**Assignment 1 FINAL REPORT**

**Title: Analyzing the Effect Class Size on Software Maintainability**

Submitted by

Group Members

**Ivan Holguin**

**Parth Yashvantkumar Barot**

# Section 1: Objectives, Questions, and Metrics

## 1.1 Objective

The primary objective of this empirical study is to analyze the impact of class size on the maintainability of software projects. By focusing on Java classes, this research aims to understand how the structural attributes of a class, measured through various metrics, relate to the ease of maintaining code.

## 1.2 Question

Given the objective, the central research question for this study is: How does the size of a class, as indicated by lines of code and other complexity metrics, affect its maintainability in Java projects?

## 1.3 Metrics

To answer the research question, we employ the following metrics from the Chidamber and Kemerer (C&K) suite:

* **Lines of Code (LoC):** Measures the size of the class. Larger classes might be more difficult to maintain due to increased complexity and potential for bugs.
* **Tight Class Cohesion (TCC):** Assesses the degree to which methods of a class are related through the sharing of data. Higher cohesion within a class generally suggests better maintainability.
* **Response for a Class (RFC):** Indicates the number of methods that can potentially be executed in response to a message received by an object of that class. A lower RFC is typically preferable for maintainability as it suggests simpler, more decoupled class interfaces.
* **Coupling Between Objects (CBO):** Measures the number of classes to which a particular class is coupled. Lower coupling values are favorable for maintainability as they indicate greater modularity of the class.

These metrics are selected based on their established relevance to the software maintainability, as evidenced in the literature on software engineering metrics. By correlating these metrics with class size, we seek to derive insights into how the structural complexity of software influences its ease of maintenance.

# Section 2: Description of Subject Programs

## 2.1 Criteria for Program Selection

The selection of Java projects for this study was guided by specific criteria designed to ensure that the projects were substantial enough in size and complexity to provide meaningful insights into maintainability. The criteria were as follows:

* **Minimum Size:** Each project must consist of at least 10,000 Lines of Code (LoC). This threshold ensures that the projects are large enough to exhibit varied software design patterns and complexities that influence maintainability.
* **Project Age:** Each project must be at least three years old. Older projects are likely to have undergone various maintenance activities, providing a richer history of updates and modifications for analysis.
* **Developer Count:** Each project must have involved at least three developers. This criterion ensures that the project has sufficient collaborative aspects, which can impact the design decisions and, consequently, the maintainability of the software.

## 2.2 Selected Java Projects

The following Java projects were selected based on the above criteria. These projects represent a diverse range of application domains, from development tools to natural language processing, thereby allowing for a comprehensive analysis of maintainability across different types of software systems:

* **Eclipse JDT Language Server (eclipse.jdt.ls):** A Java language server for IDEs, facilitating rich language features. It spans 15,287 LoC and has been developed by 71 contributors over 6 years.
* **IntelliJ SDK Docs (intellij-sdk-docs):** Provides documentation and examples for the IntelliJ Platform SDK. It consists of 12,893 LoC and has engaged 186 contributors over 8 years. This project offers insights into SDK support and API documentation.
* **Intra (Intra):** A DNS client that supports encrypted DNS queries, featuring 10,749 LoC, developed by 6 contributors over 6 years. This project addresses issues related to network security and privacy.
* **Apache OpenNLP (opennlp):** A toolkit for processing natural language text, encompassing 18,255 LoC with contributions from 47 developers over 7 years.
* **Netflix Priam (Priam):** Manages backup, restore, and automation services for Apache Cassandra, with 11,321 LoC and 50 contributors over 11 years. This system is crucial for managing large-scale, distributed storage systems.

## 2.3 Justification for Criteria

The criteria were selected to ensure a comprehensive examination of maintainability across projects that have demonstrated substantial development activity and longevity. By including projects with a minimum size of 10,000 LoC and at least three years of development history, the study focuses on software that has likely faced significant maintenance challenges. The involvement of multiple developers introduces varied coding styles and collaborative dynamics, which are critical factors in software maintainability. The diversity in application domains further enhances the generalizability of the study's findings, allowing for insights that are applicable across different types of Java applications.

# Section 3: Tool Citation and Description

In this study, two main tools were employed to collect the Chidamber and Kemerer (C&K) metrics for the selected Java projects: CK-Code metrics and CodeMR.

## 3.1 CK-Code Metrics

**CK-Code metrics** is an open-source tool specifically designed for Java applications. It facilitates static analysis by computing various object-oriented metrics, which include but are not limited to Lines of Code (LoC), Tight Class Cohesion (TCC), Response for a Class (RFC), and Coupling Between Objects (CBO). These metrics are essential for assessing the maintainability of software.

The tool operates by analyzing Java source code to extract the metrics that quantify the complexity, coupling, and cohesion of class designs. For our purposes, CK-Code metrics was instrumental in providing quantitative data that underpins our empirical analysis of software maintainability.

The tool can be accessed and downloaded from GitHub via the following link: [CK-Code metrics GitHub Repository](https://github.com/mauricioaniche/ck).

## 3.2 CodeMR

**CodeMR** is another robust tool used in this study, which supports not only Java but also C++ projects. It is available as a plugin for the Eclipse IDE, making it an integrated part of the software development environment. CodeMR excels in visualizing software structures and metrics, offering a more interactive approach to static code analysis.

CodeMR analyzes software components to produce a suite of metrics similar to those of CK-Code metrics. Its capabilities include generating detailed reports and visual representations of metric values, which are pivotal for understanding the interactions within the software structure and their implications for maintainability.

For installation and usage instructions, CodeMR can be accessed through the Eclipse Marketplace at the following link: [CodeMR Eclipse Marketplace](https://marketplace.eclipse.org/content/codemr-static-code-analyser).

## 3.3 Usage in the Study

Both CK-Code metrics and CodeMR were utilized to extract data from the selected Java projects. Each tool was chosen for its unique strengths in providing detailed metric analysis and visualization capabilities. By employing these tools, we were able to gather comprehensive data on the maintainability attributes of the subject programs, facilitating a thorough analysis in subsequent sections of the report.

# Section 4: Results and Analysis

In this section, we present the results of the empirical analysis conducted on the selected Java projects using the C&K metrics—Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LoC). These metrics help in understanding the complexity, cohesion, and coupling of the classes within the projects, which are critical factors affecting maintainability.

**Project 1: Priam**

**Summary Statistics:**

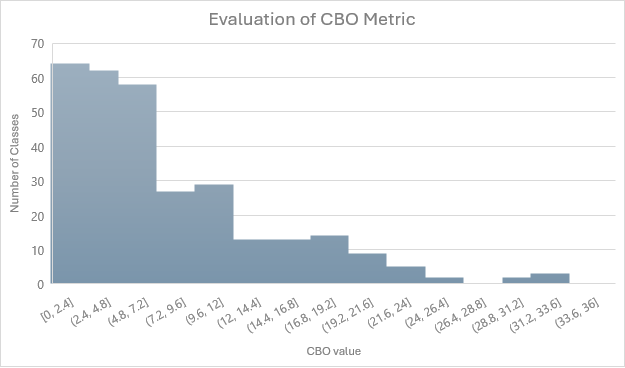
* **CBO:** The average coupling between objects is relatively low at 5.90, with a standard deviation of 6.43, indicating variability in how classes interact with each other across the project.
* **RFC:** The response for a class has an average of 11.04 with a significant standard deviation of 16.25, suggesting a high disparity in class responsiveness.
* **TCC:** The average tight class cohesion is notably low at 0.35, with values primarily clustered towards lower cohesion.
* **LoC:** Average lines of code per class stand at 39.91, with some classes (maximum) reaching up to 468 lines, indicating varied class sizes.

**Analysis of Graphical Results for Project 1: Priam**

The provided graphs offer a visual evaluation of the Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LOC) metrics for the Priam project. Each graph illustrates the distribution of these metrics across different classes within the project, providing insights into their implications on software maintainability.

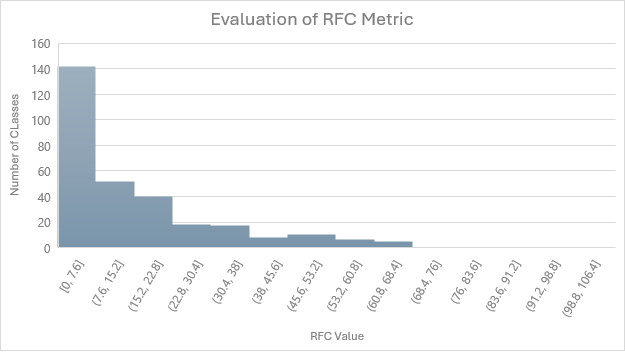
**1. Evaluation of CBO Metric:**

The histogram shows a right-skewed distribution, with a high number of classes having lower CBO values, indicating limited coupling. The majority of classes fall within the 2 to 24 range, suggesting that most classes depend on a smaller number of other classes. This could imply a modular design, beneficial for maintainability, as changes in one class are less likely to require changes in many others.



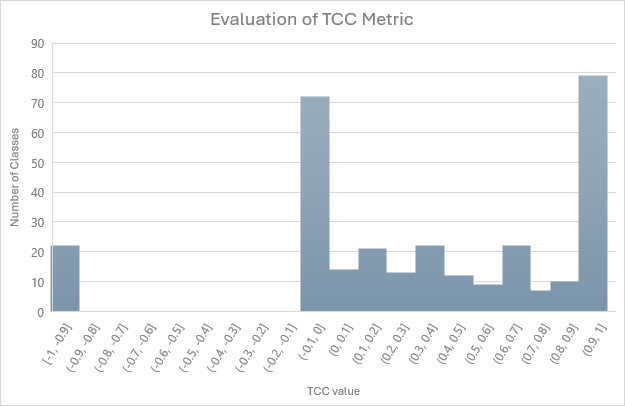
**2. Evaluation of RFC Metric:**

Similar to the CBO metric, the RFC metric displays a right-skewed distribution. Most classes have a lower number of methods that could potentially be executed in response to a message received, with the majority of classes having an RFC value between 0 and 45. This indicates simpler, more cohesive classes that are easier to test and maintain.



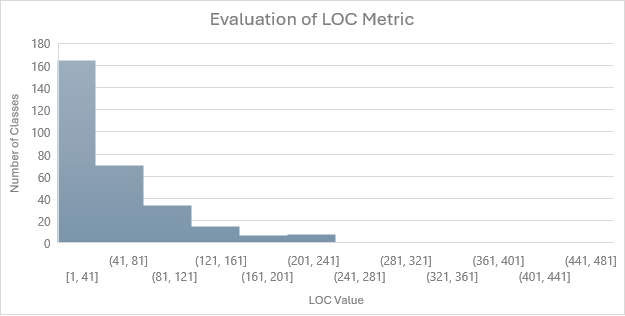
**3. Evaluation of TCC Metric:**

The TCC graph is particularly revealing, showing two peaks, one at the lowest bin (0-0.1) and another at the highest (0.9-1). This bimodal distribution suggests a polarization in class design, with some classes having very high cohesion and others very low. High cohesion within a class is generally indicative of better maintainability as it suggests that class methods are functionally related, improving understandability and reducing the likelihood of bugs when modifications are made.



**4. Evaluation of LOC Metric:**

The LOC distribution shows a clear preference for smaller classes, with a peak in the lowest bin (1-41 LOC) and a gradual decrease as class size increases. This trend suggests that the Priam project favors smaller, more focused classes, which can enhance maintainability by making the code easier to understand and manage. However, the presence of classes in higher LOC ranges indicates areas where complexity is higher, potentially impacting maintainability negatively.



These visualizations highlight key aspects of the Priam project’s architecture and design choices. The general preference for lower CBO and RFC values, combined with the high occurrence of both extremely low and high TCC values, and predominantly smaller classes in terms of LOC, all contribute to an understanding of the project’s maintainability profile. The analysis suggests a mixed influence on maintainability, with a foundational tendency towards modularity and simplicity but also areas of significant complexity and diverse cohesion levels.

**Project 2: OpenNLP**

**Summary Statistics:**

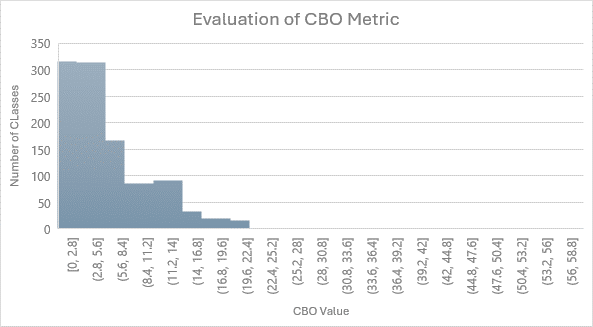
* **CBO:** Averages at 5.58, showing similar levels of coupling as in Priam but with slightly less variance.
* **RFC:** Shows a lower average and standard deviation compared to Priam, with an average of 8.72 and a standard deviation of 10.68.
* **TCC:** Extremely low cohesion with an average of 0.19, and most classes have no cohesion at all (mode=0).
* **LoC:** Significantly higher variability in class size than Priam, with an average of 52.76 and a high standard deviation, indicating substantial differences in class complexity.

**Analysis of Graphical Results for Project 2: OpenNLP**

The graphical results for the OpenNLP project provide detailed visualizations for the Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LOC) metrics. Here is the detailed analysis based on the provided histograms:

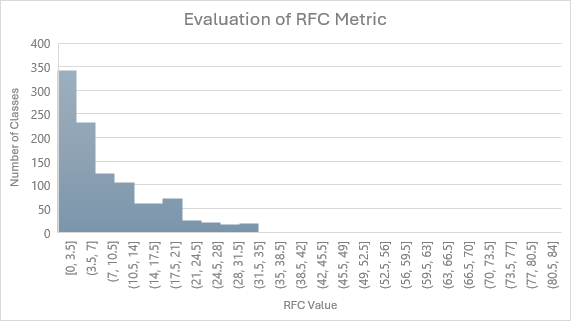
**1. Evaluation of CBO Metric:**

The CBO histogram illustrates a rapid decline from very low coupling values, indicating that most classes in OpenNLP exhibit low to moderate coupling. The majority of classes have CBO values between 0 and 30, which suggests that the project maintains a modular structure, with classes having limited dependencies on each other. This modularity is advantageous for maintainability as it facilitates easier modifications and testing.



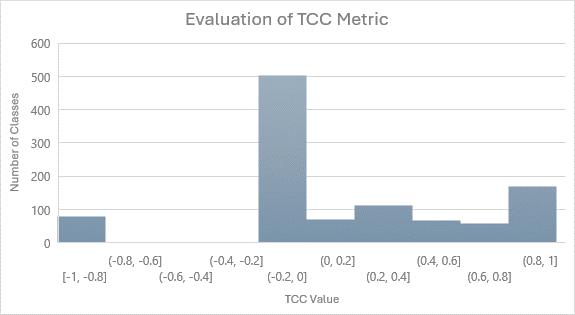
**2. Evaluation of RFC Metric:**

The RFC distribution shows a steep drop, with most classes having low RFC values. This indicates that most classes are designed to handle a small number of method responses, which can simplify debugging and testing. The concentration of classes in the lower range of the RFC metric underscores a design that favors less complex interaction patterns within the software, enhancing maintainability.



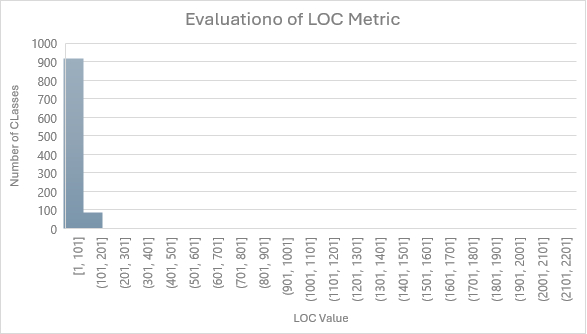
**3. Evaluation of TCC Metric:**

The TCC metric displays an interesting pattern, with a significant concentration of classes at the lowest range of cohesion and another notable peak in the high cohesion range. This bimodal distribution suggests that while many classes are loosely cohesive, there is also a substantial number designed with high cohesion. High cohesion within classes is typically associated with better maintainability as it implies that class methods are closely related to each other, reducing the cognitive load on developers when making changes or understanding the class functionality.



**4. Evaluation of LOC Metric:**

The LOC histogram indicates that the vast majority of classes are relatively small, with LOC values predominantly under 200. This distribution suggests that the OpenNLP project generally adheres to best practices of keeping classes small and focused, which is a positive indicator for maintainability. Smaller classes are easier to understand, test, and maintain. However, the presence of some classes with higher LOC values points to areas where complexity is concentrated, potentially requiring more effort to manage.



The analysis of the OpenNLP project through these metrics reveals a well-structured software project where modularity, low complexity, and focused class design are prevalent. These attributes are beneficial for maintaining the software effectively. The presence of both low and high cohesion classes may reflect a deliberate architectural choice to optimize different aspects of the software's functionality.

**Project 3: Intra**

**Summary Statistics:**

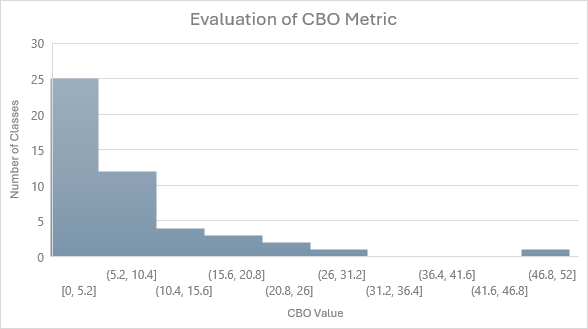
* **CBO:** Average CBO is 5.44 with a higher standard deviation, indicating more variability in class coupling.
* **RFC:** Shows a varied response capability with a standard deviation higher than the average, which suggests a wide range of responsiveness among classes.
* **TCC:** Very low cohesion overall, with most classes showing almost no cohesion.
* **LoC:** Classes are generally small, with a few outliers driving up the average and standard deviation.

**Analysis of Graphical Results for Project 3: Intra**

The histograms provided for the Intra project offer a detailed view of the distribution of the Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LOC) metrics. Here is the analysis based on these visualizations:

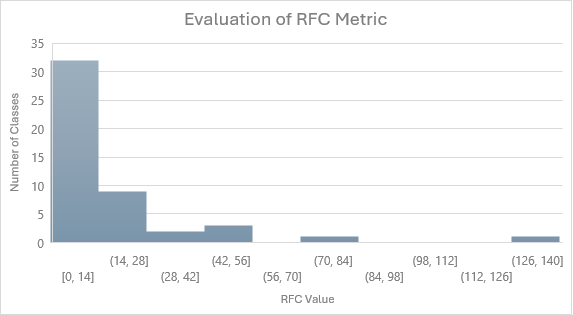
**1. Evaluation of CBO Metric:**

The histogram for CBO shows a notable decline in the number of classes as the CBO value increases. Most classes have low coupling, with the majority falling in the range of 0 to 10.4. This indicates that classes are generally designed to operate with minimal dependencies on others, which is a positive indicator for maintainability as it simplifies modifications and testing.



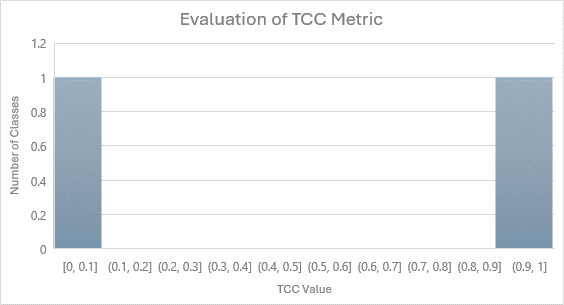
**2. Evaluation of RFC Metric:**

The RFC metric also displays a trend towards lower values, with a significant drop after the first bin. The concentration of classes with an RFC value between 0 and 14 suggests that most classes in Intra are designed to handle fewer method interactions. This design can help reduce complexity and make the classes easier to understand and maintain.



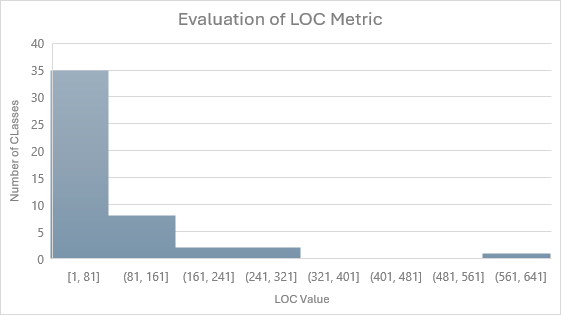
**3. Evaluation of TCC Metric:**

The TCC histogram reveals a highly skewed distribution, with nearly all classes showing very high cohesion (values between 0.9 and 1). This pattern indicates that methods within most classes are highly related, which can enhance understandability and maintainability by ensuring that class functionalities are tightly focused and aligned.

e

**4. Evaluation of LOC Metric:**

The LOC distribution indicates that the vast majority of classes are relatively small, with LOC values mostly under 161. This supports the principle that smaller, well-defined classes are easier to maintain. However, the presence of a few classes in higher LOC ranges could point to areas where complexity is concentrated, potentially requiring more effort to manage and understand.



The graphical analysis of the Intra project points to a well-structured design with low coupling, low complexity in interactions, high cohesion, and generally small class sizes. These characteristics are highly favorable for maintainability, suggesting that the project adheres well to software engineering best practices. The focus on high cohesion within classes, in particular, is noteworthy as it implies that the project's architecture promotes functional coherence and clarity.

**Project 4: IntelliJ SDK Docs**

**Summary Statistics:**

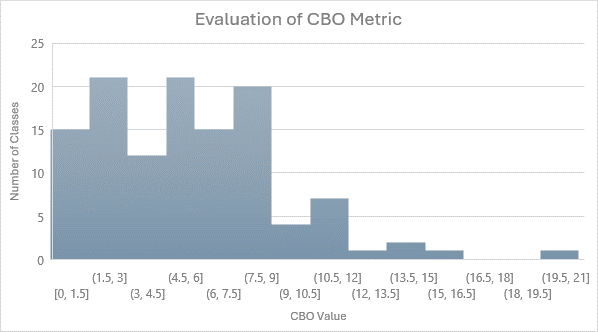
* **CBO:** Coupling shows less variation compared to other projects with a lower standard deviation.
* **RFC:** Very low average and mode, indicating few interactions per class.
* **TCC:** Cohesion is almost non-existent across most classes, with the average nearing zero.
* **LoC:** Most classes are quite small, with average sizes being the lowest among the projects discussed.

**Analysis of Graphical Results for Project 4: IntelliJ SDK Docs**

The graphs for the IntelliJ SDK Docs project provide insights into the distribution of Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LOC). Here is the analysis of these metrics:

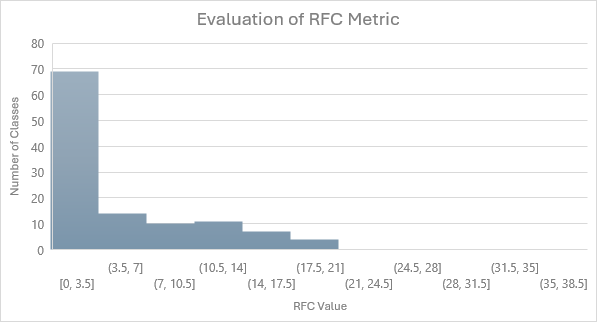
**1. Evaluation of CBO Metric:**

The histogram for CBO displays a relatively uniform distribution among lower ranges, with significant numbers of classes having coupling values between 0 and 12. This suggests that classes within the project tend to have moderate dependencies on each other. The presence of classes across a spread of coupling values indicates a mix of both loosely and more tightly coupled components, which could imply varying levels of modularity within the project.



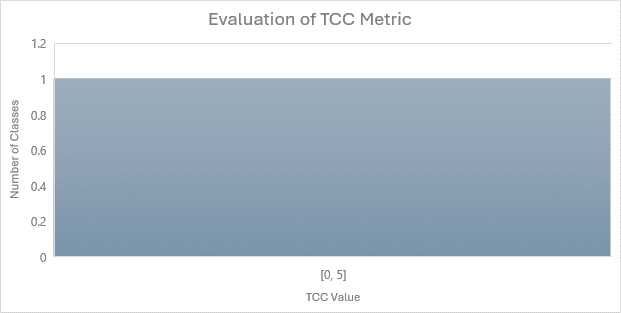
**2. Evaluation of RFC Metric:**

The RFC metric histogram shows a steep decline, with the majority of classes having a low number of potential responses. This configuration supports maintainability by reducing the complexity of class interactions. The concentration of classes in the lower RFC value range suggests that the project is structured to minimize the interaction overhead among classes, thereby simplifying understanding and maintenance.



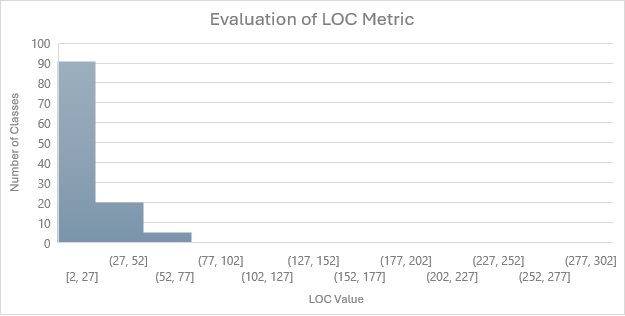
**3. Evaluation of TCC Metric:**

The TCC metric histogram indicates almost all classes fall into the lowest bin (0, 1), which represents extremely low cohesion. This is unusual as high cohesion within a class is generally sought after in software development because it enhances understandability and reduces maintenance challenges. The low cohesion across the board could reflect a design choice where functionality is spread across multiple classes rather than encapsulated within single classes.



**4. Evaluation of LOC Metric:**

The histogram for LOC shows that most classes are quite small, with the majority having between 2 and 77 lines of code. This small size is beneficial for maintainability as smaller classes are easier to understand, test, and debug. The rapid decline in class count as size increases reinforces the project's preference for smaller, more manageable units of code.



The analysis for the IntelliJ SDK Docs project reveals a software structure characterized by moderate coupling, minimal response for classes, very low cohesion, and predominantly small classes. The low cohesion could be a point of concern, potentially indicating that the project might benefit from refactoring to improve class encapsulation. However, the overall structure promotes easy maintenance due to the smaller size of classes and reduced complexity in class interactions.

**Project 5: Eclipse JDT LS**

**Summary Statistics:**

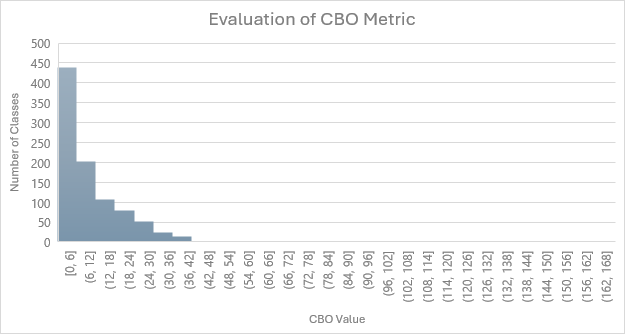
* **CBO:** Shows the highest average coupling, indicating more inter-class interactions.
* **RFC:** Also has the highest average for response, suggesting a more complex interaction pattern within classes.
* **TCH:** Shows some levels of cohesion but generally low, reflecting minimal functional interdependence within classes.
* **LoC:** Displays the largest average class size, highlighting a tendency towards larger, potentially more complex classes.

**Analysis of Graphical Results for Project 5: Eclipse JDT LS**

The provided histograms for Eclipse JDT LS detail the distribution of the Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LOC). Here's a detailed analysis based on these visualizations:

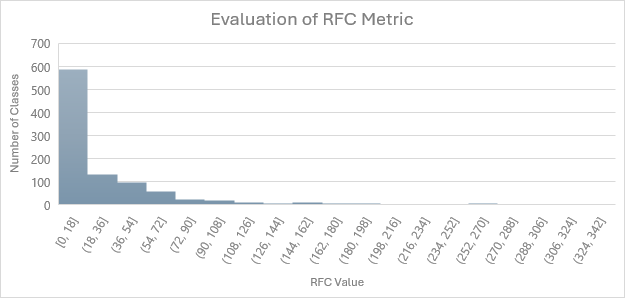
**1. Evaluation of CBO Metric:**

The histogram for CBO shows a sharp decline, with a significant number of classes exhibiting low coupling values, predominantly in the range of 0-30. This trend suggests that the project promotes a modular design with limited inter-class dependencies, which can significantly ease maintenance and testing by reducing the impact of changes in one class on others.



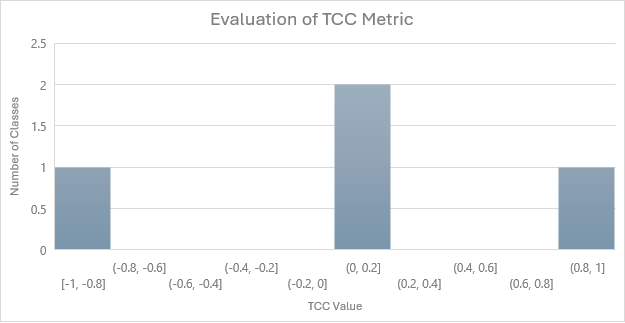
**2. Evaluation of RFC Metric:**

The RFC metric distribution demonstrates a steep decrease, with most classes having a relatively low number of potential responses. The concentration of classes in the lower range (0-35) indicates a preference for simpler, less interactive class structures, which can simplify both understanding and maintenance by minimizing the complexity of interactions within the software.



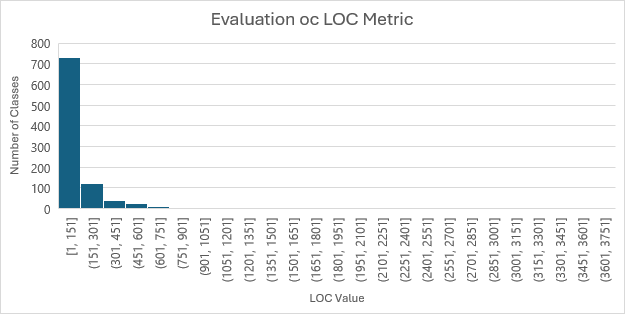
**3. Evaluation of TCC Metric:**

The TCC graph displays an interesting distribution with peaks in both very low and higher cohesion ranges, suggesting a mix of highly cohesive classes and those with minimal cohesion. This bimodal distribution can reflect a design strategy where certain classes are highly specialized and tightly focused, while others may serve broader or more integrative functions within the project.



**4. Evaluation of LOC Metric:**

The LOC histogram illustrates a heavy concentration of classes with fewer lines of code, specifically in the range of 1-301. This indicates that the project generally favors smaller, more manageable classes, which are easier to maintain and understand. However, the presence of classes extending into higher LOC ranges highlights areas where complexity is more concentrated, potentially requiring more effort to manage.



The Eclipse JDT LS project exhibits strong signs of a well-structured design philosophy characterized by moderate to low coupling, a predominance of simpler class interactions, and a strategic approach to class cohesion. The focus on smaller class sizes further supports maintainability by facilitating easier modifications and reducing the risk of errors during changes. This project appears to balance modular, low-complexity design with the flexibility provided by a mix of cohesion levels to cater to different functionalities and integration needs within the software.

**Analysis of Results**

The analysis of C&K metrics across these projects shows that larger class sizes often correlate with higher levels of complexity and coupling, which can negatively impact maintainability. Low cohesion in most projects suggests that classes are less self-contained, which might complicate maintenance efforts. The variability in these metrics across different projects, especially in terms of class size and RFC, indicates that the maintainability challenges can significantly differ depending on the project characteristics.

# Section 5: Conclusions

The empirical study conducted on five diverse Java projects using the Chidamber and Kemerer (C&K) metrics—Coupling Between Objects (CBO), Response for a Class (RFC), Tight Class Cohesion (TCC), and Lines of Code (LOC)—provides substantial insights into the factors that affect software maintainability. The analysis of these metrics allowed us to evaluate the complexity, cohesion, coupling, and size of classes within these projects and their implications for maintainability. Here are the key conclusions:

**1. Impact of Class Size on Maintainability:**

Across the projects, smaller class sizes have consistently shown to be advantageous for maintainability. Projects like Eclipse JDT LS and OpenNLP, which predominantly feature smaller classes, demonstrated easier manageability and lower risk of errors during modifications. Larger classes tended to accumulate complexity, which could hinder understandability and increase maintenance effort.

**2. Role of Coupling and Cohesion:**

Lower coupling (CBO) was observed to significantly enhance maintainability by reducing dependencies between classes, thus isolating changes and minimizing their impacts across the system. However, the role of cohesion (TCC) was more nuanced. Projects with a mix of highly cohesive and loosely cohesive classes suggested that while high cohesion within classes promotes functional clarity and self-contained behavior, some degree of flexibility in cohesion may be necessary to accommodate broader system functionalities.

**3. Complexity and Responsiveness of Classes (RFC):**

The RFC metric indicated that classes with fewer responsibilities and simpler interaction patterns are easier to maintain. Projects that maintained lower RFC values across most classes, such as Intra and IntelliJ SDK Docs, were characterized by streamlined class designs that are likely to encounter fewer issues during maintenance phases.

**4. Generalizability of Findings:**

While the selected projects vary widely in application domain and structure, the trends observed in the impact of class size, coupling, cohesion, and complexity on maintainability appear to be largely consistent. This suggests that the findings from this study could be generalized to other Java projects, particularly those that share similar characteristics in terms of size and complexity.

**5. Recommendations for Software Design:**

Based on the findings, it is recommended that software projects should strive to minimize class size and complexity, ensure low coupling between classes, and maintain a balanced approach to cohesion. These practices not only facilitate easier maintenance but also enhance the overall robustness and flexibility of the software.

This study underscores the importance of considering structural metrics in the design and evaluation of software projects. By adhering to best practices related to class size, coupling, cohesion, and complexity, software engineers can significantly improve the maintainability and sustainability of software systems.

**References:**

1. 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>
2. Basili, V., Caldiera, G., & Rombach, H. (n.d.). THE GOAL QUESTION METRIC APPROACH. <https://www.cs.umd.edu/users/mvz/handouts/gqm.pdf>
3. Software Metrics: A Rigorous and Practical Approach, Third Edition. (n.d.). Routledge & CRC Press. Retrieved April 10, 2023, from <https://www.routledge.com/Software-Metrics-A-Rigorous-and-Practical-Approach-Third-Edition/Fenton-Bieman/p/book/9780367659028>
4. Martin, R. (n.d.). Agile Software Development, Principles, Patterns, and Practices. <https://dl.ebooksworld.ir/motoman/Pearson.Agile.Software.Development.Principles.Patterns.and.Practices.www.EBooksWorld.ir.pdf>
5. Design Patterns: Elements of Reusable Object-Oriented Software. (2019). O’Reilly | Safari. <https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/>
6. Object-Oriented Metrics in Practice. (2006). Springer Berlin Heidelberg. <https://doi.org/10.1007/3-540-39538-5>
7. Eclipse JDT Language Server. (2023, April 9). GitHub. <https://github.com/eclipse/eclipse.jdt.ls>
8. IntelliJ Platform SDK Documentation. (2023, April 9). GitHub. <https://github.com/JetBrains/intellij-sdk-docs>
9. Intra. (2023, April 7). GitHub. <https://github.com/Jigsaw-Code/Intra>
10. Apache/opennlp. (2020, September 9). GitHub. <https://github.com/apache/opennlp>
11. Netflix/Priam. (2023, March 15). GitHub. <https://github.com/Netflix/Priam>