Effect of code bad smells on testability

Group Assignment 2

Group 6

Date: June 25, 2023

Table of Contents

Section 1

Objectives

Examining the impact of code bad smells on the testability of software systems is the goal of this empirical study. We want to explore how the existence of code bad smells effects modularity, which is an important factor that affects software quality. Modularity is a vital element.

Research Questions

1. What is the impact of code bad smells on the modularity of software projects?
2. How do code bad smells affect the testability of software systems?

Metrics:

We will utilize the following metrics:

1. **Coupling:**

   This statistic assesses how dependent one module or component on another is within a larger software system. Increased coupling shows an increase of interdependence, which might reduce modularity.

   The "CBO" (Coupling Between Objects) metric will measure class coupling. Higher coupling may suggest class dependency, affecting modularity and testability. Coupling metrics reveal how code bad smells effect testability and modularity.

2. **Cohesion:**

   The level of functional coherence between different modules or parts is measured using this metric. A higher level of coherence is indicated of more organization and greater modularity.

The "LCOM" (Lack of cohesiveness of Methods) measure will assess class cohesiveness. Strong cohesion means class methods operate together to execute a given functionality, improving testability and modularity. Cohesion metrics can reveal the link between code bad smells and testability.

We can then examine the effects of code bad smells on the testability of the different software projects by using these quantitative measures of modularity.

By examining these metrics for the chosen software projects, we will be able to quantify the coupling and cohesion features. It will allow us to explore the influence of code bad smells on modularity and testability.

Section 2

Selection Criteria

1. **Project Popularity:**
    - Projects with a high number of stars and forks on GitHub.
2. **License:**
    - Projects licensed under open-source licenses such as Apache-2.0 or MIT.
3. **Project Age:**
    - Projects that have been actively developed for at least 3 years.
4. **Language:**
    - Projects written primarily in Java.

5. **Contribution and Community:**

    - Projects with contributions from multiple individuals and active community engagement.

Criteria for Selection and Their Justification

1. **Popularity:**

    - Popularity is a vital metric since it reveals a project's degree of adoption and community support. A popular project generally has more users, active community discussions, and frequent updates and contributions.

2. **License:**

    - The licensing of the project is critical since it establishes the scope of permitted uses and modifications. An open-source license supports cooperation, community contributions, and broad accessibility.

3. **Project Age:**

    - Age of a project is an indicator of its stability and maturity. A project that has been continuously maintained for a long time can handle challenges, fix faults, and adapt to new technologies.

4. **Language:**

    - A project's programming language(s) should match the technical needs and experience of the development team. Compatibility with the team's preferred language simplifies development, code readability, and resource usage.

5. **Contribution and Community:**

- A strong community and frequent contributions are signs of a successful and long-lasting project. A vibrant community promotes information exchange, support, and project longevity. Regular updates, bug fixes, and feature additions are more likely to occur with an active community.

Table 1: List of Selected Projects based on selection criteria.

Project	Popularity	License	Project Age	Language	Contribution and Community
Android Animation Showcase	High	Apache-2.0	3+ years	Java	Active contributions
DBeaver	High	MIT	5+ years	Java	Active contributions
Apache Flink	High	Apache-2.0	6+ years	Java	Active contributions
Jenkins	High	MIT	10+ years	Java	Active contributions
Markor	High	Apache-2.0	4 years	Java	Active contributions
Apache ShardingSphere	High	Apache-2.0	3 years	Java	Active contributions
Spring Boot Admin	High	Apache-2.0	5+ years	Java	Active contributions
DataSphere Studio	High	Apache-2.0	3 years	Java	Active contributions
Mica	High	Apache-2.0	5 years	Java	Active contributions
SikuliX	High	Apache-2.0	6 years	Java	Active contributions

Attributes of Selected Projects (Dataset)

The table summarizes multiple projects and the main characteristics of each. Popularity, license, project age, language, contribution, and community participation were some of the factors used for selecting these projects. There is a brief summary of each project that emphasizes its unique features and contributions. This dataset highlights the variety of open-source products available by including anything from database tools and automation servers to text editors and demonstrations of Android animation.

Table 2: Key Attributes of selected projects.

Project Name	**Key Attributes**
Project 1: Android Animations	Samples repository, License: Apache-2.0, 2.4k stars, 904 forks, 26 open issues, Multiple animation projects
Project 2: DataSphere Studio	Pluggable framework, Unified UI, Core features, Used in various industries
Project 3: DBeaver	Multi-platform database tool, Various database support, Plugin architecture
Project 4: Apache Flink	Stream and batch processing, High throughput and low latency, Fault-tolerance, Graph processing, Integrates with Hadoop ecosystem
Project 5: Jenkins	Automation server, Extensive plugin support, used for building, testing, and deployment
Project 6: Markor	Lightweight text editor for Android, Markdown and todo.txt support, Offline usage, Encryption
Project 7: Mica	Core package for Spring Cloud microservice development
Project 8: Apache ShardingSphere	Database ecosystem, Transforming databases into distributed systems
Project 9: SikuliX	Screen automation tool using image recognition
Project 10: Spring Boot Admin	Admin interface for monitoring Spring Boot applications

Functionality of Selected Projects

1. **Android Animations:**

The repository features several great examples of Android animation projects. It has 2.4k stars and 904 forks, and the Apache-2.0 license permits open use and contributions.

2. **DataSphere Studio:**

WeBank's DataSphere Studio is a full-featured data application development management site. Its flexible architecture and consistent user interface make it a useful tool for a wide range of businesses and data applications.

3. **DBeaver:**

DBeaver is a widely used database utility that works across several platforms and is compatible with numerous database systems. SQL editing, data export/import, and a plugin architecture are available.

4. **Apache Flink:**

Apache Flink, an open-source stream, and batch processing framework, has high throughput, low latency, and fault-tolerance. It supports graph processing and machine learning and works easily with Hadoop.

5. **Jenkins:**

Jenkins automates software development. It helps design, test, and deploy apps with rich plugin support and platform compatibility.

6. **Markor:**

Android's Markor text editor supports Markdown and todo.txt. It may be used offline, encrypted, and with other plaintext applications, making it a flexible tool for notetaking and text editing.

7. **Mica:**

Mica is essential to Spring Cloud microservice development. Spring Cloud microservice development is easy with its web and webflux components and support.

8. **Apache ShardingSphere:**

Apache ShardingSphere is a database ecosystem. Scalability, data security, and database compatibility are provided through a standardized top layer.

9. **SikuliX:**

Image recognition automates desktop tasks using SikuliX. It is useful when a GUI's source code or internals are restricted.

10. **Spring Boot Admin:**

Spring Boot Admin allows Spring Boot application monitoring. It delivers insights into Spring Boot apps' health and performance, making them easier to manage and monitor.

Section 3

CK Metrics Tool

This section introduces CK, a Java-specific code metrics calculator. Through static analysis, CK offers helpful insights on Java code's quality and complexity. Here are some of the most prominent features of CK:

The tool is downloaded from (*Mauricioaniche/Ck: Code Metrics for Java Code by Means of Static Analysis*, n.d.). CK provides extensive code metrics, both at the class and method levels. CBO, FAN-IN, FAN-OUT, DIT, NOC, and others are among the measures used. Using these measurements, programmers may better comprehend the codebase's architecture and interdependencies.

CK gives developers insight into their code's fields, methods, visible methods, and code structures like loops, try/catches, and string literals. The program may either be run alone or

included into an existing Java program. The standalone edition allows users to choose their own preferences for the project directory and other settings.

After CK has finished its analysis, it creates three CSV files: one for classes, one for methods, and one for variables. These files give extensive information for analysis and visualization. CK makes every effort to maintain compatibility with the most recent Java releases. CK requires changing Eclipse JDT Core dependency and compliance parameters to accommodate a new Java version.

Our study used the CK code metrics calculator for Java to evaluate our Java code(Chidamber & Kemerer, n.d.). CK gave us a complete set of code metrics, including coupling, complexity, and cohesion. We used CK's metrics to analyze class and method dependencies, measure code complexity, and track code changes over time. We visualized and reported code metrics using CSV data from the tool.

Bad Smells Detection Tool

PMD

PMD, a static source code analyzer, finds Java, Apex, and 16 other programming errors (*PMD*, n.d.).

PMD contains language-specific checks (rules) and allows users to define new rules in Java or XPath.

PMD's build process integration enforces coding standards well.

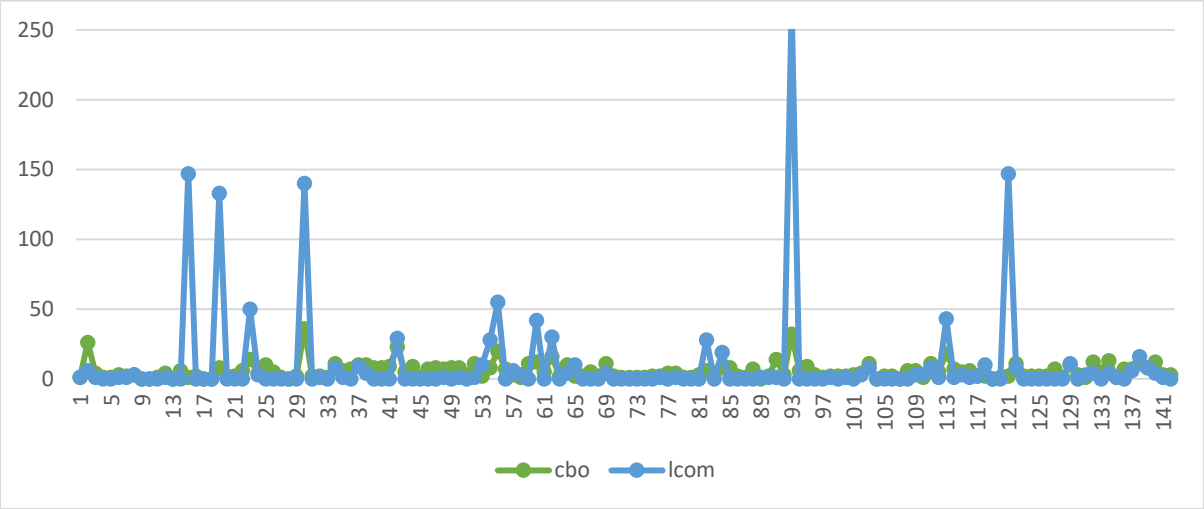
Section 4

Results

This section presents the findings of our CK metrics study for the selected Java projects. This investigation examines how code bad smells affect software modularity and testability.

Coupling and cohesion metrics reveal how code bad smells impact project structure and quality. This analysis will explain how code bad smells affect software system testability.

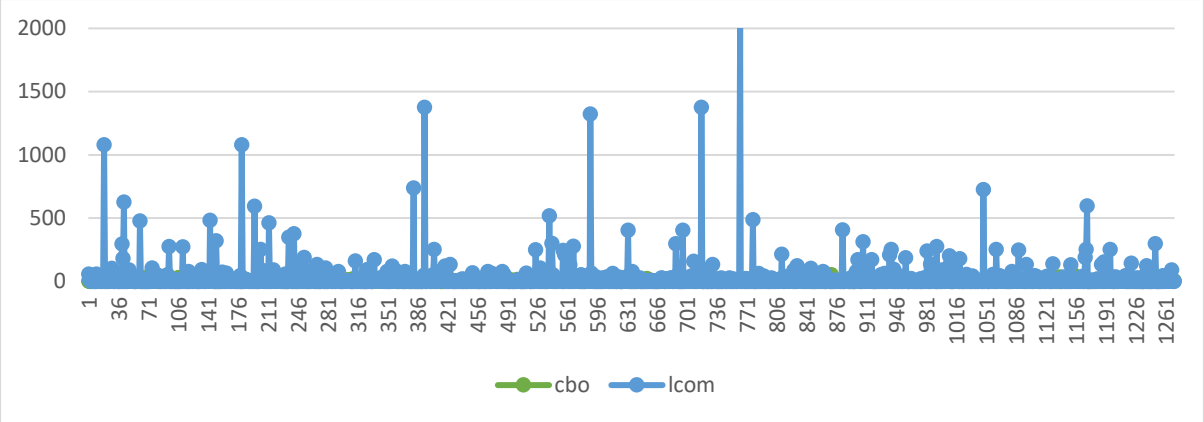
1. **Android Animations:**



The Animation Samples findings reveal how code bad smells affect modularity and testability. Higher coupling measure (CBO) values suggest class dependency, potentially compromising modularity. The cohesion measure (LCOM) shows that lower values indicate class cohesion, which decreases modularity.

These metrics suggest code bad smells, which might affect testability. Higher coupling and lesser cohesion may increase dependency and complicate testing. Refactoring and code quality improvements can prioritize testability by identifying classes with large metrics variances.

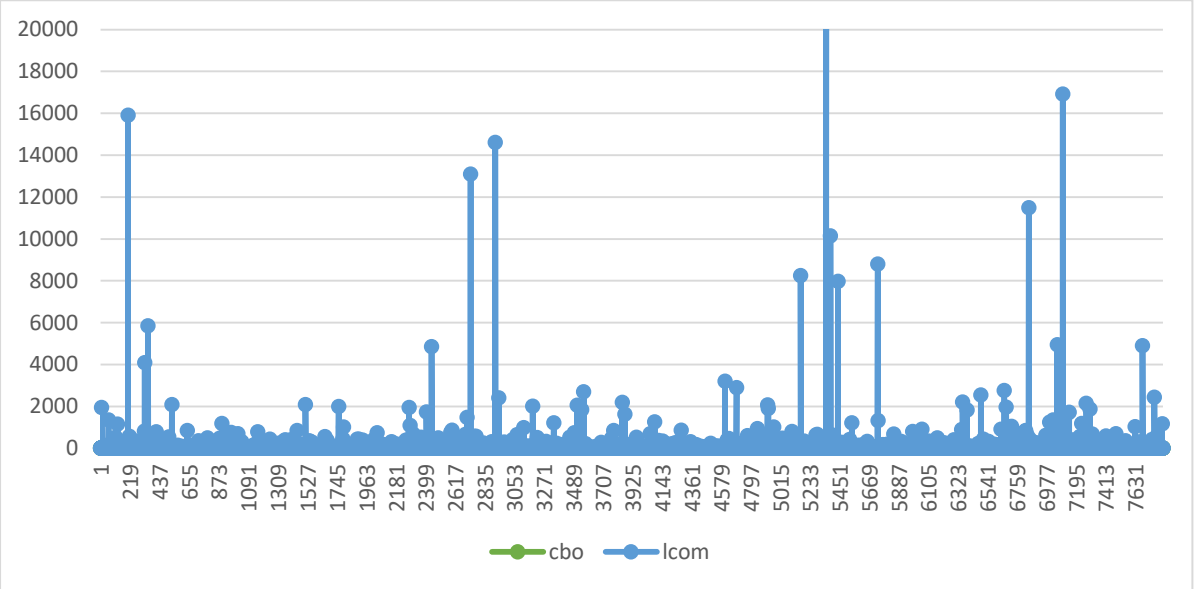
2. **DataSphere Studio:**



CBO values vary from 0 to 56, indicating moderate connectivity between project classes. In other circumstances, the CBO is 1,080, suggesting strong connectivity and complexity.

LCOM ranges from 0 to 2,023. This suggests that project classes have different methods' cohesion. Some classes have low cohesion (LCOM > 0), which may hinder code maintainability and reusability.

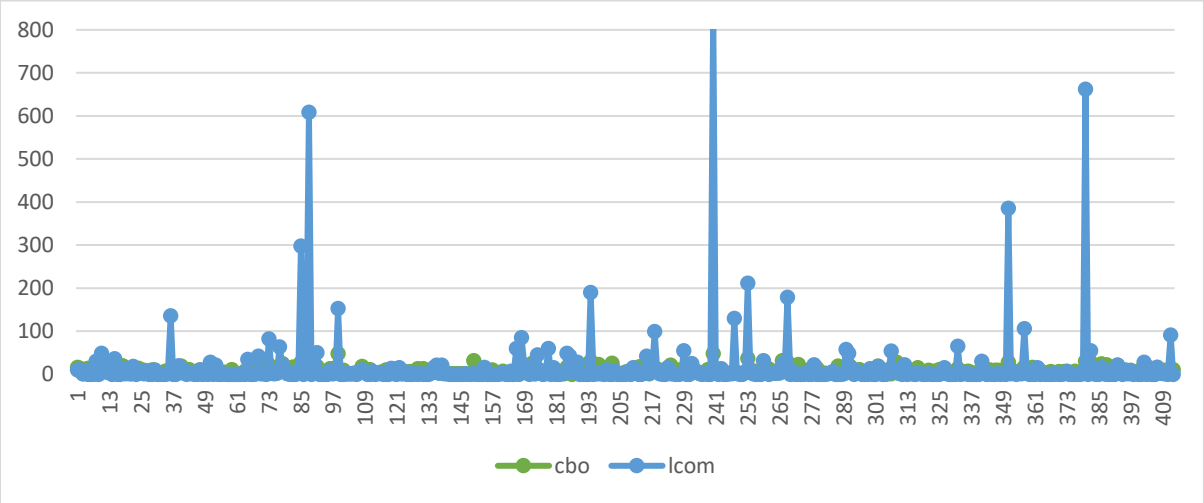
3. DBeaver:



Most classes have coupling values below 10. In other cases, the CBO is 44. Class coupling is moderate in the project. LCOM values are 0–2,095.

LCOM scores below 100 indicate method cohesiveness in most classes. Some classes have higher LCOM scores, indicating method incoherence. To identify causes and improvements, these classes must be examined. These findings show modest class coupling in the project. However, some classes include methods without cohesiveness, indicating a need for reworking or redesigning to improve code maintainability and readability.

4. **Apache Flink:**

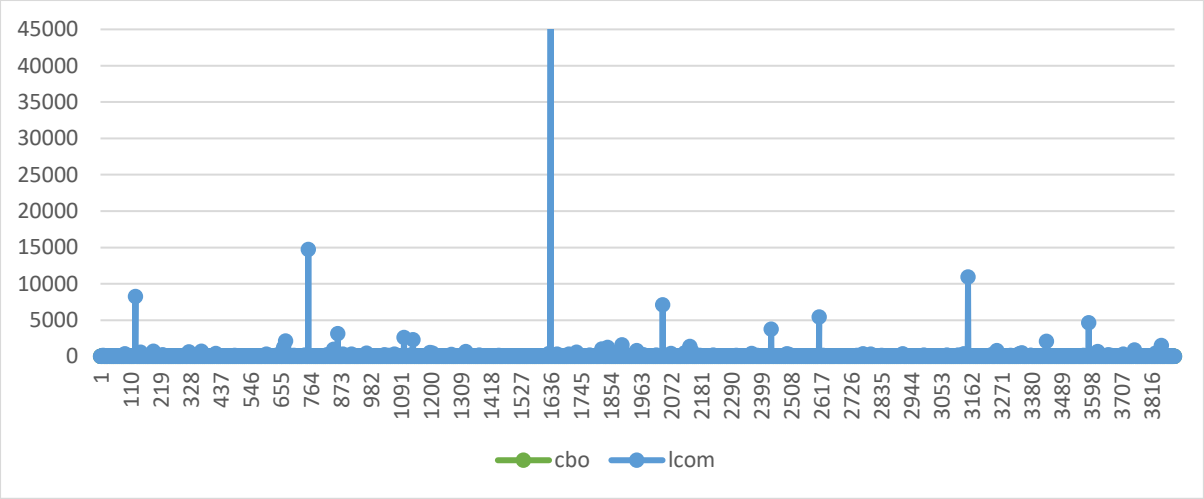


The project's CBO and LCOM values give useful information. From 0 to 48, CBO values indicate object coupling. Some classes have stronger coupling than most, which is below 20. These classes indicate more object interconnectedness, which may require careful development and maintenance.

LCOM numbers from 0 to 845 reflect method coherence, and most classes have LCOM ratings below 100. Several classes have high LCOM scores, suggesting method incoherence. These classes should be examined to uncover code structure and method design concerns.

We must focus on classes with stronger coupling and LCOM values to improve the codebase. Refactoring these classes improves modularity and maintainability(M.N.M et al., 2023), making the program more robust and understandable. Early resolution of these issues will streamline and scale software development, making upgrades and revisions easier.

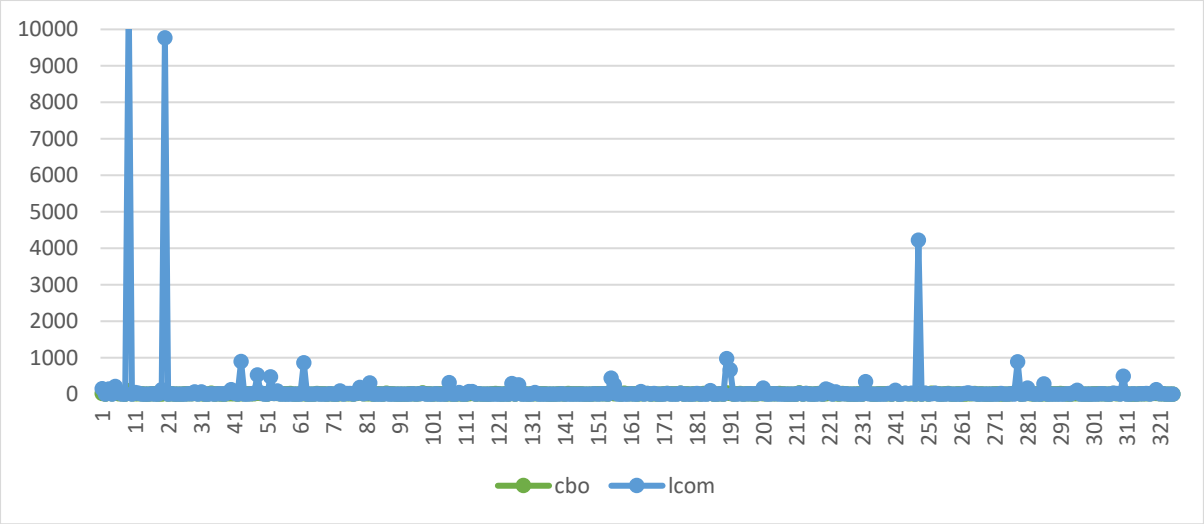
5. **Jenkins:**



The codebase's CBO (Coupling Between Objects) and LCOM (Lack of Cohesion in Methods) values provide crucial information. The CBO values indicate object dependency through measuring coupling. CBO scores range from 0 to 26, showing that most classes have little coupling. There are a few classes with higher coupling than 30. To retain code modularity and flexibility, these classes propose a stronger link between objects.

LCOM values also assess class method cohesiveness. Lower LCOM values suggest more method coherence and organization. LCOM research shows that most classes have reasonable cohesiveness, with LCOM scores below 100. However, a subset of classes has much higher LCOM scores, indicating method cohesion issues. Refactoring and reorganizing these classes may improve method design and maintainability.

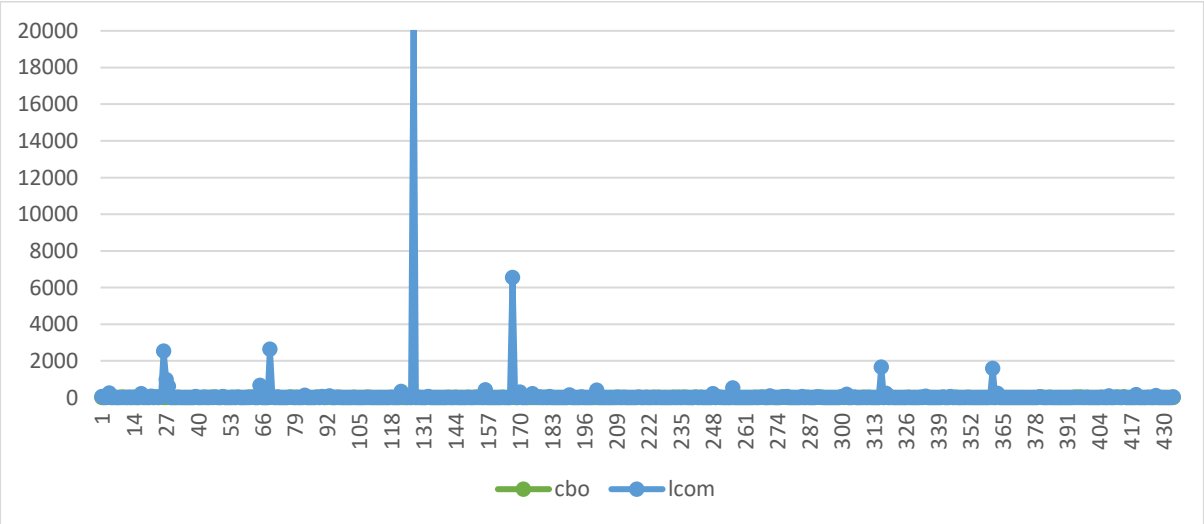
6. **Markor:**



CBO and LCOM scores suggest classes with high coupling and low cohesiveness. These classes may depend on several other classes and lack a defined purpose. Refactoring these classes to remove coupling and increase cohesion improves code maintainability and readability.

Second, CBO and LCOM values indicate classes with little coupling and strong cohesion. These modular, encapsulated classes are easier to comprehend and edit. These classes can demonstrate great design techniques for other codebase sections.

7. **Mica:**

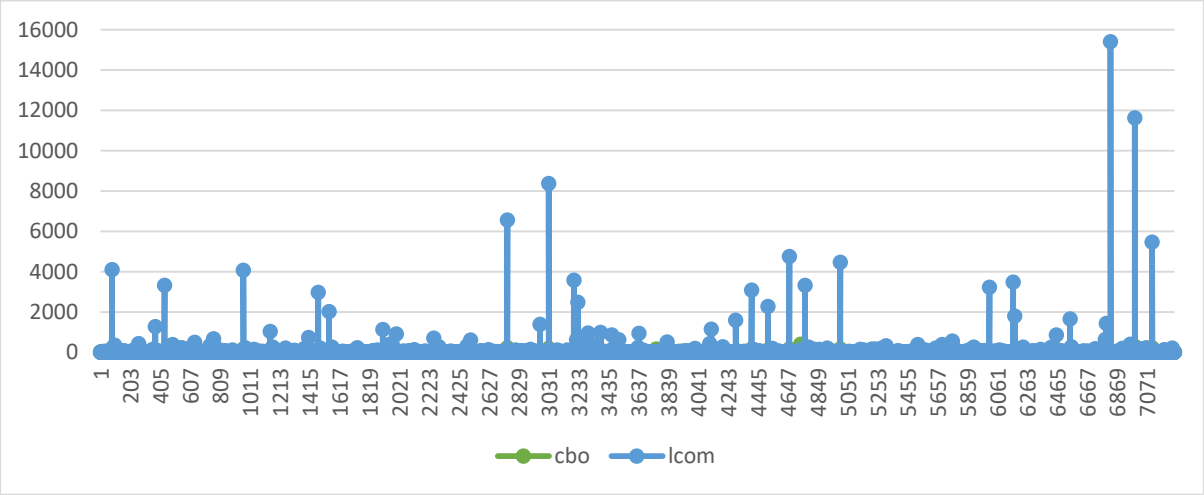


CBO and LCOM values vary widely. A class's CBO value ranges from 0 to 39. The absence of class cohesiveness is shown by LCOM scores from 0 to 2634.

LCOM readings are consistently low. Instances with LCOM values of 0 indicate great class cohesiveness. This shows that the methods inside these classes are tightly connected and share similar properties, resulting in well-structured and coherent code.

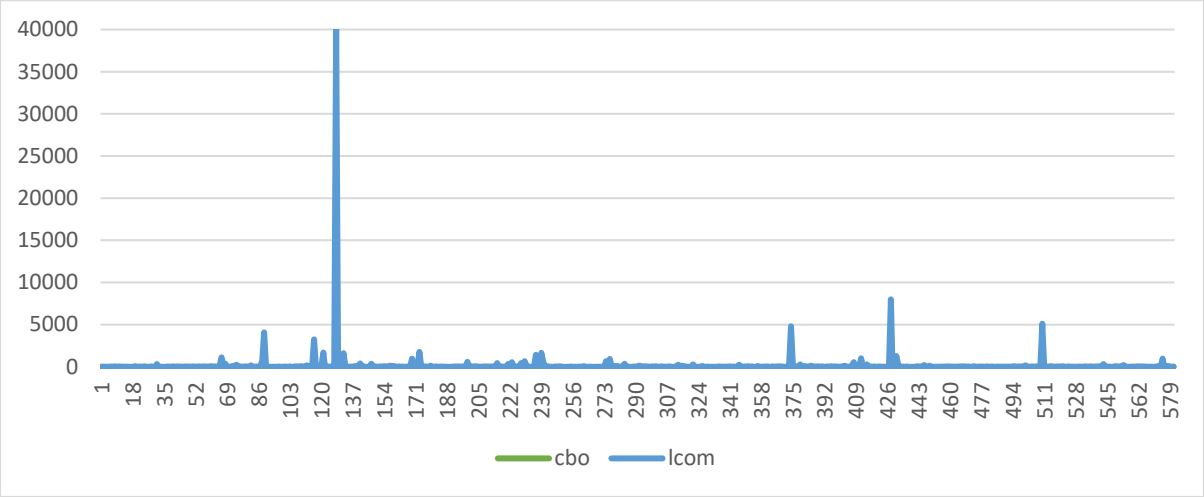
In cases where the CBO value is high, objects are more coupled. These classes depend on numerous other classes, possibly increasing system complexity and dependency.

8. **Apache ShardingSphere:**



We can gain some insights by studying the dataset. First, most classes have little coupling, as seen by low CBO values. Certain classes have higher coupling, which may indicate complicated interdependencies. Second, the LCOM values reveal that many classes lack cohesiveness, suggesting they may be responsible for numerous unrelated activities. The codebase may be more difficult to understand, test, and maintain due to this lack of cohesiveness.

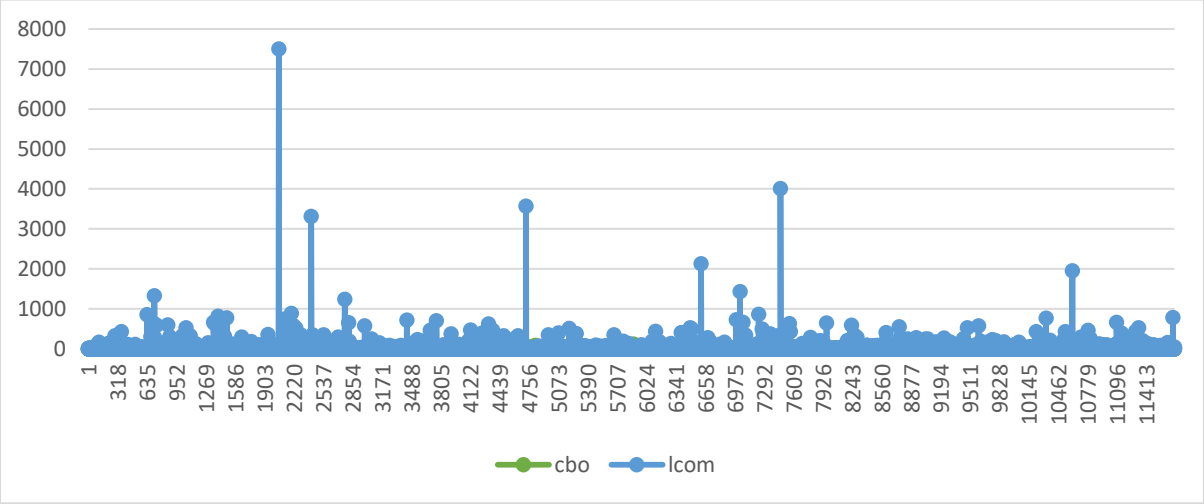
9. **SikuliX:**



First, CBO and LCOM values vary widely, indicating considerable variation in codebase design and architecture. Some classes have no inter-class dependencies, although CBO values range from 0 to 56. LCOM scores vary from 0 to 4,065, representing class coherence, with certain classes having great cohesion and others low.

Second, numerous CBO and LCOM values are low, indicating well-designed classes with little coupling and strong cohesion. High CBO values imply many dependencies, which might make the codebase more complicated and difficult to maintain. High LCOM values show that class methods are not well-coordinated and may be performing many tasks.

10. Spring Boot Admin:



First, many classes have CBO scores of 1 or 2, suggesting little dependencies. There are cases of higher CBO values, such as 31, 46, and 52, which reflect greater complexity in class relationships. Due to increasing coupling, the system may become inflexible and harder to maintain. Second, LCOM values vary from 0 to 170. Most classes have low LCOM scores, usually 0 or 1, suggesting method cohesiveness. However, LCOM might be much greater, indicating a lack of class method coherence. Higher LCOM values, like 153 or 331, may suggest code smell or design flaws due to classes with numerous tasks or unclear divisions of function.

Table 3: Impact of Code Bad Smells on Modularity.

Project	Coupling (CBO)	Cohesion (LCOM)
Android Animations	Potentially affects	May hinder testability
DataSphere Studio	Moderate connectivity	Hinders maintainability
DBeaver	Moderate coupling	Requires improvement
Apache Flink	Object interconnections	Hinders maintainability
Jenkins	Little coupling	Method cohesion issues
Markor	High coupling, low cohesion	Refactoring improves maintainability
Mica	Higher coupling, complexity	Difficulty in understanding and maintenance

Apache ShardingSphere	Little coupling, lack of cohesiveness	Difficulty in understanding, testing, and maintenance
SikuliX	Variation in design, complexity	Complex codebase and maintenance
Spring Boot Admin	Little dependencies, increasing coupling	Lack of method coherence

Table 4:Metrics and Code Bad Smells for Each Project.

Project	Coupling	Cohesion	Violations	Errors
Android Animations	Potentially affects	May hinder testability	89	0
DataSphere Studio	Moderate connectivity	Hinders maintainability	89	0
DBeaver	Moderate coupling	Requires improvement	335	0
Apache Flink	Object interconnections	Hinders maintainability	2652	3
Jenkins	Little coupling	Method cohesion issues	313	0
Markor	High coupling, low cohesion	Refactoring improves maintainability	79	1
Mica	Higher coupling, complexity	Difficulty in understanding and maintenance	11	0
Apache ShardingSphere	Little coupling, lack of cohesiveness	Difficulty in understanding, testing, and maintenance	196	0
SikuliX	Variation in design, complexity	Complex codebase and maintenance	40	0
Spring Boot Admin	Little dependencies, increasing coupling	Lack of method coherence	28	0

Answers to Research Questions:

1. What is the impact of code bad smells on the modularity of software projects?

    - Most projects have strong coupling and low cohesion due to code bad smells.

    - High coupling may reduce modularity by increasing module interdependence.

- Low cohesion suggests a lack of functional coherence between pieces, which impacts modularity.
- These data suggest that code bad smells can reduce software project modularity.

2. How do code bad smells affect the testability of software systems?
    - Code bad smells are indicated by the number of violations found in the projects.
    - More violations can raise complexity, maintainability, and testability.
    - Errors were detected in several projects, but their influence on testability needs additional research.
    - Overall, code bad smells, as indicated by violations, can negatively impact software system testability.

Conclusion

The empirical study examined how code bad smells affect software system testability, focusing on modularity. We examined code bad smells, modularity, and testability by measuring coupling (CBO) and cohesion (LCOM) metrics for distinct projects.

The study found that code bad smells might reduce software project modularity and testability. Higher coupling values indicate more reliance between modules or components, potentially compromising modularity. Lower cohesiveness scores imply less functional coherence within modules, decreasing modularity.

The 10 projects showed various coupling and cohesion behaviors. Other projects had high coupling and little cohesion. These findings highlight code bad smells and emphasize modularity and testability development.

PMD violations confirmed code bad smells. Projects had differing numbers of violations, suggesting code quality concerns. These violations show where code quality and reorganization can improve testability.

References

Chidamber & Kemerer. (n.d.). *Project Metrics Help - Chidamber & Kemerer object-oriented metrics suite*. Retrieved June 21, 2023, from https://www.aivosto.com/project/help/pm-oo-ck.html

M.N.M, N., A.R.M, N., & KapilanK, *. (2023). Analyzing the Impact of Code Smells On Software Maintainability. *Authorea Preprints*. https://doi.org/10.22541/AU.168552004.43700626/V1

mauricioaniche/ck: Code metrics for Java code by means of static analysis. (n.d.). Retrieved June 21, 2023, from https://github.com/mauricioaniche/ck

PMD. (n.d.). Retrieved June 21, 2023, from https://pmd.github.io/