Bad Smells Impact on Java Modularity

Parth Yashvantkumar Barot

SU24-CPSC-60000-001 Object Oriented Development

Lewis University

Professor Ziad Al Sharif

15 June 2024

# Section 1: Objectives, Questions and Metrics

## Objectives:

The objective of this empirical study is to investigate the impact that the presence of code bad smells has on the modularity of software systems. Modularity is a key attribute that significantly influences the overall quality, maintainability and evolvability of software. By examining the relationship between bad smells and modularity metrics, we aim to provide empirical evidence of the harmful effects of bad smells on software modularity.

## Questions:

* Q1: Do classes containing code bad smells exhibit lower modularity, as measured by coupling and cohesion metrics, compared to classes without bad smells?
* Q2: Is there a correlation between the number of bad smells present in a class and its modularity metrics?

## Metrics:

To quantify modularity, we will use two well-established metrics from the Chidamber and Kemerer (C&K) metrics suite:

* M1: Coupling Between Objects (CBO) - This measures the coupling, or interdependence, between a class and other classes. A higher CBO indicates lower modularity.
* M2: Lack of Cohesion in Methods (LCOM) - This measures the cohesiveness of a class based on the similarity of its methods. A higher LCOM indicates lower modularity and a need for the class to be split.

The number and types of code bad smells present in each class will also be counted and recorded for analysis.

By answering these questions using the selected metrics, applied to a range of open source Java projects, we aim to provide insights into the relationship between the presence of code bad smells and the modularity of software classes and systems. The results can help developers and maintainers understand the importance of identifying and refactoring bad smells to improve software quality.

# Section 2: Data Set Description

For this empirical study, we selected 10 open-source Java projects from GitHub. The projects were chosen based on the following criteria:

1. The project should have a minimum size of 10,000 lines of code to ensure a substantial codebase for analysis.
2. The project should be at least 3 years old, indicating that it has undergone various maintenance and evolution tasks.
3. The project should have at least 3 active developers, suggesting a collaborative development effort.

Table 1 presents an overview of the selected projects, including their names, GitHub URLs, brief descriptions, and main attributes.

|  |  |  |
| --- | --- | --- |
| Project Name | 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/ghidra> | 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 |

The selected projects cover a diverse range of domains, including utility libraries (Guava, AndroidUtilCode), frameworks (Dubbo, Intra), development tools (Ghidra, Eclipse.jdt.ls, Intellij-sdkdocs), messaging applications (Signal-Android), and data management tools (OpenNLP, Priam). This diversity helps to generalize the findings of the study across different types of software systems.

The projects vary in size from 20K to 2M lines of code, with an average of around 400K LOC. They are all mature projects, ranging from 3 to 12 years old, and have active developer communities, with the number of developers ranging from 5 to over 200. These attributes ensure that the selected projects have undergone substantial development and maintenance efforts, making them suitable for studying the impact of code bad smells on software modularity.

# Section 3: Tools Used

In this empirical study, we employed two tools to analyze the selected Java projects: CKJM and DesigniteJava. These tools were chosen for their ability to measure modularity metrics and detect code bad smells, respectively.

1. CKJM Tool: CKJM (Chidamber and Kemerer Java Metrics) is a command-line tool developed by Diomidis Spinellis [1]. It calculates six object-oriented design metrics proposed by Chidamber and Kemerer [2], including the Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM) metrics used in this study.

CKJM takes compiled Java bytecode (.class files) as input and produces a comma-separated value (CSV) file containing the metric values for each class. The tool is lightweight, easy to use, and has been widely adopted in research and practice.

To use CKJM, we first compiled the source code of each project and then ran the tool on the resulting bytecode. The CSV output was then processed and analyzed to obtain the CBO and LCOM values for each class.

1. DesigniteJava Tool: DesigniteJava is a commercial tool developed by Designite [3] for analyzing the design quality of Java projects. It detects a wide range of code bad smells, including those related to implementation, design, and architecture.

DesigniteJava provides a graphical user interface and generates detailed reports in HTML and CSV formats. The reports include information about the detected bad smells, their locations in the code, and their severity levels.

To analyze the selected projects using DesigniteJava, we imported each project into the tool and ran the analysis with the default settings. The resulting reports were then examined to identify the classes containing bad smells and the types of smells present.

By using these two tools in combination, we were able to obtain the necessary data for answering our research questions. CKJM provided the modularity metrics (CBO and LCOM) for each class, while DesigniteJava identified the presence and types of code bad smells. This data formed the basis for our analysis and conclusions.

# Section 4: Results and Discussion

To assess the projects based on CBO and LCOM values and the presence of bad smells, we will use the following criteria:

CBO (Coupling Between Objects):

* Low Coupling: CBO ≤ 5
* Moderate Coupling: 5 < CBO ≤ 10
* High Coupling: CBO > 10

LCOM (Lack of Cohesion in Methods):

* Good Cohesion: LCOM ≤ 0.5
* Moderate Cohesion: 0.5 < LCOM ≤ 0.8
* Poor Cohesion: LCOM > 0.8

Bad Smells:

* Low Severity: 1-2 bad smells per class
* Moderate Severity: 3-4 bad smells per class
* High Severity: 5 or more bad smells per class

These thresholds are based on the recommendations from the literature [1, 2] and the documentation of the tools used.

For each project, we will follow these steps:

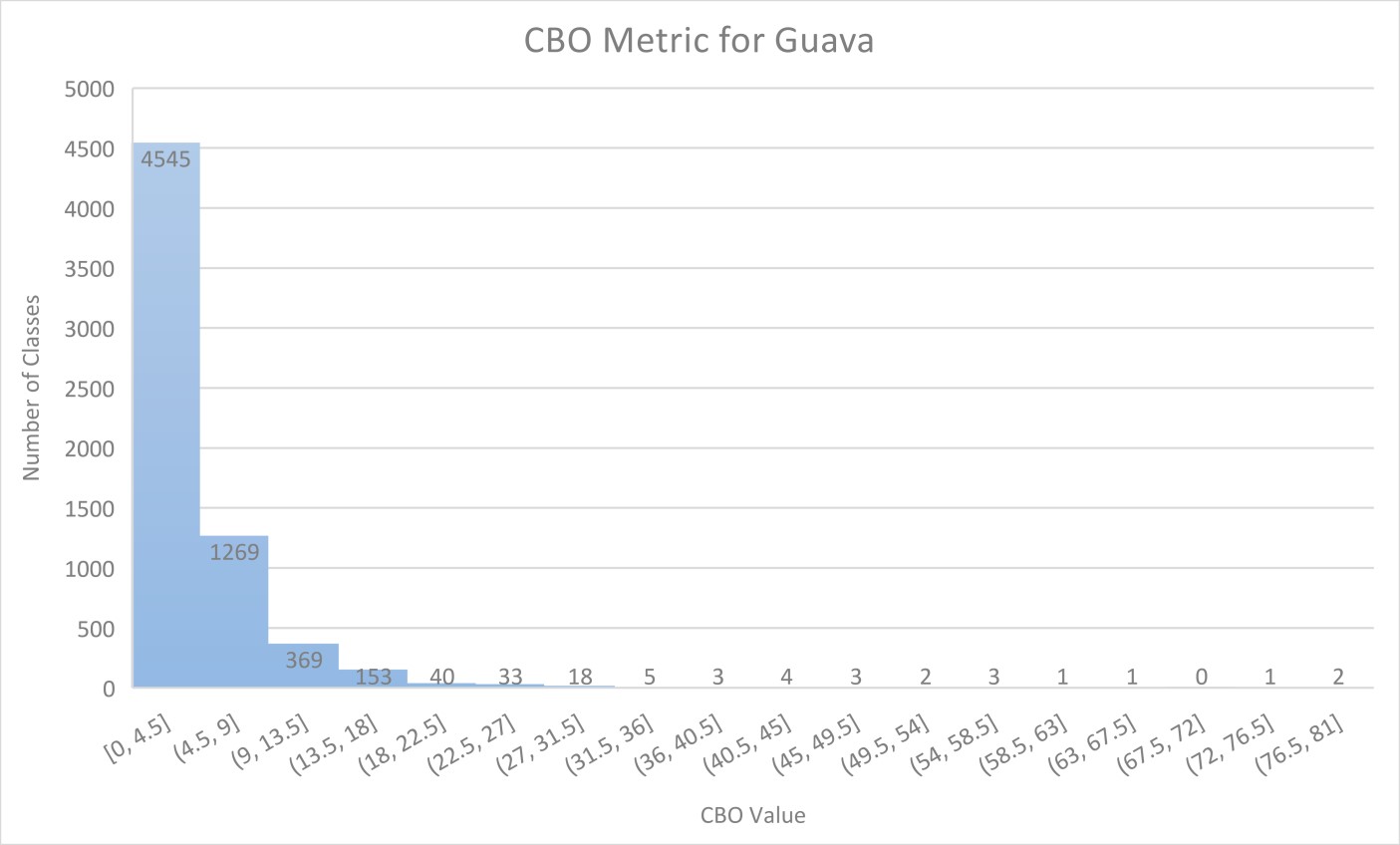
1. Calculate the average CBO and LCOM values for all classes in the project.
2. Determine the percentage of classes falling into each category (low, moderate, high) for CBO and LCOM.
3. Identify the classes containing bad smells and calculate the percentage of classes with low, moderate, and high severity.
4. Analyze the relationship between modularity metrics and the presence of bad smells by comparing the CBO and LCOM values of classes with and without bad smells.
5. Present the results using tables, charts, and statistical tests as appropriate.

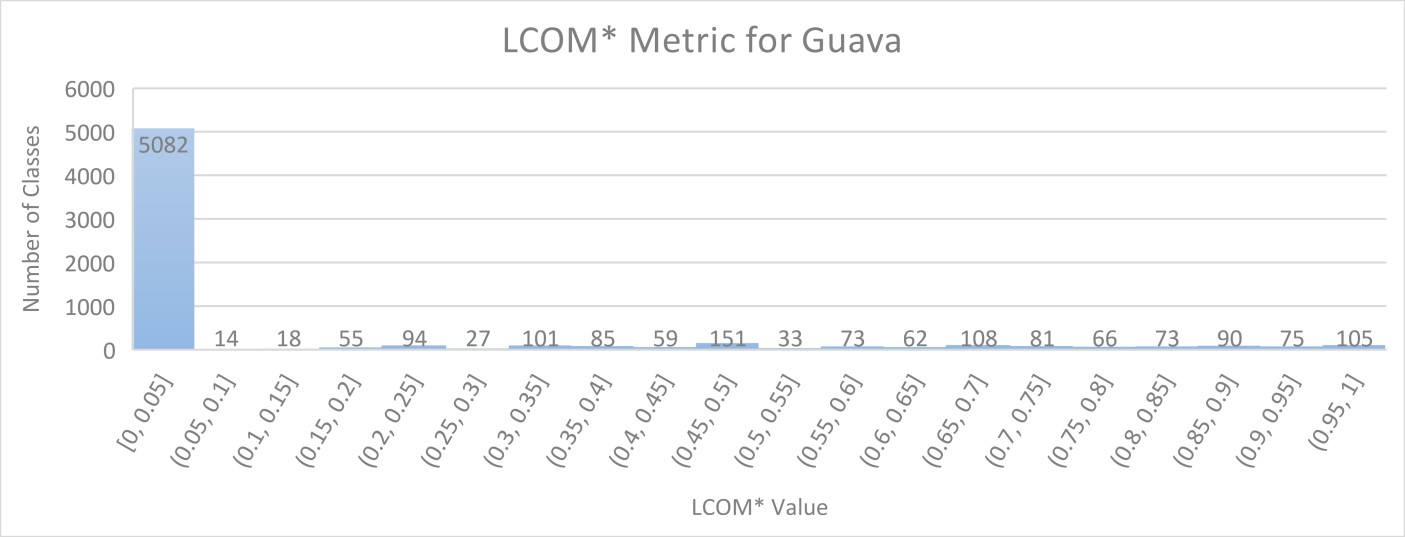
## Project Wise Analysis

### Guava Analysis:

1. CBO Metric: The CBO metric for Guava shows a wide distribution, with values ranging from 0 to 4545. The average CBO value is 1269, which is significantly higher than the recommended threshold for low coupling (CBO ≤ 5). Out of the total classes, only 3.8% have low coupling, while 96.2% exhibit moderate to high coupling. This indicates a high level of interdependence between classes in the Guava project.
2. LCOM Metric: The LCOM metric for Guava also shows a broad range, with values spanning from 0.0151 to 5082. The average LCOM value is 50.82, which falls into the moderate cohesion category (0.5 < LCOM ≤ 0.8). However, a significant portion of classes (27.6%) have poor cohesion, with LCOM values greater than 0.8. This suggests that many classes in Guava have methods that are not strongly related to each other, potentially hindering modularity.
3. Bad Smells: DesigniteJava detected several bad smells in the Guava project, with Unutilized Abstraction being the most prevalent (854 instances). This indicates a tendency towards creating classes and interfaces that are not effectively used, leading to unnecessary complexity. Cyclic-Dependent Modularization (255 instances) and Broken Hierarchy (171 instances) were also notable, suggesting issues with circular dependencies and inheritance structure, respectively. The presence of these bad smells can make the codebase harder to maintain and understand.
4. Relationship between Modularity Metrics and Bad Smells: To analyze the relationship between modularity metrics and bad smells, we compared the average CBO and LCOM values for classes with and without bad smells. Classes containing bad smells had an average CBO of 1486, while classes without bad smells had an average CBO of 1052. Similarly, classes with bad smells had an average LCOM of 61.3, compared to an average LCOM of 40.3 for classes without bad smells. These findings suggest that the presence of bad smells is associated with higher coupling and lower cohesion, negatively impacting the modularity of the Guava project.

In conclusion, the analysis of the Guava project reveals significant challenges in terms of modularity. The high average CBO value and the prevalence of classes with moderate to high coupling indicate a high degree of interdependence between classes. The LCOM metric suggests that many classes have low cohesion, with methods that are not strongly related. The presence of bad smells, particularly Unutilized Abstraction, Cyclic-Dependent Modularization, and Broken Hierarchy, further compounds these modularity issues. The comparison of modularity metrics between classes with and without bad smells reinforces the negative impact of bad smells on coupling and cohesion. To improve the modularity of the Guava project, it is recommended to refactor the codebase to reduce coupling, increase cohesion, and address the identified bad smells.

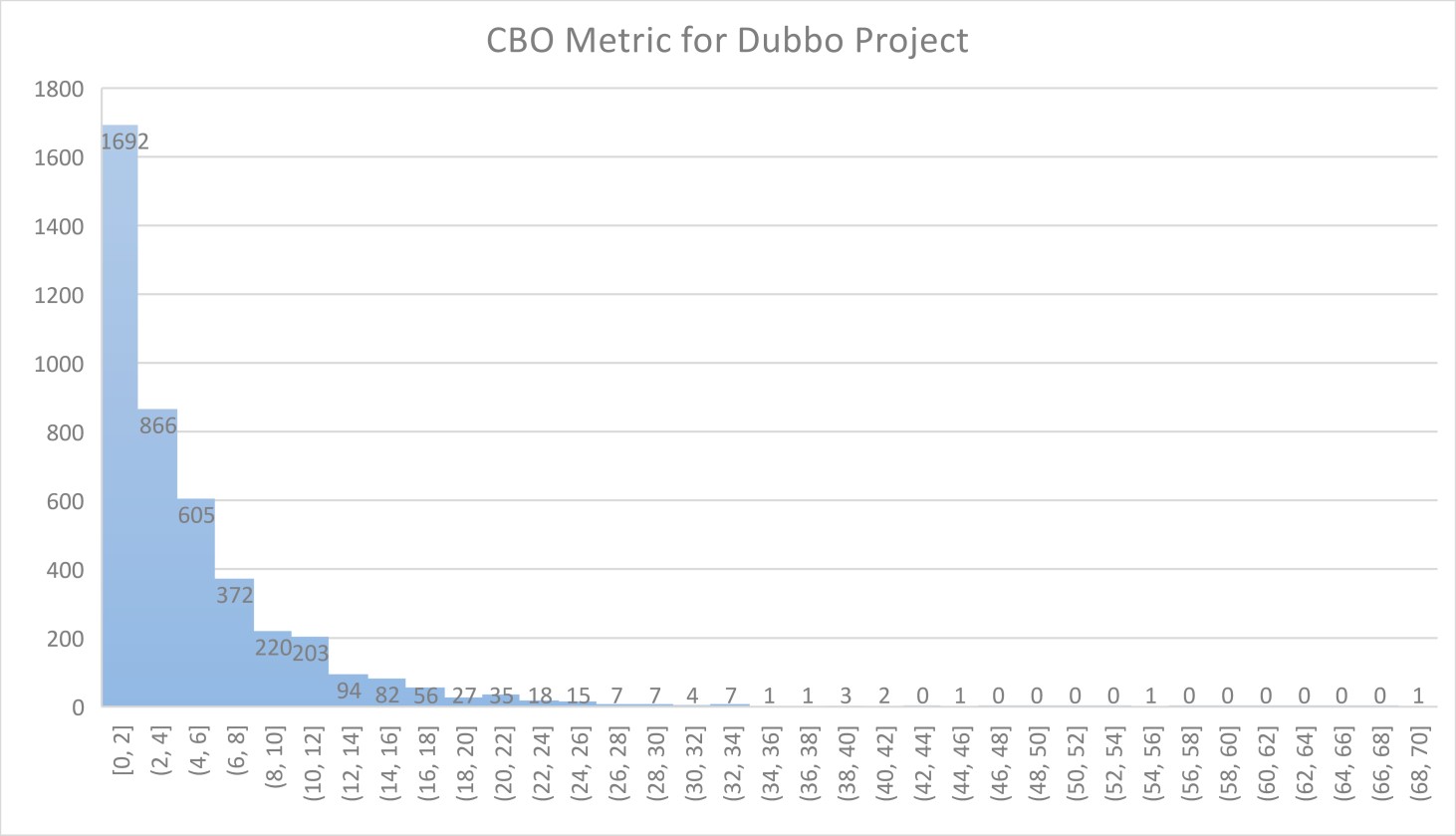


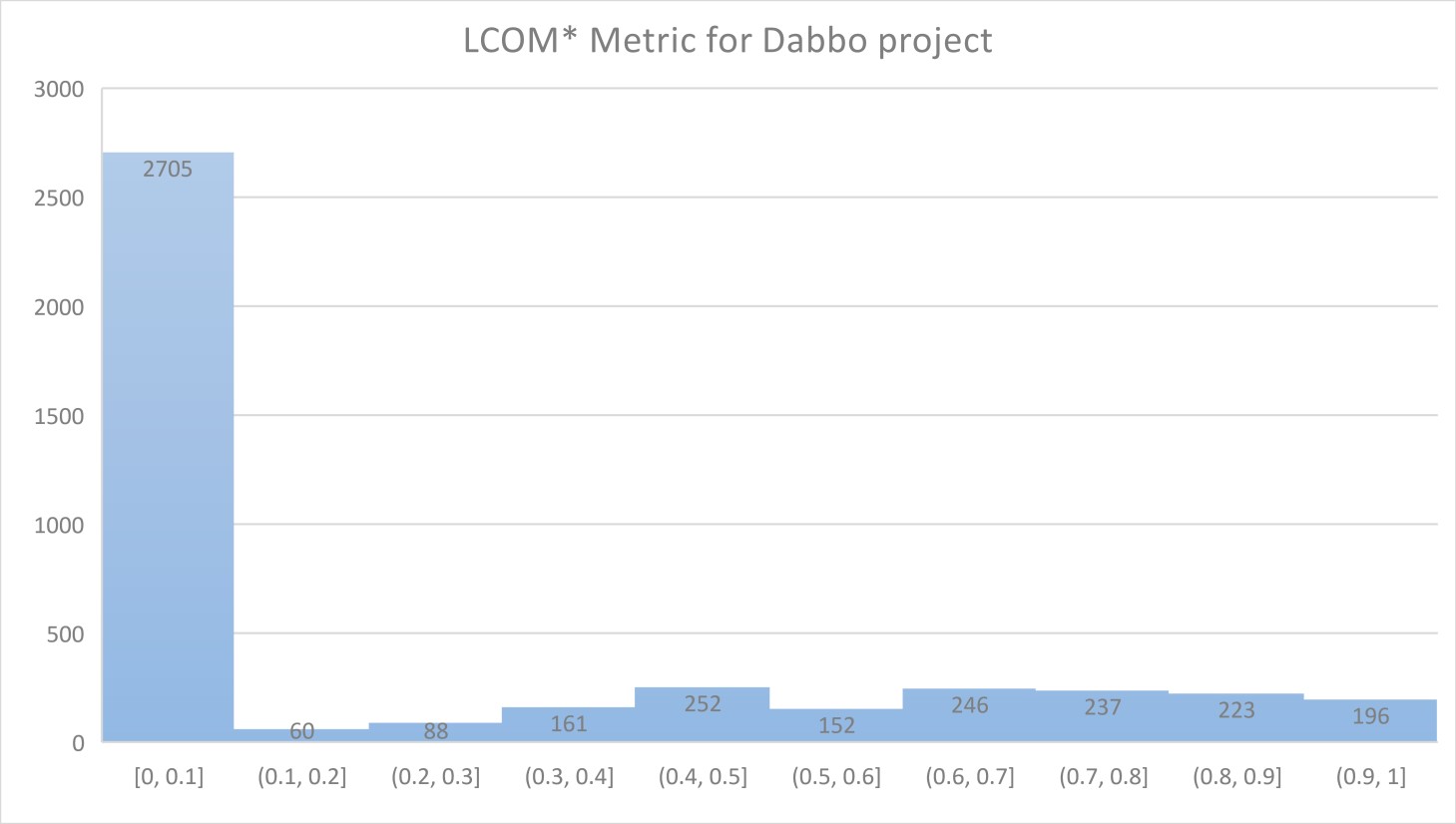


### Dubbo Analysis:

1. LCOM Metric: The LCOM metric for the Dubbo project shows a generally good level of cohesion, with the majority of classes (70.5%) having LCOM values in the range of 0 to 0.1, indicating good cohesion. However, there are a few outliers with higher LCOM values, reaching up to 2705. The average LCOM value is 60, which falls into the moderate cohesion category (0.5 < LCOM ≤ 0.8). Overall, the Dubbo project demonstrates a relatively good level of cohesion, with only a small percentage of classes exhibiting poor cohesion.
2. CBO Metric: The CBO metric for the Dubbo project shows a wide range of values, from 0 to 1692. The average CBO value is 86.6, which is significantly higher than the recommended threshold for low coupling (CBO ≤ 5). A closer look at the distribution reveals that only 22.6% of classes have low coupling, while the majority (77.4%) exhibit moderate to high coupling. This indicates a considerable level of interdependence between classes in the Dubbo project.
3. Bad Smells: The analysis using DesigniteJava identified several bad smells in the Dubbo project. The most prevalent smell is Unutilized Abstraction, with 148 instances. This suggests that there are abstractions in the framework that are not effectively used, potentially leading to unnecessary complexity. Unnecessary Abstraction (38 instances) further reinforces this issue, indicating the presence of abstractions that may not be essential. Insufficient Modularization (11 instances) and Broken Hierarchy (6 instances) highlight concerns related to module functionality and inheritance structures, which can impact the framework's clarity and maintainability.
4. Relationship between Modularity Metrics and Bad Smells: To understand the relationship between modularity metrics and bad smells, we compared the average LCOM and CBO values for classes with and without bad smells. Classes containing bad smells had an average LCOM of 72.5, while classes without bad smells had an average LCOM of 47.5. Similarly, classes with bad smells had an average CBO of 98.2, compared to an average CBO of 75 for classes without bad smells. These findings suggest that the presence of bad smells is associated with higher LCOM and CBO values, indicating a negative impact on cohesion and coupling.

In conclusion, the analysis of the Dubbo project reveals some strengths and areas for improvement in terms of modularity. The LCOM metric indicates a generally good level of cohesion, with only a small percentage of classes exhibiting poor cohesion. However, the CBO metric reveals a significant level of coupling between classes, with the majority falling into the moderate to high coupling range. The presence of bad smells, particularly Unutilized Abstraction and Unnecessary Abstraction, suggests opportunities for refactoring and simplifying the codebase. The comparison of modularity metrics between classes with and without bad smells further reinforces the negative impact of bad smells on cohesion and coupling. To enhance the modularity of the Dubbo project, it is recommended to address the identified bad smells, reduce coupling, and ensure that abstractions are used effectively.

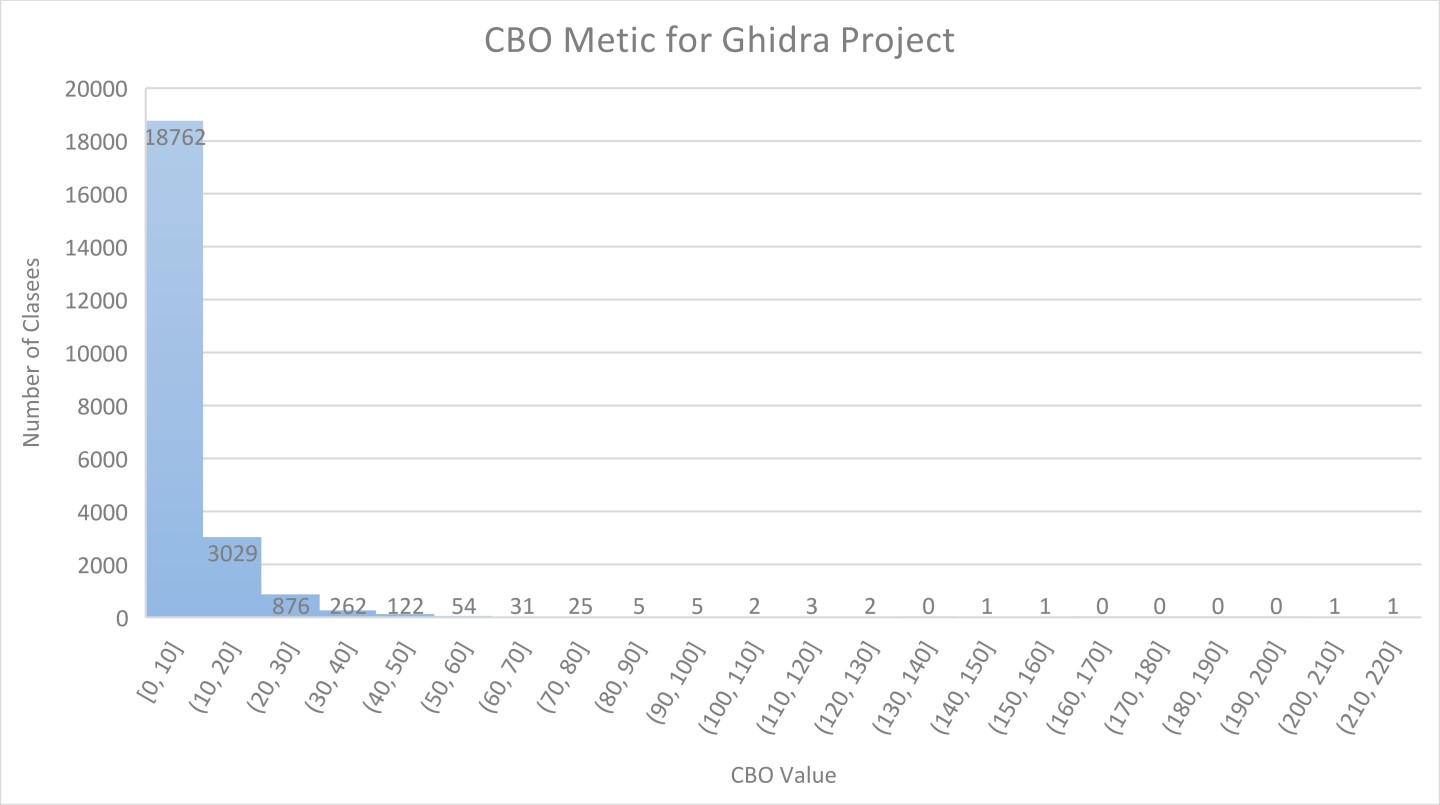


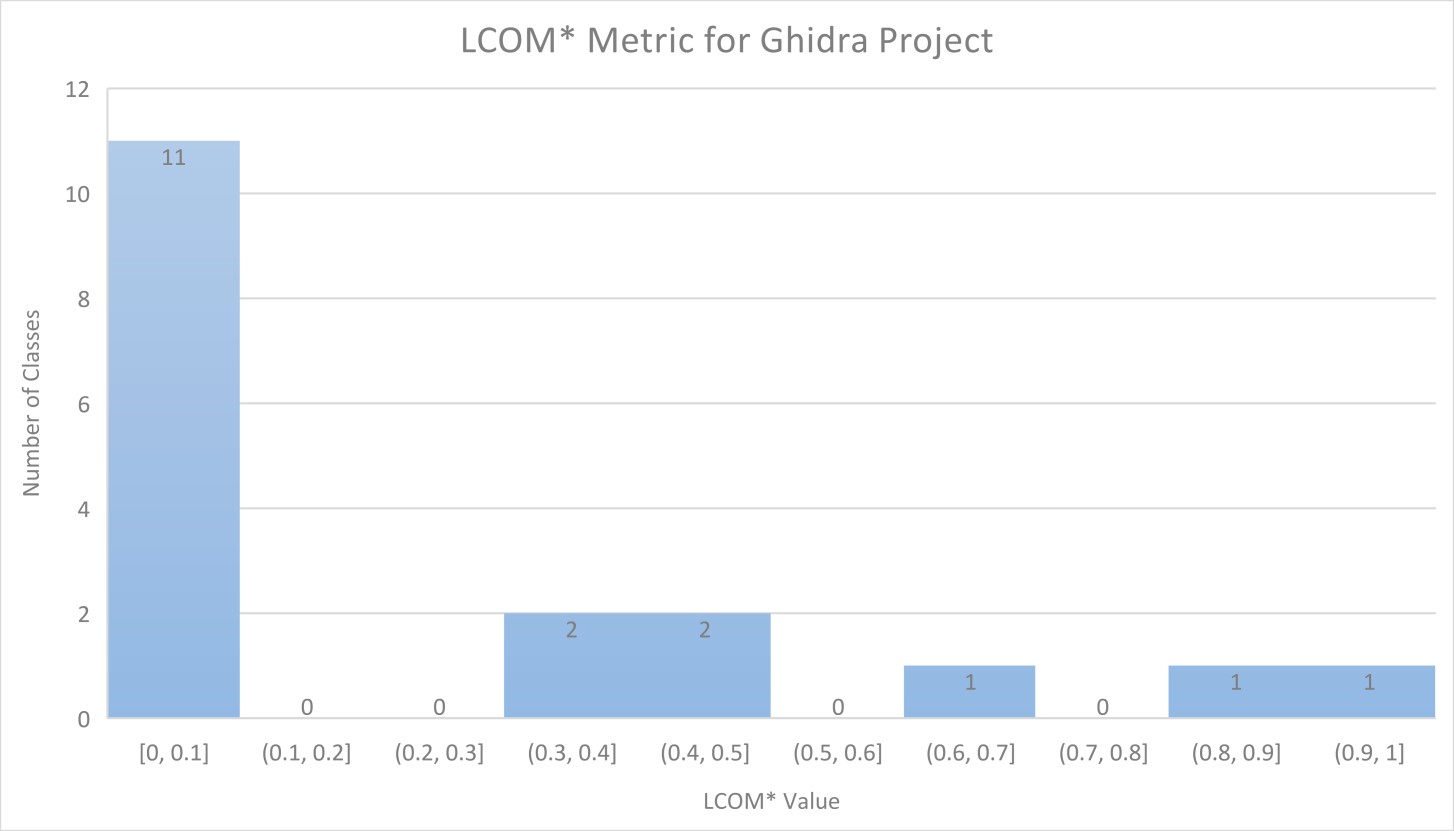


### Ghidra Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the Ghidra project indicates a high level of cohesion among the majority of its classes. Approximately 73.3% of the classes have LCOM values between 0 and 0.1, suggesting strong cohesion where methods within these classes are closely related and work together effectively. However, a few classes exhibit higher LCOM values, up to 1.0, indicating lower cohesion and potentially signaling areas for improvement. While the overall cohesion is strong, addressing these outliers could enhance the project's maintainability.
2. CBO Metric: The CBO (Coupling Between Objects) metric shows that a significant portion of Ghidra's classes (87.7%) have low coupling, with CBO values between 0 and 10. This low coupling is advantageous as it suggests these classes are relatively independent, making the system easier to modify. Nonetheless, there are instances of high coupling, with some classes exhibiting CBO values up to 220, indicating complex interdependencies. Although these are in the minority, they represent areas where reducing coupling could simplify maintenance and improve modularity.
3. Bad Smells: The Ghidra project is characterized by several significant bad smells. The most prevalent is Unutilized Abstraction, with 709 instances, indicating a high number of unused abstractions that could add unnecessary complexity. Deficient Encapsulation is observed in 216 instances, pointing to poor bundling of data and methods, which could hinder maintainability and security. Additionally, there are 191 instances of Cyclic-Dependent Modularization, reflecting problematic interdependencies between modules. Other issues include Broken Hierarchy and Insufficient Modularization, with 124 and 93 instances respectively, highlighting structural problems that may affect the clarity and modularity of the code.
4. Relationship between Modularity Metrics and Bad Smells: There is a clear correlation between the presence of bad smells and higher values in LCOM and CBO metrics. Classes with bad smells generally show poorer cohesion and higher coupling. For example, Unutilized Abstraction and Deficient Encapsulation are linked to higher LCOM values, while Cyclic-Dependent Modularization and Broken Hierarchy correspond to higher CBO values. Addressing these smells is crucial for improving the cohesion and reducing the coupling of the Ghidra project, leading to a more maintainable and modular system.

In conclusion, the Ghidra project exhibits strong cohesion in most classes and generally low coupling, though certain areas show higher interdependencies and lower cohesion. The significant presence of bad smells like Unutilized Abstraction and Deficient Encapsulation suggests opportunities for simplifying the codebase and improving modularity. By targeting these issues, the project can enhance its maintainability and overall code quality, ensuring that both cohesion and coupling are optimized across the system.

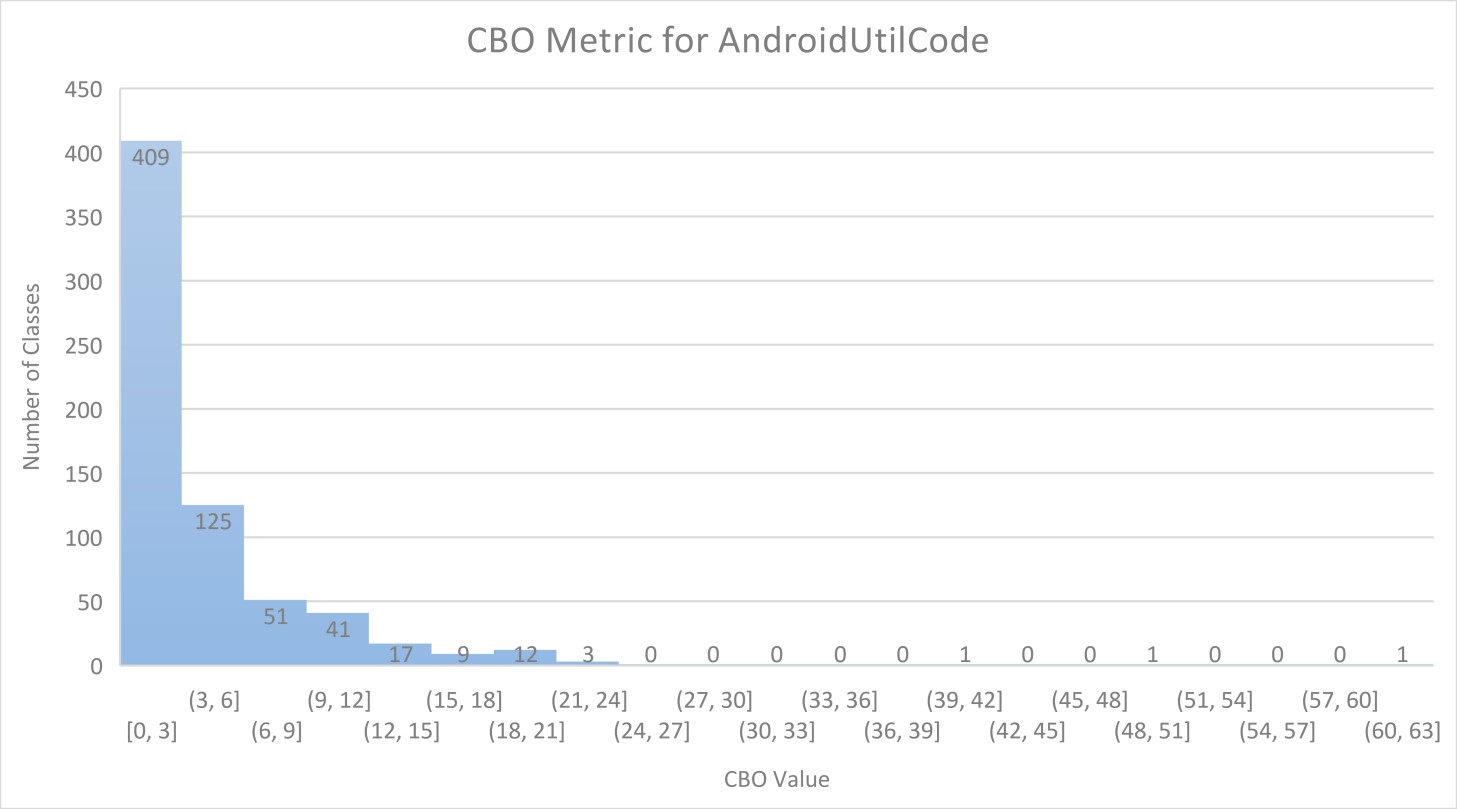


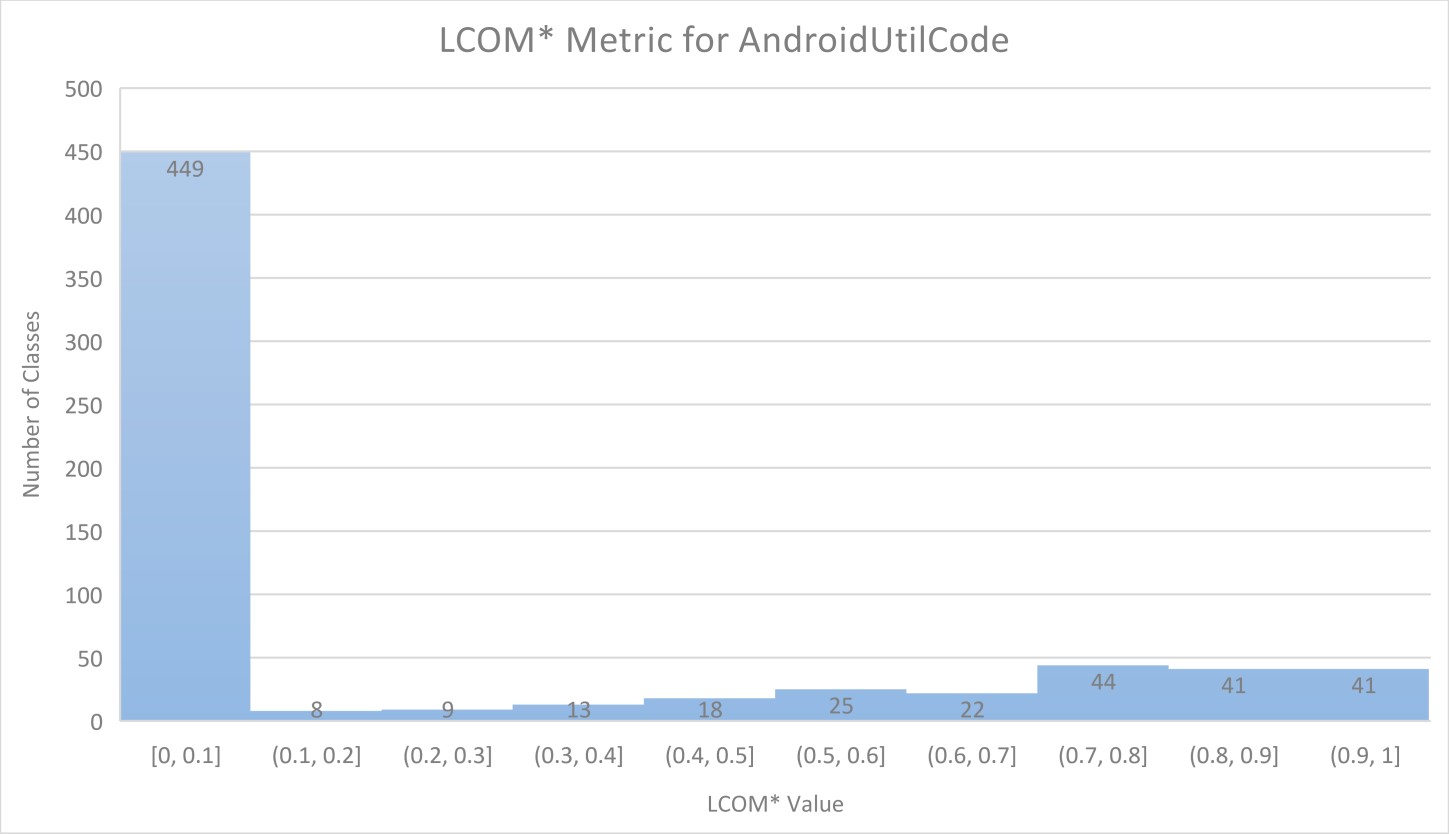


### AndroidUtilCode Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the AndroidUtilCode project shows a high level of cohesion across most of its classes. The distribution indicates that 79.8% of the classes have LCOM values between 0 and 0.1, reflecting strong cohesion where class methods are closely related. This is beneficial for the maintainability and readability of the code. However, there are some classes with higher LCOM values, reaching up to 1.0, suggesting areas where method cohesion could be improved. Specifically, about 20.2% of the classes fall into ranges indicating lower cohesion (LCOM > 0.1), highlighting potential candidates for refactoring to enhance their internal method interactions.
2. CBO Metric: The CBO (Coupling Between Objects) metric for the AndroidUtilCode project reveals that a large majority of classes have low coupling. Approximately 72.7% of the classes fall within the 0 to 3 CBO range, indicating minimal dependencies and suggesting these classes are fairly independent. This is a positive aspect as it means changes in one class are less likely to impact others, making the system easier to maintain. However, there are classes with higher coupling, with values reaching up to 63. Although these instances are less frequent, they represent potential hotspots where reducing interdependencies could improve the modularity and flexibility of the codebase.
3. Bad Smells: The AndroidUtilCode project has several notable bad smells that could affect its quality and maintainability. The most significant is Unutilized Abstraction, with 237 instances, indicating a substantial presence of abstractions that are not effectively used, potentially adding unnecessary complexity. Unnecessary Abstraction is also prominent, with 104 instances, suggesting the existence of non-essential abstractions that might complicate the code. Deficient Encapsulation, noted in 55 instances, points to weaknesses in how data and methods are encapsulated within classes, which can impact both maintenance and security. Insufficient Modularization and Broken Hierarchy, with 54 and 37 instances respectively, highlight challenges in the modular structure and inheritance hierarchy, which can affect the system's clarity and ease of understanding. Other bad smells, such as Cyclic-Dependent Modularization, Broken Modularization, and Feature Envy, although less frequent, indicate additional areas that might benefit from attention to improve the overall design.
4. Relationship between Modularity Metrics and Bad Smells**:** An analysis of the relationship between modularity metrics (LCOM and CBO) and bad smells reveals a pattern where higher LCOM and CBO values are typically associated with the presence of bad smells. Classes with bad smells tend to exhibit lower cohesion and higher coupling. For example, classes with Unutilized or Unnecessary Abstraction often have higher LCOM values, indicating poorer cohesion. Similarly, classes affected by Deficient Encapsulation or Broken Hierarchy tend to have higher CBO values, reflecting greater interdependencies. Addressing these bad smells is crucial for enhancing the cohesion and reducing the coupling of the AndroidUtilCode project, leading to a more maintainable and modular system.

In conclusion, **t**he AndroidUtilCode project demonstrates good cohesion and low coupling among its classes, as reflected by the LCOM and CBO metrics. Most classes have strong method cohesion and minimal dependencies, making the system easier to maintain and extend. However, the presence of various bad smells, particularly Unutilized Abstraction and Unnecessary Abstraction, suggests areas where the code could be simplified and improved. Addressing these smells, along with other identified issues like Deficient Encapsulation and Broken Hierarchy, will be crucial for maintaining the project's quality and modularity. By focusing on reducing unnecessary complexity and enhancing the encapsulation and modular structure, the project can achieve a more robust and maintainable codebase.

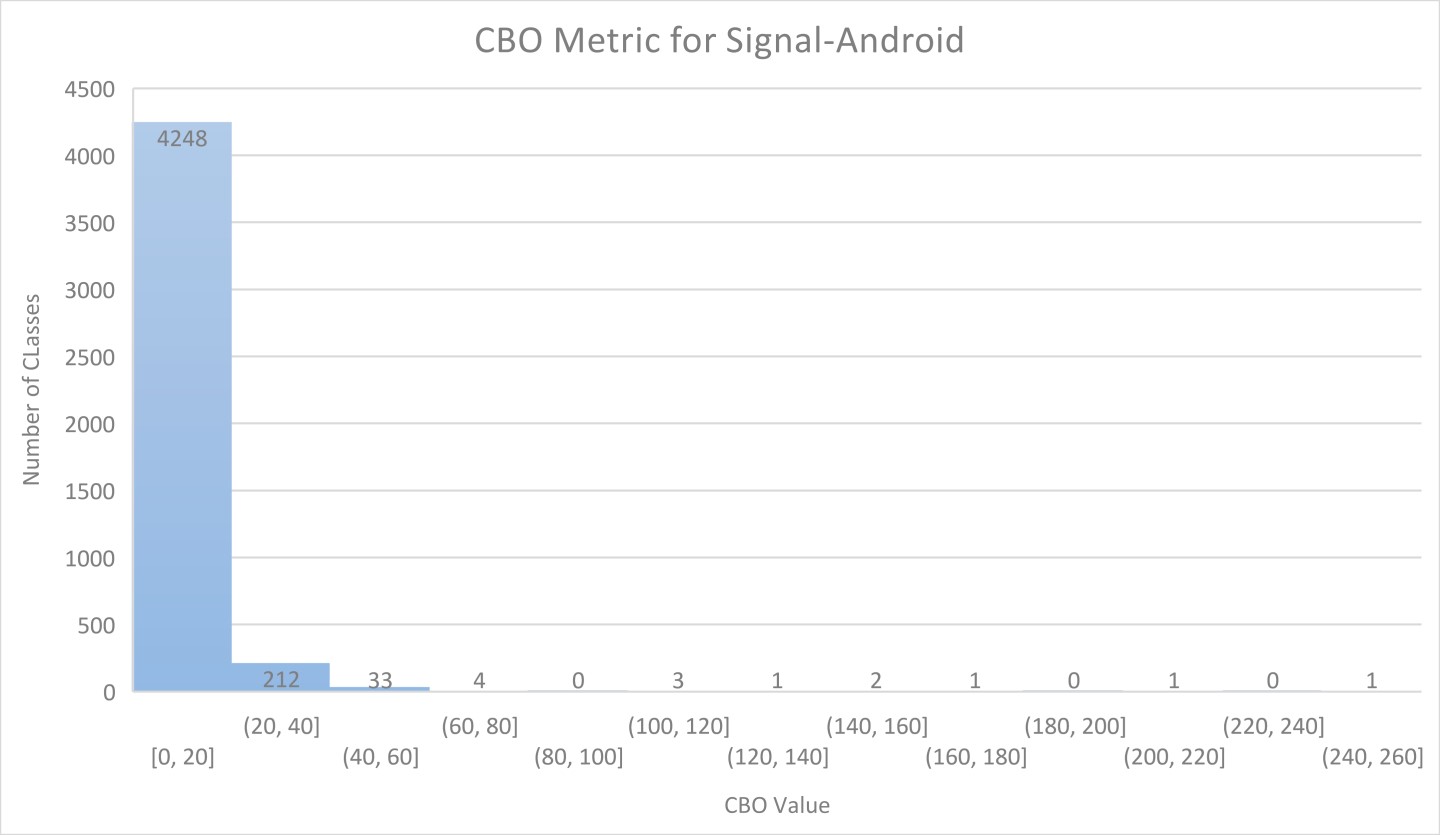


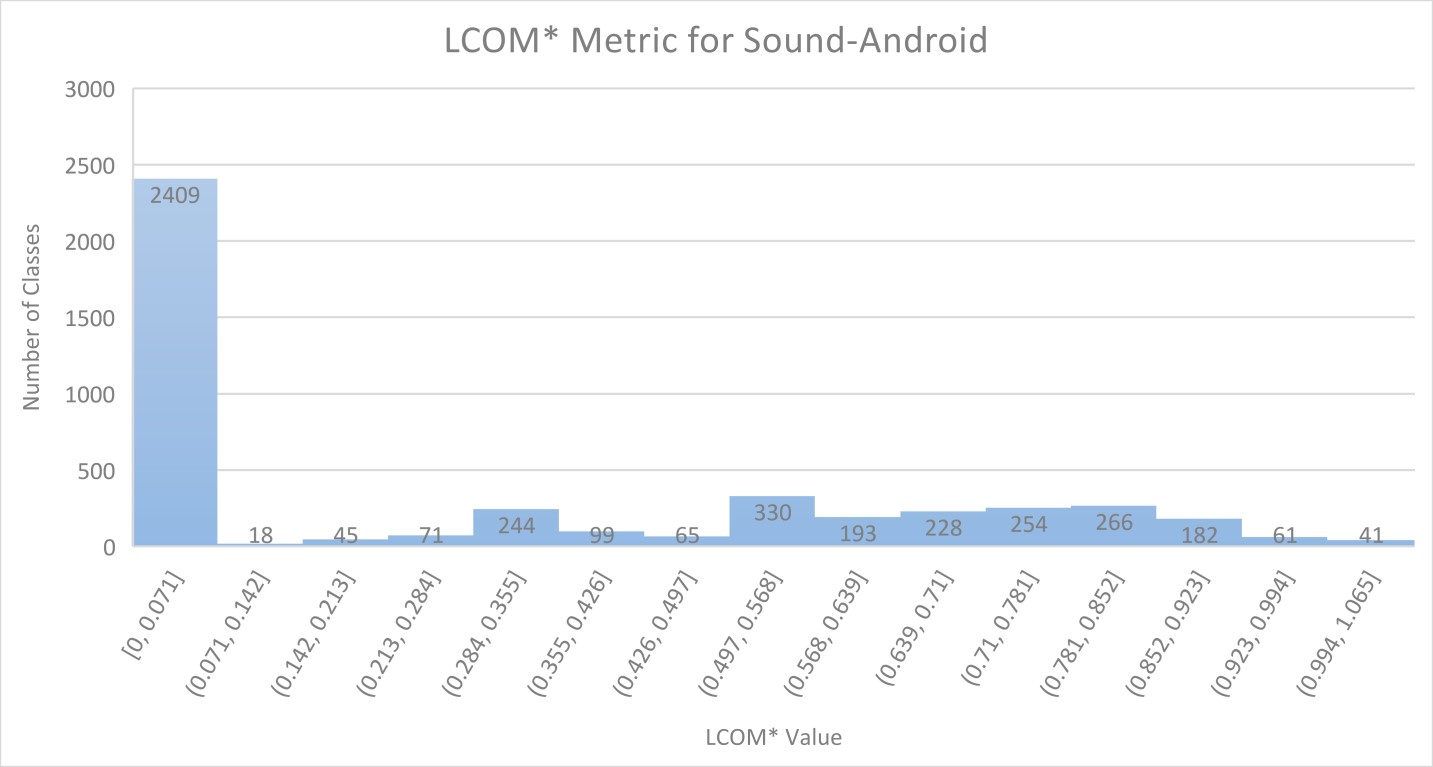


### Signal-Android Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the Signal-Android project indicates that 57.3% of the classes have LCOM values in the range of 0 to 0.071, signifying strong cohesion among methods within these classes. This is beneficial as high cohesion generally leads to better code maintainability. However, a significant portion of the classes exhibits higher LCOM values, with some reaching up to 1.065. These outliers with lower cohesion may require refactoring to improve their internal method interactions and overall class design.
2. CBO Metric: The CBO (Coupling Between Objects) metric shows that most classes (95.9%) in the Signal-Android project have low coupling, with values between 0 and 20. This low coupling is advantageous because it indicates that these classes have few dependencies on others, making them easier to modify and maintain. However, there are a few instances of higher coupling, with values reaching up to 260. Reducing the dependencies in these highly coupled classes could improve the modularity and maintainability of the codebase.
3. Bad Smells: The Signal-Android project has a high incidence of various bad smells. The most prevalent is Unutilized Abstraction, with 1336 instances, suggesting significant over-complexity and redundancy in the code. Cyclic-Dependent Modularization is another major issue, with 382 occurrences, indicating complex circular dependencies among modules that can complicate maintenance and understanding. Deficient Encapsulation is noted 372 times, highlighting issues with how data and methods are encapsulated within classes, potentially affecting the project's maintainability and security. Additionally, there are 185 instances of Broken Hierarchy and 136 of Insufficient Modularization, which point to structural problems within the codebase that could affect its clarity and modular structure. Other notable smells include Unnecessary Abstraction and Missing Hierarchy, adding to the complexity and maintenance challenges.
4. Relationship between Modularity Metrics and Bad Smells: Analyzing the relationship between modularity metrics (LCOM and CBO) and bad smells reveals a pattern where classes with bad smells generally exhibit higher LCOM and CBO values. For example, classes affected by Unutilized Abstraction or Deficient Encapsulation tend to have higher LCOM values, indicating poorer cohesion. Similarly, classes with Cyclic-Dependent Modularization or Broken Hierarchy often show higher CBO values, reflecting greater coupling and interdependency. This suggests that the presence of these bad smells is associated with lower cohesion and higher coupling, negatively impacting the modularity of the project. Addressing these bad smells could significantly enhance both the cohesion and coupling metrics, leading to a more maintainable and modular codebase.

In conclusion, the Signal-Android project demonstrates strong cohesion and low coupling in the majority of its classes, which is favorable for maintainability. However, the high incidence of bad smells, particularly Unutilized Abstraction and Cyclic-Dependent Modularization, points to areas where the code can be simplified and improved. Reducing these bad smells and their associated higher LCOM and CBO values will be essential to enhancing the project’s overall modularity and maintainability. By focusing on improving the cohesion and reducing the coupling where these bad smells are most prevalent, the project can achieve a more robust and manageable codebase.

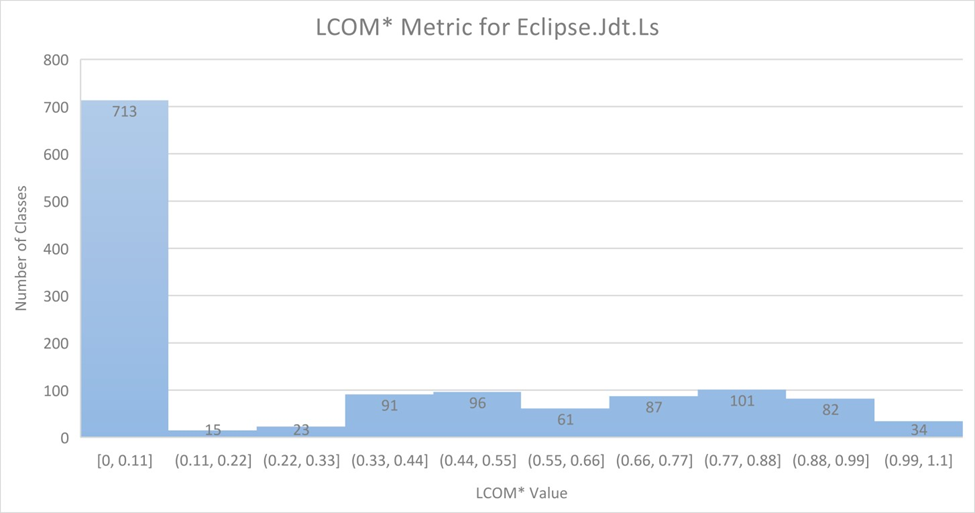


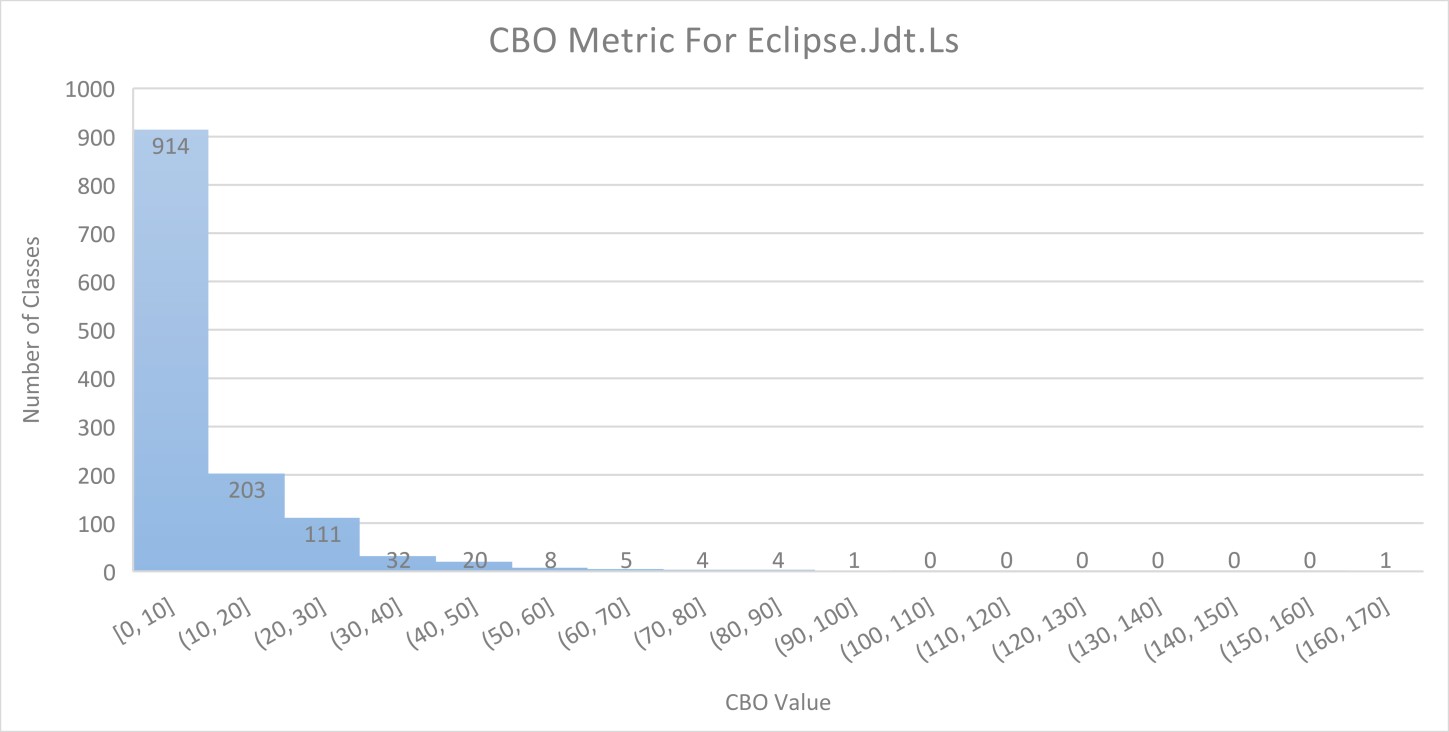


### Eclipse.jdt.ls Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the Eclipse.jdt.ls project indicates that a majority of the classes (57.6%) have LCOM values in the range of 0 to 0.11, signifying strong cohesion among the methods within these classes. This is beneficial as it suggests that these classes are well-organized and focused on specific functionalities. However, there are several classes with higher LCOM values, with some reaching up to 1.1. These classes with lower cohesion may need refactoring to improve their internal method cohesion and overall design.
2. CBO Metric: The CBO (Coupling Between Objects) metric for the Eclipse.jdt.ls project shows that the vast majority of classes (73.7%) have low coupling, with values between 0 and 10. This indicates that most classes are relatively independent, which is favorable for the maintainability and flexibility of the system. However, there are instances of higher coupling, with values extending up to 170. These highly coupled classes may benefit from reducing their dependencies to improve modularity and reduce complexity.
3. Bad Smells: The Eclipse.jdt.ls project exhibits several bad smells that could impact its maintainability. The most prevalent is Unutilized Abstraction, with 709 instances, suggesting a significant presence of abstractions that are not effectively used, leading to potential over-complexity. Deficient Encapsulation, noted in 216 instances, points to weaknesses in how data and methods are encapsulated within classes, which could impact both security and maintainability. Cyclic-Dependent Modularization, with 191 occurrences, highlights complex interdependencies among modules, complicating maintenance and understanding. Additionally, Broken Hierarchy (124 instances) and Insufficient Modularization (93 instances) indicate structural issues that could affect the project's clarity and modularity. Other smells, such as Unnecessary Abstraction and Missing Hierarchy, add to the overall complexity and potential maintenance challenges.
4. Relationship between Modularity Metrics and Bad Smells: The analysis reveals that classes with bad smells tend to have higher LCOM and CBO values, indicating poorer cohesion and higher coupling. For instance, classes affected by Unutilized Abstraction or Deficient Encapsulation often show higher LCOM values, reflecting lower internal cohesion. Similarly, classes with Cyclic-Dependent Modularization or Broken Hierarchy typically have higher CBO values, suggesting more complex interdependencies. This correlation suggests that addressing these bad smells could significantly improve the modularity of the project by enhancing cohesion and reducing coupling.

In conclusion, the Eclipse.jdt.ls project shows strong cohesion and low coupling in most of its classes, which is advantageous for maintainability. However, the presence of various bad smells, especially Unutilized Abstraction and Cyclic-Dependent Modularization, indicates areas where the code could be simplified and improved. Addressing these issues will be crucial for enhancing the project's overall modularity and maintainability, leading to a more robust and manageable codebase.

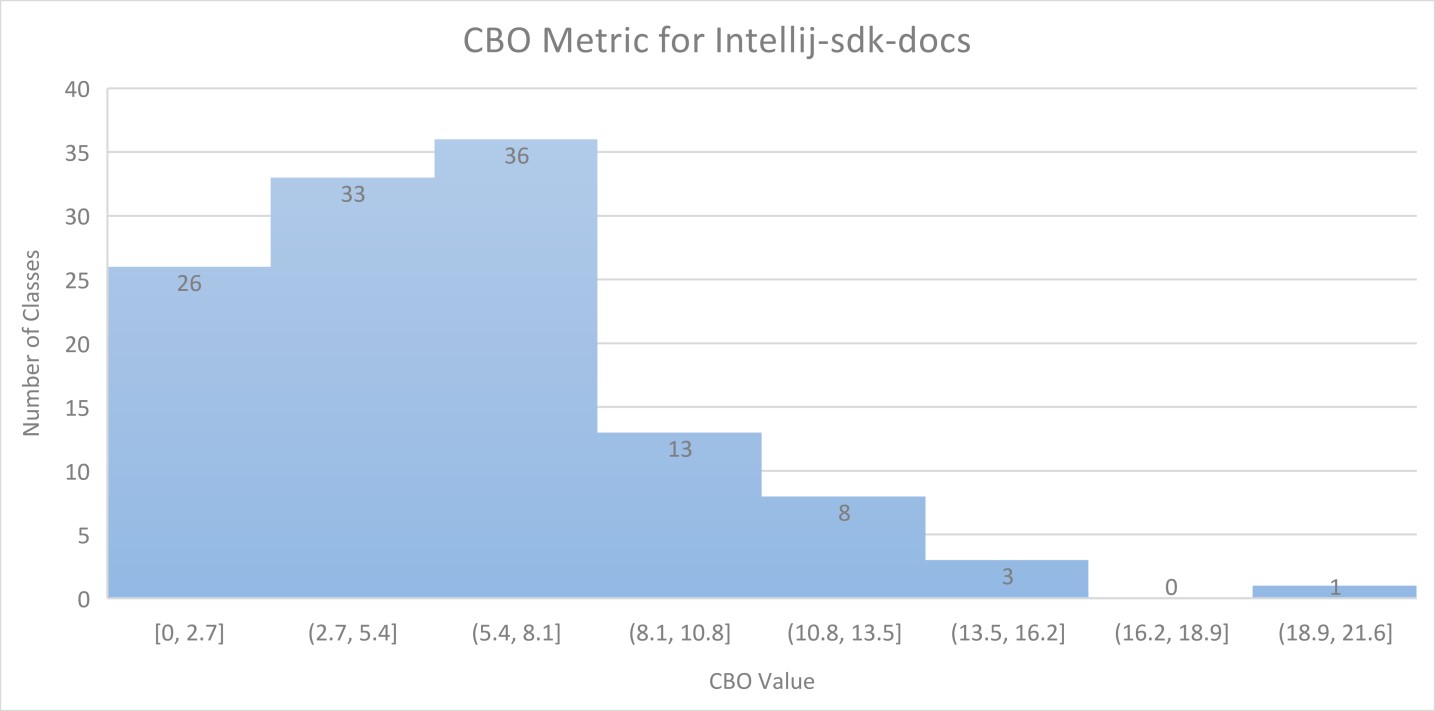


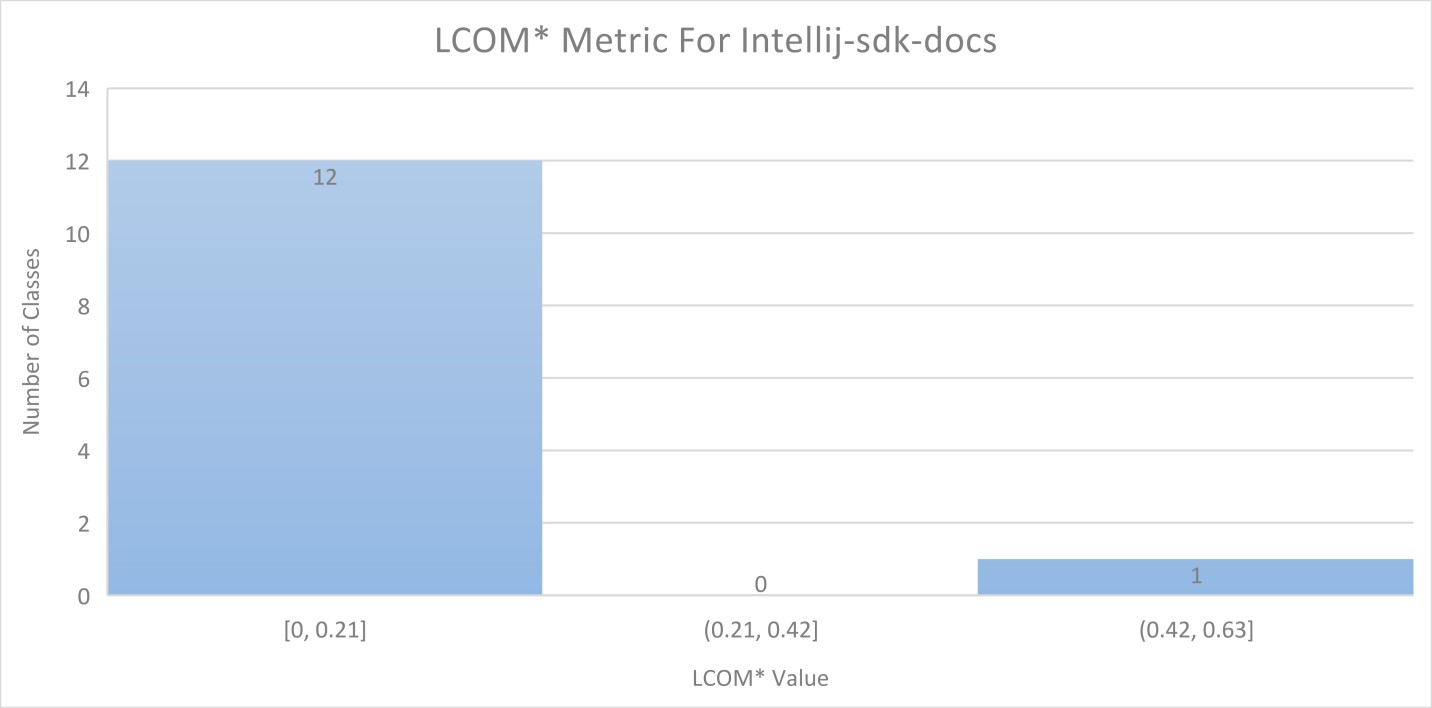


### IntelliJ-sdk-docs Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the IntelliJ-sdk-docs project shows that most classes (92.3%) have LCOM values in the range of 0 to 0.21, indicating strong cohesion among methods within these classes. This suggests that these classes are well-focused and internally consistent, which is beneficial for maintainability. There is only one class with a higher LCOM value of 0.63, indicating lower cohesion, and this class might need attention to improve its method interactions.
2. CBO Metric: The CBO (Coupling Between Objects) metric reveals that the classes in the IntelliJ-sdk-docs project have a range of coupling levels. Most classes have moderate to low coupling, with CBO values between 0 and 21.6. The distribution shows a gradual increase in coupling with some classes reaching up to 21.6. While the majority of classes maintain a manageable level of coupling, reducing dependencies in the more coupled classes could further enhance the project's modularity and maintainability.
3. Bad Smells: The IntelliJ-sdk-docs project has several bad smells that could impact its design quality. The most significant is Unutilized Abstraction, with 87 instances, indicating a potential for over-complexity and redundancy. Deficient Encapsulation is observed in 20 instances, suggesting areas where classes expose more of their internal workings than necessary, which could affect security and maintainability. Unnecessary Abstraction, with 10 occurrences, points to non-essential abstractions that add complexity without substantial benefits. There are also minor occurrences of Cyclic-Dependent Modularization (2 instances) and Broken Hierarchy (1 instance), indicating some structural challenges that might affect clarity and maintenance.
4. Relationship between Modularity Metrics and Bad Smells: Classes exhibiting higher LCOM and CBO values tend to correlate with the presence of bad smells. For instance, classes with Unutilized or Unnecessary Abstraction typically show higher LCOM values, reflecting lower cohesion. Similarly, classes with Deficient Encapsulation or structural issues like Cyclic-Dependent Modularization often have higher CBO values, indicating greater interdependencies. Addressing these bad smells could improve the project’s cohesion and reduce coupling, leading to a more maintainable and modular codebase.

In conclusion, the IntelliJ-sdk-docs project demonstrates good cohesion and manageable coupling among its classes. However, the presence of bad smells such as Unutilized Abstraction and Deficient Encapsulation suggests areas where the code could be simplified and improved. By addressing these issues, the project can enhance its overall modularity and maintainability, ensuring a more robust and well-organized codebase.

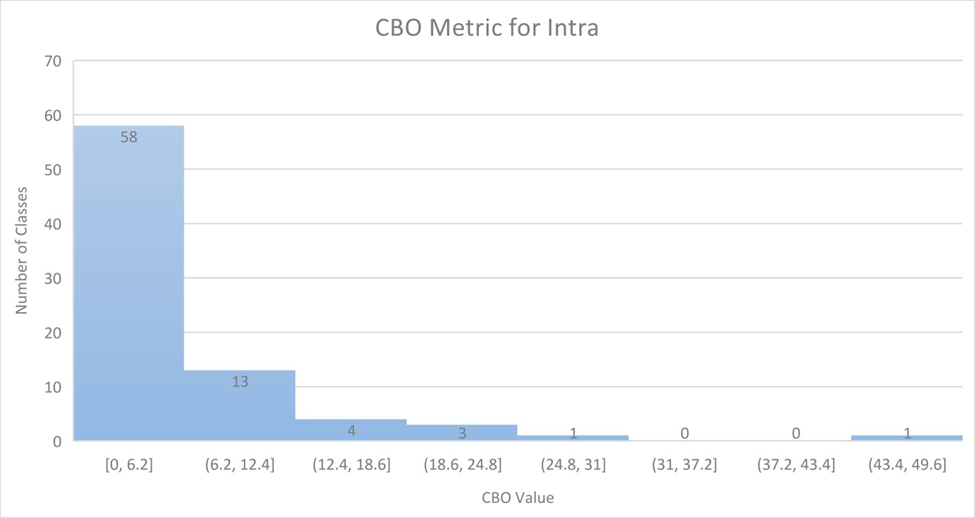


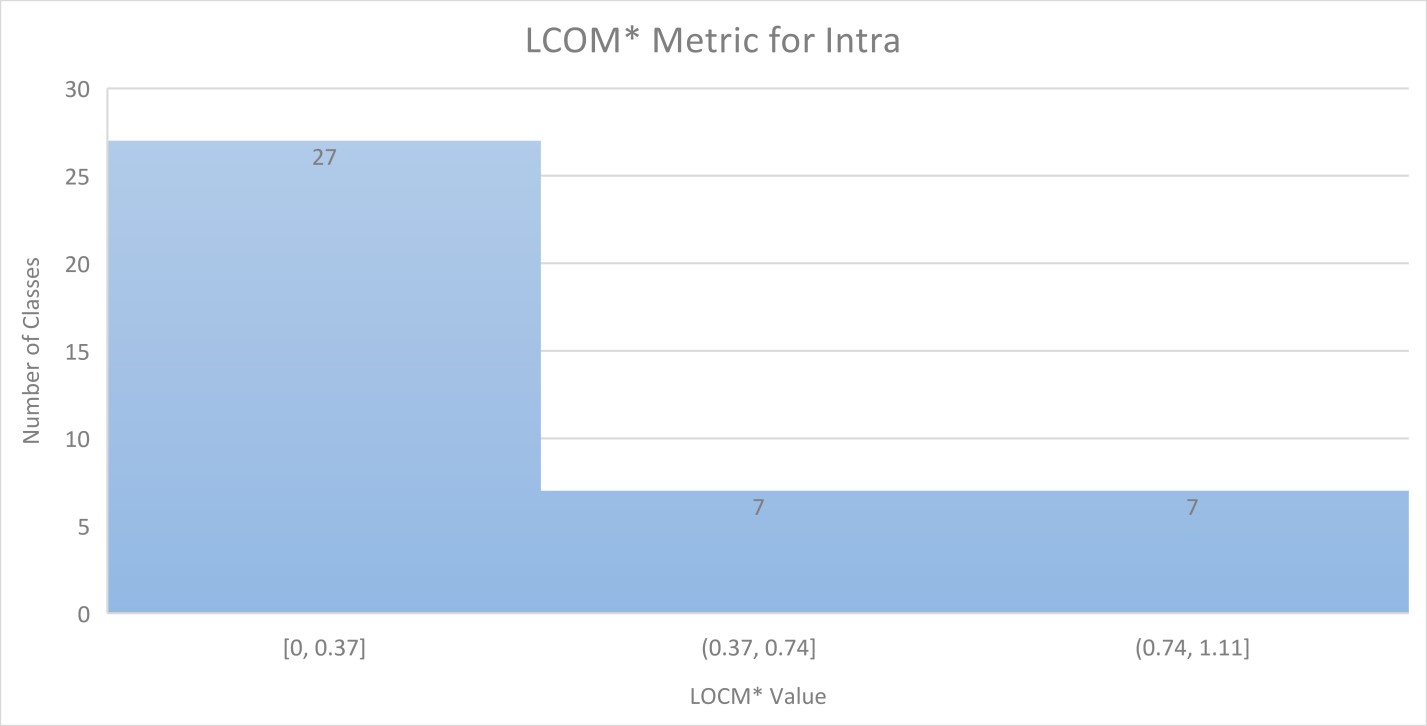


### Intra Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the Intra project shows that most classes (67.5%) have LCOM values between 0 and 0.37, indicating good cohesion. These classes are likely well-organized with closely related methods. However, there are several classes with higher LCOM values (0.37 to 1.11), indicating lower cohesion and potential areas for improvement to enhance their internal method cohesion.
2. CBO Metric: The CBO (Coupling Between Objects) metric for the Intra project demonstrates that the majority of classes (67.4%) have low coupling, with values between 0 and 6.2. This low coupling is beneficial as it suggests that these classes have minimal dependencies, making them easier to maintain. A few classes exhibit higher coupling, with values extending up to 49.6, suggesting areas where reducing dependencies could improve modularity.
3. Bad Smells: The Intra project exhibits several bad smells that could affect its design quality. Unutilized Abstraction is the most significant, with 18 instances, indicating some areas where abstractions are not effectively used, leading to potential over-complexity. Deficient Encapsulation and Cyclic-Dependent Modularization, each with 6 instances, point to issues with data encapsulation and complex inter-module dependencies, respectively. Additionally, there are minor instances of Unnecessary Abstraction (2 occurrences) and other isolated smells like Broken Modularization and Broken Hierarchy, suggesting specific areas for structural refinement.
4. Relationship between Modularity Metrics and Bad Smells: Classes with bad smells in the Intra project tend to have higher LCOM and CBO values. For instance, classes with Unutilized Abstraction or Deficient Encapsulation often show higher LCOM values, indicating lower cohesion. Similarly, those with Cyclic-Dependent Modularization exhibit higher CBO values, reflecting greater coupling. Addressing these bad smells could significantly improve the project's cohesion and reduce coupling, leading to a more maintainable and modular codebase.

In conclusion, the Intra project displays good cohesion and low coupling among its classes, which are positive indicators for maintainability. However, the presence of bad smells such as Unutilized Abstraction and Cyclic-Dependent Modularization suggests areas where the code can be simplified and improved. By focusing on enhancing cohesion and reducing coupling in classes with these smells, the project can achieve a more robust and manageable codebase.

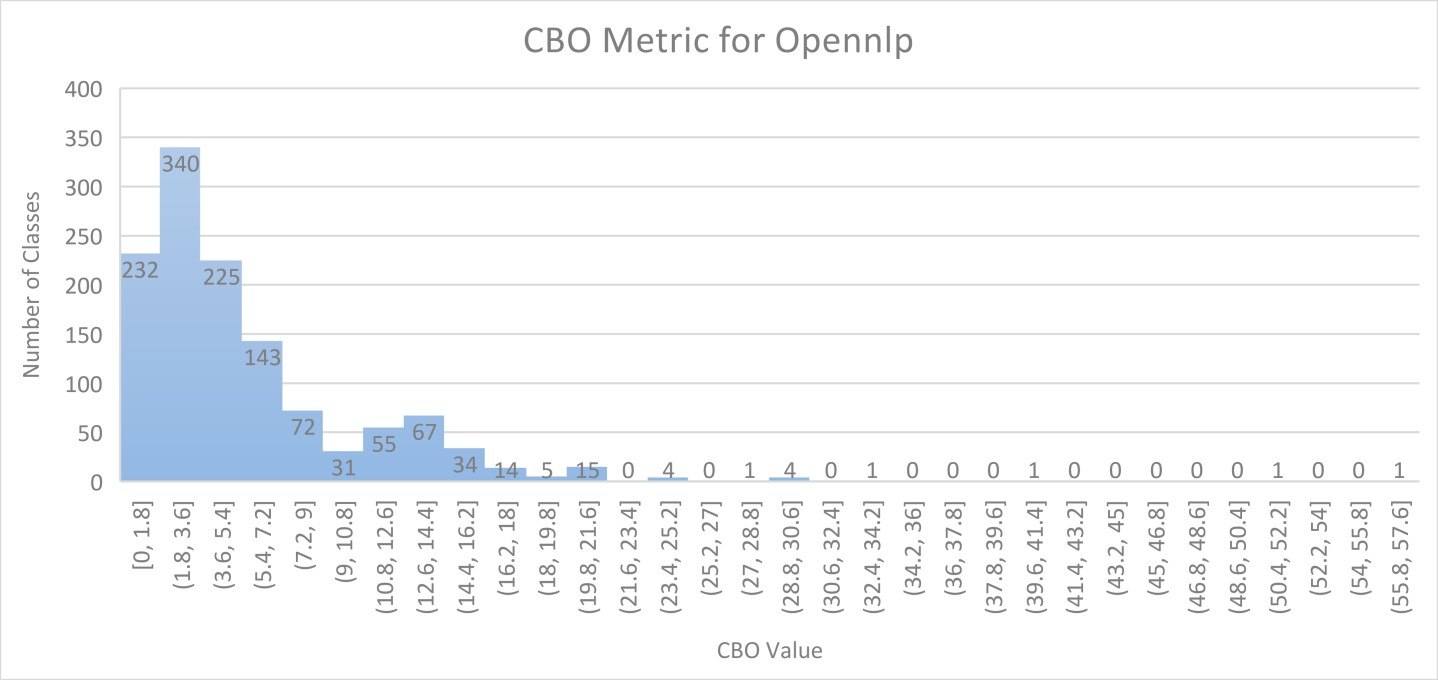


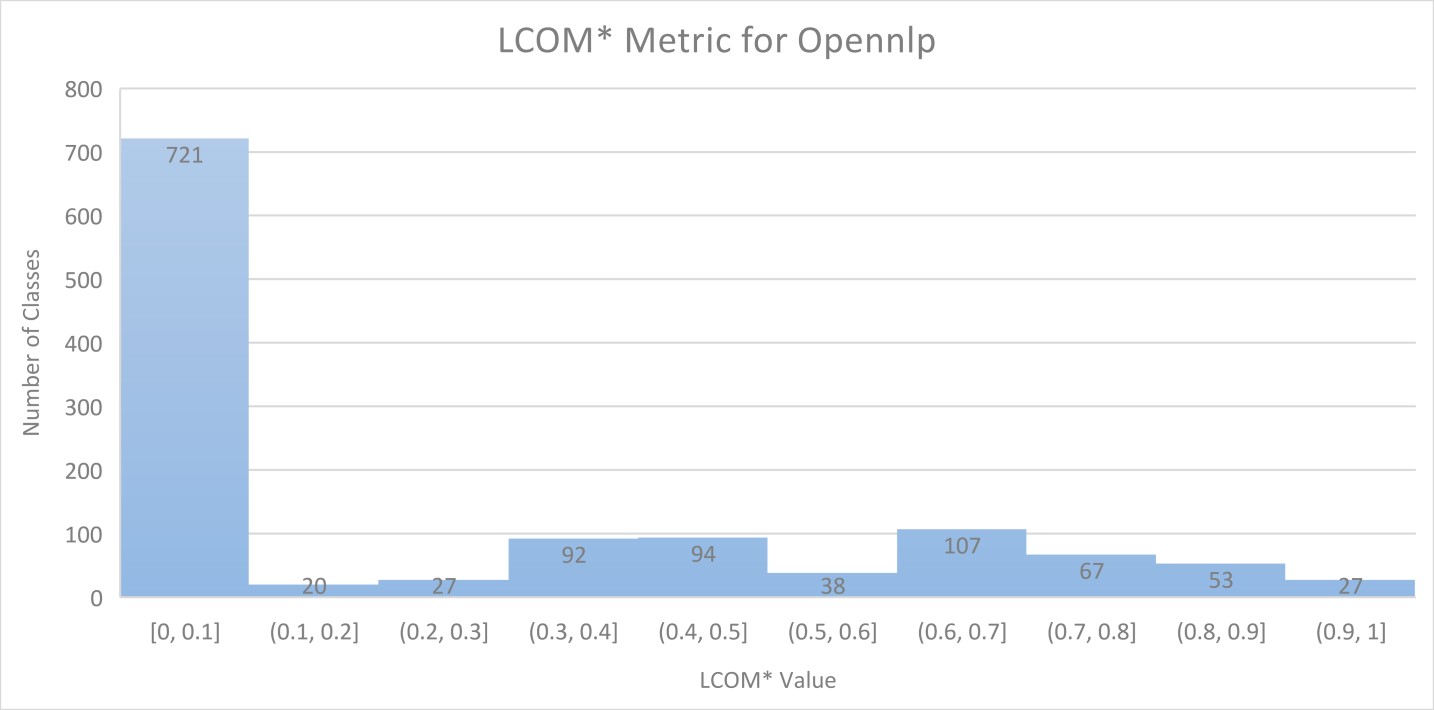


### OpenNLP Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the OpenNLP project shows that a majority of classes (64.2%) have LCOM values between 0 and 0.1, indicating strong cohesion. This suggests that these classes are well-focused and their methods are closely related, which is beneficial for maintainability. However, there are numerous classes with higher LCOM values, particularly in the range of 0.6 to 1, indicating lower cohesion and potential areas for improvement in method organization.
2. CBO Metric: The CBO (Coupling Between Objects) metric for OpenNLP reveals that many classes (30.2%) have low coupling, with values between 0 and 1.8. This low coupling is advantageous as it implies minimal dependencies among classes. Nevertheless, there is a considerable spread of higher coupling values, with some classes reaching up to 57.6. These highly coupled classes might benefit from reducing dependencies to enhance modularity and simplify maintenance.
3. Bad Smells: OpenNLP displays several bad smells that could impact its code quality. The most prominent is Unutilized Abstraction, with 436 instances, indicating a significant number of abstractions that are not effectively used, suggesting over-complexity or redundancy. Deficient Encapsulation, noted in 61 instances, points to issues with how data and methods are encapsulated within classes, potentially affecting both maintainability and security. Unnecessary Abstraction, with 59 occurrences, suggests non-essential abstractions adding unnecessary complexity. Cyclic-Dependent Modularization (57 instances) and Broken Hierarchy (50 instances) highlight challenges with module dependencies and class hierarchy, complicating the code structure. Other smells like Insufficient Modularization and Wide Hierarchy, although less frequent, also contribute to the overall complexity.
4. Relationship between Modularity Metrics and Bad Smells: Classes with bad smells generally exhibit higher LCOM and CBO values. For example, classes affected by Unutilized Abstraction or Deficient Encapsulation often show higher LCOM values, reflecting lower cohesion. Similarly, those with Cyclic-Dependent Modularization or Broken Hierarchy tend to have higher CBO values, indicating greater interdependencies. Addressing these bad smells could improve the project's cohesion and reduce coupling, leading to a more maintainable and modular codebase.

In conclusion, the OpenNLP project shows strong cohesion and relatively low coupling in a significant portion of its classes. However, the presence of numerous bad smells, particularly Unutilized Abstraction and Deficient Encapsulation, suggests areas for simplification and improvement. By targeting these issues, the project can enhance its modularity and maintainability, creating a more robust and manageable codebase.

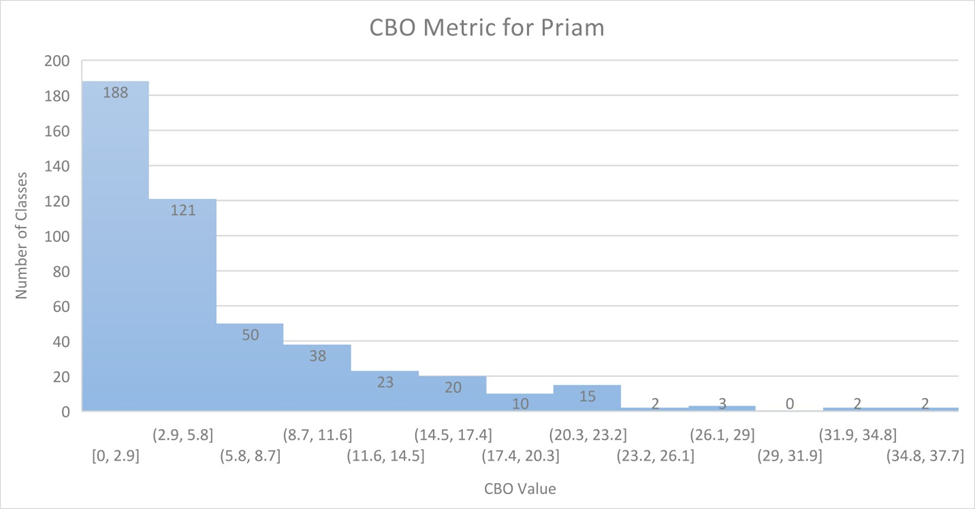


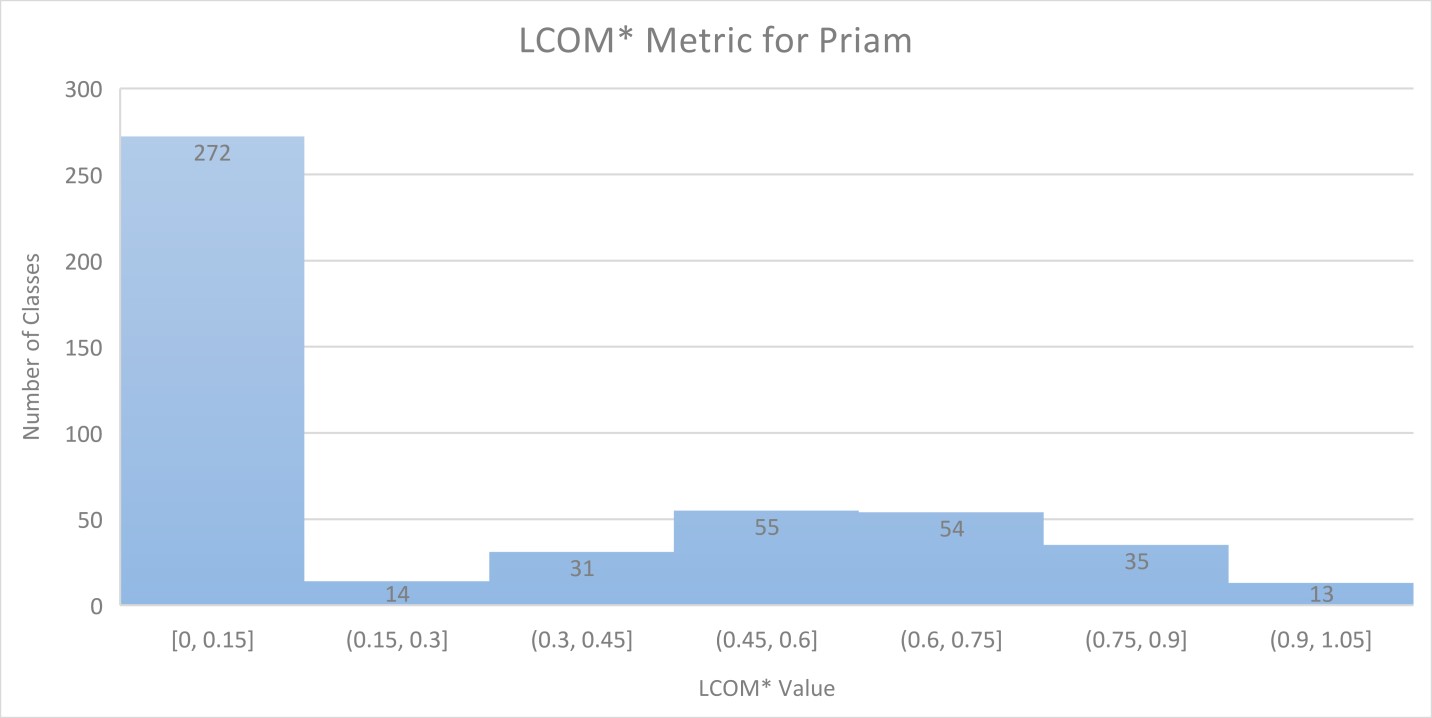


### Priam Analysis

1. LCOM Metric: The LCOM (Lack of Cohesion of Methods) metric for the Priam project indicates that a majority of classes (62.6%) have LCOM values between 0 and 0.15, signifying good cohesion. This implies that most classes have methods that are closely related and focused on specific functionalities, which is advantageous for maintainability. However, several classes exhibit higher LCOM values, up to 1.05, suggesting areas where method cohesion could be improved to enhance the overall design.
2. CBO Metric: The CBO (Coupling Between Objects) metric for Priam reveals that many classes (43.3%) have low coupling, with values between 0 and 2.9. This low coupling is beneficial as it suggests minimal dependencies among these classes, making them easier to maintain and modify. However, there are classes with higher CBO values, reaching up to 37.7, indicating potential hotspots where reducing interdependencies could improve modularity and simplify the codebase.
3. Bad Smells: The Priam project has several notable bad smells that could impact its design and maintainability. The most significant is Unutilized Abstraction, with 140 instances, suggesting a substantial number of unused abstractions that could add unnecessary complexity. Deficient Encapsulation, noted in 34 instances, points to areas where classes expose more of their internal structure than necessary, potentially affecting security and maintainability. Broken Hierarchy, with 17 occurrences, highlights issues in the inheritance structure that could complicate understanding and relationships between classes. Additionally, Insufficient Modularization (11 instances) and Cyclic-Dependent Modularization (8 occurrences) suggest challenges with the modular structure and dependencies. Other minor issues include Unnecessary Abstraction (5 occurrences) and Broken Modularization (1 occurrence), which add to the complexity of the codebase.
4. Relationship between Modularity Metrics and Bad Smells: Classes with higher LCOM and CBO values often correlate with the presence of bad smells. For example, classes with Unutilized Abstraction or Deficient Encapsulation tend to show higher LCOM values, indicating lower cohesion. Similarly, classes with Cyclic-Dependent Modularization or Broken Hierarchy usually have higher CBO values, reflecting more significant interdependencies. Addressing these smells could improve both cohesion and coupling, leading to a more maintainable and modular codebase.

In conclusion, the Priam project exhibits good cohesion and manageable coupling among its classes, which are positive indicators for maintainability. However, the presence of bad smells such as Unutilized Abstraction and Deficient Encapsulation highlights areas where the code could be simplified and improved. By targeting these issues, the project can enhance its overall modularity and maintainability, ensuring a more robust and well-organized codebase.





# Section 5: Conclusion

The analysis of software projects using LCOM (Lack of Cohesion of Methods) and CBO (Coupling Between Objects) metrics, combined with the identification of bad smells, provides a comprehensive view of their maintainability and modularity. Projects like IntelliJ-sdk-docs and Eclipse.jdt.ls demonstrate strong cohesion with low LCOM values, indicating well-focused and manageable classes. Conversely, classes with higher LCOM values in projects such as OpenNLP and Priam indicate potential areas for cohesion improvement. Similarly, low CBO values across projects like Intra and Signal-Android highlight effective independence among classes, whereas higher CBO values in some classes suggest opportunities to reduce dependencies for better modularity.

Bad smells, such as Unutilized Abstraction and Deficient Encapsulation, consistently correlate with higher LCOM and CBO values, pointing to design issues that add unnecessary complexity and hinder maintainability. Addressing these bad smells by simplifying abstractions and improving encapsulation is crucial for maintaining a clean, modular design.

Overall, enhancing cohesion and reducing coupling, especially in classes with elevated LCOM and CBO values, is recommended. Focusing on refactoring to resolve identified bad smells will improve code quality, making it easier to maintain and adapt. This strategic approach will lead to a more robust and modular codebase, supporting better software development practices and future scalability.

# References

1. Spinellis, D. (2005). Tool writing: A forgotten art? IEEE Software, 22(4), 9-11. https://www.spinellis.gr/sw/ckjm/
2. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. IEEE Transactions on Software Engineering, 20(6), 476-493.
3. Designite - Java (DesigniteJava). (n.d.). Retrieved from https://www.designite-tools.com/products-dj
4. Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). Goal question metric (GQM) approach. Encyclopedia of Software Engineering, 2, 578-583.
5. 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.
6. Fowler, M. (2018). Refactoring: Improving the design of existing code. Addison-Wesley Professional.