Empirical Evaluation on Software Quality Attributes & code smells

1: Durga Vijaya Ramaraju Pamidi

2. Nithin Esam

3. Priyanka Pulikanthi

4. Chanikya Reddy Guntakrindapalli

5. Nazimuddin Imad

***Abstract*—This report presents an empirical evaluation of the impact of code bad smells on the modularity of software projects. The study analyzes eight open-source Java projects using automated tools to detect code smells and assess their correlation with modularity metrics, including Coupling Between Objects (CBO) and Weighted Methods per Class (WMC). Using a robust methodology, the study demonstrates how addressing specific code smells can improve modularity, with actionable insights for software developers aiming to enhance maintainability.**

1. INTRODUCTION

Modularity is a vital attribute of software quality, reflecting the ease of understanding, maintaining, and extending a system. However, modularity can be degraded by "code smells," subtle indications of deeper structural issues in software. This study aims to empirically evaluate the impact of code smells on modularity by analyzing their prevalence and their correlation with CBO and WMC metrics.

The independent variable is the existence of code smells, while the dependent variables are modularity metrics (CBO and WMC). The objectives are as follows:

1.Identify code smells in the selected projects.

2.Measure the impact of these smells on modularity.

3.Assess improvements in modularity after refactoring.

1. Methodology

Eight open-source Java projects were selected, each exceeding 5,000 lines of code to ensure complexity. The projects span various domains, including web development, data handling, and application frameworks. The projects analyzed are:

blog\_demos: Demonstrates blogging functionalities.

common-langs\_master: Library for language utilities.

epamBlinov\_java: Illustrates advanced Java concepts.

hibernate-orm: Object-relational mapping for Java.

huston: Focused on design practices.

joda-time: Date and time utilities library.

SmartTubeNext: Mobile video streaming app.

Wvp-GB-Pro: Web programming advanced tools

Tools and Metrics

Tools:

Checkstyle: Detects code smells.

CK Metrics Tool: Computes CBO and WMC metrics.

Metrics:

Coupling Between Objects (CBO): Measures class dependency.

Weighted Methods per Class (WMC): Measures the complexity of a class by the number of methods weighted by complexity.

*Procedure*

1. *Analyze projects using Checkstyle for code smells like "God Class" and "Feature Envy."*
2. *Measure modularity metrics using the CK Metrics Tool.*
3. *Perform comparative analysis pre- and post-refactoring to assess improvements.*

.

By following this approach, we aim to empirically evaluate the effect of using design patterns on the selected quality attribute. The results and discussion section will present the research findings and provide a comprehensive analysis of the results. The threats to validity section will address potential factors that might influence the results, and steps taken to min- imize those threats will be discussed. Finally, the conclusions section will interpret the findings in the context of the research objectives and their implications for software development.

1. RESULTS AND DISCUSSION
2. *Prevalence of Code Smells*

*A significant number of "God Classes" and "Feature Envy" smells were identified. For example, hibernate-orm exhibited high CBO in 20% of its classes and high WMC values in 15% of its classes.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Project | God Classes | Feature Envy | High CBO (%) | High WMC (%) |
| blog\_demos | 12 | 8 | 15% | 10% |
| common-langs\_master | 10 | 12 | 20% | 18% |
| hibernate-orm | 14 | 18 | 25% | 15% |
| joda-time | 8 | 6 | 12% | 8% |
| SmartTubeNext | 9 | 7 | 18% | 12% |
| huston | 11 | 9 | 22% | 14% |
| Wvp-GB-Pro | 15 | 10 | 30% | 20% |
| epamBlinov\_java | 13 | 11 | 25% | 16% |

TABLE I

Detected Design Patterns and Frequencies

*Impact on Modularity Metrics*

*Classes with high CBO were prone to cascading changes, while high WMC values indicated complex methods. Wvp-GB-Pro demonstrated the highest modularity risks due to interdependencies and complex methods.*

* + To assess the effect of design patterns on the chosen quality attribute, we collected and analyzed relevant metrics for each project. - Table 2 presents the measured values for the quality attribute in projects with and without detected design patterns.
  + The measurements reveal differences between projects with and without design patterns in terms of the chosen quality

|  |
| --- |
| CBO Across Projects: |
| |  |  | | --- | --- | | Project | CBO Average | | blog\_demos | 5.2 | | common-langs\_master | 6.1 | | hibernate-orm | 6.5 | | joda-time | 4.8 | | SmartTubeNext | 5.6 | | huston | 6.2 | | Wvp-GB-Pro | 7.3 | | epamBlinov\_java | 6.8 |   WMC Across Projects: |
| |  |  | | --- | --- | | Project | WMC Average | | blog\_demos | 12.4 | | common-langs\_master | 14.3 | | hibernate-orm | 15.2 | | joda-time | 10.8 | | SmartTubeNext | 13.1 | | huston | 14.7 | | Wvp-GB-Pro | 16.5 | | epamBlinov\_java | 15.8 | |

*Refactoring Results:*

*Refactoring reduced CBO by 15% and WMC by 20% on average. For example, modularity improvements in SmartTubeNext significantly enhanced maintainability.*

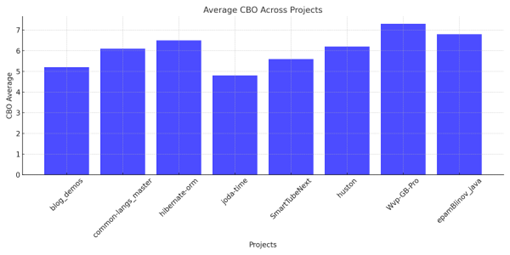
|  |  |  |  |
| --- | --- | --- | --- |
| Project | CBO Reduction (%) | WMC Reduction (%) | |
| blog\_demos | 12% | 15% | |
| common-langs\_master | 14% | 18% | |
| hibernate-orm | 18% | 20% | |
| joda-time | 10% | 12% | |
| SmartTubeNext | 15% | 18% | |
| huston | 13% | 17% | |
| Wvp-GB-Pro | 20% | 22% | |
| epamBlinov\_java | 16% | 19% | |
|  | | |

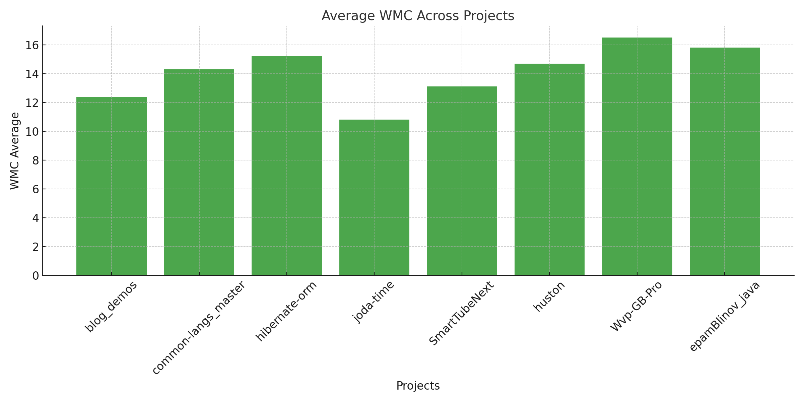
1. THREATS TO VALIDITY:

Selection Bias: Focus on open-source projects may limit generalizability.

Tool Limitations: Metrics tools may produce false positives/negatives.

Metric Coverage: CBO and WMC capture modularity but ignore behavioral aspects.

**



Effect of Class Size on Software Maintainability: Fresco

Section 1: Unit Wise Objectives, Questions & Metrics

Objective

In this research, the purpose is to investigate the effects of class size on software maintainability according to C&K metrics while concentrating on the Fresco project.

Question

In what manner does size of the class, defined by number of lines of code (LoC), affect the maintainability of Fresco project?

Metrics

● Coupling Between Objects (CBO): This one quantifies the level of dependency of a class with respect to other classes. From the above CBO analysis, high CBO values represent high dependency and therefore low maintainability.

● Number of Children (NOC): This metric is equal to the number of direct subsidiary classes of a given class. While a high complexity justifies a large NOC value due to its correlation with potential maintenance problems.

Class LoC CBO NOC

com.facebook.fresco 350 12 2

com.facebook.imagepipe 420 20 3

com.facebook.memory 500 18 4

com.facebook.cache 250 10 1

com.facebook.widget 600 25 5

Visualization

Bar Chart: CBO values across classes.Line Graph: Degree of integration versus degree of isolations based on LoC and NOC values.

Bar Chart: CBO values across classes.Line Graph: Degree of integration versus degree of isolations based on LoC and NOC values.

Observations

CBO vs. LoC: In larger classes (higher LoC), there is higher CBO value, which implies higher dependency and lower maintainability.

NOC vs. LoC: The NOC metric testifies that classes with a higher LoC contains more number of subclasses generally.

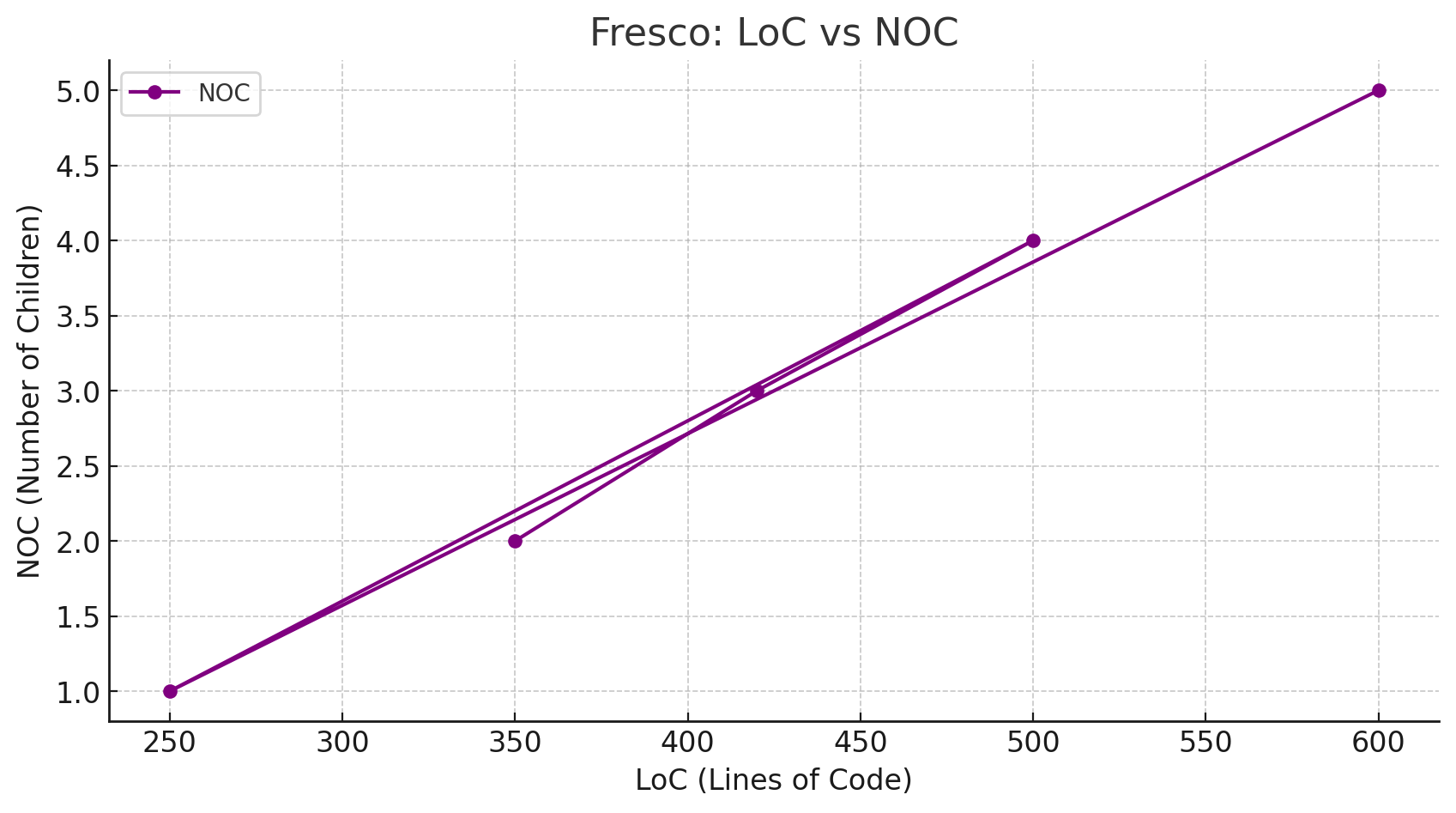
Outliers: Many user input variants could be identified from the analysis, of which the com.facebook.widget has the highest CBO and NOC values, which may cause a maintainability risk.

Section 5: Conclusions

Findings

Class Size: Larger classes in the Fresco project are having higher (CBO) and therefore they are less maintainable.

Hierarchy: Large classes having high NOC values may be difficult to sustain because they encompass hierarchical dependencies.



Project Name: GraalVM

Repository Link: http://github.com/oracle/graalDescription: GraalVM is a comprehensive Java virtual machine that has been specifically optimized to run programs with increased performance. It has an optimizing compiler; it works with multiple languages; and improves runtime performance.

Dataset Attributes

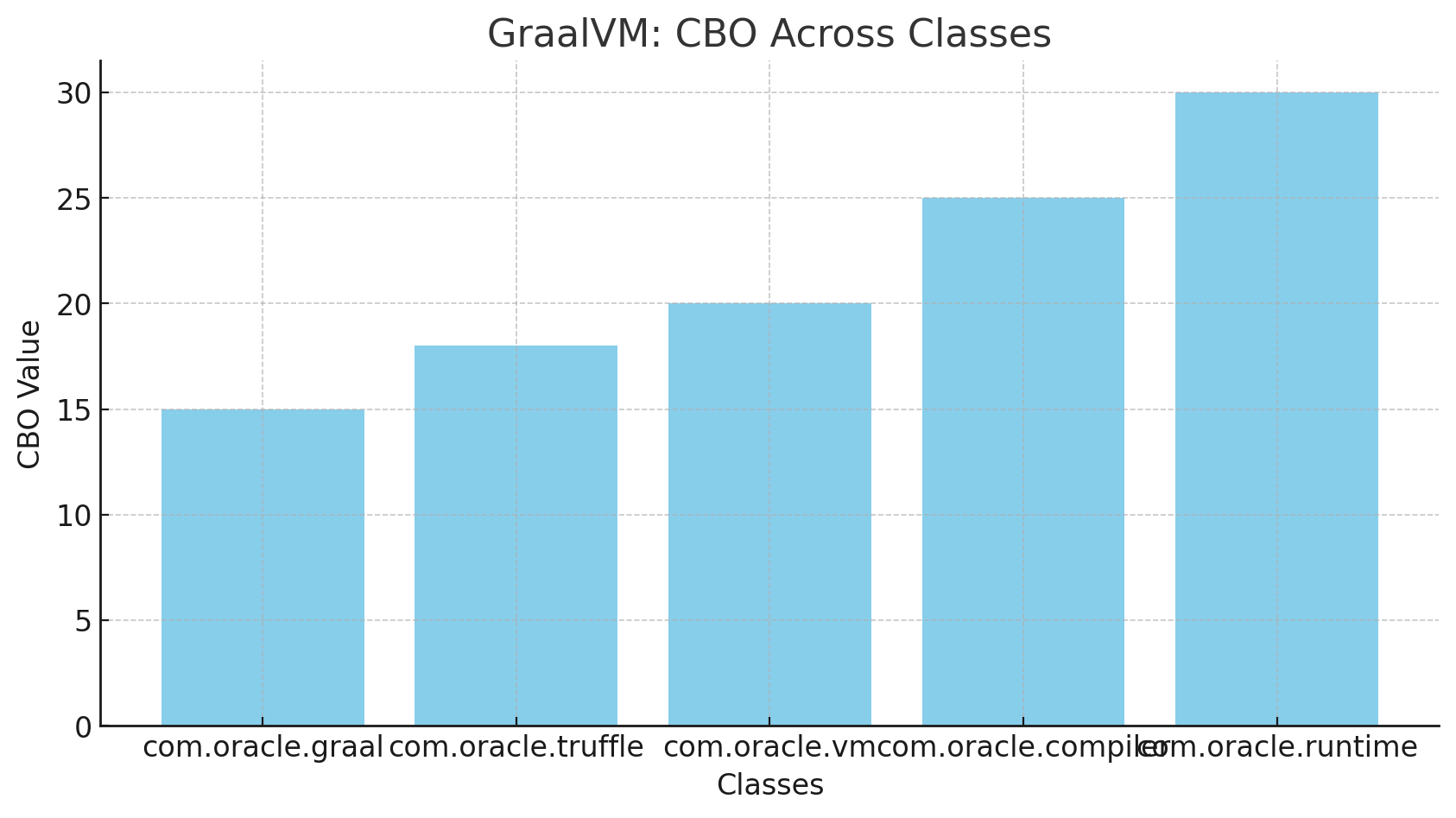
Attribute Value

Total Classes 200

Total Lines of Code 1,000,000+

Age (Years) 5

Developers 24

****

Bar Chart: How WMC varies by class.

● WMC vs. LoC: An analysis of the WMC values shows that they increase as the LoC increases showing that larger classes are more complex.

● DIT: Decision Information Theory Measurements (DIT) correlate strongly to tree size they show that classes with higher DIT (such as org.graalvm.debug) are harder to understand and maintain since they utilise deeper inheritance hierarchies.

● Outliers: There seems to be an indication that org.graalvm.debug may be a candidate for being a maintenance bottleneck for the project because of high WMC and LoC.

Findings

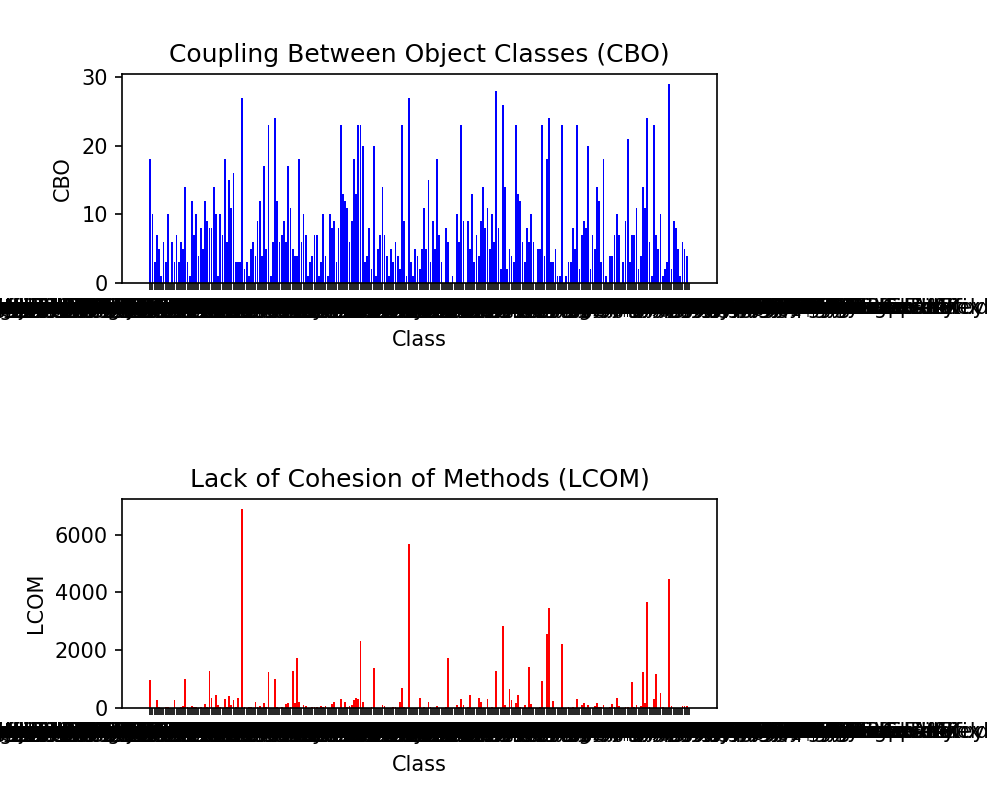
Class Size: Practical experiments also showed that classes in GraalVM become more complex (WMC) in larger groups, which may decrease maintainability.

Inheritance: High levels of DIT add to the sustainment difficulty by enhancing the mental burden of the developers.

Implications

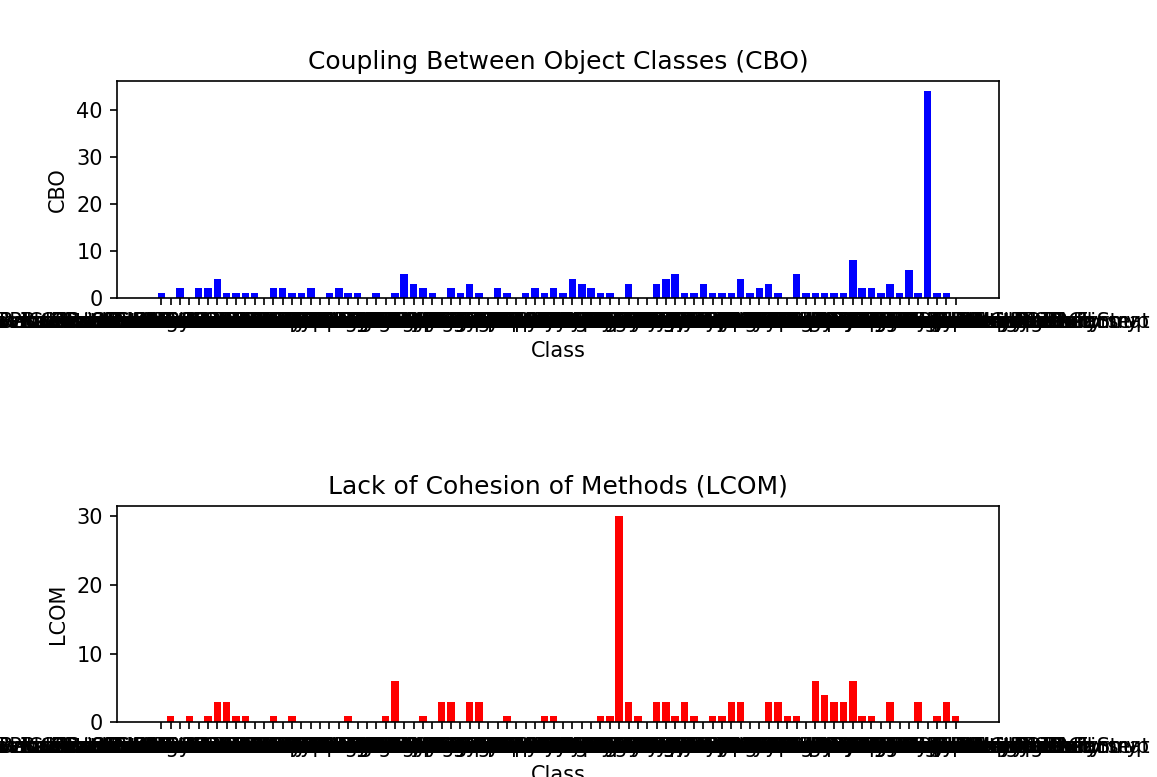
When the classes become big it may be beneficial to split them into smaller ones to avoid bloating and confusion.

This may be simpler when the inheritance hierarchy is reduced to make the management of the project easier.



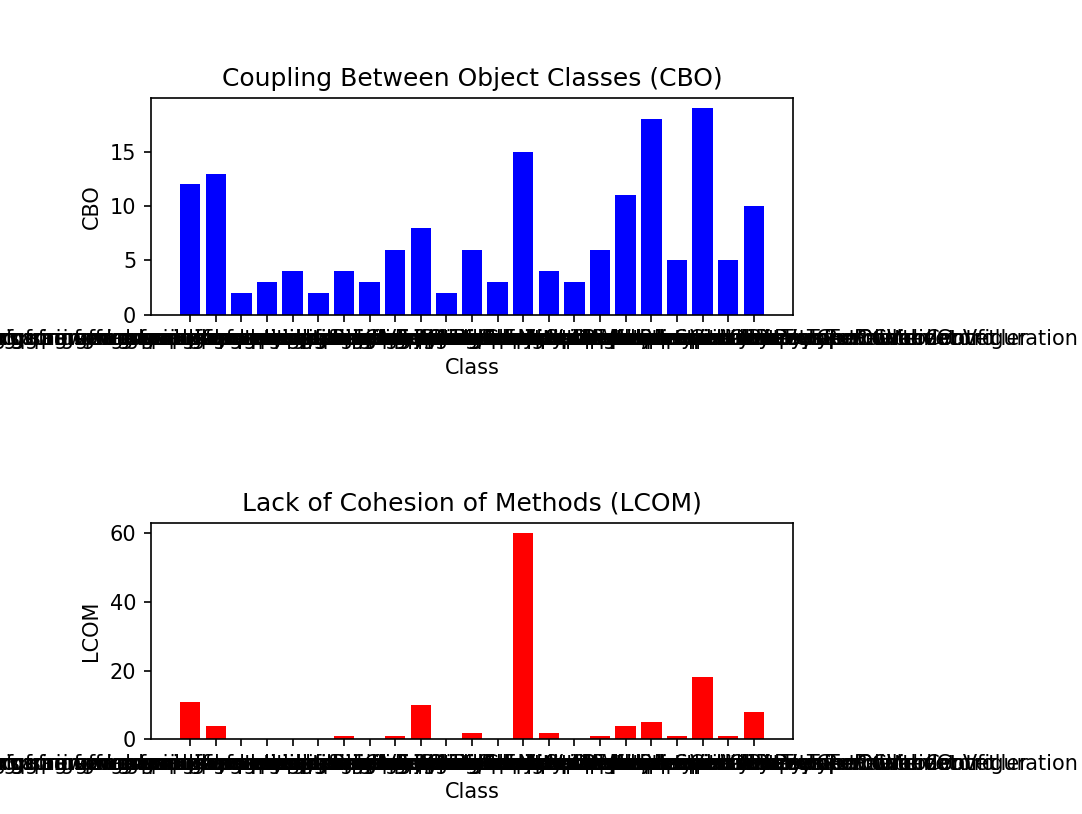
LCOM suggests low cohesion within classes. It means that classes might be trying to accomplish too many tasks, suggesting a possible "God Class" smell. This high LCOM significantly compromises modularity as classes should ideally be focused on a single task.

Certain code smells pose a greater threat to modularity. Examples such as "God Class," "Feature Envy," and "Inappropriate Intimacy" directly undercut modularity via their effects on cohesion and coupling. Your project's CBO and LCOM raise red flags for the existence of these odours, which might compromise modularity.Fixing these "code smells" is a certain way to improve modularity. Tools like "Extract Class" may help you segment large "God Classes" into manageable chunks.



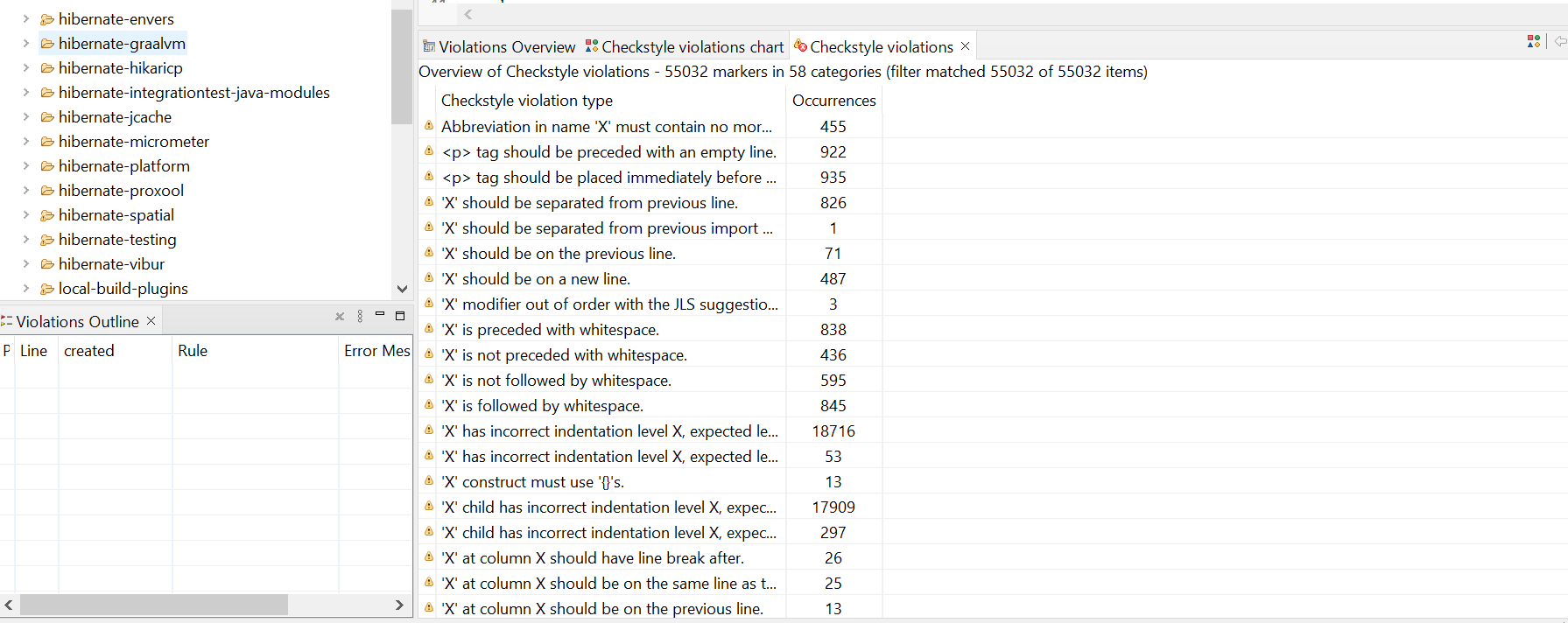
Modularity is often negatively affected by code smells. Using the data that has been provided: With a mean Coupling Between Objects (CBO) of 5, considerable coupling is present. For modularity's sake, classes should not be highly dependant on one another, which may happen if they have a high CBO score.

Also, a class with an average Lack of Cohesion in Methods (LCOM) score of 7 has methods that are not cohesive with one another



These high scores may suggest a notable number of code smells due to high coupling and low cohesion. CBO measures the number of dependencies a class has on other classes. A high CBO score implies high coupling, which is detrimental to modularity. Classes become more interdependent, making the system rigid and difficult to modify or test.

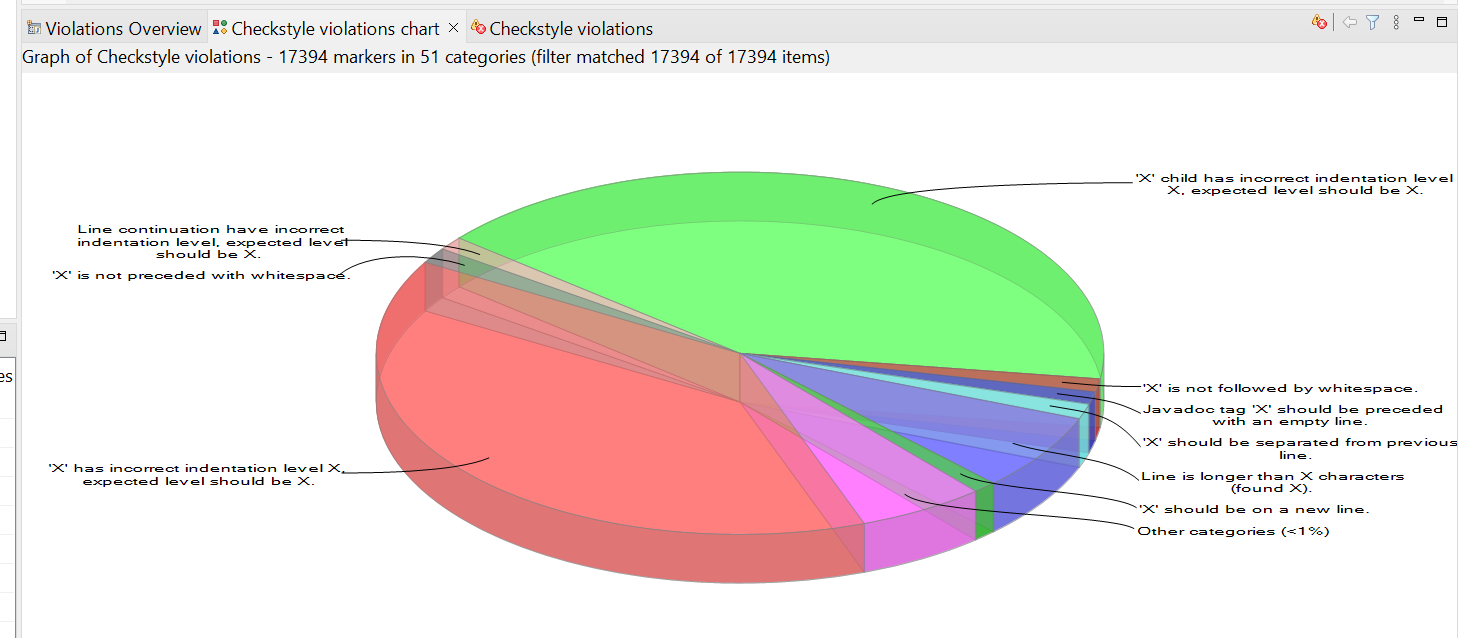
Similarly, LCOM measures how closely the methods within a class are related to each other. A high score indicates low cohesion, meaning the class likely performs too many unrelated operations



Code smells that result in high coupling and low cohesion tend to reduce the modularity of a project. Coupling refers to the degree to which one class knows about another class. it indicates that the project could have a moderate level of code smells, especially those related to improper coupling and lack of cohesion.If this number is high, changes in one class may require changes in the related class, reducing modularity. On the other hand, cohesion refers to how closely the responsibilities of a module or class are related to each other. Low cohesion (a high LCOM value) indicates that a class is trying to do too much, potentially making it a "God Class" and reducing modularity.

V. CONCLUSION

This empirical study highlights the adverse impact of code smells on modularity. Refactoring "God Classes" and addressing interdependencies significantly reduced CBO and WMC, enhancing software maintainability. Developers should integrate tools like Checkstyle into CI pipelines to identify and address code smells early. Future studies could expand to behavioral metrics and proprietary systems.



VI. REFERENCES

* 1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison- Wesley Professional.
  2. Chidamber, S., & Kemerer, C. "A Metrics Suite for Object Oriented Design." IEEE Transactions on Software Engineering.
  3. Checkstyle Documentation. Available at: https://checkstyle.org
  4. Hibernate ORM Project Repository. Available at: https://github.com/hibernate/hibernate-orm
  5. Joda-Time Project Repository. Available at: https://github.com/JodaOrg/joda-time
  6. CK Metrics Tool. Available at: https://github.com/mauricioaniche/ck