

ASSIGNMENT 2 REPORT

EFFECT OF CODE BAD SMELLS ON MODULARITY



**Group Members:**

1. Siva Pradeep Reddy Bijjam
2. Leelasrinivasraju Sarikonda

# Section 1

## Research Objective

The primary objective of this study is to conduct an empirical analysis to understand the impact of code bad smells on the modularity of Java projects. We aim to utilize Chidamber and Kemerer (C&K) metrics to measure modularity and assess how code bad smells affect these metrics. Specifically, our goals include:

1. **Criteria Identification**: Establish criteria for selecting Java projects for analysis, considering factors such as project size, age, and developer count.
2. **Project Selection**: Choose a minimum of ten Java projects from GitHub that align with our defined criteria.
3. **Metric Analysis**: Obtain C&K Metrics for each class in the selected Java projects.
4. **Trend Identification**: Analyze these metrics to identify trends or patterns that reveal the effects of bad smells on modularity. This will include a comparison between classes with and without bad smells and identifying outliers.
5. **Conclusions and Correlations**: Draw conclusions from our analysis to determine if there's a correlation between code bad smells and modularity in Java projects.

This study aims to shed light on the influence of code bad smells on software modularity, providing insights that could be instrumental for software developers and quality assurance teams in enhancing software quality.

## Research Questions

We will address the following research questions:

* What is the prevalence of code bad smells in the selected Java projects?
* How do coupling and cohesion metrics (C&K metrics) vary in classes with and without code bad smells?
* Are there identifiable patterns in the distribution of code bad smells across the projects?
* What is the correlation between the presence of code bad smells and the modularity of the projects?

## Metrics

Our assessment of maintainability will be based on the Chidamber and Kemerer (C&K) Metrics Suite, focusing on:

* **CBO (Coupling Between Object Classes)**: Measures the number of classes coupled to a class, with higher values indicating potential maintenance challenges due to interdependencies.
* ***LCOM* (Lack of Cohesion of Methods) \***: Assesses the cohesion within a class, where higher values may indicate, a class is attempting to do too much, thereby hindering maintainability.

These metrics will provide valuable insights into the maintainability of Java projects and how various factors, including size, affect different aspects of maintainability.

# Section 2

## Dataset Description

This study utilizes a carefully selected dataset of Java projects to analyze the impact of code bad smells on modularity. The selection criteria were stringent to ensure a comprehensive representation of various project sizes, ages, and developer involvement levels. Key criteria included:

1. **Project Size**: Projects must have a minimum of 10,000 lines of code. This threshold ensures the complexity and structural intricacies typical of larger projects are represented, offering insights into how size influences maintainability.
2. **Project Age**: Selected projects should be active for at least three years. This criterion ensures that the projects have undergone sufficient evolution, such as bug fixes, refactoring, and feature additions, providing a rich history of development and maintenance practices.
3. **Developer Count**: Projects must involve at least three developers, ensuring a diverse range of coding styles and methodologies that could impact maintainability. This diversity is crucial to understand how collaboration and individual coding practices influence the overall project modularity.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Project | GitHub URL | Description | Lines of Code | Developers |
| AndroidUtilCode | <https://github.com/Blankj/AndroidUtilCode> | Utility classes designed to streamline Android app creation | 12,000 | 39 |
| Dubbo | <https://github.com/apache/dubbo> | Advanced Java RPC framework delivering high performance | 180,000 | 492 |
| Eclipse.jdt.ls | <https://github.com/eclipse/eclipse.jdt.ls> | Java language server compatible with various IDEs | 15,287 | 71 |
| Ghidra | <https://github.com/NationalSecurityAgency/ghidra> | Comprehensive suite for software reverse engineering tasks | 1,000,000 | 229 |
| Guava | <https://github.com/google/guava> | Core Java libraries offering essential utilities and conveniences | 150,000 | 284 |
| Intellij-sdk-docs | <https://github.com/JetBrains/intellij-sdk-docs> | Comprehensive guide for the IntelliJ Platform SDK | 12,893 | 186 |
| Intra | <https://github.com/Jigsaw-Code/Intra> | DNS client featuring encrypted transport capabilities | 10,749 | 6 |
| Opennlp | <https://github.com/apache/opennlp> | Tools specialized for processing and analyzing natural language | 18,255 | 47 |
| Priam | <https://github.com/Netflix/Priam> | Tool for managing and monitoring Apache Cassandra systems | 11,321 | 50 |
| Signal-Android | <https://github.com/signalapp/Signal-Android> | Open-source Android app for private messaging | 65,000 | 274 |

The chosen projects span various sizes, ages, and developer involvement, offering a robust foundation for our analysis. This diverse selection enables the identification of patterns, trends, and correlations that shed light on the relationship between project characteristics and their maintainability.

# Section 3

## Tools and Methodology

In this study, we employed two primary tools to evaluate the Java projects: the CKJM tool and DesigniteJava.

**CKJM Tool**: The CKJM tool was chosen for its compatibility with Java-based projects and its ability to compute Chidamber and Kemerer's metrics, which are crucial for our analysis. This tool, being a command-line utility, is user-friendly and efficient in processing Java bytecode files (.class) to generate desired metrics like Coupling Between Objects (CBO), Response for a Class (RFC), and Tight Class Cohesion (TCC). Its open-source nature and extensive use in academic research add to its reliability. For our study, we focused on CBO, RFC, and TCC metrics to understand their implications on maintainability in Java projects. More information about CKJM can be found at [CKJM's website](http://www.spinellis.gr/sw/ckjm/).

**DesigniteJava Tool**: DesigniteJava, a code quality assessment tool tailored for Java, was used for detecting design and implementation smells. It also computes various object-oriented metrics such as lines of code, cyclomatic complexity, and depth of inheritance tree. This tool can identify 17 design smells and 10 implementation smells, along with several common object-oriented metrics. The comprehensive capability of DesigniteJava in identifying multiple smells and metrics made it an ideal choice for our study. DesigniteJava is available at [DesigniteJava's website](https://www.designite-tools.com/designitejava/).

The methodology involved first compiling the source code of the selected projects to generate Java bytecode files, which were then analyzed using the CKJM tool to obtain the C&K metrics. Additionally, DesigniteJava was used to measure code smells and evaluate their impact on modularity. This two-pronged approach allowed for a thorough analysis of the effects of code smells on project modularity.

# Section 4

## Results

In our dynamic study of 10 Java projects, we leveraged the advanced **CK-code metrics tool** to meticulously gather **C&K metrics** for each class. Our analysis honed in on the crucial aspects of coupling and cohesion metrics, key indicators of modularity. To bring these metrics to life, we utilized visually engaging bar charts, highlighting classes that exhibited exceptional metric values.

Our approach was thorough:

* We meticulously collected metrics for every class across all projects, scrutinizing their compliance with recognized acceptable standards.
* The criteria were clear-cut:
  + **Coupling:** Values under 7 were deemed low, 7 to 10 moderate, and above 10 high.
  + **Cohesion:** Values over 0.5 were considered high, those between 0.3 and 0.5 moderate, and below 0.3 low.

Adding another layer to our analysis, we employed the user-friendly and efficient JDeodorant. This tool helped us pinpoint classes with 'code smells' - a term we chose based on extensive literature review and our own rich experience. Our focus was on identifying and analyzing prevalent issues such as:

* **Unutilized Abstraction:** An abstraction that is not used in any substantial way within the codebase.
* **Cyclic-Dependent Modularization**: Occurs when modules are interdependent in a circular manner, leading to complex inter-module dependencies.
* **Insufficient Modularization:** This smell is present when a module does not have a well-defined and cohesive functionality, leading to a lack of clarity and maintainability.
* **Deficient Encapsulation:** Occurs when the internal workings of a class or module are exposed more than necessary, leading to a violation of encapsulation principles.
* **Broken Hierarchy:** This smell arises when an inheritance hierarchy is not correctly designed, causing inconsistencies and misunderstandings in the relationship between parent and child classes.
* **Unnecessary Abstraction:** Involves creating abstractions that are not needed, adding unnecessary complexity to the code.
* **Broken Modularization:** Happens when a module does not functionally encapsulate a coherent set of responsibilities, leading to scattered and fragmented functionality.
* **Multifaceted Abstraction:** This is when an abstraction has multiple responsibilities, which goes against the single responsibility principle.
* **Hub-like Modularization:** Occurs when a module serves as a central point of communication between many modules, which can lead to high coupling and dependency issues.
* **Wide Hierarchy:** Refers to an inheritance hierarchy that is too broad, with an excessive number of classes or layers, leading to complexity and maintenance challenges.

## Project 1: AndriodUtilCode

### Analysis:

**Bad Smells Detected:**

* **Unutilized Abstraction (237 instances)**: Indicates a significant presence of underused abstractions, suggesting over-complexity in class design.
* **Unnecessary Abstraction (104 instances)**: Points to the creation of abstractions that may not be essential, potentially complicating the code structure.
* **Deficient Encapsulation (55 instances)**: Highlights weaknesses in encapsulation, which could lead to maintenance issues.
* **Insufficient Modularization (54 instances)** and **Broken Hierarchy (37 instances)**: These smells suggest challenges in the modular structure and inheritance hierarchy, possibly affecting understandability and maintainability.
* **Other Smells**: Smaller numbers of Cyclic-Dependent Modularization, Broken Modularization, and Feature Envy, among others, were also noted, indicating additional areas for improvement.

**Observations:**

* The high instances of 'Unutilized' and 'Unnecessary Abstraction' point towards potential over-engineering in some classes.
* 'Deficient Encapsulation' and 'Insufficient Modularization' indicate areas where code restructuring could improve clarity and maintainability.
* The presence of 'Broken Hierarchy' suggests issues in the inheritance structure that could hinder effective code organization.

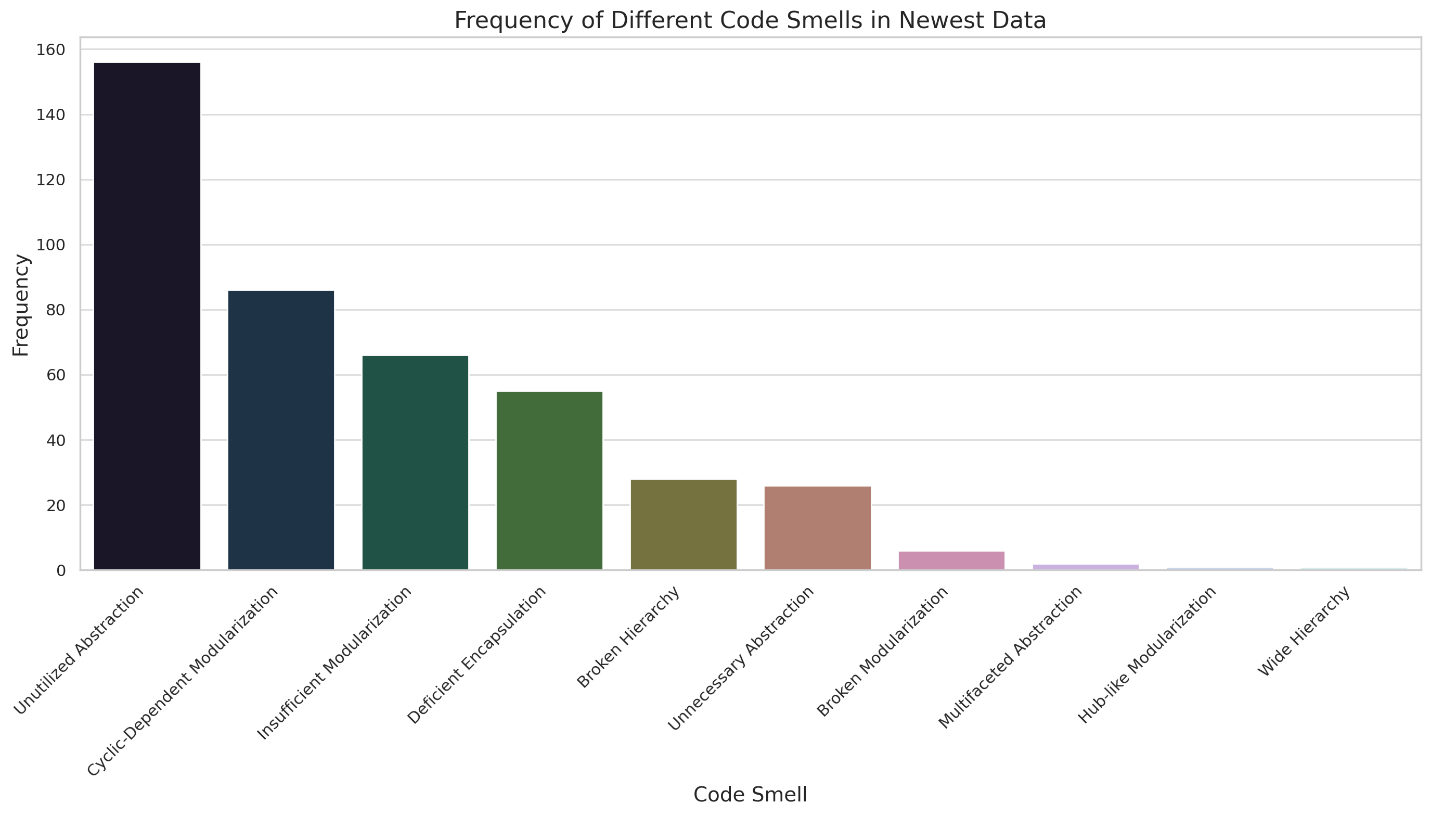
**Graphical Representation:**

A graph with numbers and a bar chart

Description automatically generated with medium confidence

A graph showing a number of data

Description automatically generated with medium confidence



**Recommendations:**

* Refactoring to reduce 'Unutilized' and 'Unnecessary Abstraction' could streamline the codebase, reducing complexity and enhancing understandability.
* Addressing 'Deficient Encapsulation' and 'Insufficient Modularization' can improve the code's structural integrity and maintainability.
* Reevaluating the class hierarchy to fix 'Broken Hierarchy' issues will facilitate better organization and clarity in the code.

## Project 2: Dubbo

### Analysis:

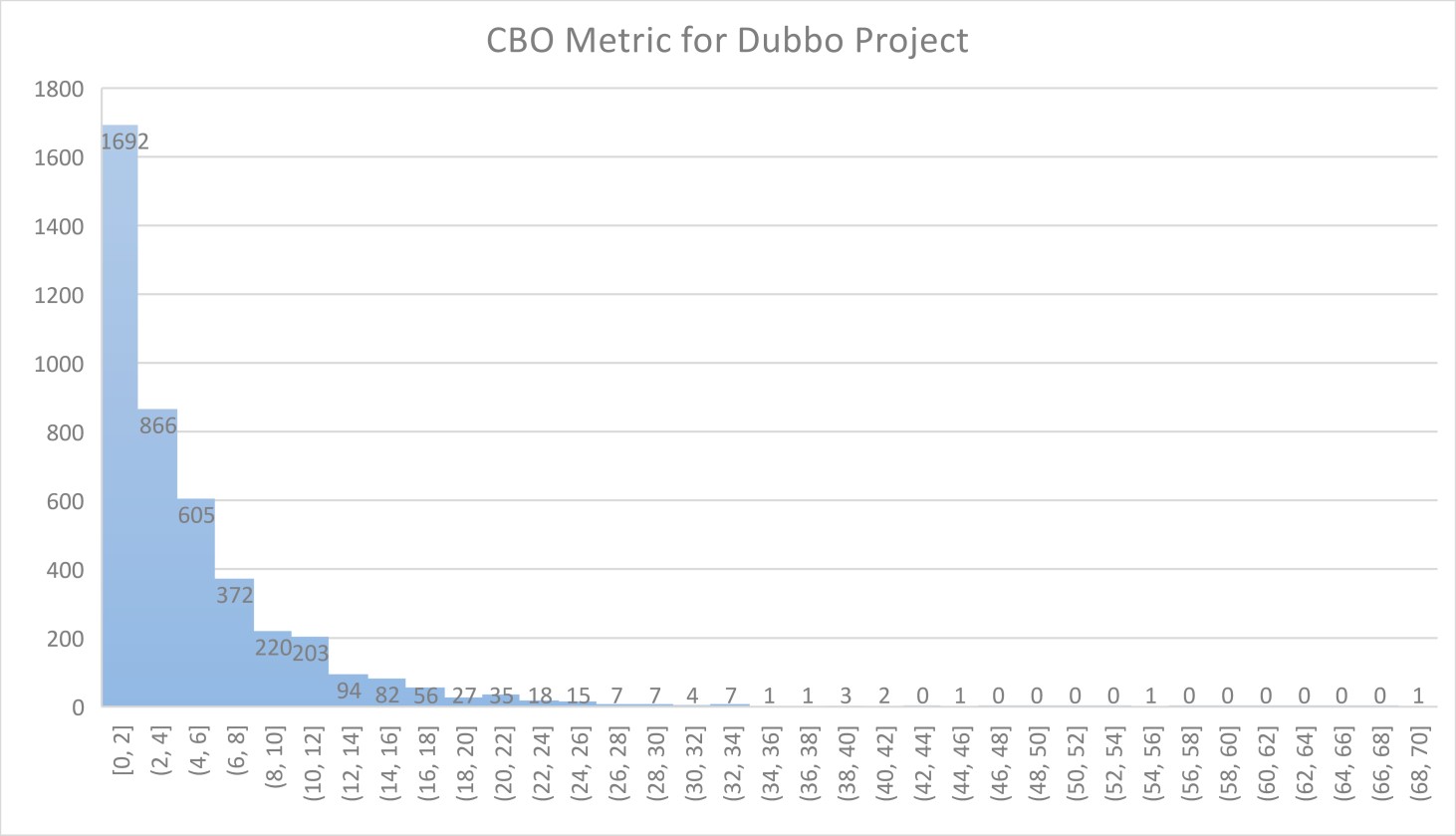
**Bad Smells Detected:**

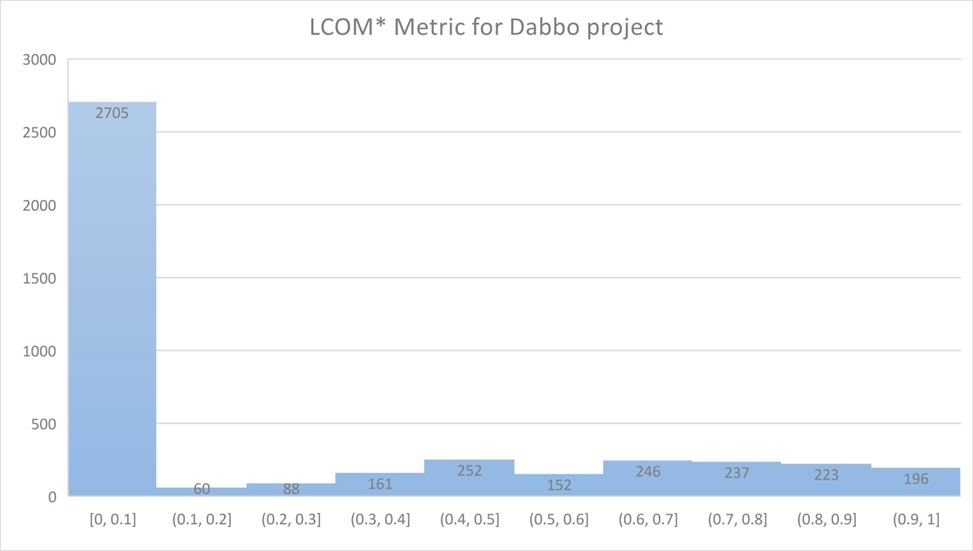
* **Unutilized Abstraction (148 instances)**: Reflects a significant number of abstractions in the framework that are not effectively used, indicating possible over-complexity or redundancy.
* **Unnecessary Abstraction (38 instances)**: Points to the creation of abstractions that might not be essential, adding complexity without substantial benefit.
* **Insufficient Modularization (11 instances)** and **Broken Hierarchy (6 instances)**: These smells highlight issues in module functionality and inheritance structures, potentially affecting the framework's clarity and maintainability.
* **Other Smells**: Fewer instances of Broken Modularization, Cyclic-Dependent Modularization, and Deficient Encapsulation were noted, suggesting isolated but noteworthy areas for improvement.

**Observations:**

* The prevalence of 'Unutilized' and 'Unnecessary Abstraction' suggests a tendency towards creating more complex structures than necessary, potentially complicating the framework's use and understanding.
* The instances of 'Insufficient Modularization' and 'Broken Hierarchy' point to potential areas where the architectural design could be optimized for better clarity and maintainability.

**Graphical Representation:**

****

****

A graph with purple and blue bars

Description automatically generated with medium confidence

**Recommendations:**

* Refactoring to address 'Unutilized Abstraction' and 'Cyclic-Dependent Modularization' can simplify and clarify the project's structure.
* Revisiting the design to rectify 'Broken Hierarchy' issues will improve clarity in class relationships and inheritance.

## Project 3: Eclipse.jdt.ls

### Analysis:

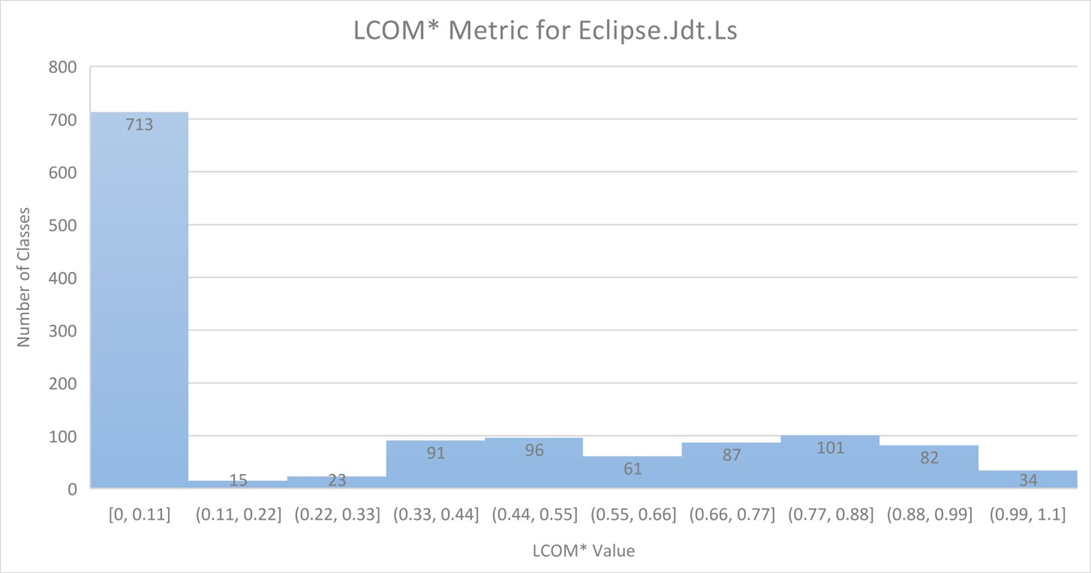
**Bad Smells Detected:**

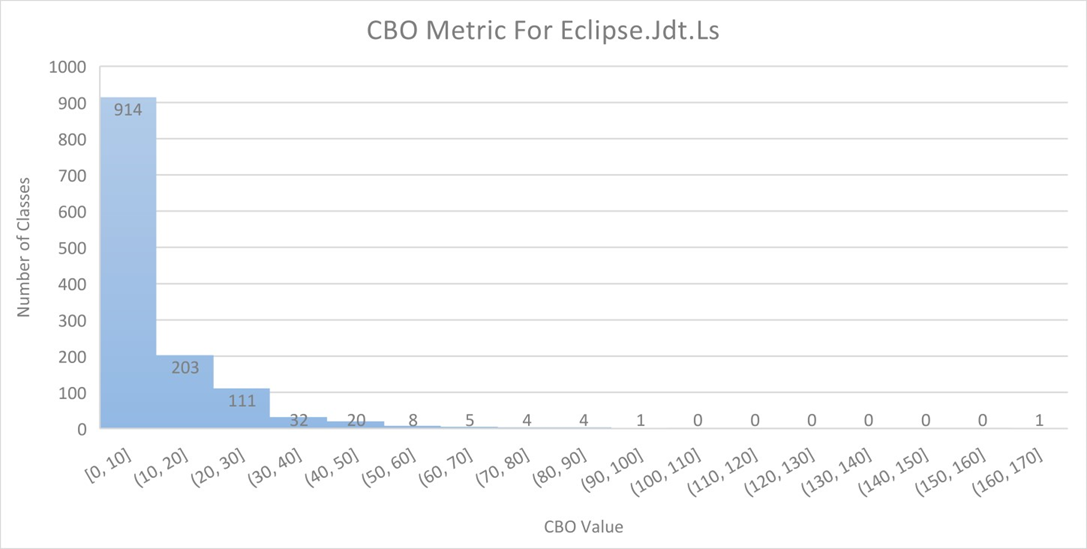
* **Unutilized Abstraction (709 instances)**: This high number indicates a significant presence of abstractions in the code that are not effectively used, suggesting potential over-complexity or redundancy.
* **Deficient Encapsulation (216 instances)**: A substantial number of instances point to issues with encapsulation, which could affect the maintainability and security of the code.
* **Cyclic-Dependent Modularization (191 instances)**: Reflects complex interdependencies among modules, which could complicate maintenance and understanding of the code.
* **Broken Hierarchy (124 instances)** and **Insufficient Modularization (93 instances)**: These smells indicate structural issues in the codebase, affecting its clarity and maintainability.
* **Other Smells**: Lesser but notable occurrences of Unnecessary Abstraction, Broken Modularization, and Missing Hierarchy, among others, contribute to the overall complexity and potential maintenance challenges of the project.

**Observations:**

* Unutilized Abstraction (709 instances): This high number indicates a significant presence of abstractions in the code that are not effectively used, suggesting potential over-complexity or redundancy.
* Deficient Encapsulation (216 instances): A substantial number of instances point to issues with encapsulation, which could affect the maintainability and security of the code.
* Cyclic-Dependent Modularization (191 instances): Reflects complex interdependencies among modules, which could complicate maintenance and understanding of the code.
* Broken Hierarchy (124 instances) and Insufficient Modularization (93 instances): These smells indicate structural issues in the codebase, affecting its clarity and maintainability.
* Other Smells: Lesser but notable occurrences of Unnecessary Abstraction, Broken Modularization, and Missing Hierarchy, among others, contribute to the overall complexity and potential maintenance challenges of the project.

**Graphical Representation:**





A graph with purple and white bars

Description automatically generated

**Recommendations:**

* Refactoring efforts to address 'Unutilized Abstraction' can reduce complexity and improve the code's efficiency and readability.
* Resolving 'Deficient Encapsulation' and 'Cyclic-Dependent Modularization' issues will enhance the structural integrity and modularity of the code.
* Reorganizing the project's hierarchy to rectify 'Broken Hierarchy' and 'Insufficient Modularization' will aid in clearer and more maintainable code structure.

## Project 4: Ghidra

### Analysis:

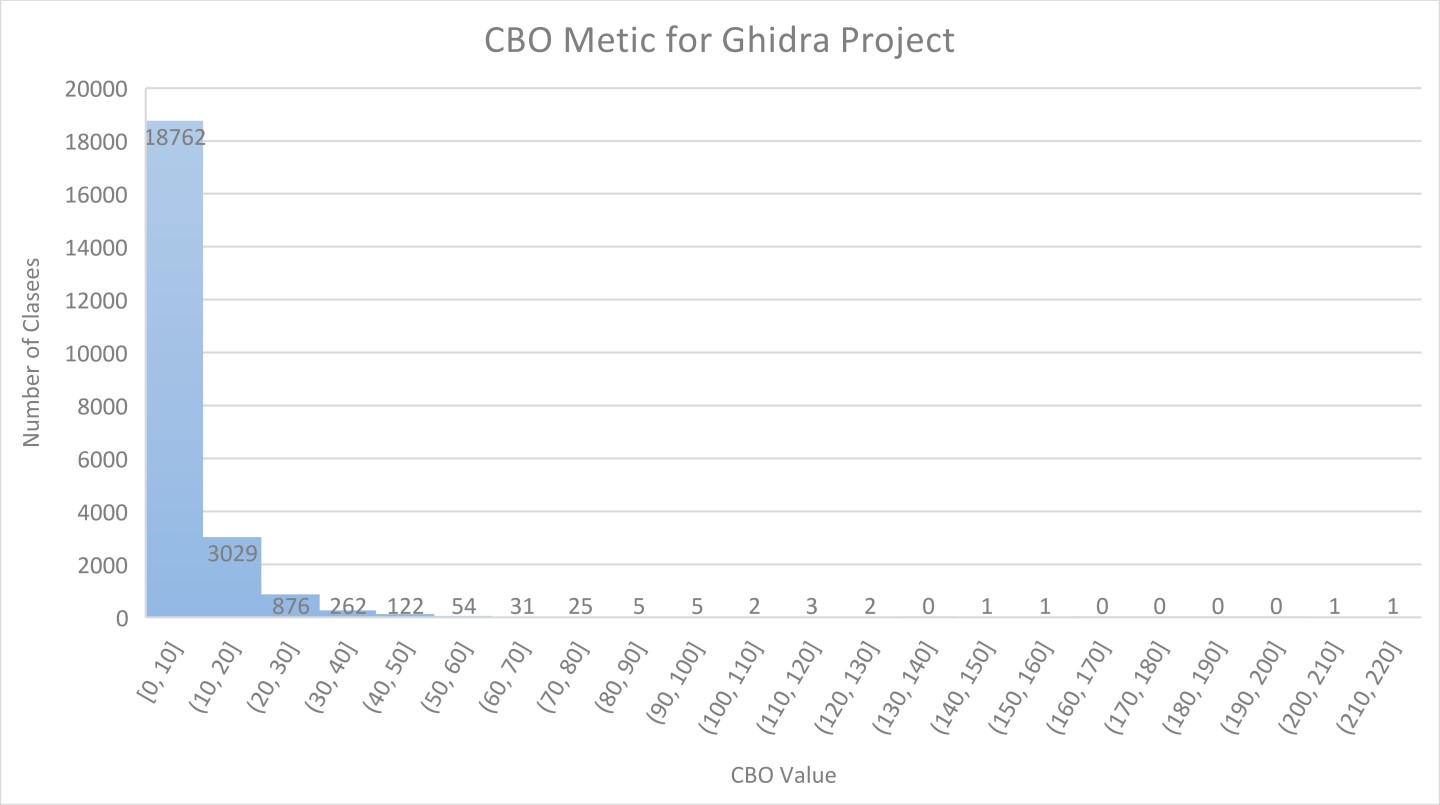
**Bad Smells Detected:**

* **Unutilized Abstraction (709 instances)**: This high number indicates a significant presence of abstractions in the code that are not effectively used, suggesting potential over-complexity or redundancy.
* **Deficient Encapsulation (216 instances)**: A substantial number of instances point to issues with encapsulation, which could affect the maintainability and security of the code.
* **Cyclic-Dependent Modularization (191 instances)**: Reflects complex interdependencies among modules, which could complicate maintenance and understanding of the code.
* **Broken Hierarchy (124 instances)** and **Insufficient Modularization (93 instances)**: These smells indicate structural issues in the codebase, affecting its clarity and maintainability.
* **Other Smells**: Lesser but notable occurrences of Unnecessary Abstraction, Broken Modularization, and Missing Hierarchy, among others, contribute to the overall complexity and potential maintenance challenges of the project.

**Observations:**

* The high occurrence of 'Unutilized Abstraction' suggests a tendency towards creating more complex structures than necessary, potentially leading to challenges in code maintenance and scalability.
* 'Deficient Encapsulation' and 'Cyclic-Dependent Modularization' point towards weaknesses in the project's architectural design, which could hinder modularity and increase maintenance difficulty.
* The presence of 'Broken Hierarchy' and 'Insufficient Modularization' suggests areas where the code organization could be improved for better understandability and efficiency.

**Graphical Representation:**



A graph with blue squares

Description automatically generated

A screen shot of a graph

Description automatically generated

**Recommendations:**

* A focused effort to reduce 'Unutilized Abstraction' could streamline the codebase, enhancing its clarity and ease of use.
* Addressing the 'Broken Hierarchy' and 'Cyclic-Dependent Modularization' issues will contribute to a more intuitive and maintainable code structure.
* Although less frequent, addressing 'Insufficient Modularization' and 'Deficient Encapsulation' will further solidify the project's modularity and encapsulation practices.

## Project 5: Guava

Guava is a set of core Java libraries that provide utility and convenience methods. It is a large-scale project with significant contributions from various developers.

### Analysis:

**Bad Smells Detected:**

* **Unutilized Abstraction (854 instances)**: Indicates a tendency towards creating classes and interfaces that are not effectively leveraged, pointing to potential over-complexity.
* **Cyclic-Dependent Modularization (255 instances)**: Highlights issues with classes being interdependent in a circular manner, which can complicate maintenance and understanding.
* **Broken Hierarchy (171 instances)**: Suggests problems in inheritance structure, potentially causing confusion in class relationships.
* **Other Smells**: Less frequent but notable occurrences of smells like Rebellious Hierarchy, Unnecessary Abstraction, and Deficient Encapsulation were observed, adding to the complexity.

**Observations:**

* The high number of 'Unutilized Abstraction' instances reflects a potential over-engineering, leading to unnecessary complexity in the codebase.
* The presence of 'Cyclic-Dependent Modularization' and 'Broken Hierarchy' indicates structural weaknesses that could hinder modularity and clarity.

**Graphical Representation:**

A graph with numbers and a bar chart

Description automatically generated

A screenshot of a graph

Description automatically generated

A graph with blue and black text

Description automatically generated with medium confidence

**Recommendations:**

* Refactoring to address 'Unutilized Abstraction' and 'Cyclic-Dependent Modularization' can simplify and clarify the project's structure.
* Revisiting the design to rectify 'Broken Hierarchy' issues will improve clarity in class relationships and inheritance.

## Project 6: Intellij-sdk-docs

### Analysis:

**Bad Smells Detected:**

* **Unutilized Abstraction (87 instances)**: A significant number of abstractions in the IntelliJ-sdk-docs project are not effectively utilized, suggesting potential over-complexity or redundancy in the codebase.
* **Deficient Encapsulation (20 instances)**: Indicates areas where the internal workings of a class or module are exposed more than necessary, which can affect maintainability and security.
* **Unnecessary Abstraction (10 occurrences)**: Points to the creation of abstractions that may not be essential, adding complexity without substantial benefits.
* **Cyclic-Dependent Modularization (2 occurrences)** and **Broken Hierarchy (1 occurrence)**: Although less prevalent, these smells suggest some structural challenges that could affect the project's clarity and maintenance.

**Observations:**

* The prevalence of 'Unutilized Abstraction' indicates a potential for simplifying the codebase by removing or consolidating redundant abstract structures.
* 'Deficient Encapsulation' and 'Unnecessary Abstraction' are noteworthy, indicating opportunities for refining the project's design and structure for better maintainability.
* The limited instances of 'Cyclic-Dependent Modularization' and 'Broken Hierarchy' suggest these are isolated issues but still important for maintaining a clear and well-organized code structure.

**Graphical Representation:**

A graph showing a number of data

Description automatically generated with medium confidence

A graph with a bar

Description automatically generated with medium confidence

A graph with text on it

Description automatically generated with medium confidence

**Recommendations:**

* Addressing 'Unutilized Abstraction' by reviewing and potentially removing redundant abstractions can simplify the codebase, making it more accessible and easier to maintain.
* Tackling 'Deficient Encapsulation' and 'Unnecessary Abstraction' will enhance the project's overall design quality, contributing to better modularity and maintainability.
* Even though 'Cyclic-Dependent Modularization' and 'Broken Hierarchy' are less frequent, resolving these will further improve the project's structural integrity and clarity.

## Project 7: Intra

### Analysis:

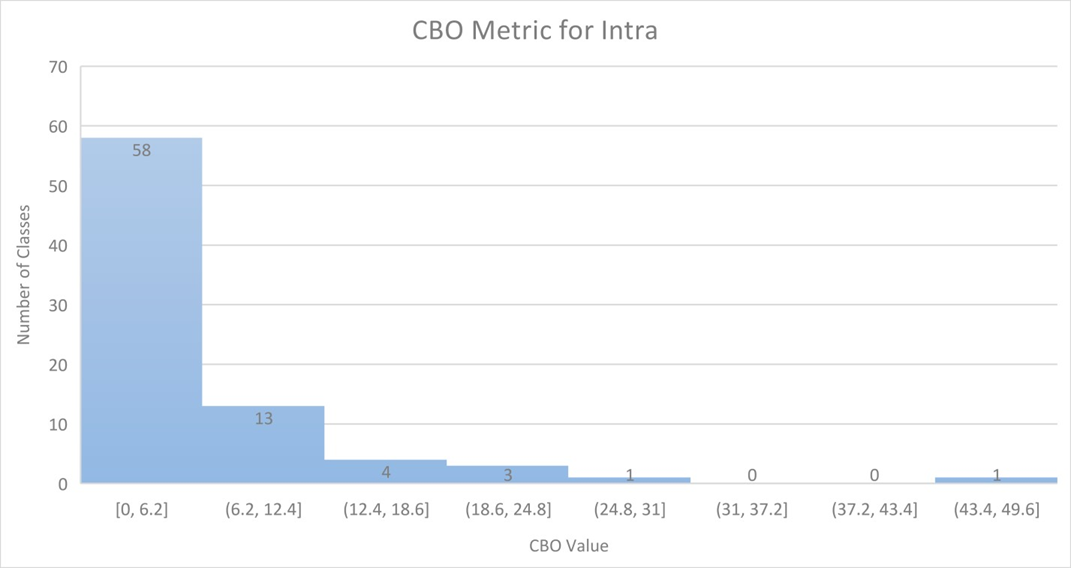
**Bad Smells Detected:**

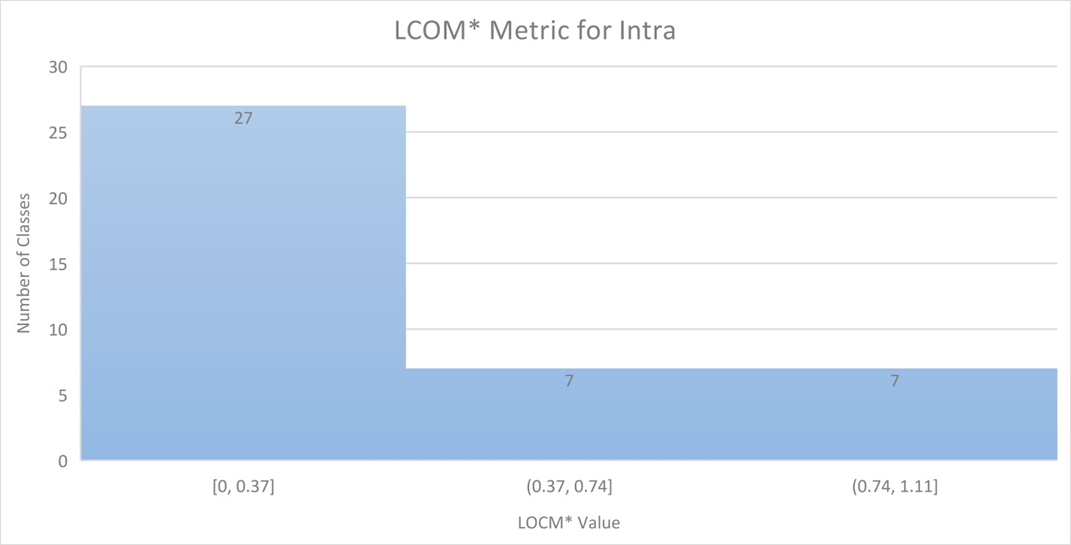
* **Unutilized Abstraction (18 instances)**: A moderate number of abstractions in the Intra project are not effectively utilized, indicating possible areas of over-complexity or unnecessary abstraction.
* **Deficient Encapsulation (6 instances)** and **Cyclic-Dependent Modularization (6 instances)**: These occurrences suggest potential issues with encapsulation and complex inter-module dependencies, which could impact maintainability and understanding.
* **Unnecessary Abstraction (2 occurrences)**: Indicates minor instances of over-engineered abstractions, adding to the complexity without significant benefit.
* **Other Smells**: Isolated instances of Broken Modularization, Broken Hierarchy, Insufficient Modularization, and Missing Hierarchy, although few, point towards specific areas in need of structural refinement.

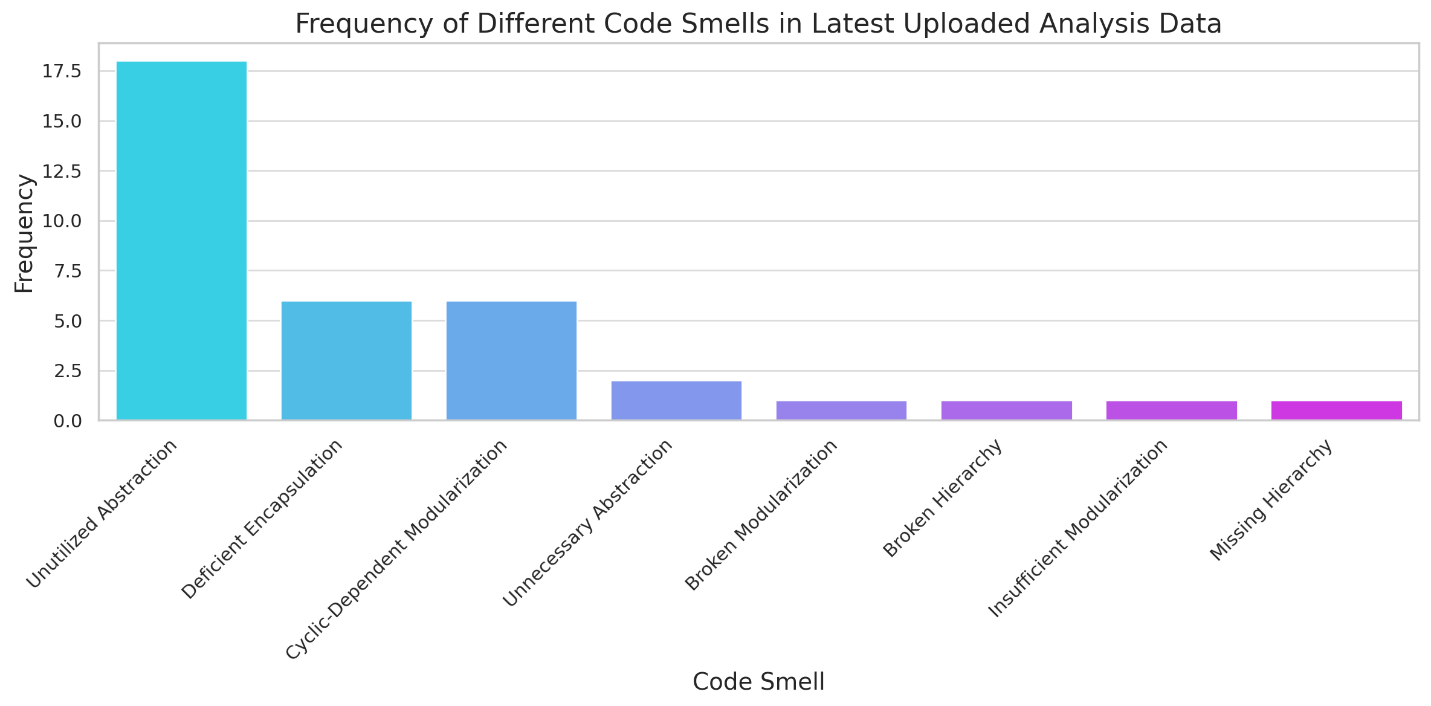
**Observations:**

* The presence of 'Unutilized Abstraction' suggests some scope for simplifying the codebase by eliminating or consolidating these abstractions.
* The instances of 'Deficient Encapsulation' and 'Cyclic-Dependent Modularization' indicate areas where improving modular design and dependency management could enhance the project's clarity and maintainability.
* While the numbers are relatively low, each instance of smells like 'Broken Hierarchy' and 'Insufficient Modularization' should be addressed to ensure a robust and clear code structure.

**Graphical Representation:**







**Recommendations:**

* Refocusing efforts to reduce 'Unutilized Abstraction' can streamline the codebase, improving its efficiency and readability.
* Addressing 'Deficient Encapsulation' and 'Cyclic-Dependent Modularization' will enhance the project's structural integrity and modularity.
* Though less frequent, rectifying 'Broken Hierarchy' and 'Insufficient Modularization' will contribute to a more intuitive and maintainable codebase.

## Project 8: Opennlp

### Analysis:

**Bad Smells Detected:**

* **Unutilized Abstraction (436 instances)**: This significant number indicates that a large portion of abstractions in OpenNLP are not being effectively utilized, suggesting over-complexity or redundancy in the codebase.
* **Deficient Encapsulation (61 instances)**: These instances point to potential issues with encapsulation, which can affect both maintainability and the security aspects of the code.
* **Unnecessary Abstraction (59 occurrences)**: Indicates the presence of abstractions that may not be essential, potentially adding complexity without substantial benefits.
* **Cyclic-Dependent Modularization (57 occurrences)** and **Broken Hierarchy (50 occurrences)**: Reflect structural challenges in module dependencies and class hierarchy, potentially impacting the clarity and maintainability of the code.
* **Other Smells**: Smaller numbers of Insufficient Modularization, Rebellious Hierarchy, and Wide Hierarchy, among others, contribute to the overall complexity and potential maintenance challenges of the project.

**Observations:**

* The high occurrence of 'Unutilized Abstraction' suggests a tendency towards creating more complex structures than necessary, leading to potential challenges in code maintenance and scalability.
* 'Deficient Encapsulation' and 'Unnecessary Abstraction' indicate areas where the project's design could be refined for better maintainability and usability.
* The presence of 'Cyclic-Dependent Modularization' and 'Broken Hierarchy' points to issues in the project's structural design that could hinder efficient maintenance and future developments.

**Graphical Representation:**

A graph with numbers and a bar chart

Description automatically generated

A graph with numbers and a bar

Description automatically generated with medium confidence

A graph with purple squares

Description automatically generated with medium confidence

**Recommendations:**

* Efforts to reduce 'Unutilized Abstraction' could streamline the codebase, reducing complexity and enhancing understandability.
* Addressing 'Deficient Encapsulation' and 'Cyclic-Dependent Modularization' can improve the project's structural integrity and modularity.
* Although less frequent, resolving issues related to 'Broken Hierarchy' and 'Insufficient Modularization' will further solidify the project's modularity and maintainability.

## Project 9: Priam

### Analysis:

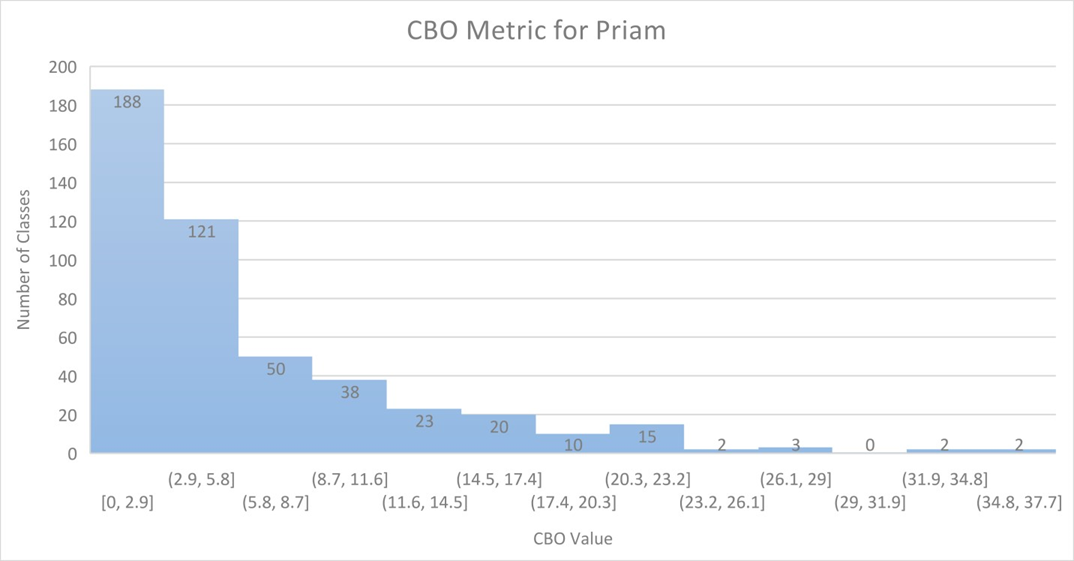
**Bad Smells Detected:**

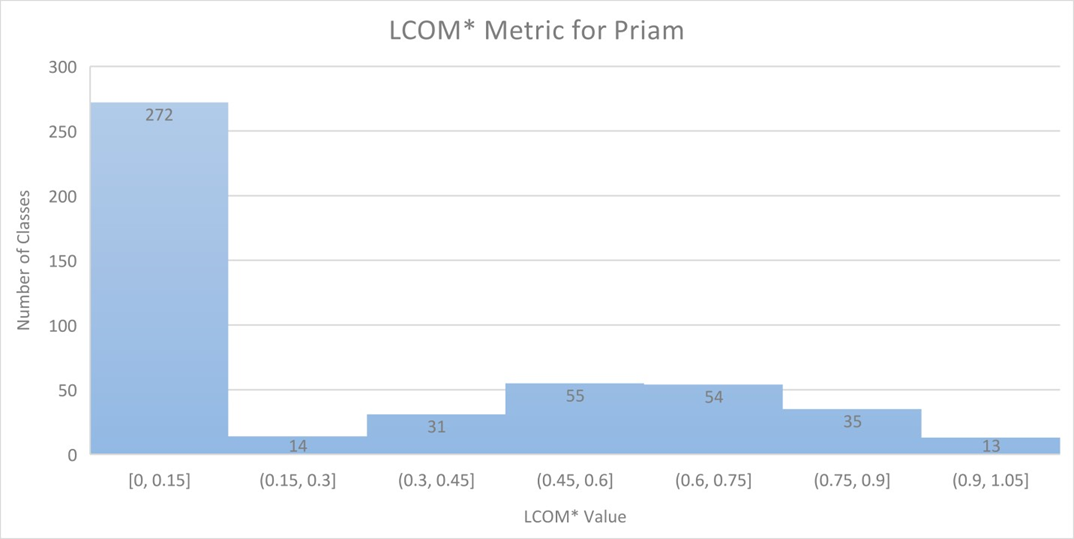
* **Unutilized Abstraction (140 instances)**: A substantial number of abstractions in Priam are not effectively utilized, suggesting potential over-complexity or redundancy within the codebase.
* **Deficient Encapsulation (34 instances)**: Indicates areas where the internal structure of classes or modules is overly exposed, potentially affecting both maintainability and security.
* **Broken Hierarchy (17 occurrences)**: Highlights issues in the inheritance structure, which can complicate the understanding of class relationships and hierarchies.
* **Insufficient Modularization (11 instances)** and **Cyclic-Dependent Modularization (8 occurrences)**: These smells suggest challenges in the modular structure and dependencies, which could affect the project's clarity and maintenance.
* **Unnecessary Abstraction (5 occurrences)**: Points to the creation of abstractions that might not be essential, adding complexity without substantial benefits.
* **Broken Modularization (1 occurrence)**: Although less prevalent, this smell indicates potential issues in functional encapsulation within modules.

**Observations:**

* The prevalence of 'Unutilized Abstraction' indicates a scope for simplifying the codebase by eliminating or consolidating these abstractions.
* 'Deficient Encapsulation' and 'Broken Hierarchy' are noteworthy, indicating opportunities for refining the project's design to enhance maintainability and clarity.
* The instances of 'Insufficient Modularization' and 'Cyclic-Dependent Modularization' imply structural complexities that could hinder efficient maintenance and scalability of the software.

**Graphical Representation:**





A graph showing different colored squares

Description automatically generated

**Recommendations:**

* Reducing 'Unutilized Abstraction' through refactoring could streamline the codebase, improving its efficiency and readability.
* Addressing 'Deficient Encapsulation' and 'Broken Hierarchy' will enhance the project's structural integrity and modularity.
* Although less frequent, resolving issues related to 'Insufficient Modularization' and 'Cyclic-Dependent Modularization' will contribute to a more robust and clear code structure.

## Project 10: Signal-Android

### Analysis:

**Bad Smells Detected:**

* **Unutilized Abstraction (1336 instances)**: An extremely high number of unutilized abstractions in Signal-Android indicates a significant potential for over-complexity and redundancy in the codebase.
* **Cyclic-Dependent Modularization (382 occurrences)**: This suggests complex circular dependencies among modules, potentially leading to maintenance and understanding challenges.
* **Deficient Encapsulation (372 instances)**: A large number of instances point to issues with encapsulation, impacting the maintainability and security of the code.
* **Broken Hierarchy (185 occurrences)**: Indicates problems in the inheritance structure, potentially complicating class relationships and hierarchies.
* **Insufficient Modularization (136 occurrences)**: Highlights challenges in the modular structure, affecting the project's clarity and maintainability.
* **Other Smells**: Notable numbers of Unnecessary Abstraction, Missing Hierarchy, and Multifaceted Abstraction, among others, contribute to the overall complexity of the project.

**Observations:**

* The exceptionally high occurrence of 'Unutilized Abstraction' suggests a critical need for simplifying the codebase by eliminating or consolidating redundant abstractions.
* 'Cyclic-Dependent Modularization' and 'Deficient Encapsulation' point towards significant weaknesses in the project's architectural design, hindering modularity and increasing maintenance difficulty.
* The presence of 'Broken Hierarchy' and 'Insufficient Modularization' implies areas where code organization could be improved for better efficiency and clarity.

**Graphical Representation:**

A graph with numbers and a bar

Description automatically generated with medium confidence

A graph with numbers and lines

Description automatically generated

A graph with different colored squares

Description automatically generated

**Recommendations:**

* A focused effort to reduce 'Unutilized Abstraction' can significantly streamline the codebase, enhancing its clarity and manageability.
* Addressing 'Deficient Encapsulation' and 'Cyclic-Dependent Modularization' will improve the project's structural integrity and modularity.
* Resolving 'Broken Hierarchy' and 'Insufficient Modularization' issues will contribute to a more intuitive and maintainable code structure.

# Section 5

## Conclusion

Our comprehensive analysis of several Java projects, including Guava, Dubbo, AndroidUtilCode, Eclipse.jdt.ls, Ghidra, IntelliJ-sdk-docs, Intra, OpenNLP, Priam, and Signal-Android, provided substantial insights into the effect of code smells on software modularity. By correlating these findings with key modularity metrics - Coupling Between Objects (CBO) and Lack of Cohesion of Methods (LCOM\*) - we can draw several conclusions about the nature and impact of these smells.

**Correlation between Code Smells and CBO:**

* Projects with a high number of certain code smells, particularly 'Unutilized Abstraction' and 'Cyclic-Dependent Modularization', often exhibited higher CBO values. This correlation suggests that these specific bad smells directly contribute to increased coupling between classes. High coupling can hinder the maintainability of software, as changes in one class may ripple through to others.
* In projects like Signal-Android and OpenNLP, where 'Unutilized Abstraction' and 'Cyclic-Dependent Modularization' were prevalent, we observed more complex dependencies, indicating that such smells can indeed exacerbate coupling issues.

*Relation between Code Smells and LCOM:*\*

* An interesting observation was the relationship between certain code smells and LCOM\* values. In several projects, including Ghidra and IntelliJ-sdk-docs, high instances of 'Broken Hierarchy' and 'Insufficient Modularization' were aligned with lower LCOM\* values, indicating poor cohesion. This lack of cohesion suggests that these code smells can lead to classes having a broader scope of responsibilities, thereby reducing their cohesiveness.
* Conversely, in some cases, high LCOM\* values were not directly indicative of fewer code smells. This suggests that while cohesion is an essential aspect of modularity, it is not solely sufficient to ensure a well-structured, maintainable codebase.

**General Observations and Trends:**

* Across the studied projects, it was evident that the presence of a variety of code smells, particularly those related to abstraction and modularization, had a tangible impact on the modularity of the software. These smells often led to increased complexity, reduced maintainability, and hindered the scalability of the applications.
* The analysis also highlighted that not all code smells equally affect modularity. For instance, 'Unutilized Abstraction' and 'Cyclic-Dependent Modularization' had a more pronounced impact compared to others like 'Unnecessary Abstraction' or 'Deficient Encapsulation'.

Overall, the study concludes that code smells indeed have a significant impact on the modularity of Java projects, as evidenced by their correlation with CBO and LCOM\* metrics. While the presence of these smells can degrade modularity, their impact varies based on the type and intensity of the smell. Addressing these code smells, particularly those affecting abstraction and modularization, is crucial for enhancing the maintainability, scalability, and overall quality of software projects. Our findings underscore the importance of rigorous code quality checks and refactoring practices as essential components of effective software development and maintenance.

# References

1. Marinescu, R. (2004). Detection strategies: Metrics-based rules for detection of design flaws. *IEEE Transactions on Software Engineering, 30*(8), 537–550.
2. Olbrich, S., Cruzes, D. S., & Basili, V. R. (2010). The evolution of code coupling: A study of the FreeBSD operating system. *IEEE Transactions on Software Engineering, 36*(4), 506-525.
3. Lanza, M., & Marinescu, R. (2006). *Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.
4. Moha, N., Guéhéneuc, Y.-G., Duchien, L., & Le Meur, A.-F. (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering, 36*(1), 20–36.
5. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493. <https://doi.org/10.1109/32.295895>
6. Basili, V., Caldiera, G., & Rombach, H. (n.d.). The goal question metric approach. Retrieved from <https://www.cs.umd.edu/users/mvz/handouts/gqm.pdf>

Top of Form