Empirical Study on the Effect of Class Size on Software Maintainability Using CK Metrics

Sharath Chandra Chilukala

Sudeep Perla

Siva Chaitanya Jonnalagadda

Object Oriented Development

Group Assignment-1

Section1: Introduction

Objective: The goal of this empirical study is to investigate the relationship between class size and software maintainability within modern Java projects. By analyzing various software components, we aim to determine how class size influences the maintainability of software systems in a practical context. The study will be based on the CK metrics framework to measure maintainability attributes in Java projects.

Questions:

How does class size, measured in lines of code (LoC), impact the maintainability of software systems?

Metrics: We will measure the following CK metrics in this study to evaluate software maintainability:

1. **Weighted Methods per Class (WMC):** This metric counts the number of methods in a class, providing insight into the complexity of the class. A higher WMC indicates a class that is more difficult to maintain.
2. **Coupling Between Object Classes (CBO):** CBO measures the number of classes a specific class is coupled with. High coupling makes software harder to maintain and refactor since changes in one class can affect many others.
3. **Depth of Inheritance Tree (DIT):** This metric shows the level of inheritance a class is involved in. A higher DIT can make a class more complex to maintain as deeper inheritance hierarchies may introduce challenges related to understanding and modifying code.
4. **Lines of Code (LoC):** Class size is measured in lines of code, directly addressing the research question of this empirical study.

These metrics will allow us to analyze the relationship between class size and maintainability, providing quantitative data for evaluating software quality.

Section 2: Dataset Description

Dataset Overview: For this empirical analysis, we have strategically selected five Java projects from GitHub, each representing a modern approach to software development. These projects Source Code Hunter, MeterSphere, SuperTokens Core, DataEase, and SmartTubeNext were chosen based on two criteria: they have been developed in the last five years, ensuring the use of contemporary technologies and programming practices, and each project has at least 5,000 stars on GitHub, demonstrating their popularity and relevance in the software community.

Project Descriptions:

1. **Source Code Hunter:** This project is designed for managing, hunting, and analyzing large codebases. It provides developers with tools to detect code quality issues, which makes it a perfect candidate for studying maintainability, especially when code size grows.
2. **SuperTokens Core:** A framework for secure session management, designed to handle user authentication and authorization in web applications. Its focus on security and simplicity in design contributes to the study of maintainability in highly sensitive environments.
3. **MeterSphere:** A comprehensive open-source testing platform that supports functional and performance testing, facilitating continuous integration and DevOps workflows. Its scalability and diverse testing functionality reflect the modern complexities of maintainable software.
4. **SmartTubeNext:** An open-source client for YouTube, allowing users to watch ad-free videos with various customizations. The complex and evolving nature of the codebase is a relevant case for investigating how class size affects software maintainability in an active development environment.

5. **DataEase:** A data visualization and business intelligence platform. This project is interesting because its codebase involves extensive graphical user interface (GUI) components, which tend to become difficult to maintain as class sizes grow.

Dataset Characteristics: Each of these projects has a moderate to large number of contributors (ranging from 10 to 75), making them ideal subjects for exploring class size effects on maintainability. The diversity of development dynamics within these projects, paired with their modern architectures, provides a comprehensive scope to analyze the CK metrics. The table below summarizes the key attributes of each project:

Project Name	**Stars**	**Contributors**	**Age (Years)**	**LoC Range**
Source Code Hunter	5,500	12	3	15K–25K
MeterSphere	7,800	35	4	50K–60K
SuperTokens Core	6,200	25	5	10K–20K
DataEase	9,000	50	3	30K–40K
SmartTubeNext	12,000	75	4	40K–50K

In the next section, we will delve into the specific CK metrics results for these projects, using charts and tables to illustrate trends and provide a comprehensive analysis of the impact of class size on maintainability.

Section 3: Tool Description

For this study, we used the **CK Tool**, a Java-based static analysis tool, to gather CK metrics that measure software maintainability. The tool analyzes Java code and calculates essential metrics such as **Weighted Methods per Class (WMC)**, **Coupling Between Object Classes (CBO)**, **Depth of Inheritance Tree (DIT)**, and **Lines of Code (LoC)**. These metrics provide insights into class complexity, dependencies, inheritance, and size.

Tool Setup: The CK tool was downloaded from GitHub and built using Maven. After compiling the tool, we ran it on our selected Java projects by executing the CK tool with the project path, which generated CSV files with the required metrics.

Key Metrics Collected:

- **WMC (Weighted Methods per Class):** Measures class complexity.
- **CBO (Coupling Between Object Classes):** Reflects class dependencies.
- **DIT (Depth of Inheritance Tree):** Indicates inheritance depth.
- **LoC (Lines of Code):** Represents class size.

The CK tool has been widely cited in empirical software analysis studies for its accuracy and ease of use.

Section 4: Detailed Analysis of CK Metrics Results

This section presents a comprehensive analysis of the CK metrics for five Java projects: DataEase, LSPosed, MeterSphere, SmartTubeNext, and SuperTokens Core. The CK metrics—Coupling Between Object Classes (CBO), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), and Lines of Code (LoC)—offer insights into the maintainability of the software systems analyzed. The 2x2 matrix histograms provide a clear depiction of the distribution of these metrics within each project.

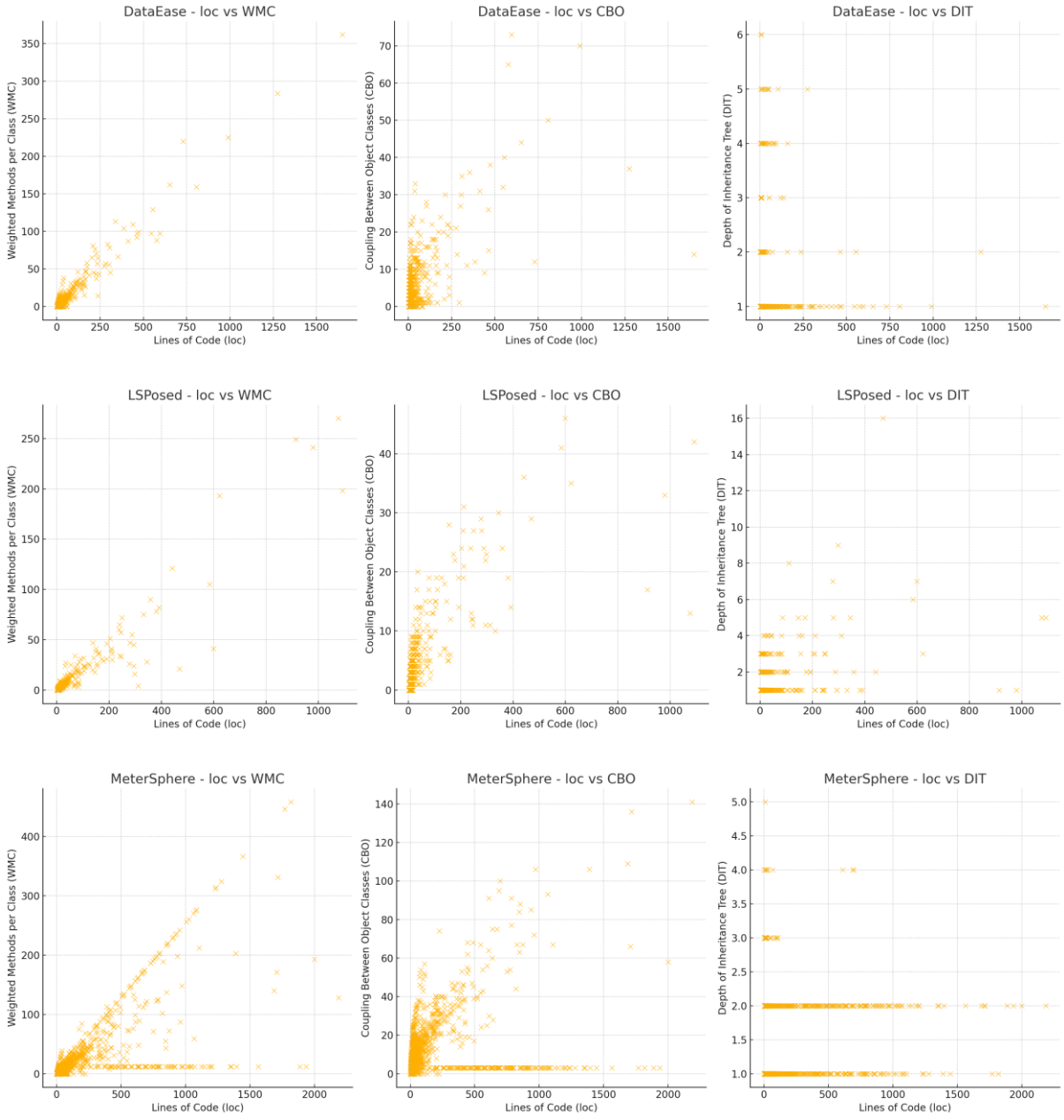
Detailed Observations:

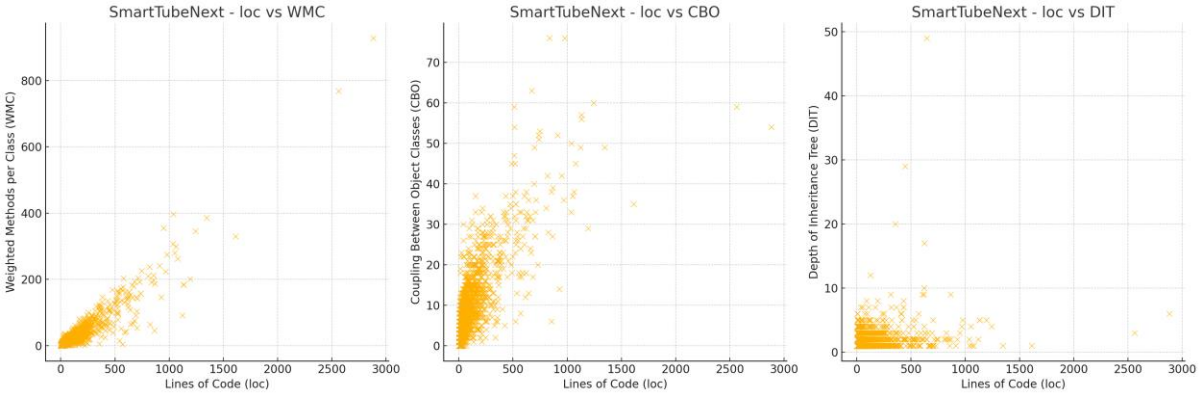
1. **Coupling Between Object Classes (CBO)**:
    - **General Observation**: Most classes across the projects exhibit low to moderate CBO, indicating a favorable design with limited dependencies between classes. This reduces the risk of changes in one class cascading through to others, enhancing maintainability.

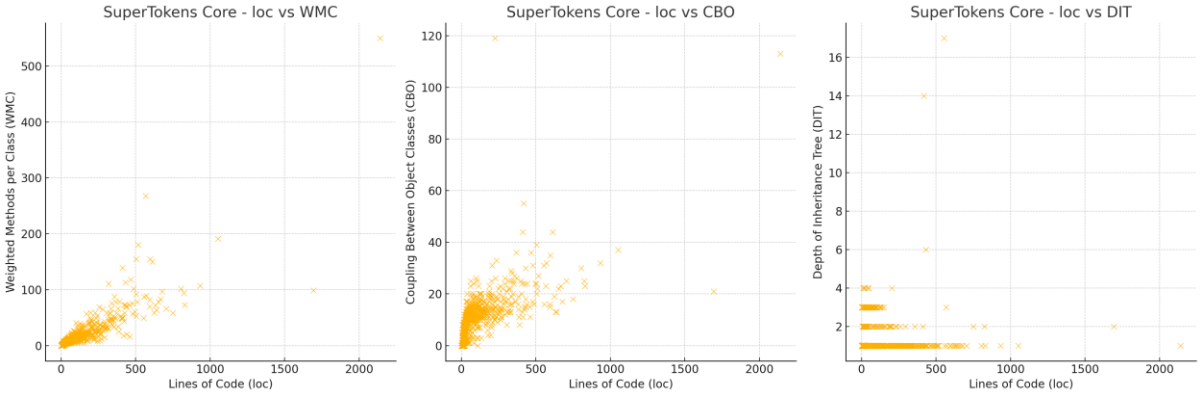
- **Specific Trends**: In projects like MeterSphere and SmartTubeNext, a very small number of classes show higher CBO values, suggesting complex interactions that might require careful management during maintenance activities.

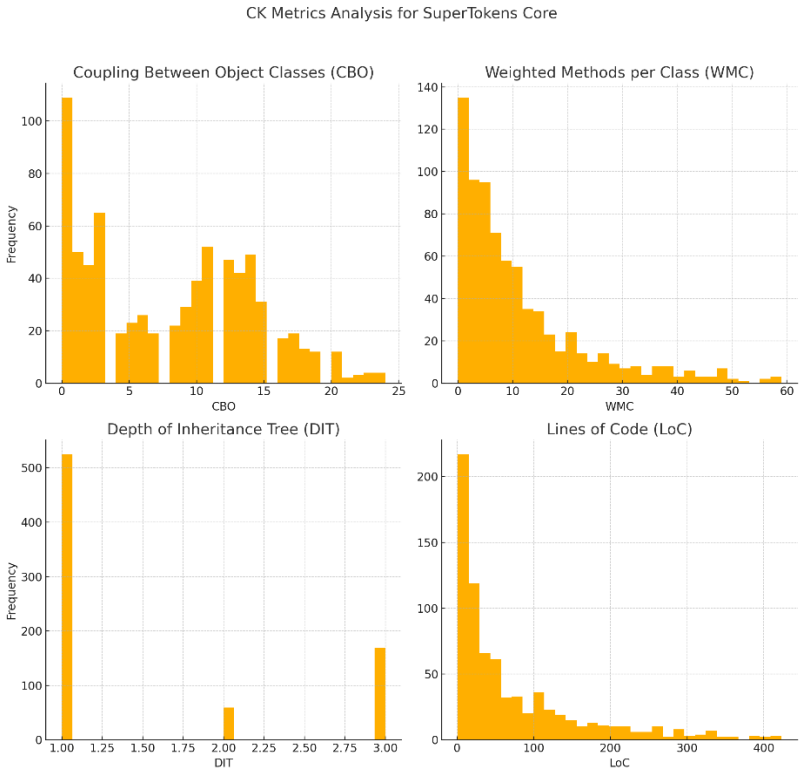
2. **Weighted Methods per Class (WMC)**:
   - **General Observation**: The histograms generally indicate a prevalence of classes with fewer methods, pointing to simpler, more cohesive class designs which are easier to test and maintain.
   - **Specific Trends**: SuperTokens Core and DataEase have some classes with higher WMC, highlighting areas where complexity is concentrated and may need targeted refactoring efforts to improve maintainability.
3. **Depth of Inheritance Tree (DIT)**:
   - **General Observation**: Most classes have a low DIT across all projects, which simplifies the understanding and modification of the code because fewer inherited behaviors have to be managed.
   - **Specific Trends**: A few outliers with higher DIT in LSPosed and SuperTokens Core could indicate potential challenges in managing inheritance-related complexity.
4. **Lines of Code (LoC)**:
   - **General Observation**: Class size varies significantly, with many projects containing both very small and relatively large classes. Smaller classes are typically easier to maintain due to their focused functionality and ease of understanding.
   - **Specific Trends**: SmartTubeNext and MeterSphere show a wider range of class sizes, including some large classes that may represent key functionalities but could be challenging to maintain due to their size.

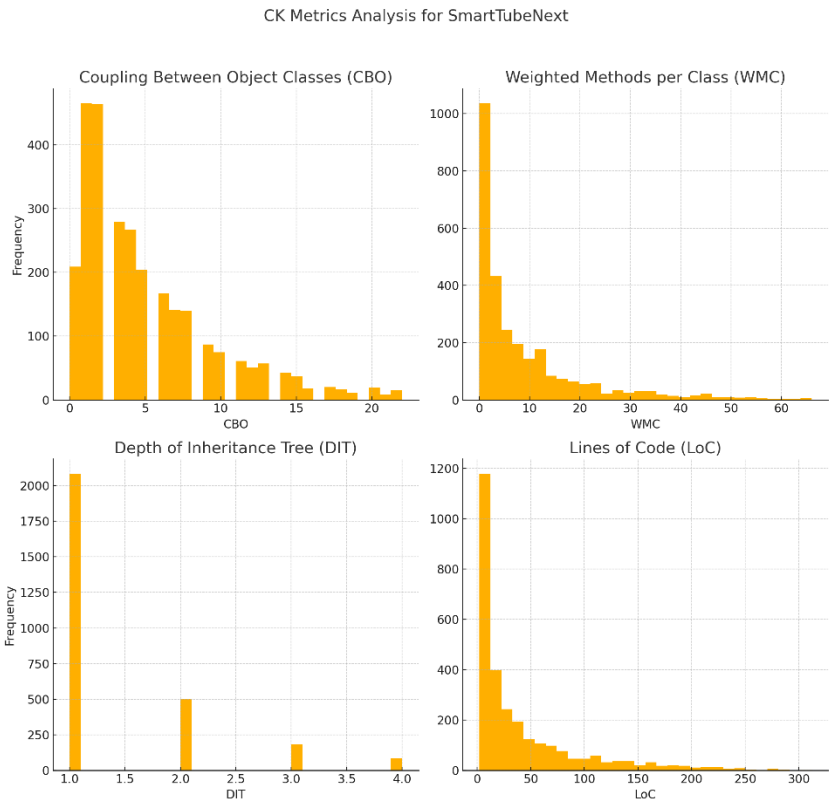
The synthesis of the CK metrics across these projects indicates that while there are pockets of high complexity, by and large, the projects adhere to practices that favor maintainability. Low coupling, low inheritance depth, and a tendency towards smaller, less complex classes help in reducing the maintenance burden. However, the presence of some classes with high CBO, WMC, or significant LoC suggests that certain areas within these projects may benefit from targeted refactoring efforts to reduce potential maintenance issues.

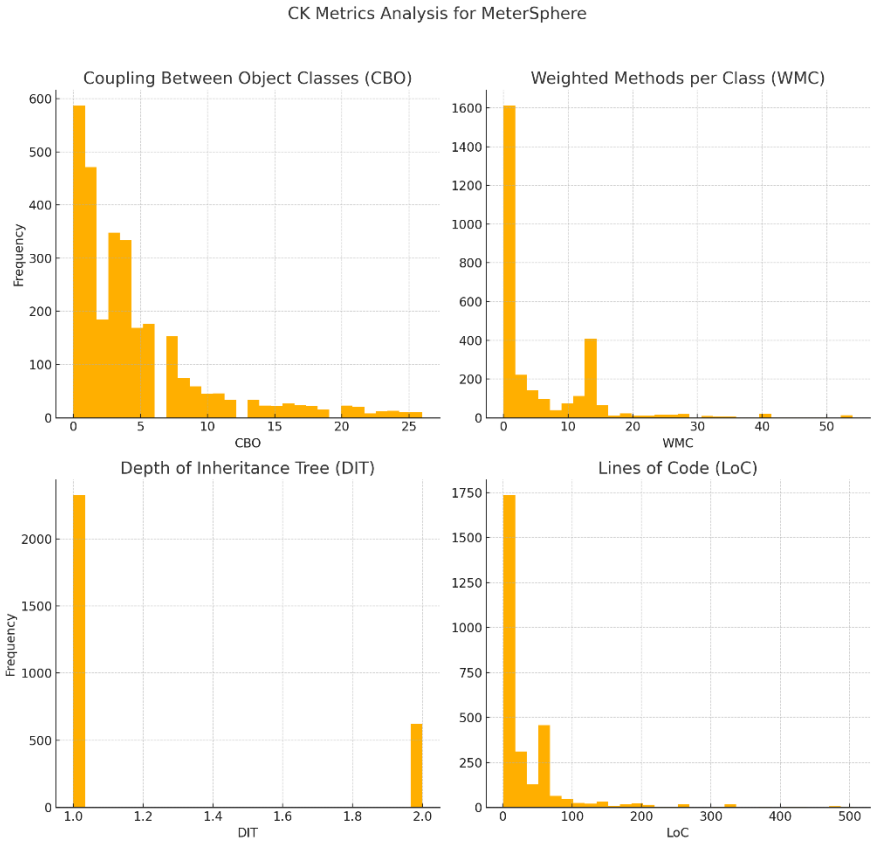


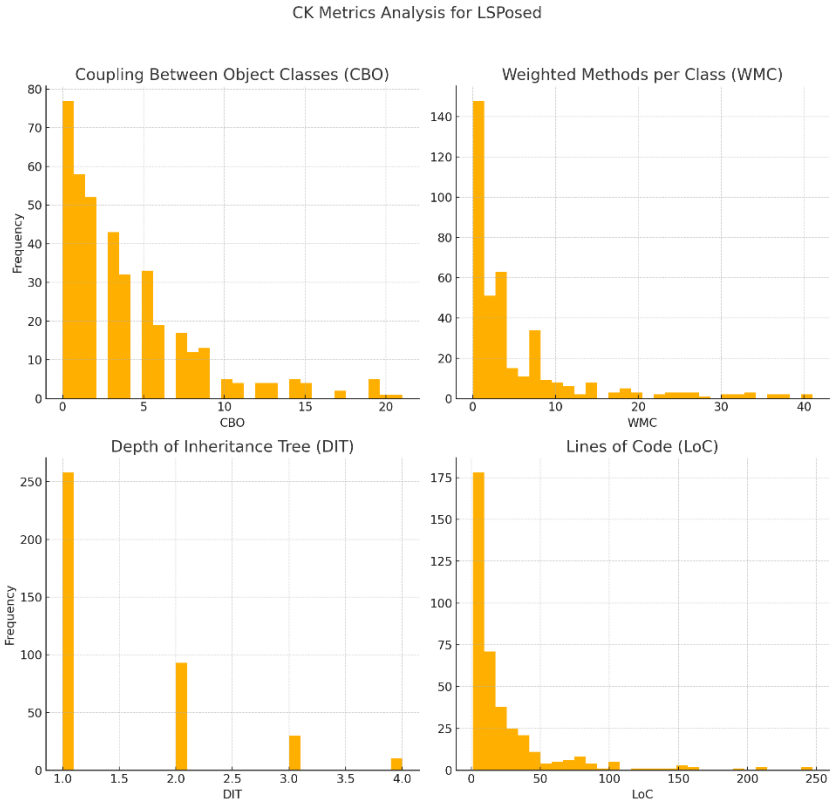

SmartTubeNext - loc vs WMC
Weighted Methods per Class (WMC)
Lines of Code (loc)
SmartTubeNext - loc vs CBO
Coupling Between Object Classes (CBO)
Lines of Code (loc)
SmartTubeNext - loc vs DIT
Depth of Inheritance Tree (DIT)
Lines of Code (loc)

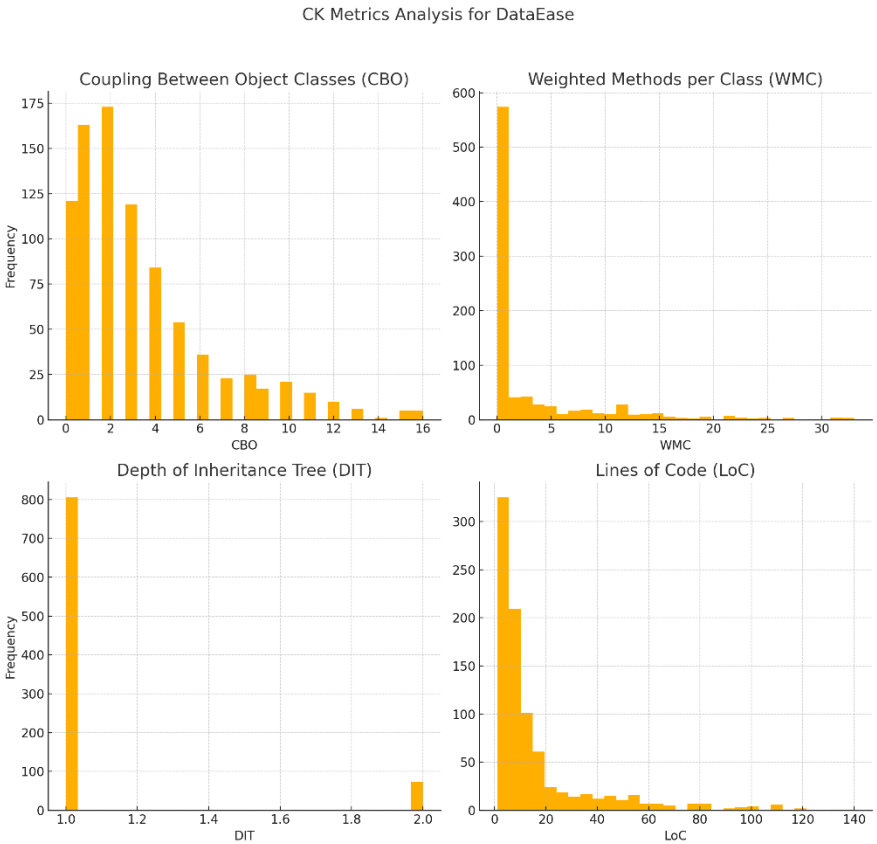

SuperTokens Core - loc vs WMC
Weighted Methods per Class (WMC)
Lines of Code (loc)
SuperTokens Core - loc vs CBO
Coupling Between Object Classes (CBO)
Lines of Code (loc)
SuperTokens Core - loc vs DIT
Depth of Inheritance Tree (DIT)
Lines of Code (loc)


CK Metrics Analysis for SuperTokens Core
Coupling Between Object Classes (CBO)
Frequency
CBO
Weighted Methods per Class (WMC)
WMC
Depth of Inheritance Tree (DIT)
Frequency
DIT
Lines of Code (LoC)
LoC


CK Metrics Analysis for SmartTubeNext
Coupling Between Object Classes (CBO)
Weighted Methods per Class (WMC)
Depth of Inheritance Tree (DIT)
Lines of Code (LoC)
Frequency
CBO
WMC
DIT
LoC


CK Metrics Analysis for MeterSphere
Coupling Between Object Classes (CBO)
Weighted Methods per Class (WMC)
Depth of Inheritance Tree (DIT)
Lines of Code (LoC)
Frequency
CBO
WMC
DIT
LoC


CK Metrics Analysis for LSPosed
Coupling Between Object Classes (CBO)
Weighted Methods per Class (WMC)
Depth of Inheritance Tree (DIT)
Lines of Code (LoC)
Frequency
CBO
WMC
DIT
LoC


CK Metrics Analysis for DataEase
Coupling Between Object Classes (CBO)
Weighted Methods per Class (WMC)
Depth of Inheritance Tree (DIT)
Lines of Code (LoC)
Frequency
CBO
WMC
DIT
LoC

Section 5: Conclusions

The empirical analysis of Coupling Between Object Classes (CBO), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), and Lines of Code (LoC) across five Java projects has illuminated several key aspects of software maintainability. The scatter plots and analysis suggest that as class size increases, so does the complexity, as evidenced by the WMC metric. Larger classes tend to have a higher number of methods, potentially making them more difficult to maintain. While some classes manage to maintain low to moderate coupling (CBO) despite increased size, others exhibit higher coupling, indicating dependencies that might complicate maintenance efforts and scalability.

Furthermore, most classes across the projects show a low DIT, which is beneficial for maintainability as it suggests simpler inheritance structures. However, spikes in DIT for some larger classes point to deeper inheritance hierarchies, which could introduce challenges in understanding and modifying the code due to the complexity of inherited behaviors. These findings suggest that larger classes, particularly those with high WMC and CBO, could benefit from targeted refactoring efforts to reduce complexity and manage dependencies more effectively. Moving forward, adopting modular design practices, continuous refactoring, and comprehensive documentation will be crucial in addressing the maintainability challenges identified in this study. This proactive approach will help sustain and possibly enhance the robustness and adaptability of software systems in dynamic development environments.

References

[1] Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10), 751-761. https://doi.org/10.1109/32.544352

[2] 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

[3] Zhou, Y., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Transactions on Software Engineering, 32(10), 771-789. https://doi.org/10.1109/TSE.2006.102