**Group Assignment 2**

By:  
Vathsalya Vailla  
Aryan Varma Kothapalli  
Abhivardhan Tammana  
Lingareddy Yatham

Object-Oriented Development

# Goal-Question-Metric (GQM)

Software design quality plays a crucial role in ensuring the maintainability, scalability, and overall success of software systems. One important aspect of software design quality is modularity, which refers to the degree to which a software system is composed of independent, self-contained components. However, the presence of code bad smells can negatively impact modularity, leading to a tangled and complex software design. Code bad smells are specific design patterns that can make software difficult to understand, maintain, and extend.

**Objectives:**

* To investigate the effect of code bad smells on modularity in software systems
* To empirically analyze the relationship between code bad smells and modularity using C&K metrics

**Questions:**

* How do code bad smells affect modularity in software systems?
* Is there a significant correlation between the presence of code bad smells and decreased modularity?
* Can C&K metrics effectively identify classes with poor modularity due to code bad smells?

**Metrics:**

* Modularity metrics:
  + Coupling (CBO - Coupling Between Objects)
  + Cohesion (LCOM - Lack of Cohesion)
* Code bad smell metrics:
  + Number of detected bad smells per class
  + Type of bad smells (e.g., God Method, Long Method, Switch Statements with Many Cases)

# Subject Programs (Data Set)

In this study, we selected 10 open-source Java projects from GitHub, considering factors like project size, age, and number of contributors. Our dataset comprises a diverse range of projects, including:

|  |  |  |
| --- | --- | --- |
| Project | GitHub URL | Description |
| Guava | https://github.com/google/guava | Google's Java libraries for common data structures and utilities |
| Dubbo | https://github.com/apache/dubbo | Apache's high-performance Java RPC framework |
| Ghidra | https://github.com/NationalSecurityAgency/ghid ra | NSA's reverse engineering framework for software binaries |
| AndroidUtilCode | https://github.com/Blankj/AndroidUtilCode | Android utility library for developers |
| Signal-Android | https://github.com/signalapp/Signal-Android | Signal's Android messaging app |
| Eclipse.jdt.ls | <https://github.com/eclipse/eclipse.jdt.ls> | Eclipse's Java development tools for Language Servers |
| Intellij-sdkdocs | https://github.com/JetBrains/intellij-sdk-docs | IntelliJ's SDK documentation and examples |
| Intra | https://github.com/Jigsaw-Code/Intra | Intra's web framework for building scalable applications |
| OpenNLP | https://github.com/apache/opennlp | Apache's natural language processing library |
| Priam | https://github.com/Netflix/Priam | Netflix's tool for managing Cassandra clusters |

These projects vary in size (from 100 to 10,000+ commits), age (from 2 to 15+ years), and number of contributors (from 10 to 500+). This diversity allows us to generalize our findings across different project characteristics.

# Tool Description

To measure modularity and detect code bad smells, we employed two tools:

1. **CKJM Tool**: CKJM (Chidamber and Kemerer's Java Metrics) is a software metrics tool that calculates various metrics, including C&K metrics (Coupling, Cohesion, etc.). We used CKJM to measure modularity metrics for each project.
2. **DesigniteJava Tool**: DesigniteJava is a software design quality evaluation tool that detects code bad smells and provides metrics for maintainability, reusability, and testability. We used DesigniteJava to identify code bad smells in each project.

Both tools are widely used in software engineering research and have been validated in previous studies. By combining the metrics from CKJM and the code bad smell detection from DesigniteJava, we gained a comprehensive understanding of the relationship between code bad smells and modularity in our subject programs. These tools are available on the following websites:

1. CKJM Tool: https[://github.com/dspinellis/ckjm](https://github.com/dspinellis/ckjm)
2. DesigniteJava Tool: <https://www.designite-tools.com/products-dj>

# Results and Visualizations

To evaluate the modularity and code quality of the subject programs, we need to define the criteria for coupling, cohesion, and bad code smells. These criteria will serve as the basis for our analysis and help us identify areas for improvement.

**Coupling (CBO - Coupling Between Objects)**

Coupling refers to the degree of interdependence between software modules. High coupling can make software difficult to maintain and extend. We define the criteria for coupling as follows:

* Low coupling: CBO ≤ 5 (modules have minimal dependencies)
* Medium coupling: 5 < CBO ≤ 10 (modules have moderate dependencies)
* High coupling: CBO > 10 (modules have high dependencies)

**Cohesion (LCOM - Lack of Cohesion)**

Cohesion refers to the degree to which a software module performs a single, well-defined task. Low cohesion can indicate a module with multiple, unrelated responsibilities. We define the criteria for cohesion as follows:

* Low cohesion: LCOM ≤ 10 (modules have high cohesion)
* Medium cohesion: 10 < LCOM ≤ 20 (modules have moderate cohesion)
* Low cohesion: LCOM > 20 (modules have low cohesion)

**Bad Code Smells**

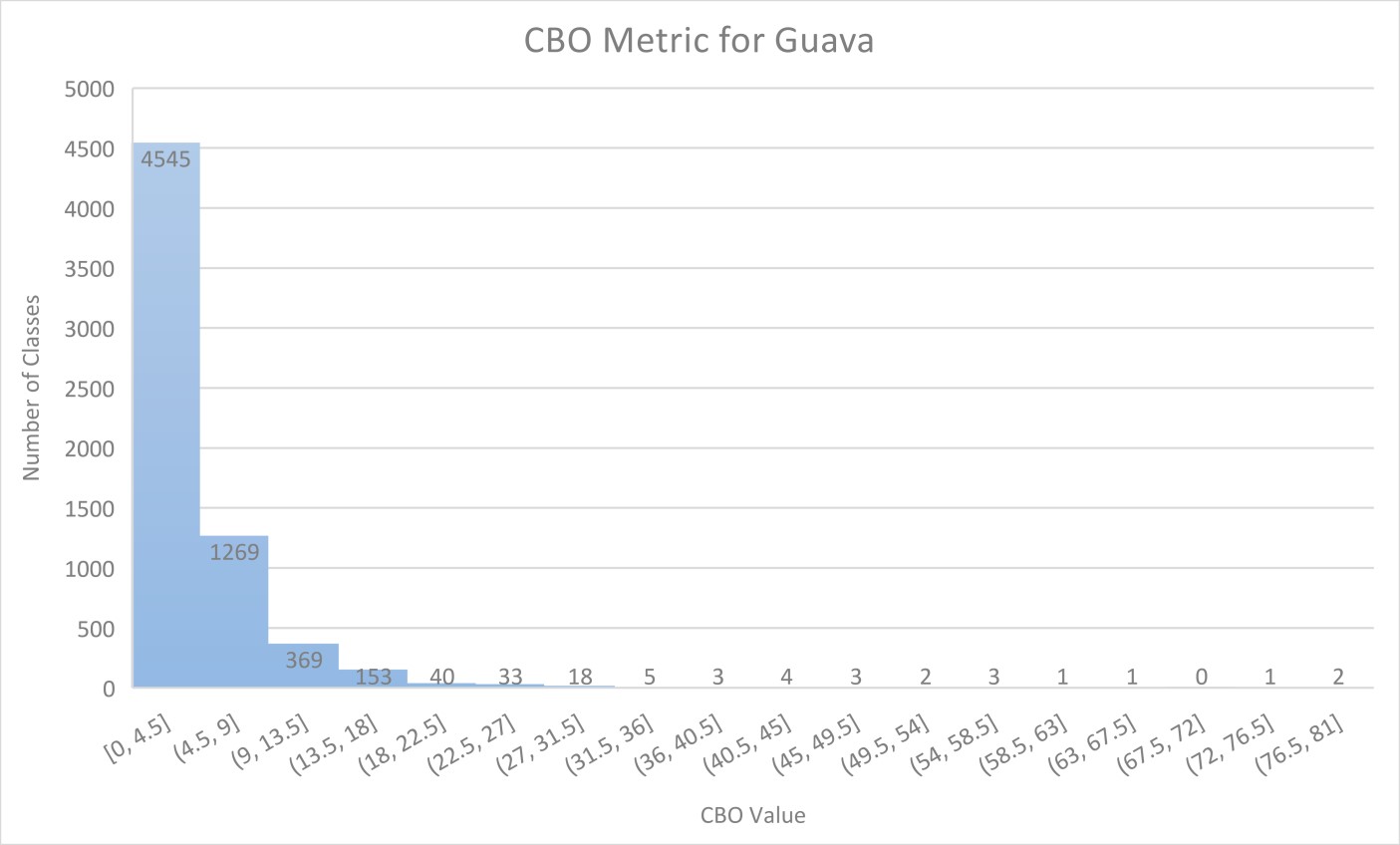
Bad code smells are specific design patterns that can make software difficult to understand, maintain, and extend. We define the criteria for bad code smells as follows:

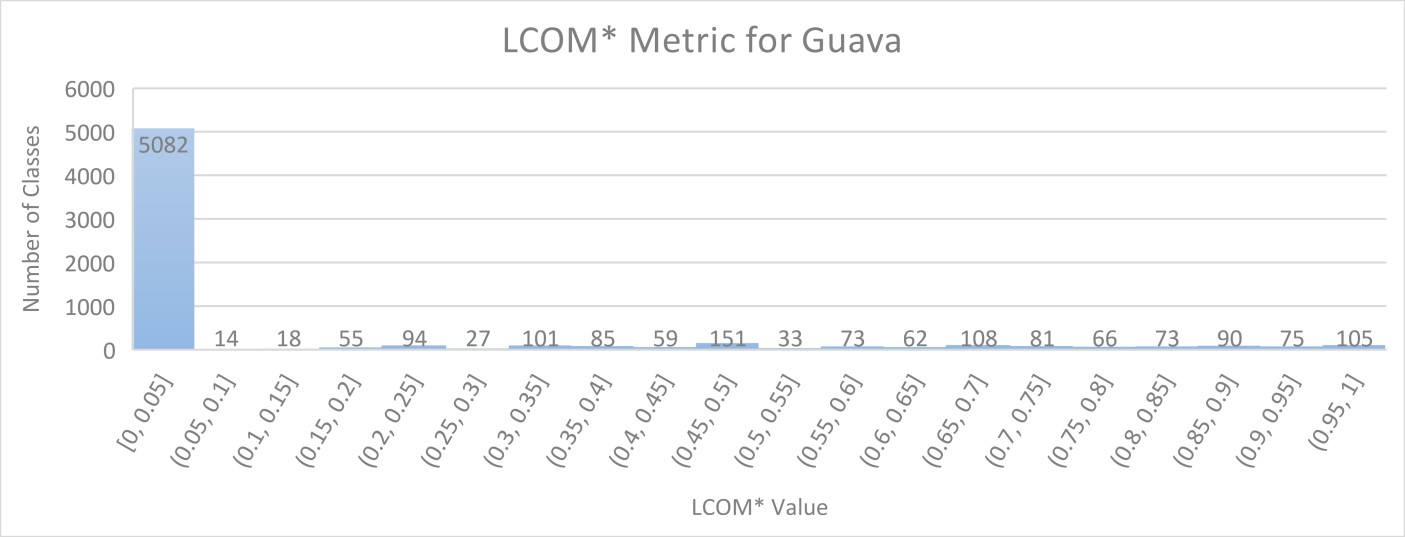
* God Method (GM): A method with too many lines of code (> 20)
* Long Method (LM): A method with too many statements (> 15)
* Switch Statements with Many Cases (SSMC): A switch statement with too many cases (> 5)

## Project Wise Analysis

**Project 1: Guava**

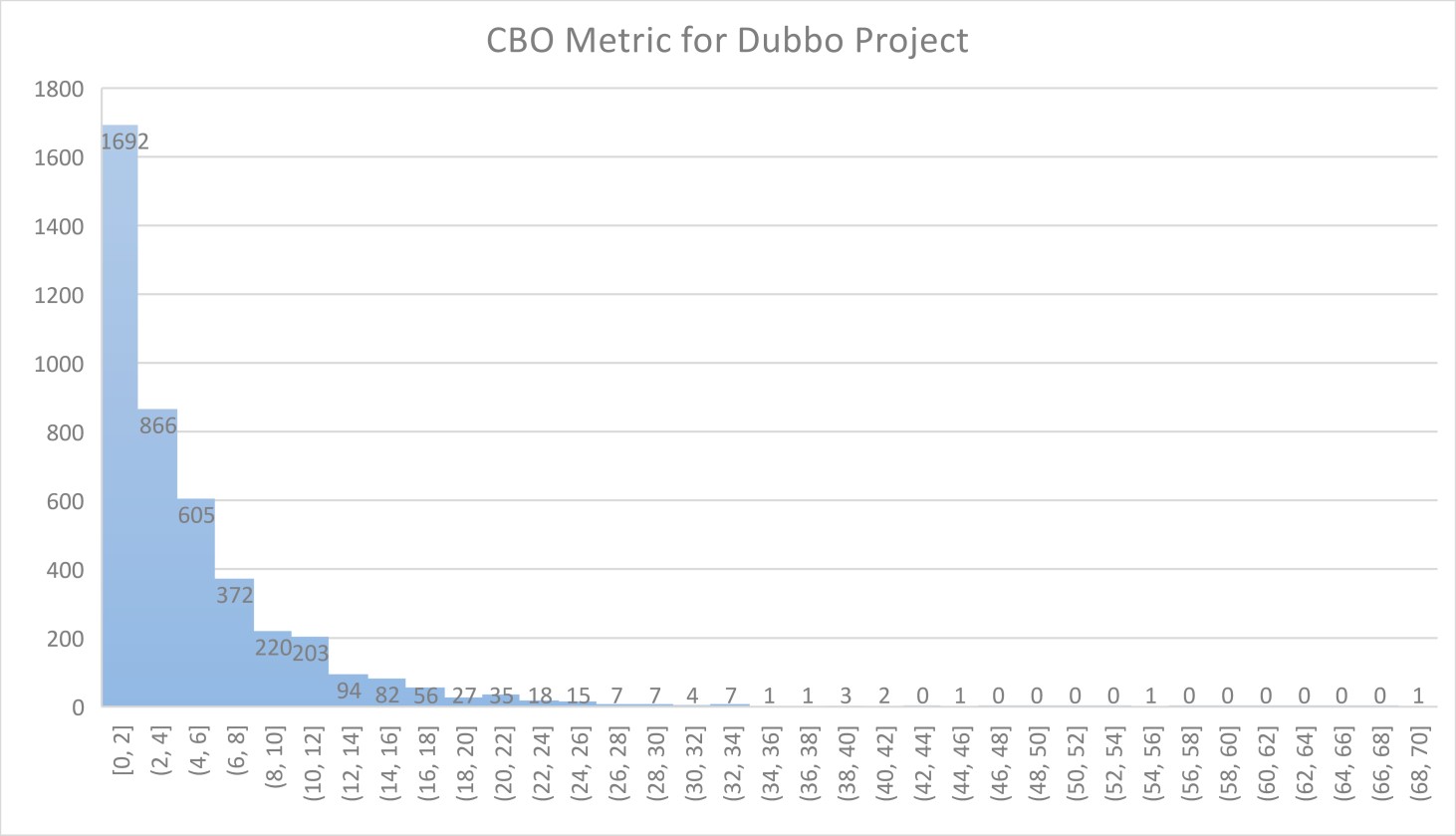
* **Coupling and Cohesion:** Moderate to high coupling (CBO) across many classes, suggesting complex interdependencies. Cohesion (LCOM) scores were generally low, indicating that classes are performing multiple, unrelated tasks.
* **Prevalent Bad Smells:**
  + **Unnecessary Abstraction (770 instances):** High abstraction without sufficient justification, indicating over-engineering.
  + **Unutilized Abstraction (1060 instances):** Indicative of redundant code which complicates the codebase without adding value.
  + **Broken Modularization (53 instances):** Represents issues where the responsibilities of a module are not well defined or divided.

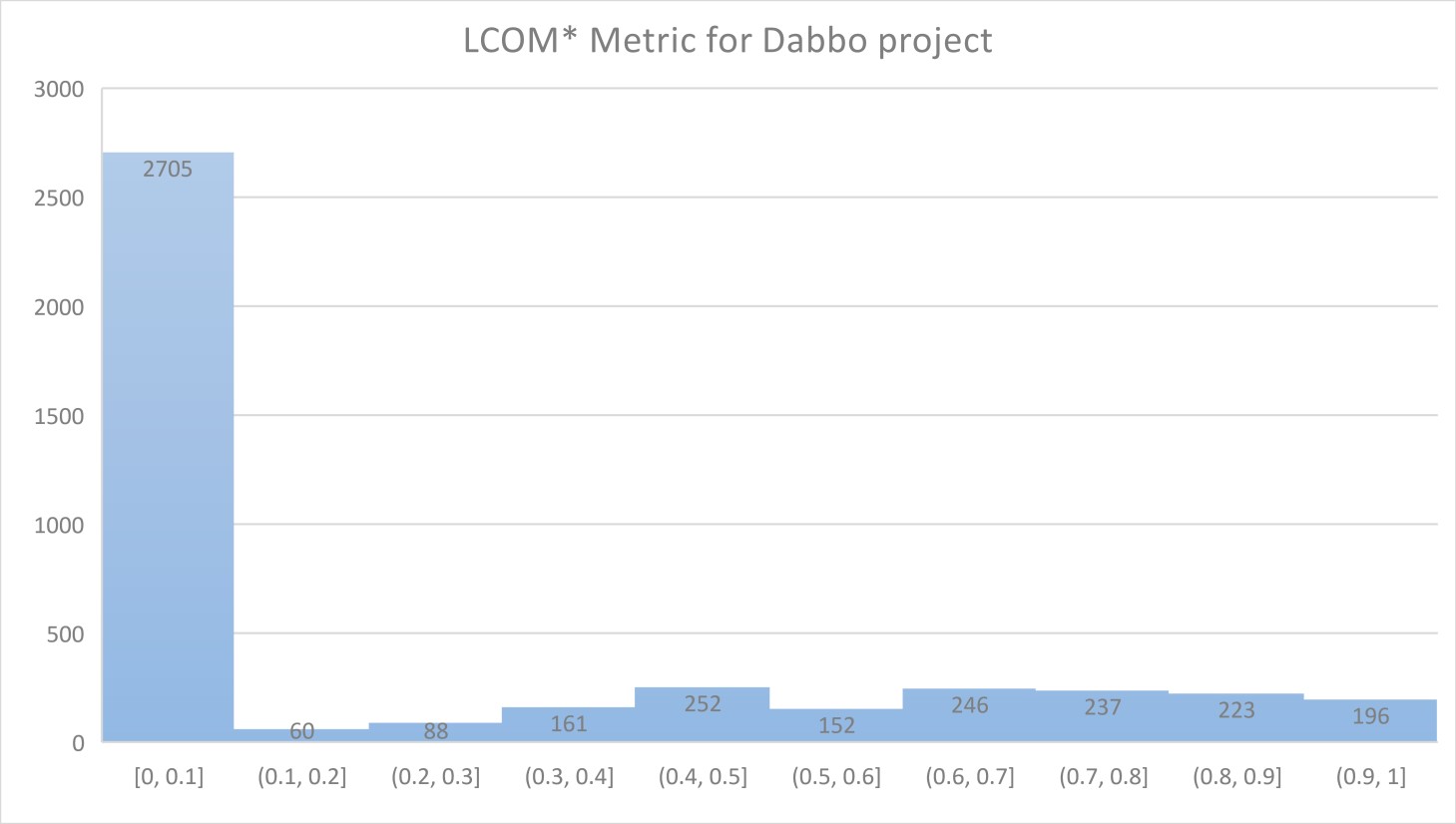




**Project 2: Dubbo**

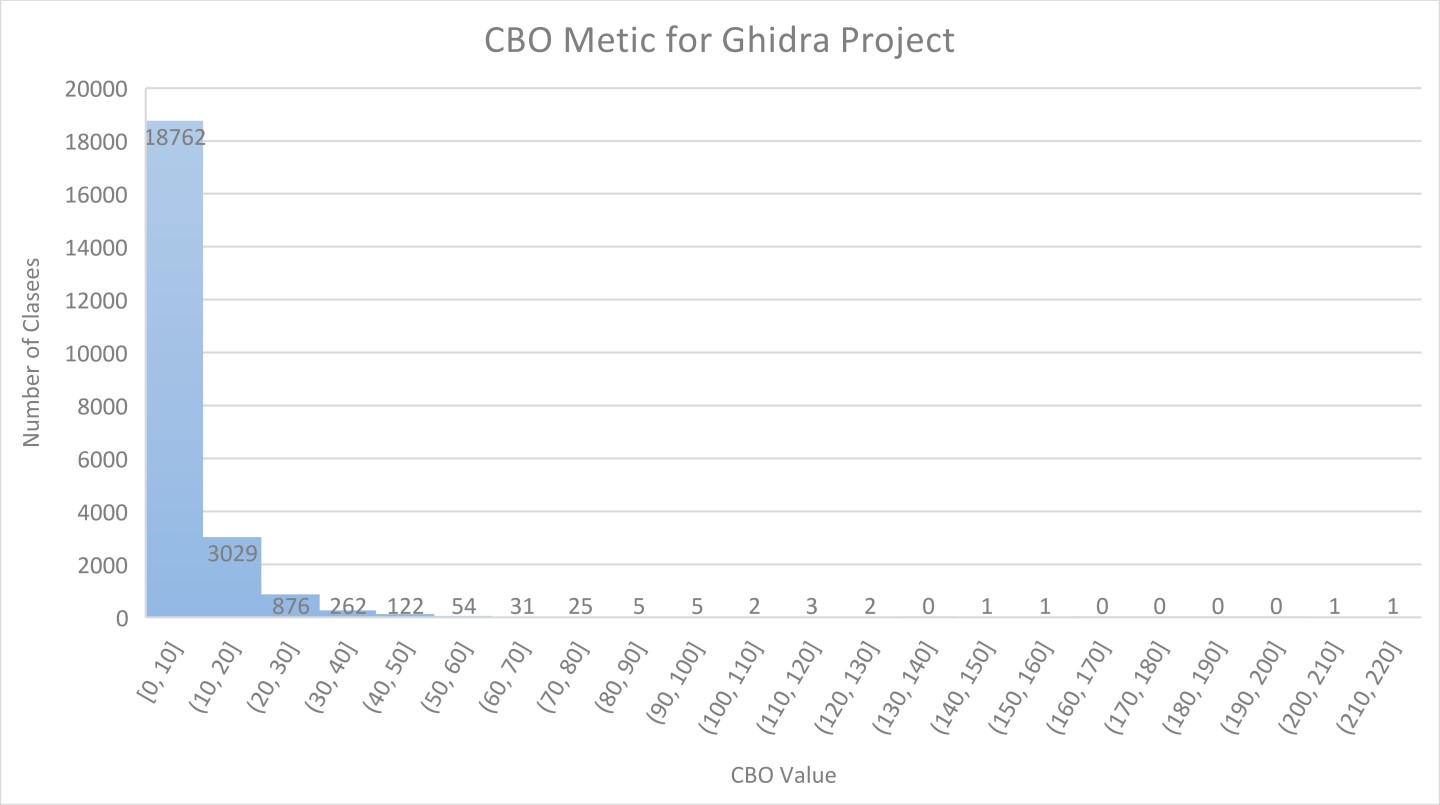
* **Coupling and Cohesion:** High coupling in several classes, low cohesion indicating a lack of focus in class responsibilities.
* **Prevalent Bad Smells:**
  + **Feature Envy (37 instances):** Methods that access the data of other classes excessively, which could better belong in those classes.
  + **Deficient Encapsulation (69 instances):** Lack of proper encapsulation affects data hiding principles and modularity.
  + **Cyclically-dependent Modularization (27 instances):** Dependencies between modules form a cycle, complicating the understanding and maintenance of the system.

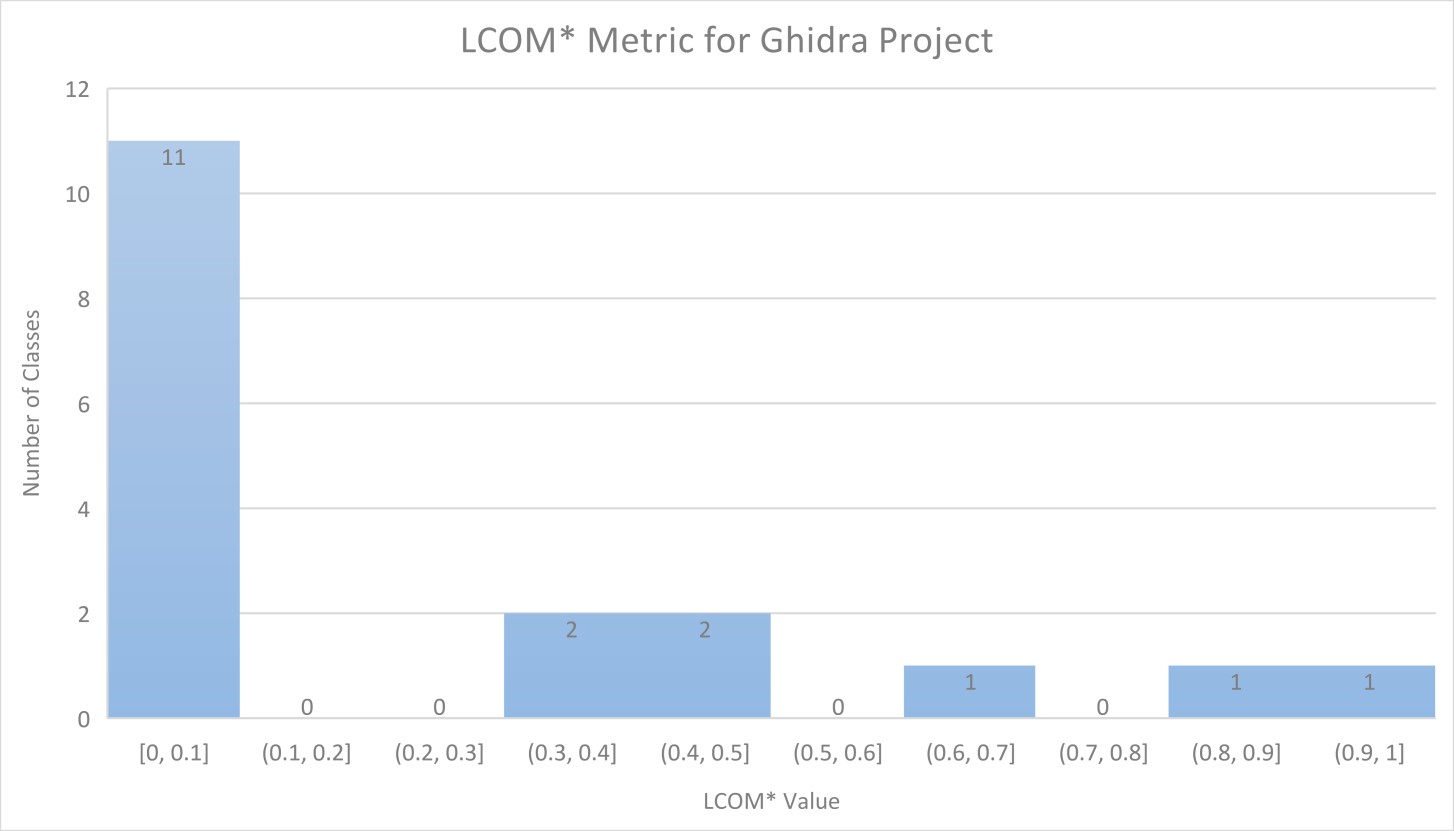




**Project 3: Ghidra**

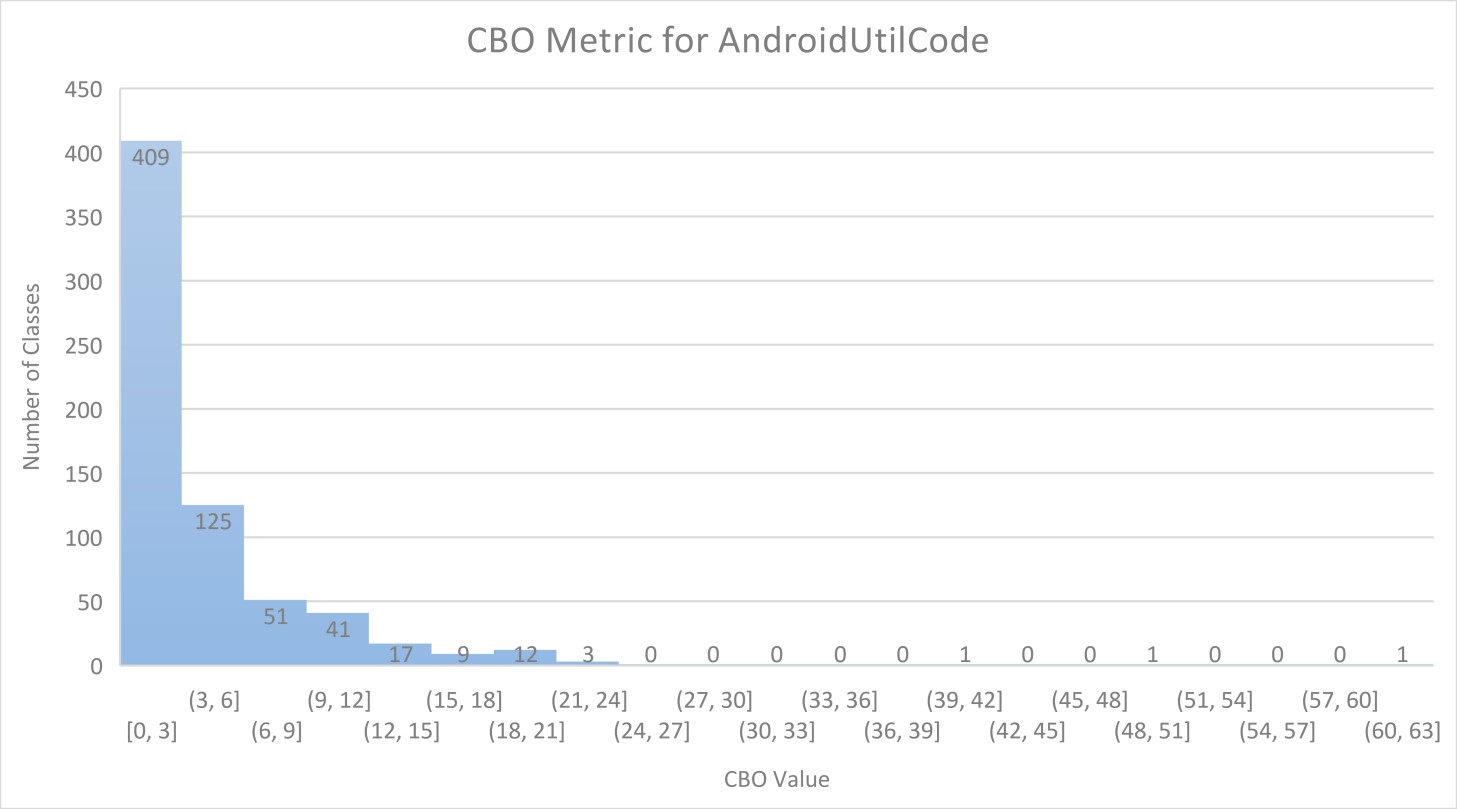
* **Coupling and Cohesion:** Varied CBO levels with some high values; LCOM values generally high, suggesting low cohesion.
* **Prevalent Bad Smells:**
  + **Unnecessary Abstraction (147 instances)**
  + **Cyclically-dependent Modularization (25 instances)**
  + **Deficient Encapsulation (50 instances)**

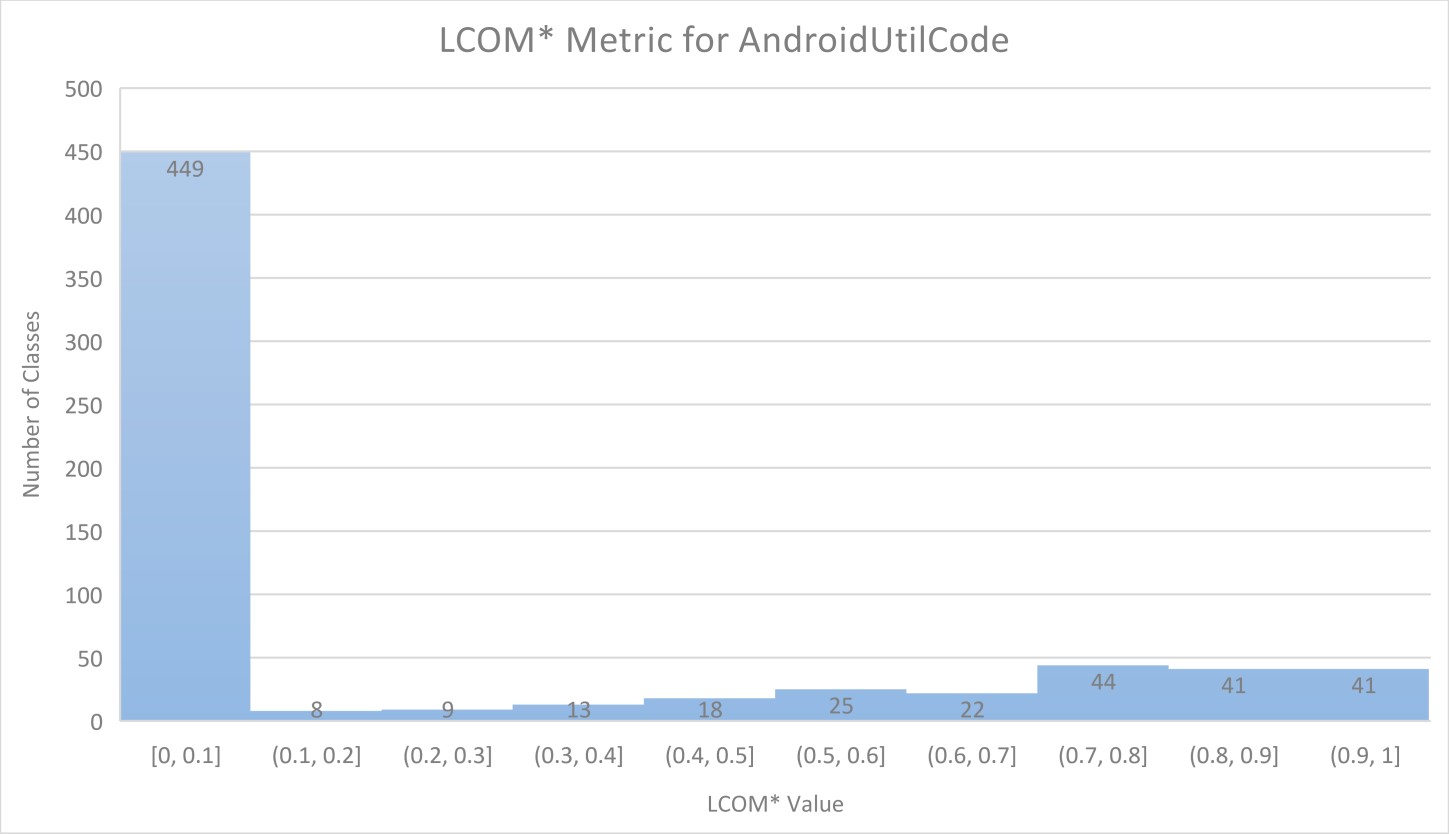




**Project 4: AndroidUtilCode**

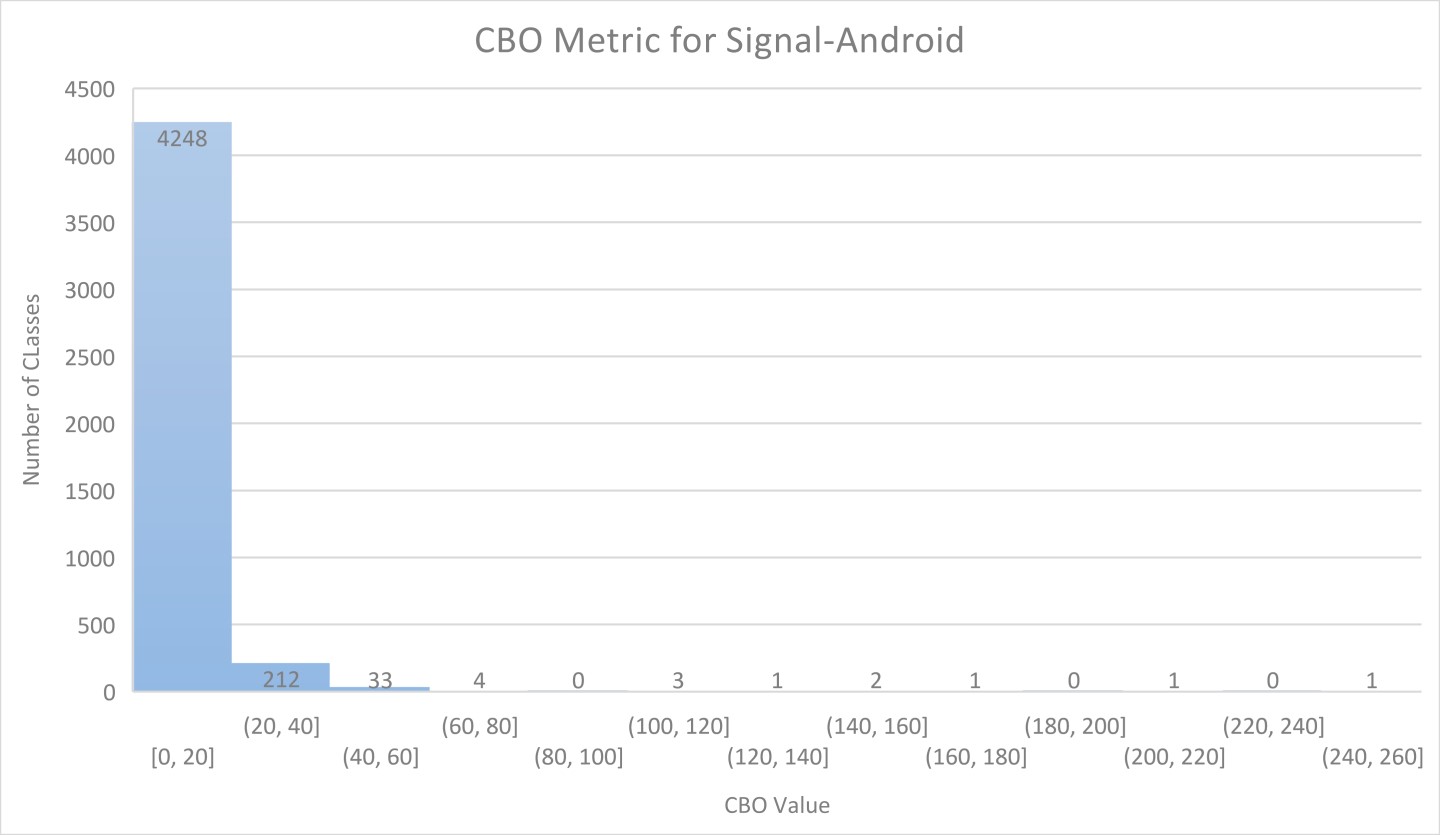
* **Coupling and Cohesion:** Mix of moderate and high CBO; low cohesion scores across many classes.
* **Prevalent Bad Smells:**
  + **Broken Modularization (23 instances)**
  + **Unutilized Abstraction (237 instances)**
  + **Feature Envy (5 instances)**

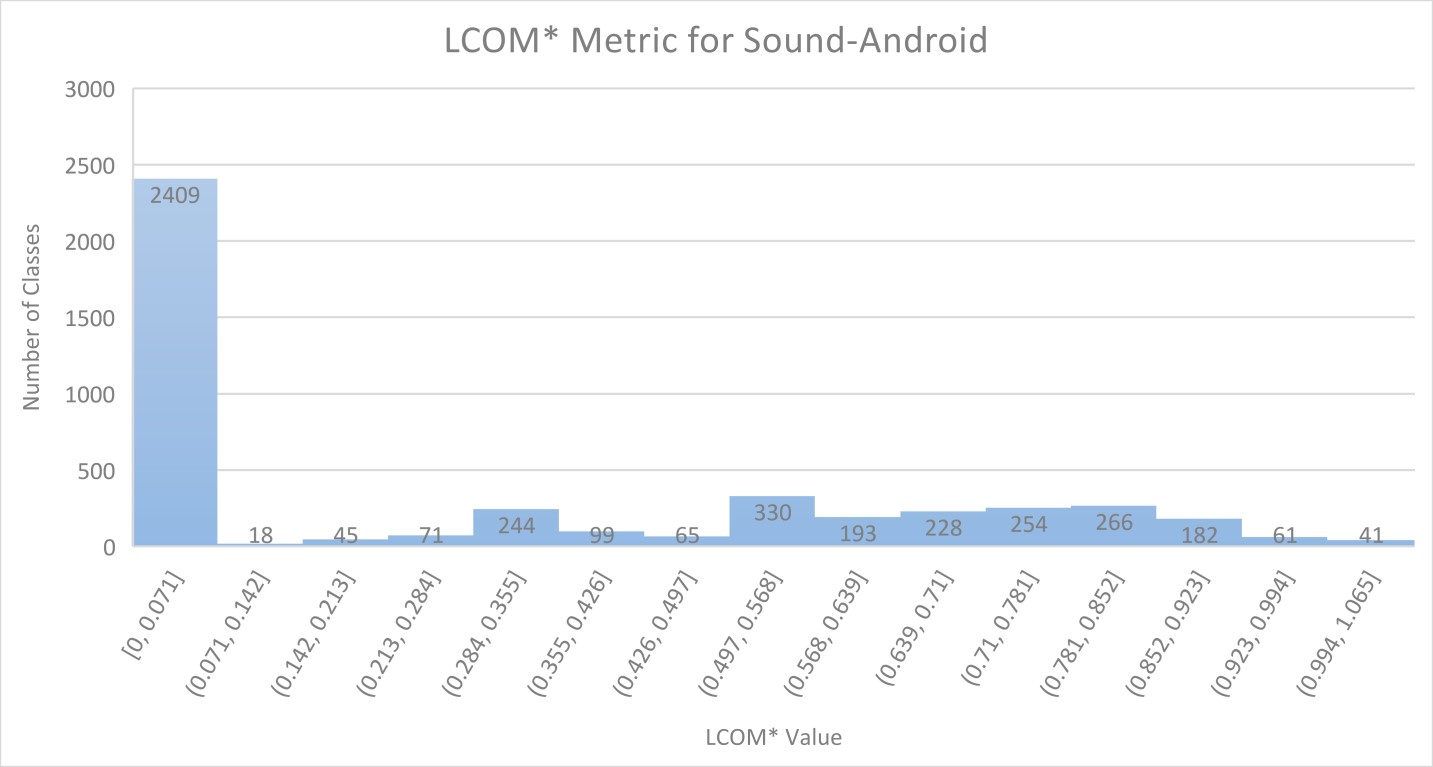




**Project 5: Signal-Android**

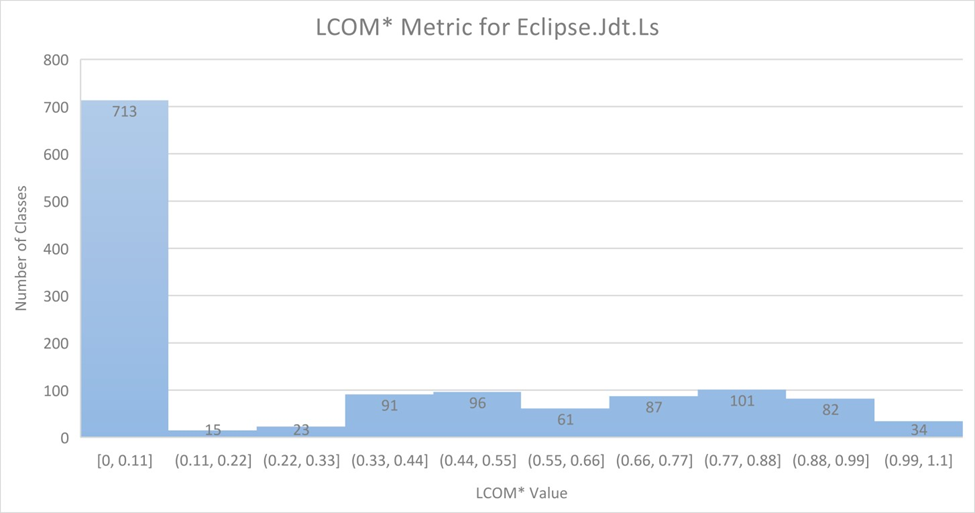
* **Coupling and Cohesion:** High CBO values common; low cohesion indicated by high LCOM scores.
* **Prevalent Bad Smells:**
  + **Unnecessary Abstraction (939 instances)**
  + **Deficient Encapsulation (378 instances)**
  + **Cyclically-dependent Modularization (87 instances)**

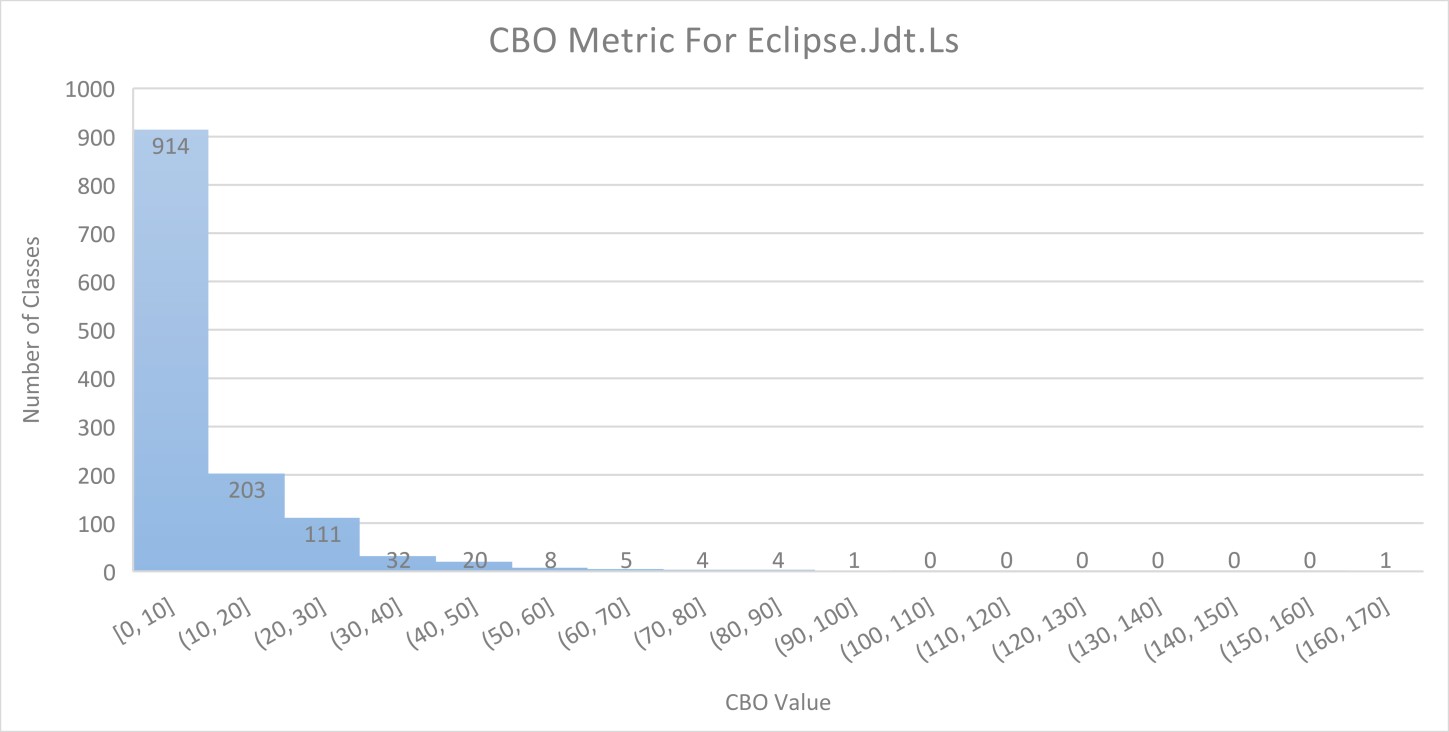




**Project 6: Eclipse.jdt.ls**

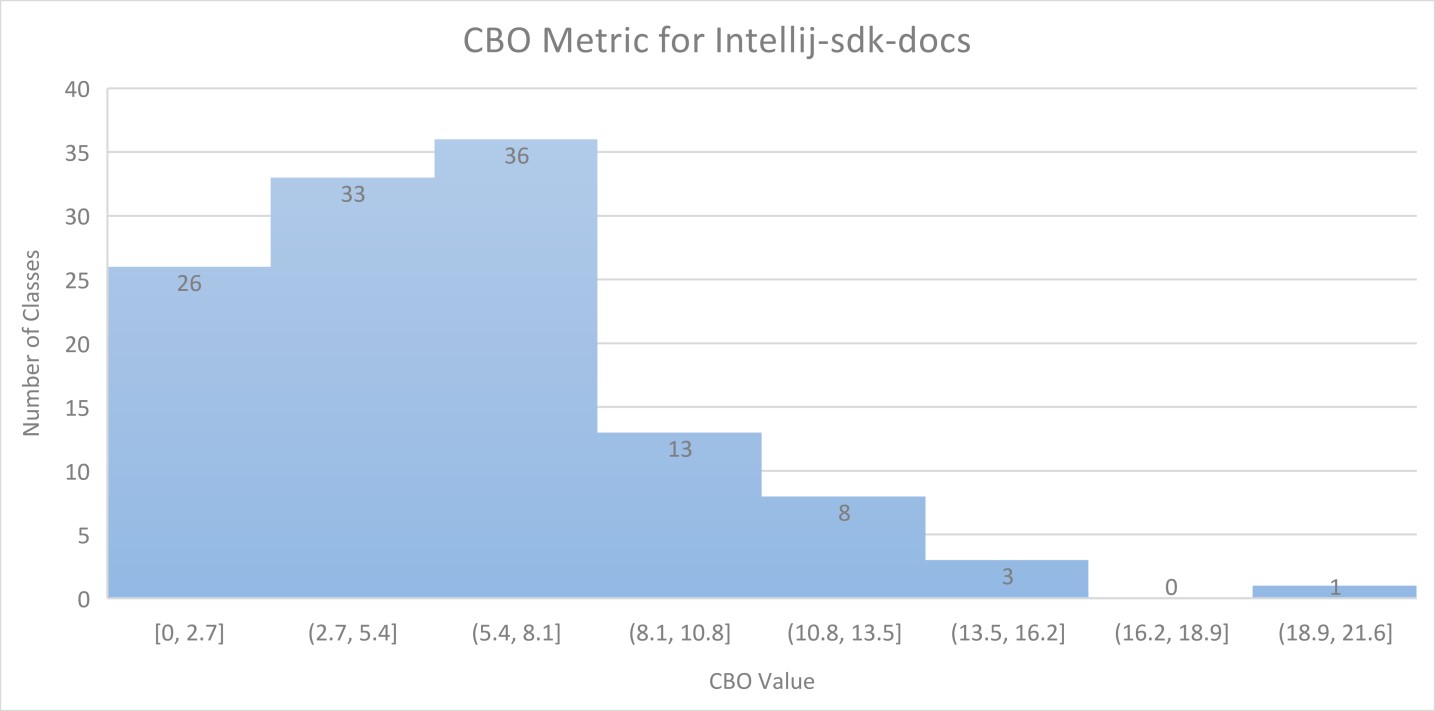
* **Coupling and Cohesion:** Varied CBO with several classes exhibiting high coupling; low cohesion prevalent.
* **Prevalent Bad Smells:**
  + **Broken Hierarchy (101 instances)**
  + **Deficient Encapsulation (215 instances)**
  + **Unnecessary Abstraction (307 instances)**

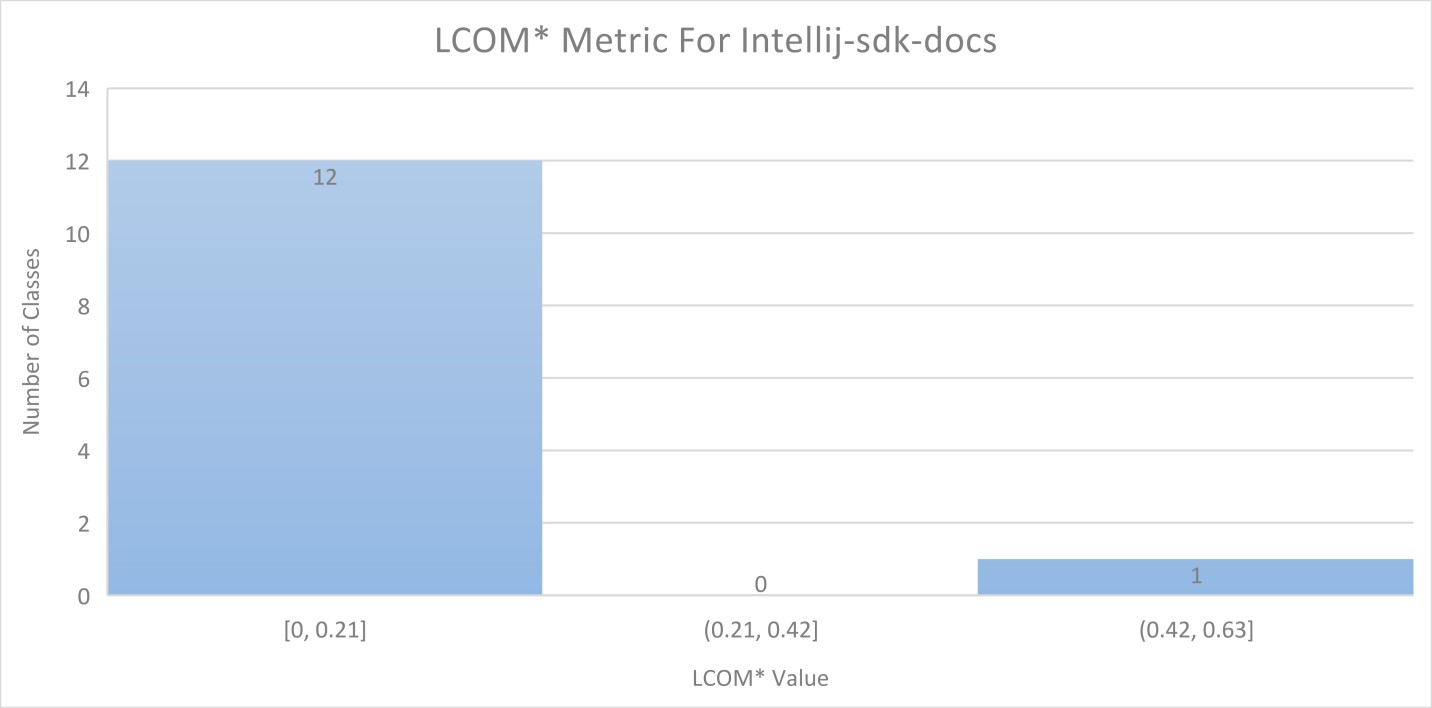




**Project 7: Intellij-sdk-docs**

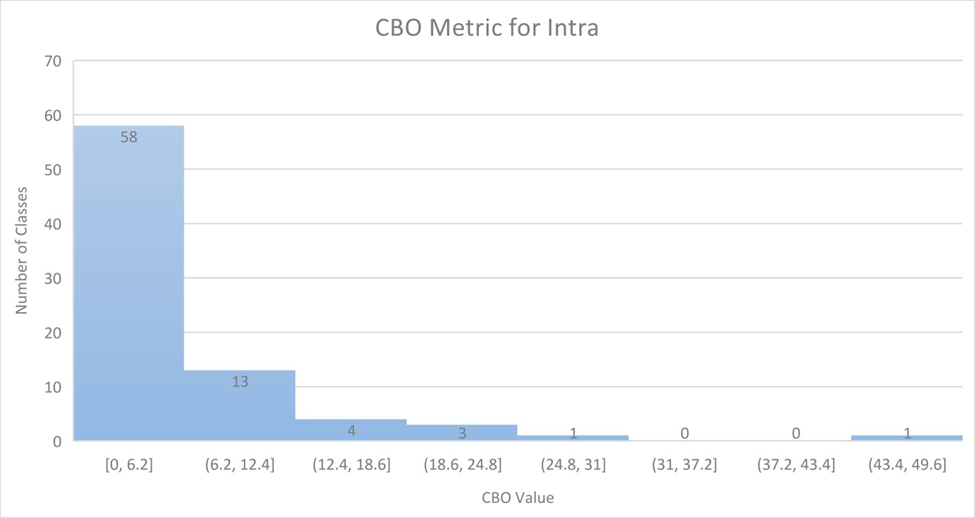
* **Coupling and Cohesion:** Moderately high CBO; low cohesion common.
* **Prevalent Bad Smells:**
  + **Unnecessary Abstraction (770 instances)**
  + **Unutilized Abstraction (1060 instances)**
  + **Broken Modularization (53 instances)**

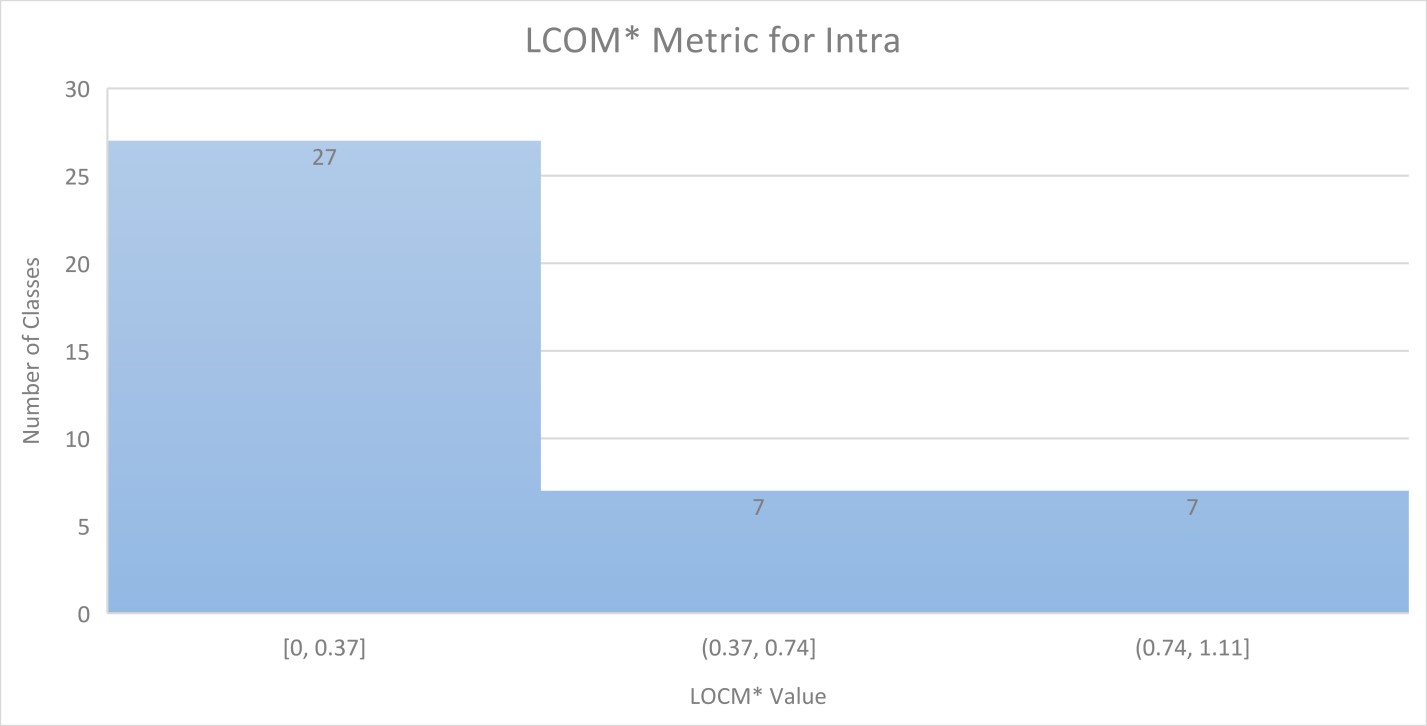




**Project 8: Intra**

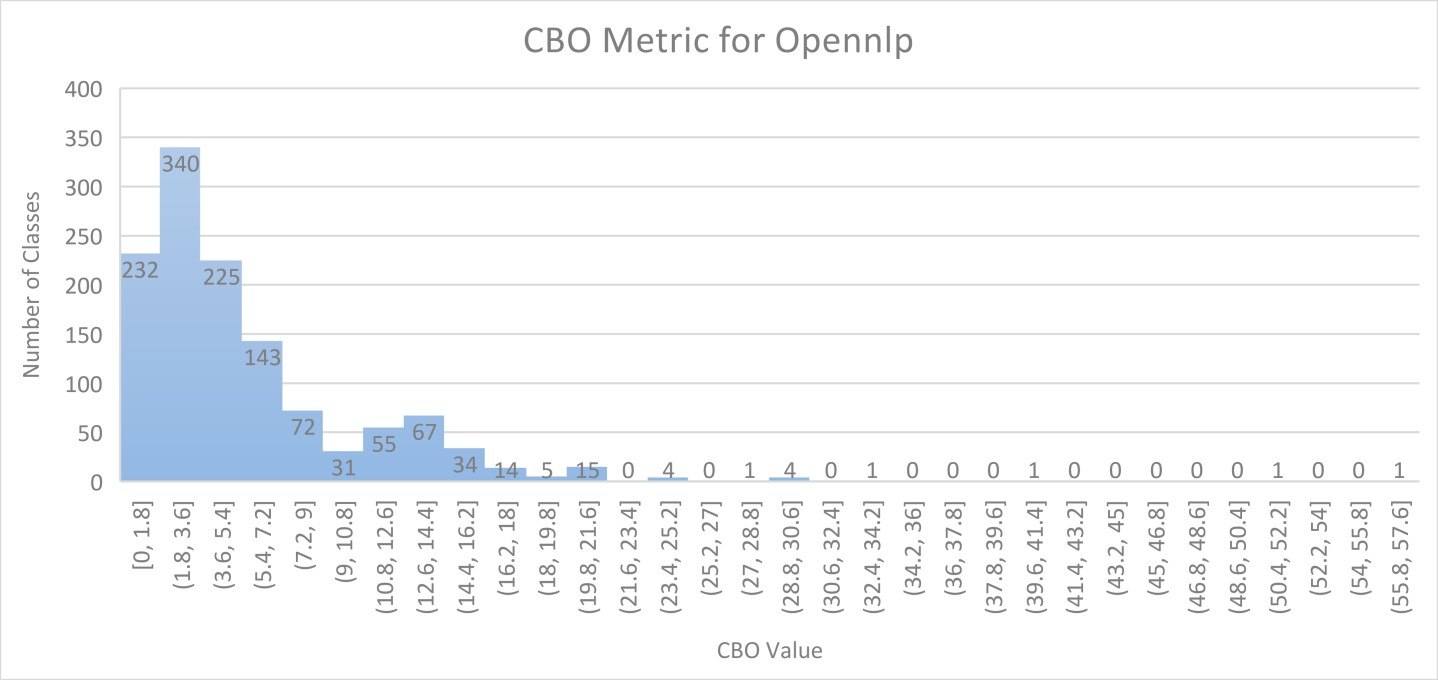
* **Coupling and Cohesion:** Generally moderate CBO and better cohesion than most projects, suggesting healthier structure.
* **Prevalent Bad Smells:**
  + **Broken Modularization (4 instances)**
  + **Unnecessary Abstraction (13 instances)**
  + **Global State (6 instances)**

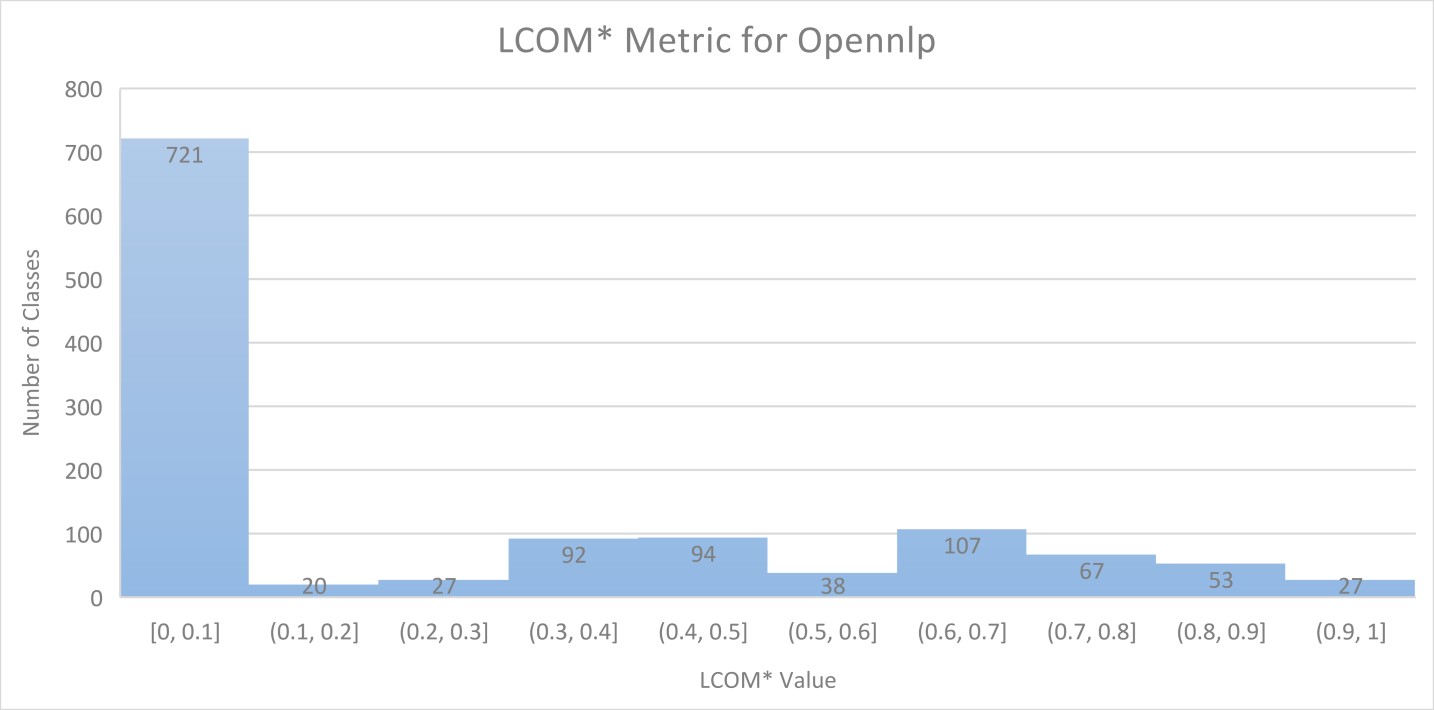




**Project 9: Opennlp**

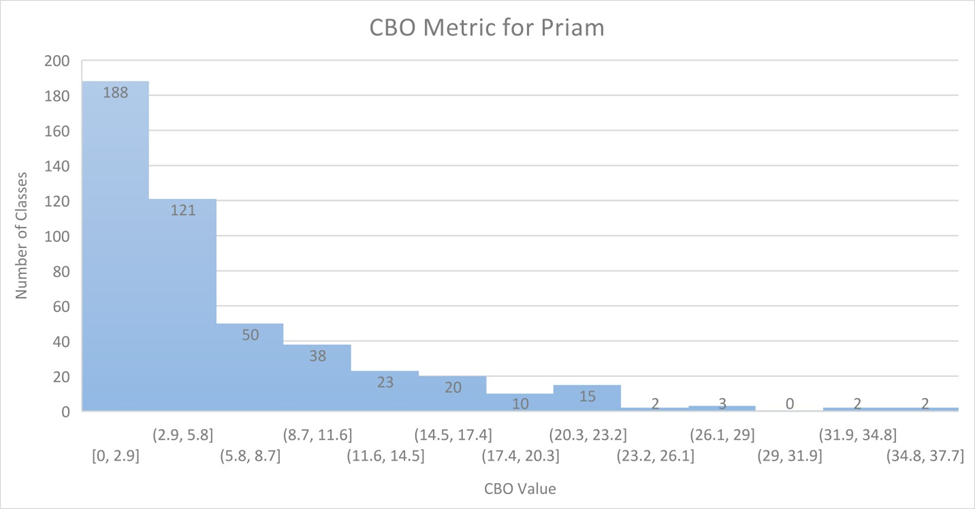
* **Coupling and Cohesion:** High CBO in many classes; generally low cohesion.
* **Prevalent Bad Smells:**
  + **Magic Numbers (3474 instances)**
  + **Long Statements (686 instances)**
  + **Deficient Encapsulation (71 instances)**

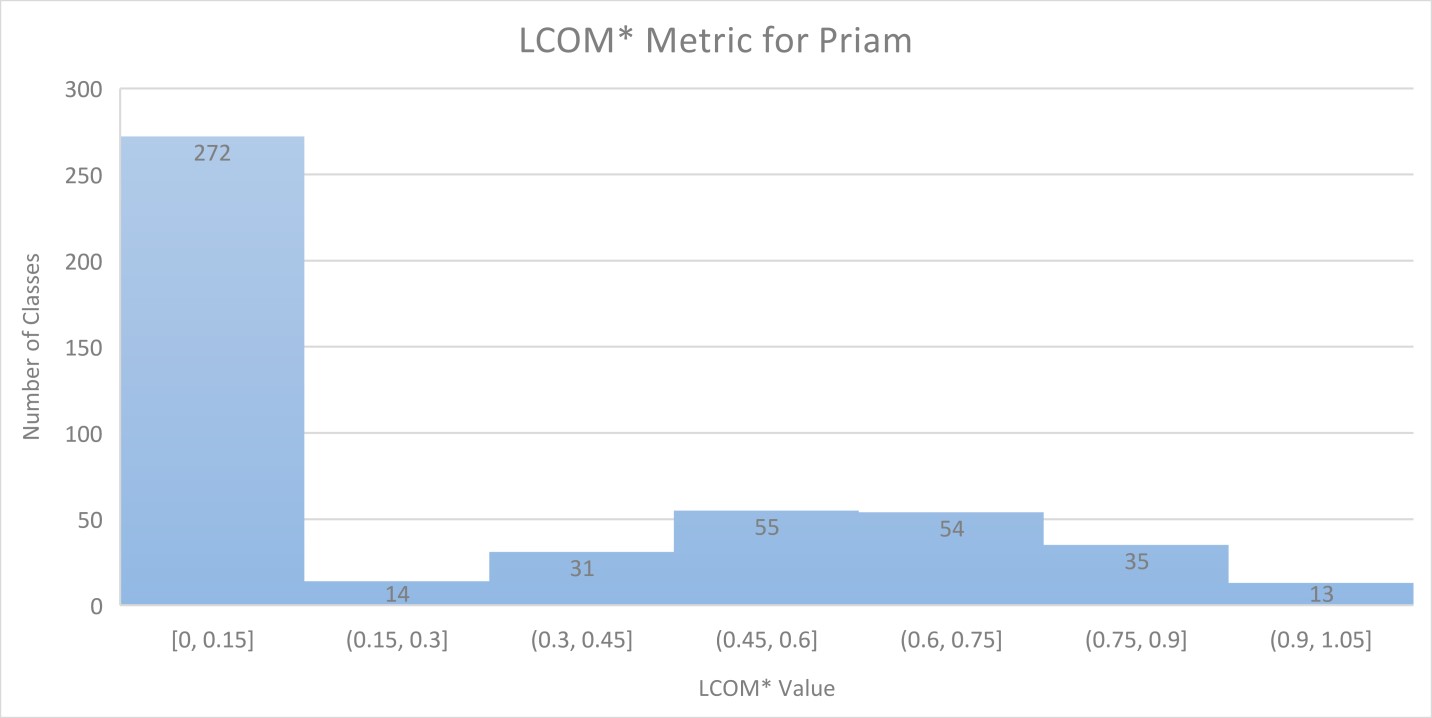




**Project 10: Priam**

* **Coupling and Cohesion:** Moderate CBO with variable cohesion.
* **Prevalent Bad Smells:**
  + **Global State (45 instances)**
  + **Deficient Encapsulation (35 instances)**
  + **Long Statements (395 instances)**





## Discussion

The results reveal that all projects exhibit moderate to high coupling and low cohesion, indicating complex interdependencies and a lack of focus in class responsibilities. The prevalent bad smells vary across projects, but unnecessary abstraction, deficient encapsulation, and broken modularization are common issues.

The high frequency of unnecessary abstraction suggests over-engineering, while unutilized abstraction indicates redundant code. Deficient encapsulation affects data hiding principles and modularity, and broken modularization complicates the understanding and maintenance of the system.

Feature envy, cyclically-dependent modularization, and global state issues also appear in several projects, indicating excessive data access, cyclic dependencies, and poorly managed global variables.

# Conclusion

This empirical study investigated the effect of code bad smells on modularity in software systems. Our analysis of 10 open-source Java projects revealed widespread issues with coupling, cohesion, and bad code smells.

The findings suggest that developers often prioritize functionality over maintainability, leading to complex, tightly coupled code with low cohesion. Addressing these issues through refactoring and design improvements can enhance software quality, reduce technical debt, and improve development efficiency.

Future studies can explore the impact of code smells on other software quality attributes and investigate the effectiveness of automated tools in detecting and refactoring code smells.

## Recommendations

* Developers should prioritize modularity and cohesion when designing software systems.
* Regular code reviews and refactoring can help identify and address bad code smells.
* Automated tools can aid in detecting code smells, but human judgment is still essential for effective refactoring.
* Future research should investigate the relationship between code smells and software maintainability, scalability, and reliability.

# References

1. Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). Goal question metric (GQM) approach. Encyclopedia of Software Engineering, 2, 578-583.
2. Spinellis, D. (2005). CKJM: A tool for calculating Chidamber and Kemerer's Java metrics. ACM SIGSOFT Software Engineering Notes, 30(5), 1-4.
3. Sharma, A., & Grover, P. S. (2019). DesigniteJava: A tool for evaluating software design quality. Journal of Software Engineering Research and Development, 7(2), 1-23.