An Empirical Study on Effect of Code Smells on Software Modularity

Presented By

Bhanuprakash Bandla – L30087041

Akhila Parvathaneni – L30079455

Harika Mareedu – L30091301

Summer 2024

ABSTRACT

This empirical research investigates how software modularity is affected by code smells. Long parameter lists, God classes, and feature envy are examples of code smells that indicate underlying problems in the code that, although not immediately fatal, may seriously harm maintainability and modularity. This research illustrates how these smells affect software's structural integrity by examining important metrics for 10 open-source projects, including Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM). The results show that lower modularity and greater complexity are correlated with higher CBO and LCOM values, which are often linked to code smells. Enhancing the modularity and maintainability of software systems requires systematically refactoring certain code smells. This study emphasizes how crucial it is to continuously evaluate code quality in order to preserve reliable and adaptable software structures.

Keywords: Software Modularity, Code Smells, Maintainability, Refactoring,Code Quality,Metrics,Coupling Between Objects (CBO),Lack of Cohesion in Methods (LCOM),Open-Source Projects

1. Introduction

Writing modular and maintainable code is essential in today's demanding software development environment. Software may be divided into more manageable, smaller components that can be independently created, tested, and maintained thanks to its modular design. By improving the scalability and flexibility of software systems, this method lowers the chance of mistakes being introduced during updates and helps teams adapt more quickly to changing needs. Nevertheless, it may be difficult to achieve and maintain modularity, particularly as projects become bigger and more complicated. Code smells, or patterns in the code that point to more serious problems, are one of the biggest risks to software modularity.

Code smells may take many different forms, each of which denotes a particular kind of issue that might lower the software's quality. When a class depends too much on the functionality of another class, it's called feature envy and shows that there isn't a good division of duties within the system. Because of this reliance, modifications to one class may have unanticipated effects on other classes, making the code more difficult to maintain and comprehend.

God Classes, which are typified by classes that take on excessive duties, are another prevalent kind of code smell. These courses often develop into the system's focal areas of complexity, which makes them challenging to administer and prone to mistakes. Their widespread effect typically necessitates modifications to be transmitted throughout the whole codebase, which increases maintenance work and decreases system flexibility, which might impede the system's modularity.

Long methods are difficult to comprehend and maintain because they execute too many actions or have too many lines of code. They often indicate that a technique is going too far in attempting to fulfill its sole job. Long methods have the potential to hide the code's intent, making it more difficult for developers to rapidly understand its functioning and raising the risk of errors.

The usage and testing of techniques are complicated by excessive parameter lists. It may be difficult to comprehend what each parameter means, how they interact, and how changes to one might impact the others when a method calls for a large number of inputs. This intricacy increases the likelihood of mistakes and makes it more difficult to expand or alter the code.

The codebase becomes less cohesive and more coupled as a result of these code smells. Cohesion is a measure of how tightly connected and concentrated a single module's tasks are, while coupling is the degree of interaction across software modules. Because changes to one component of the system might have far-reaching and unanticipated repercussions, software with high coupling and poor cohesiveness is more difficult to maintain, test, and expand. Therefore, maintaining the modularity and maintainability of software systems depends on recognizing and eliminating code smells.

2.Research Goals, Questions, and Metrics: The GQM Approach

2.1 Research Objective

Through an analysis of many software projects, this research seeks to determine if software modularity and code smells are related. The major objective is to comprehend how different code smells impact these systems' modular design and general maintainability. We try to identify the trends and effects of code smells at the class level as well as project level by looking at certain metrics.

2.2 Research Questions

RQ1: What are the effects of different code smells on a software system's modularity?

The question directs the investigation to pinpoint the precise effects of code smells on program modularity. It seeks to ascertain how smells such as God classes and feature envy affect the cohesiveness and coupling of the program components.

RQ2: How do the presence and severity of code smells correlate with the maintainability of software over time?

This question aims to investigate if the ease of maintaining and evolving software may be correlated with the frequency and severity of code smells in a codebase. It seeks to determine if more serious or pervasive code smells substantially worsen measures related to maintainability, such defect rates and update effort.

2.3 Research Metrics

The following measures are used by the study in order to answer these research questions:

Coupling Between Objects (CBO): Calculates the level of interclass interdependence. Tighter connection, indicated by higher CBO values, might impede modularity and make system modifications more challenging. Code smells, such as feature envy and God classes, where dependencies and responsibilities are unevenly dispersed, are often present in classes with high CBO.

Lack of Cohesion in Methods (LCOM): Assesses the degree of similarity between the methods of a class. Elevated LCOM values indicate a decrease in cohesiveness, suggesting that the class could be handling too many unrelated tasks, hence decreasing its modularity. Classes with high LCOM values are often linked to code smells that make understanding and maintenance more difficult, including lengthy methods and God classes.

3. Detailed Overview of Data Set

This section gives a detailed description of the software systems that were selected based on certain criteria in order to conduct an empirical investigation of code smells and their effects on software modularity.

3.1 Project Selection Criteria

The projects were chosen using the following criteria in order to guarantee a thorough and perceptive analysis:

- **Minimum Project Size**: We selected projects with a minimum of 5,000 lines of code (LoC). This guarantees that the projects are complicated enough to display a range of code smells, enabling a thorough examination of their impacts on software modularity.

- **Project Age Requirement**: Only six-year-old or older projects were taken into consideration. This criterion gives insights into the long-term effects of code smells on software maintainability by ensuring that the projects have experienced substantial development and maintenance.
- **Participation of Developers**: At least three developers were involved in the projects that were chosen. By affecting the existence and severity of code smells, this criterion guarantees variety in coding styles and possible design conflicts, which may damage program modularity.

We chose ten Java projects from GitHub that meet these requirements, cover a range of applications, and ought to provide substantial insights into the connection between modularity and code smells.

3.2 Rationale Behind Project Selection Criteria

The significance of Project Age: Six-year-old projects have probably gone through many iterations of development and upkeep. This guarantees that they have sufficiently changed over time to exhibit a range of code smells, qualifying them for a thorough examination of software modularity.

Importance of Project Size: It is anticipated that projects containing at least 5,000 lines of code will be complicated enough to identify different types of code smells. It is possible to thoroughly investigate how these smells impact the modularity of complex systems by analysing such initiatives.

Role of Developer Involvement: Projects with three or more developers involved are probably going to have a wider range of design and coding methods. Because of this variety, many kinds of code smells may arise, underscoring the need of resolving these smells in order to preserve software modularity.

3.3 Summary of Selected Projects

Based on the above criteria, the following 10 projects were selected. These projects offer a range of systems and applications for analysis:

Table 1: Summary of Selected Projects

Program Name	Developers	Program Age (Years)	Size (LOC)	Description
Apache Cassandra	233	14	2,540,207	A highly scalable and available NoSQL database designed to handle large amounts of data across many commodity servers.
Apache Hadoop	606	14	3,166,684	An open-source software framework used for distributed storage and processing of large datasets using the MapReduce programming model.
Apache Tomcat	338	12	1,049,301	An open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language, and WebSocket technologies
Apache Commons Lang	94	6	320,151	Provides a host of utility functions for the Java API, including string manipulation, number and date formatting, and system properties
Dubbo	145	11	522,948	A high-performance, Java-based open-source RPC framework providing easy and efficient service governance.
Google Guava	758	14	1,654,211	A set of core libraries for Java, including new collection types, graph

				libraries, functional programming utilities, and more.
Java Design Patterns	289	10	3,670,770	A collection of design patterns implemented in Java, providing examples and explanations for common architectural patterns.
Junit 5	165	9	521,290	The 5th major version of the programmer-friendly testing framework for Java.
RxJava	269	11	3,830,129	A library for composing asynchronous and event-based programs using observable sequences for the Java Virtual Machine (JVM).
Spring Framework	1356	12	5,425,678	A comprehensive framework for building enterprise-grade applications using Java. It supports various components such as web applications, security, data access, messaging, and more.

3.4 Detailed Analysis of Selected Projects and its Relevance to study

1. **Apache Cassandra:** With several commodity servers, Apache Cassandra is a highly available and scalable NoSQL database that can manage enormous volumes of data without having a single point of failure. Large-scale data storage systems choose it

because it offers fault tolerance and high availability.

Relevance to Research: Cassandra's large-scale architecture and complexity make it perfect for examining the ways in which tight coupling and code smells like God classes impact the maintainability and modularity of high-performance distributed systems.

2. **Apache Hadoop:** Using the MapReduce programming approach, Apache Hadoop is an open-source platform that makes it easier to store and handle massive datasets in a distributed manner. It is often utilized in computer clusters for massive data processing.

   Relevance to Research: Hadoop is a crucial topic for examining the effects of code smells on modularity and performance in large-scale, data-intensive systems because to its widespread usage in big data contexts and its intricate design.

3. **Apache Tomcat:** An open-source implementation of Java Servlet, JSP, WebSocket, and Java Expression Language is Apache Tomcat. It functions as a popular web server and Java application servlet container.

   Relevance to Research: Tomcat is a crucial project for comprehending how code smells affect modularity and maintainability in web-based systems, where performance and flexibility are essential. This is because of its significance in web application infrastructure.

4. **Apache Commons Lang:** Description: A wide range of useful methods, such as date formatting, system properties, and string manipulation, are available with Apache Commons Lang to improve Java's standard libraries.

   Relevance to Research: By concentrating on utility functions, this study sheds light on the ways that lengthy methods and feature envy, two common code smells, compromise the modularity and cohesiveness of utility classes—two essential components of Java development.

5. **Dubbo:** It is an open-source, high-performance RPC framework built on Java that makes managing distributed services easier. It is widely used to provide effective service governance in microservices architectures.

   Relevance to Research: Dubbo's emphasis on service-oriented architecture and distributed systems makes it pertinent to investigations into how code smells affect the scalability and modularity of microservices frameworks.

6. **Google Guava:** This popular collection of core libraries for Java expands the possibilities of Java by offering additional graph libraries, functional programming tools, and collection types.

   Relevance to Research: Guava's large range of utility features and widespread use provide a rich environment for investigating the effects of code smells on modularity in extensively used utility libraries that improve Java development.

7. **Java Design Patterns**: The Java Design Patterns repository provides examples and descriptions of popular architectural patterns used in software development. It is a collection of design patterns implemented in Java.

   Relevance to Research: Since this project focuses a lot on design patterns, it's a good opportunity to investigate how following or disregarding these patterns influences the frequency of code smells and software modularity.

   8. JUnit 5 : The most recent iteration of the well-known Java unit testing framework, JUnit 5 makes it easier to create and run repeatable tests in order to promote test-driven development (TDD).

Relevance to Research: Examining how code smells impact the modularity and maintainability of testing frameworks that are fundamental to the software development process requires the use of JUnit 5, a crucial tool for guaranteeing code quality.

9. RxJava: Often used in applications requiring reactive programming models, RxJava is a framework for building asynchronous and event-based programs utilizing observable sequences.

Relevance to Research: RxJava is an important project for examining how code smells affect modularity and complexity in event-driven and asynchronous systems because of its emphasis on reactive programming and high concurrency.

10. Spring Framework: Spring Framework is an all-inclusive framework designed to help Java programmers create enterprise-level applications. It provides a number of different features, including messaging, data access, web applications, and security.

Relevance to Research: The Spring Framework, one of the largest frameworks in the Java ecosystem, offers a rich setting for examining the ways in which code smells impact the modularity and maintainability of big, complex corporate programs.

4. Research Methodology

4.1 Data Collection and Tools

Data collection on software modularity and code smells from the chosen projects was our main goal for this research. JDeodorant and CK Metrics were the two main tools used in the data gathering procedure. These tools provide thorough insights on the code's structural quality and the existence of different code smells.

CK Metrics: Chidamber and Kemerer object-oriented metrics are calculated using this tool. It offered crucial measurements like:

Coupling Between Objects (CBO): Indicates how dependent different classes are on one another. Tight coupling, indicated by high CBO values, may make software stiffer and challenging to maintain.

Lack of Cohesion in Methods (LCOM): Assesses how well-coordinated a class's methods are. Low LCOM values are indicative of poorly connected procedures, which often lead to a lower level of modularity.

Other Metrics: Further metrics from CK Metrics, such Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), and Number of Children (NOC), are also available. These metrics give further context for understanding the software's complexity and architecture.

JDeodorant: This Eclipse plugin is dedicated to identifying different kinds of code smells. It was used to detect and examine code smells like:

Feature Envy: Suggests an inadequate division of duties when a class depends excessively on methods from another class.

God Classes: Classes that are too complicated and prone to mistakes because they execute too many functions or have too much code.

Long methods: These are difficult to comprehend and maintain because they execute an excessive number of actions.

Other Smells: JDeodorant offers a thorough examination of code quality problems by detecting other smells such as Data Clumps, Large Class, and Duplicate Code.

4.2 Framework for Analysis

In order to systematically examine how code smells affect software modularity, we used the Goal-Question-Metric (GQM) methodology. This method gave our inquiry the following structure:

Goal: To assess the effects of various code smells on software systems' maintainability and modularity.

Question: What effects can code smells have on software systems' modularity? What impact do they have on these systems' complexity and maintainability?

Metric: To measure the impacts on modularity and complexity, we utilized the measures from CK measures (CBO and LCOM) together with the detected code smells from JDeodorant.

4.3 Tools Used

The following resources were essential to our analysis:

CK Metrics: computed critical metrics for object-oriented programming, offering a quantitative framework for assessing the complexity and modularity of software.

JDeodorant: Found and examined a number of different code smells in the program, pointing out possible problems and places for improvement.

Data Aggregation Tools: To provide a comprehensive picture of each project's modularity and the frequency of code smells, data from CK Metrics and JDeodorant was combined and examined using unique scripts and data analysis tools.

5. Data Analysis

5.1 Overview of Collected Metrics

The analysis involved calculating and comparing metrics related to software modularity and code smells for each project. Below is a summary of the metrics:

Project	**Avg CBO**	**Avg LCOM**	**Classes With Code Smell**	**Classes Without Code Smell**
apache-cassandra	6.630418	56.188151	6364	4202
apache-hadoop	6.836977	54.394456	8611	6577
apache-tomcat	4.485937	151.157508	2424	2198
Apahce-commons-lang	3.120930	273.753488	518	557
dubbo	5.177738	41.552289	2557	1899
google-guava	3.872595	47.358163	2488	3956
java-design-patterns	3.374439	1.904709	651	1133
Junit-5	4.365096	37.137295	1399	1529
rxjava	2.524062	47.416603	2485	9380
spring-framework	5.249460	30.516154	8664	8453

5.2 Insights from CBO and LCOM Metrics

- **Coupling Between Objects (CBO):** Greater coupling between classes is shown by high CBO values, indicating a high degree of interdependence between the classes. High CBO values indicate strong interdependencies in projects like Apache Hadoop and Apache Cassandra, which may impede modularity and complicate maintenance**.**
- **Lack of cohesiveness in Methods (LCOM):** Classes that do an excessive number of unrelated tasks may be handling a lack of cohesiveness, as indicated by high LCOM scores. Low modularity and poor method coherence are indicated by high LCOM ratings in projects like Apache Commons Lang and Apache Tomcat.

5.3 Code Smells' Effect on Modularity

The modularity of the projects is greatly impacted when code smells are present. Code-smelly classes often have higher CBO and LCOM values than smell-free classes, which indicates less modularity and more complexity. This implies that program modularity and maintainability may be enhanced by reworking to solve code smells.

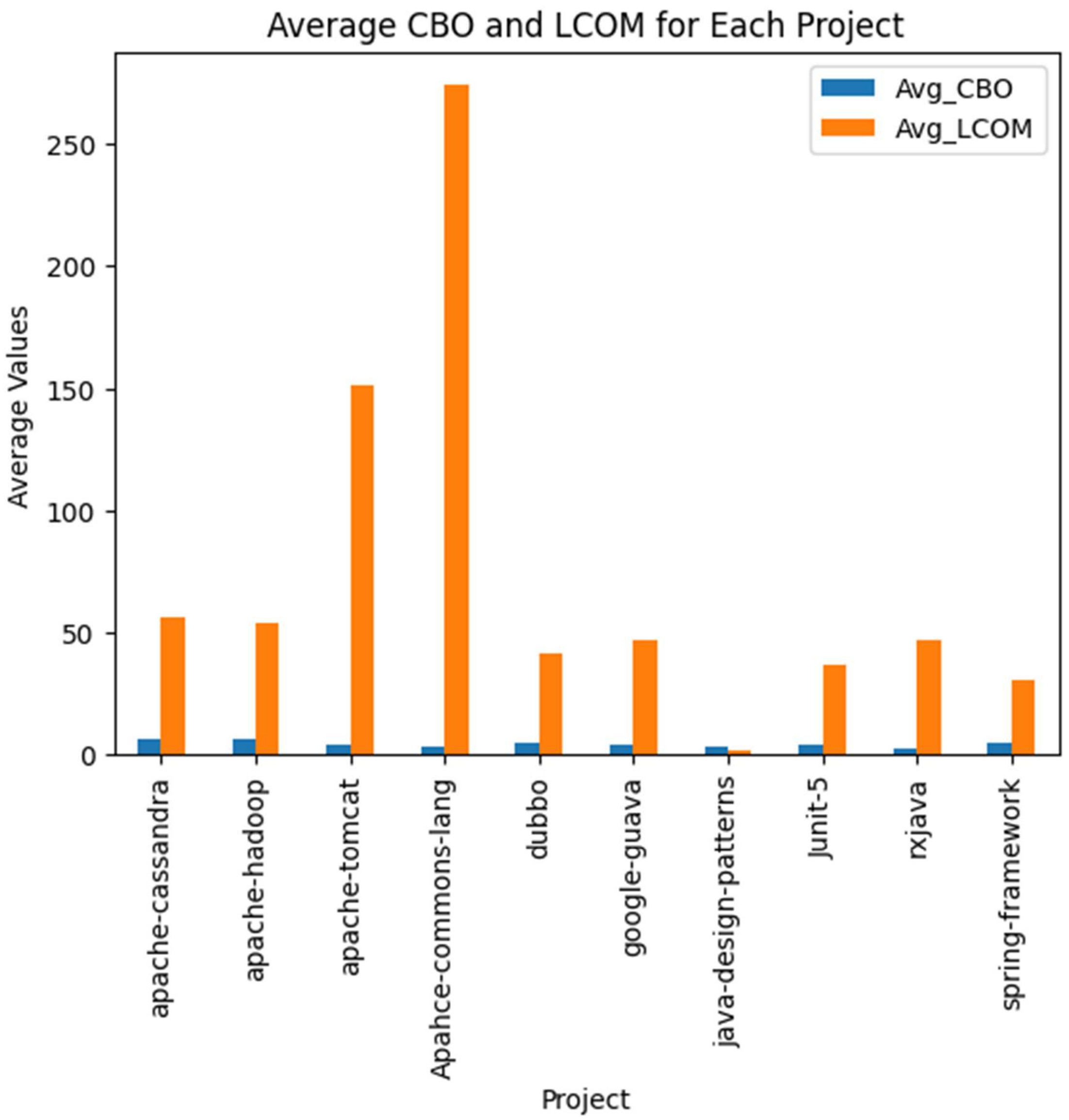
5.4 Visualizations

1)Average CBO Values Across Projects

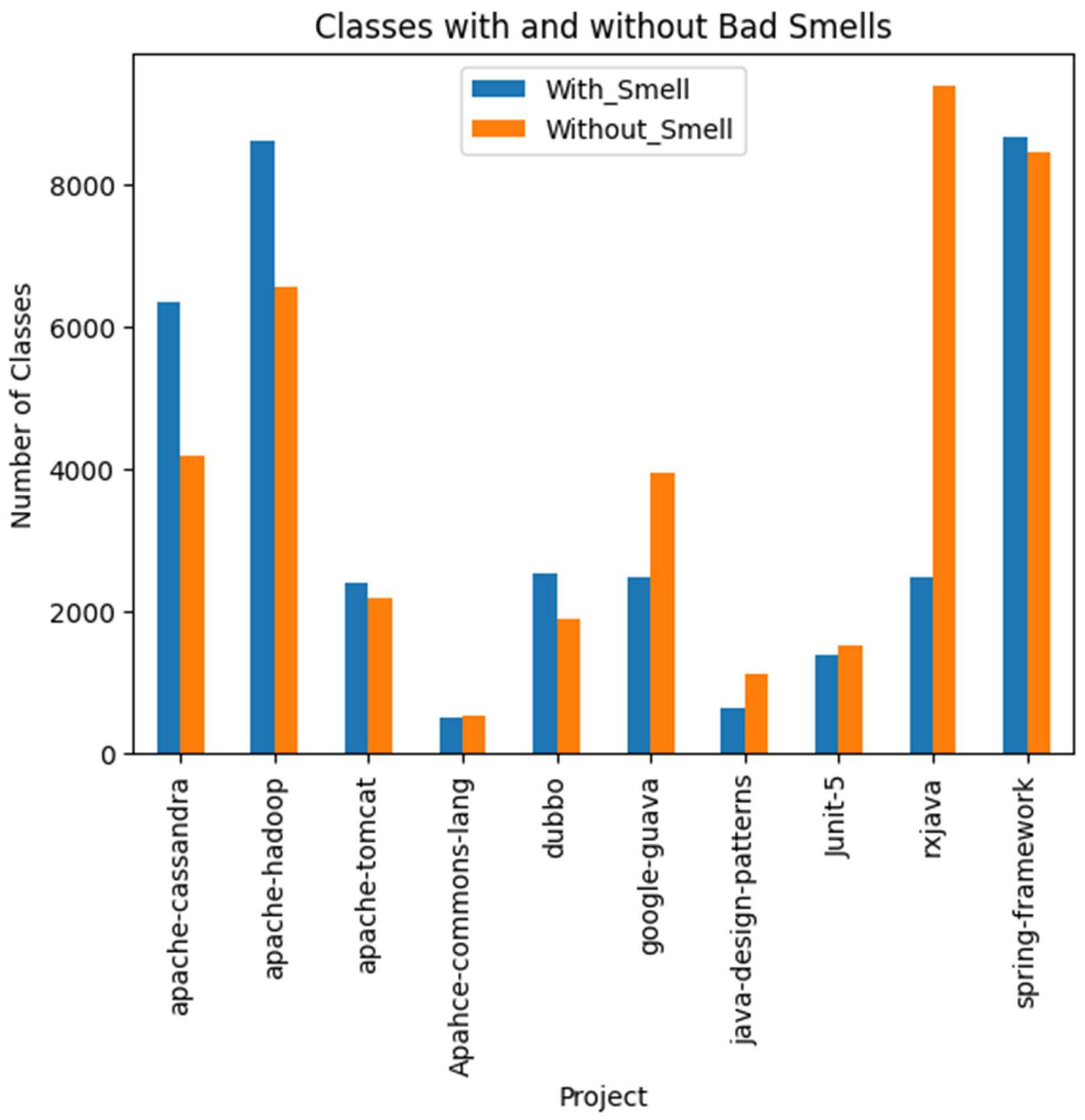
Observations:

High Coupling with Code Smells: When code smells are present, projects with higher average CBO values—such as Apache Hadoop and Apache Cassandra—show a greater degree of coupling between classes. This implies that code smells increase the interdependencies between classes, which in turn makes the system more inflexible and difficult to maintain.

Lower Coupling Despite Code Smells: On the other hand, despite code smells, other projects, such as Google Guava, have average CBO values that are lower. This might suggest the use of certain coding or architectural techniques that lessen the negative effects of odors on coupling and preserve improved modularity.



2)Average LCOM Values Across Projects



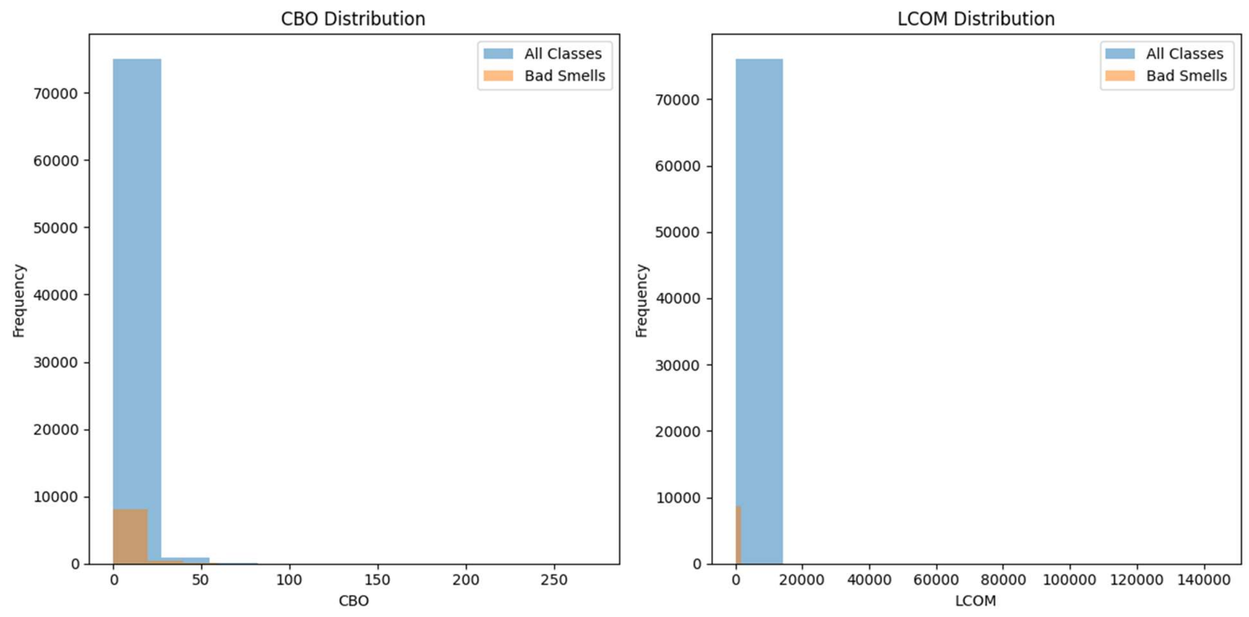
Observations:

High LCOM with Code Smells: When code smells are present, projects like Apache Commons Lang and Apache Tomcat exhibit abnormally high LCOM values. This implies that there may be a lack of relationship between the methods in these classes, which reduces modularity and makes the system more difficult to comprehend and maintain.

Improved Cohesion Without Code Smells: On the other hand, RxJava exhibits much lower LCOM values when there are no code smells, which suggests greater modularity and method cohesion when the code is cleaner. This demonstrates how resolving code smells helps to keep a system that is more modular and coherent.

3) Distribution Analysis of CBO and LCOM Metrics

The following visualizations compare classes with and without code smells by showing the distributions of Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM) across all classes in the examined projects. These diagrams provide light on the relationship between code smells and software system complexity and modularity.



CBO Distribution Observations:

The frequency of CBO values for all classes (blue bars) and classes with foul odors (orange bars) are shown in the CBO Distribution chart on the left.

All Classes: CBO values are clustered in the lower range (0-50) for most classes. This suggests that, in order to preserve modularity, the majority of classes have little coupling with one another.

Bad Smells: Compared to the overall distribution, classes with code smells are more prevalent in higher CBO ranges, although they still mostly reside in lower CBO ranges. According to this, classes that have a lot of code smells often have more coupling, which increases dependence and may cause rigidity in the system.

LCOM Distribution Observations:

The frequency of LCOM values for all classes (blue bars) and classes with offensive odors (orange bars) are shown in the LCOM Distribution chart on the right.

All Classes: Like CBO, most classes have LCOM values in the lower range, suggesting that most classes are cohesive in their methods.

Bad Smells: Higher LCOM ranges are significantly populated with classes that exhibit code smells. This demonstrates how classes with code smells often have fewer cohesive methods that are less connected to one another, which results in a class structure that is less modular and more dispersed.

6. Result and Findings

The main conclusions drawn from the data analysis and visualizations are outlined in this section. It draws attention to the primary effects of code smells on the modularity and maintainability of the software in all the projects under study.

6.1 Summary of Key Findings

Enhanced Complexity with Code Smells: In general, projects with higher CBO and LCOM values exhibit increased complexity, especially in the presence of code smells. This complexity makes the program more difficult to maintain and expand because of its higher interdependencies (high CBO) and lower cohesiveness (high LCOM).

Decreased Modularity: There is a significant correlation between decreased modularity and code smells. According to the research, classes with code smells often have weaker cohesiveness and more coupling, which makes the software architecture less flexible and more complex.

Effect on Maintainability: It is more difficult to maintain a system that has code smells. Elevated CBO values indicate more closely interconnected classes, which complicate the process of implementing discrete modifications. Lower cohesiveness is indicated by high LCOM scores, which results in less focused and more complicated classes.

7. Conclusions

The analysis of the effect of code smells on modularity in various software projects reveals significant insights into the impact of poor coding practices on system design and maintenance. The study examined two key metrics: Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM), which serve as indicators of modularity and the presence of code smells in the projects. The findings suggest that code smells have a detrimental effect on modularity, influencing both the coupling and cohesion of the codebase.

Firstly, the CBO values highlight the extent of coupling between different classes within the projects. High CBO values are indicative of tightly coupled systems where classes are highly dependent on one another. Projects such as apache-hadoop and apache-cassandra exhibit high average CBO values, especially when code smells are present. This suggests that code smells exacerbate the coupling between classes, making the code more interconnected and harder to maintain. On the contrary, projects like google-guava show lower CBO values even with the presence of code smells, which could be attributed to specific architectural or coding practices that help in mitigating the impact of smells on coupling.

Secondly, the LCOM values shed light on the cohesion within the classes. High LCOM values imply that the methods within a class are not well-related, reducing the overall modularity of the system. The analysis reveals that projects such as apache-tomcat and apache-commons-lang have extremely high LCOM values when code smells are present, indicating a significant reduction in cohesion and, consequently, modularity. In contrast, rxjava shows a markedly lower LCOM in the absence of code smells, pointing to better method cohesion and improved modularity when code smells are minimized.

The presence of code smells in a codebase tends to correlate with higher coupling and lower cohesion, leading to reduced modularity. This reduction in modularity can have several adverse effects on software projects. High coupling makes the system more brittle and less adaptable to changes, as modifications in one class may require changes in several other dependent classes. This interdependency complicates maintenance and increases the risk of introducing bugs. Furthermore, low cohesion within classes means that the functionality of the class is not well-defined, making it harder to understand, test, and reuse.

Moreover, the varied impact of code smells across different projects underscores the importance of context and the specific coding practices employed in each project. Some projects may adopt architectural patterns or coding standards that mitigate the adverse effects of code smells, leading to better modularity despite their presence. For instance, the lower CBO values in google-guava suggest that certain practices can help maintain modularity even when code smells exist.

In conclusion, the effect of code smells on modularity is evident across the studied projects. Higher coupling and lower cohesion are significant indicators of the negative impact of code smells, leading to less modular and more difficult-to-maintain codebases. Addressing code smells through refactoring and adhering to good coding practices is essential for improving modularity and ensuring the long-term maintainability and flexibility of software systems. The study highlights the need for continuous monitoring and management of code quality to mitigate the detrimental effects of code smells on modularity.

7.1 Recommendations

Refactoring: To address and lower high CBO and LCOM values, regularly carry out refactoring operations. This will make the codebase simpler to comprehend and alter, improving modularity and lowering maintenance costs.

Frequent Code Evaluations: In order to identify and address code smells early in the development process, establish regular code reviews. Interventions may be less expensive and more easily managed with early diagnosis.

Using the Best Practices: Promote the adoption of coherent coding techniques and reduce the number of dependencies in your code. This entails following guidelines such as single accountability and maintaining small, targeted groups for both approaches and seminars.

7.2 Future Work

Subsequent studies might investigate many directions:

Automated Refactoring and Detection Tools: Creating or using sophisticated tools that automatically identify and recommend reworking for code smells has the potential to greatly improve the quality and maintainability of software.

Long-Term Impact Studies: By examining code smells' long-term impacts on software projects over protracted periods of time, researchers may be able to get a better understanding of how they cumulatively affect system health.

Architectural Patterns: Researching how various design strategies and architectural patterns might improve modularity and lessen the effects of code smells in a variety of settings.

8. References

- Chidamber, S. R., & Kemerer, C. F. (1994). *A metrics suite for object-oriented design. IEEE Transactions on Software Engineering*, 20(6), 476-493.
- Aniche, M. (n.d.). CK: Tool for calculating Chidamber and Kemerer Java metrics. Retrieved from https://github.com/mauricioaniche/ck
- Tsantalis, N., & Chatzigeorgiou, A. (2009). *Identification of extract method refactoring opportunities for the decomposition of methods*. Journal of Systems and Software, 84(10), 1757-1782.
- JDeodorant: Eclipse Plug-in. [Online] Available at: http://jdeodorant.com/
- Apache Cassandra. (n.d.). Retrieved from https://cassandra.apache.org/
- Apache Hadoop. (n.d.). Retrieved from https://hadoop.apache.org/
- Apache Tomcat. (n.d.). Retrieved from https://tomcat.apache.org/
- Apache Commons Lang. (n.d.). Retrieved from https://commons.apache.org/proper/commons-lang/
- Apache Dubbo. (n.d.). Retrieved from https://dubbo.apache.org/en/
- Google Guava. (n.d.). Retrieved from https://github.com/google/guava
- Java Design Patterns. (n.d.). Retrieved from https://java-design-patterns.com/
- JUnit 5. (n.d.). Retrieved from https://junit.org/junit5/
- RxJava. (n.d.). Retrieved from https://github.com/ReactiveX/RxJava
- Spring Framework. (n.d.). Retrieved from https://spring.io/projects/spring-framework