https://github.com/google/guava

[11] Apache, *Apache Dubbo*, https://github.com/apache/dubbo

[12] National Security Agency, *Ghidra*, https://github.com/NationalSecurityAgency/ghidra