Prithu Kathet, Harini Radhavaram, and Madhuri Bajjuri

prithukathet@lewisu.edu , Hariniradhavaram@lewisu.edu , madhuribajjuri@lewisu.edu

# 1. Introduction

## 1.1 Objectives:

The primary objective of this empirical study is to assess the maintainability of Java software projects using selected C&K metrics. By evaluating these metrics across multiple open-source projects, we aim to identify patterns and factors influencing software maintainability. The findings will help in understanding the relationship between various class-level and method-level metrics and their impact on the ease of maintaining software systems.

## 1.2 Questions:

Based on the GQM (Goal-Question-Metric) approach, the following questions guide our study:

a. **Q1: How does coupling between classes (CBO) affect the maintainability of the software?**
   - *Rationale*: Higher coupling may indicate that a class is more dependent on other classes, potentially complicating maintenance activities such as refactoring, testing, and debugging.

b. **Q2: What is the impact of class complexity (WMC) on maintainability?**
   - *Rationale*: Classes with higher complexity can be harder to understand and modify, making them less maintainable. High complexity may lead to increased effort and time for maintenance tasks.

c. **Q3: Is there a correlation between class size (LOC) and maintainability?**
   - *Rationale*: Large classes may be difficult to manage and comprehend, potentially affecting maintainability. This question seeks to explore if larger classes consistently have lower maintainability.

d. **Q4: How do variations in method-level metrics contribute to overall class maintainability?**
   - *Rationale*: Method-level metrics like method complexity and variable usage might offer insights into the maintainability of classes at a finer granularity.

## 1.3 Metrics:

To answer the above questions, we will use the following metrics, derived from the C&K suite:

a. **CBO (Coupling Between Objects)**:
   - *Definition*: Measures the number of classes that a given class is coupled with. High CBO values can indicate higher interdependencies between classes.

- *Purpose*: Used to evaluate the level of coupling in the software and its impact on maintainability.

b. **WMC (Weighted Methods per Class)**:
   - *Definition*: Measures the total complexity of all methods in a class. A higher WMC suggests that a class is more complex and potentially harder to maintain.
   - *Purpose*: Indicates the complexity and understandability of a class.

c. **LOC (Lines of Code)**:
   - *Definition*: Measures the size of a class based on the number of lines of code it contains.
   - *Purpose*: Used to assess the size of a class, which can impact its maintainability.

d. **Additional Metrics**:
   - *Method Complexity*: Measures the complexity of individual methods within classes.
   - *Variable Usage*: Assesses the number of times variables are used within methods, indicating method complexity and readability.

## 1. Subject Programs

The following table summarizes the main attributes of the selected programs:

| Project Name | Stars | LOC | Age (Year) | Forks | Contributors | Descriptions | No. of files |
|---|---|---|---|---|---|---|---|
| Conductor | 1300 | 10415 | 8 | 2300 | 248 | Orchestrates microservices with workflows. | 49 |
| Error-prone | 6800 | 7284 | 13 | 741 | 214 | A tool for catching common Java mistakes. | 24 |
| Litho | 7700 | 1712 | 7 | 762 | 308 | A declarative framework for building UIs. | 72 |
| Spring-Security | 8700 | 28208 | 15 | 5900 | 740 | Provides authentication and access control. | 42 |
| VirtualXposed | 15400 | 93322 | 8 | 2500 | 38 | A framework for Android app modification. | 9 |

**Justification for Selection**

The selected projects are all at least 3 years old, ensuring they have undergone various development and maintenance phases, allowing for a meaningful analysis of maintainability.

## 2. Tool Used

For obtaining C&K metrics, we utilized the **CK-Code** metrics tool, developed by a team of 24 developers. This tool analyzes Java code through static analysis, providing various metrics that can be used to evaluate software quality attributes, including maintainability. The tool can be accessed at: CK-Code Metrics Tool (Aniche, 2015)

CK is a Java code metrics collection tool, streamlined into a simple structure that revolves around three primary packages:

*ck:* Contains the core classes that drive the metrics collection process.
*ck.metrics*: Hosts the metric definitions and implementations.
*ck.utils:* Utilities supporting the metrics collection process.

CK is the core class of a metrics collection tool that orchestrates the entire process of gathering code metrics from Java projects. It initializes metric finders, manages file partitioning based on memory, sets up Abstract Syntax Tree (AST) parsers, and controls execution flow across directories and JAR dependencies. It adjusts dynamically based on user inputs like useJars, maxAtOnce, and variablesAndFields.
The Runner class serves as the entry point, processing command-line arguments to configure the metrics collection. It handles user input for project paths, JAR inclusion, file partitioning, and output directory setup, while initializing the CK class and managing result output through ResultWriter.

MetricsExecutor extends FileASTRequestor and is crucial for coordinating the metrics collection process. It creates the AST for Java source files, essential for analyzing and extracting code metrics.

MetricsFinder, located in ck.utils, dynamically identifies and instantiates metric collector classes that implement ClassLevelMetric and MethodLevelMetric interfaces. It uses the Reflections library to load metric classes at runtime, making the CK system adaptable to new metrics.

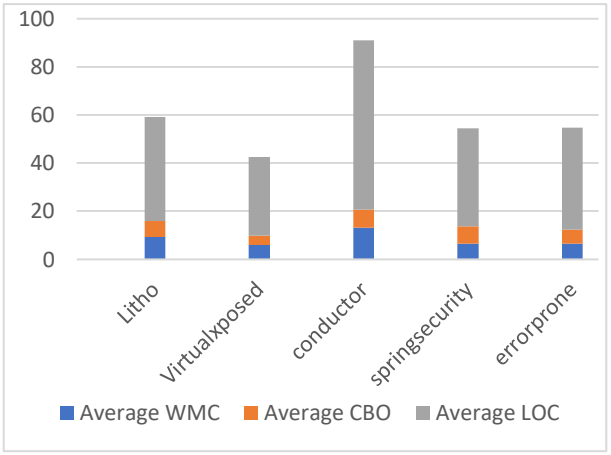
CKVisitor extends ASTVisitor, enabling detailed analysis and metric collection from the AST. It traverses AST nodes, applying specific actions and managing a stack-based hierarchy for metrics calculation.

CKASTVisitor allows metric classes to handle specific AST nodes. ClassLevelMetric and MethodLevelMetric interfaces define methods for collecting respective metrics.
After collecting metrics, MetricsExecutor uses a Notifier design pattern to broadcast results via the CKNotifier interface, while an anonymous class in Runner.main populates CKClassResult and CKMethodResult with data. ResultWriter generates and stores results in .CSV format.

## 3. Results

The comparison of average metrics across the five projects—Litho, VirtualXposed, Conductor, Spring Security, and Error-prone—reveals significant variations in maintainability-related factors. Conductor has the highest average WMC (13.09), indicating it has more complex classes compared to the other projects, which may lead to more challenging maintenance tasks. Litho and Error-prone have moderate WMC values (9.23 and 6.66, respectively), suggesting a balanced complexity in their classes, while Spring Security and VirtualXposed have the lowest WMC averages (6.50 and 6.05), indicating relatively simpler classes. In terms of coupling (CBO), Conductor and Spring Security show higher averages (7.46 and 7.33), which could imply more interdependencies between classes, potentially complicating maintenance. VirtualXposed, with the lowest CBO (3.86), suggests better modularity and maintainability. Lastly, Conductor has the largest average class size (70.56 LOC), followed by Litho (43.27 LOC), while VirtualXposed and Spring Security have smaller classes on average, indicating that Conductor's larger, more complex classes may require more effort to maintain. Overall, Conductor appears to pose the greatest maintainability challenges, while VirtualXposed shows the best potential for ease of maintenance due to its lower complexity and coupling.

| Project Name | Average WMC | Average CBO | Average LOC |
|---|---|---|---|
| **Litho** | 9.2352 | 6.6561 | 43.2786 |
| **Virtualxposed** | 6.0564 | 3.8609 | 32.704 |
| **conductor** | 13.09 | 7.4638 | 70.56 |
| **springsecurity** | 6.504 | 7.333 | 40.666 |
| **errorprone** | 6.66 | 5.8260 | 42.30 |



## 4.1 Litho Project:

**WMC Histogram:**
The histogram shows the distribution of Weighted Methods per Class (WMC) across the Litho project. Higher WMC values indicate classes with more complex methods, potentially making them harder to maintain. The distribution in the histogram reveals that the majority of the classes have moderate WMC values, though a few outliers exhibit high complexity.

**CBO Histogram:**
This histogram displays the Coupling Between Objects (CBO) for the Litho project. Lower values reflect loosely coupled classes, while higher values indicate tightly coupled classes, which may impact maintainability. The distribution suggests that most classes in Litho are moderately coupled, though a small number have higher coupling, which might complicate maintenance.

**WMC vs LOC Scatter Plot:**
This scatter plot compares the Weighted Methods per Class (WMC) against the Lines of Code (LOC) for each class in the Litho project. A positive correlation between WMC and LOC may suggest that larger classes tend to have more complex methods. The plot shows a trend where larger classes have higher WMC, indicating potential maintainability challenges for larger, more complex classes.

**WMC & CBO Boxplot:**
The boxplot contrasts WMC and CBO metrics for Litho. It visualizes the spread and variation in class complexity (WMC) and coupling (CBO). Outliers suggest that certain classes are significantly more complex or coupled than the average, which could be areas of concern for maintainability.

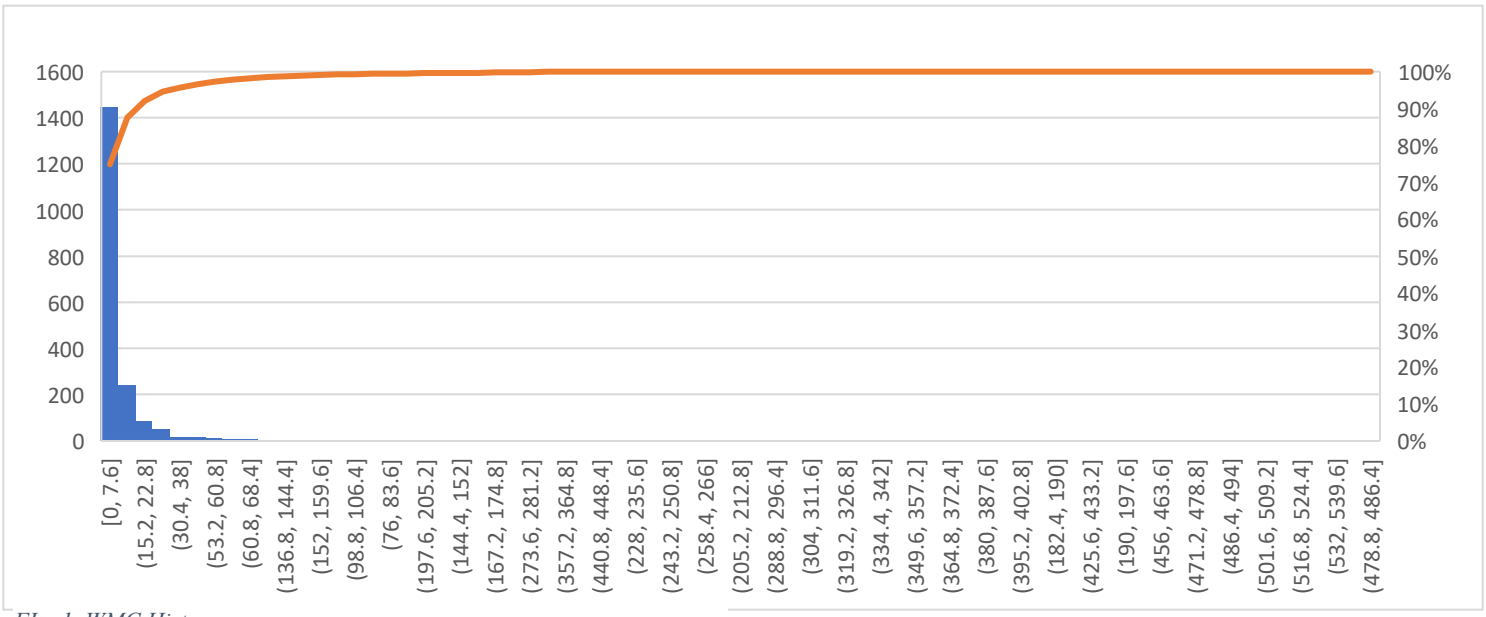



*FIg. 1 WMC Histogram*

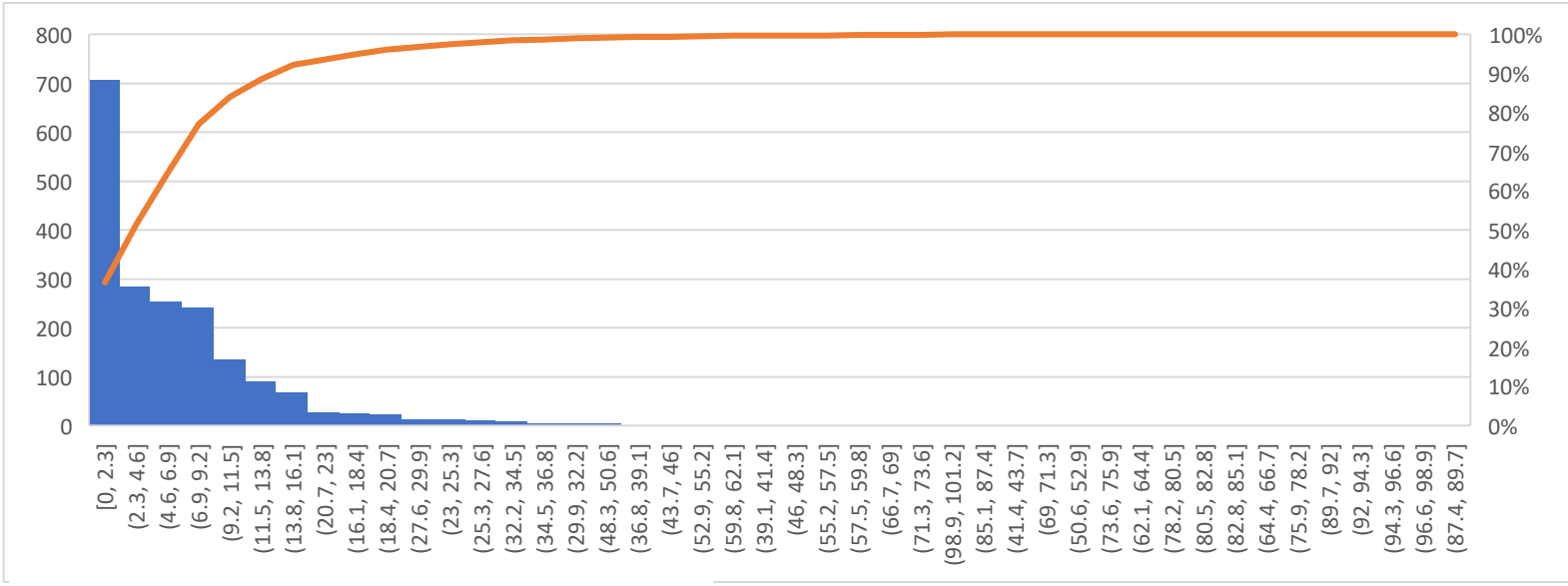



*Fig. 2 CBO Histogram*

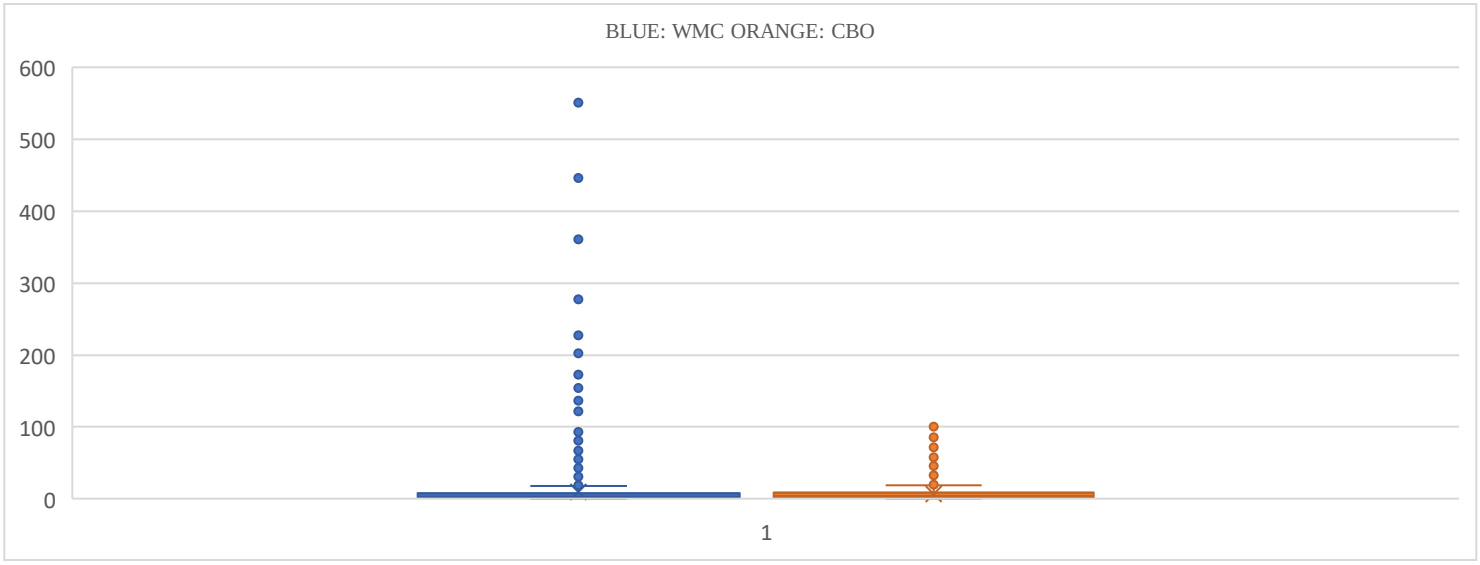



*Fig. 3 WMC & CBO Boxplot*

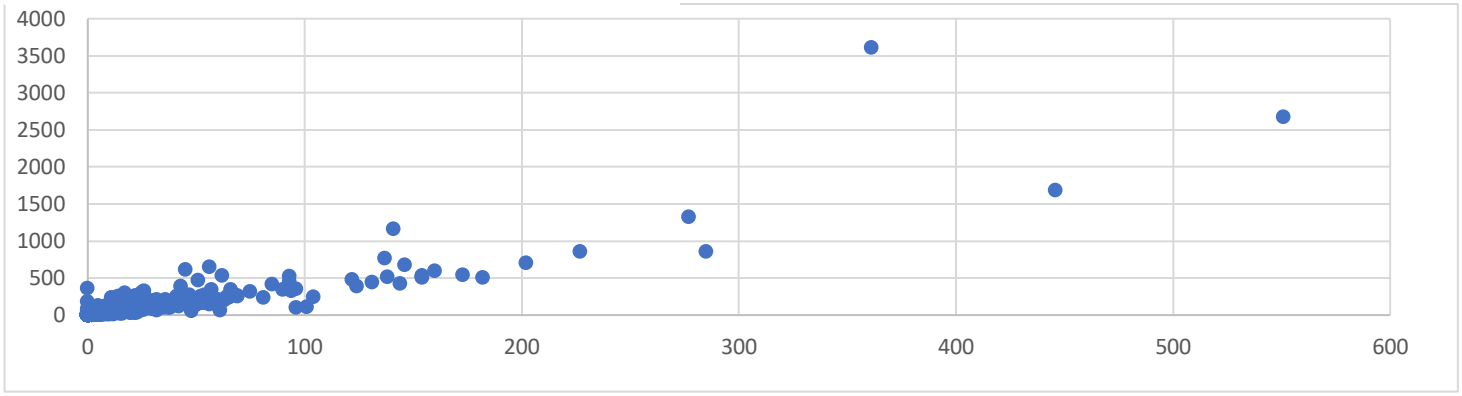



*Fig. 4 WMC VS LOC Scatter plot*

## 4.2 VirtualXposed Project:

**WMC Histogram:**
The histogram for Weighted Methods per Class (WMC) highlights the distribution of class complexity in the VirtualXposed project. Most classes have moderate WMC values, indicating manageable complexity. A few classes, however, have high WMC values, potentially indicating areas that require more maintenance effort.

**CBO Histogram:**
The CBO histogram shows how classes in VirtualXposed are coupled. The majority of the classes exhibit low to moderate coupling, suggesting that the project is well modularized. However, a few classes have higher coupling, which may pose challenges in terms of dependencies and maintainability.

**WMC vs LOC Scatter Plot:**
This scatter plot depicts the relationship between class size (LOC) and complexity (WMC) in VirtualXposed. The spread suggests that while some large classes are relatively simple, others grow in complexity as they increase in size. This could imply that certain larger classes may need more maintenance due to their complexity.

**WMC & CBO Boxplot:**
The boxplot for VirtualXposed shows the variability in class complexity and coupling. Similar to Litho, it highlights the presence of outliers with exceptionally high WMC or CBO values, indicating classes that may be more difficult to maintain.

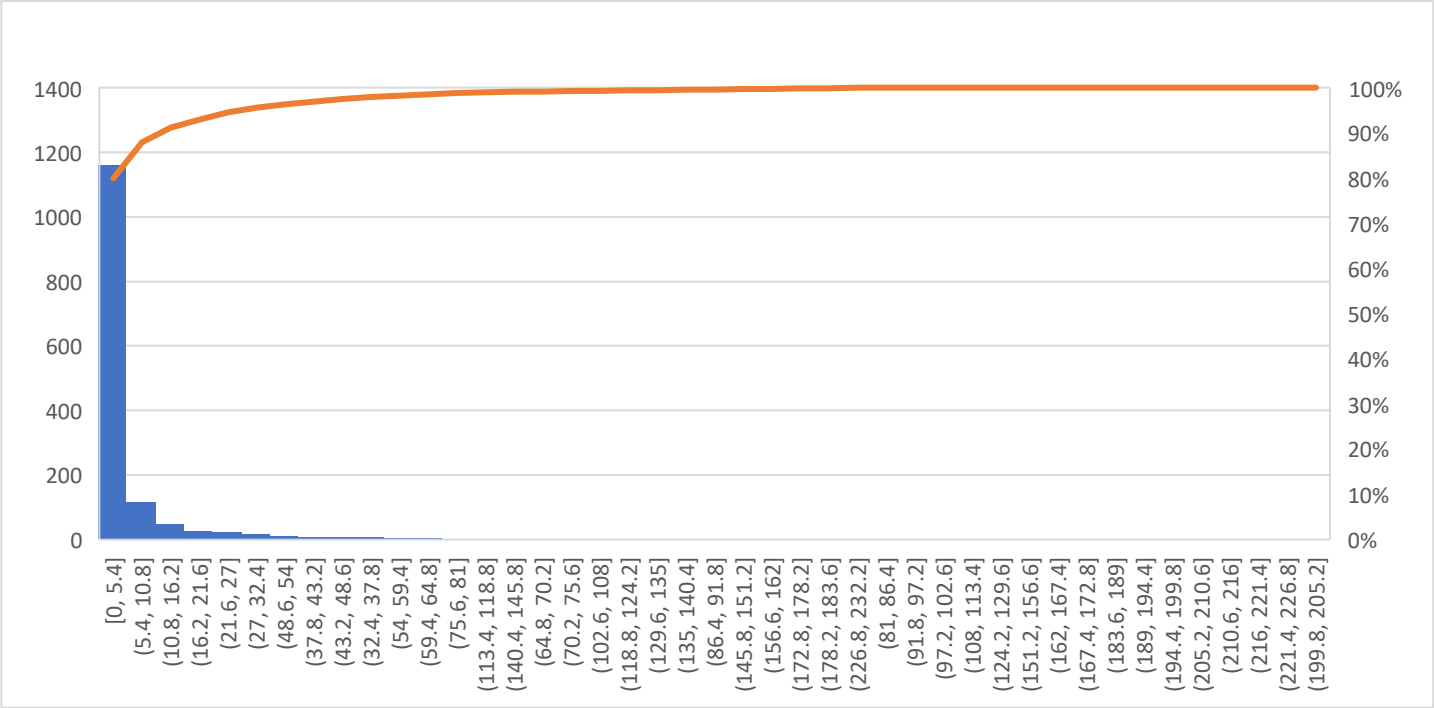



*Fig. 1 WMC HIstogram*

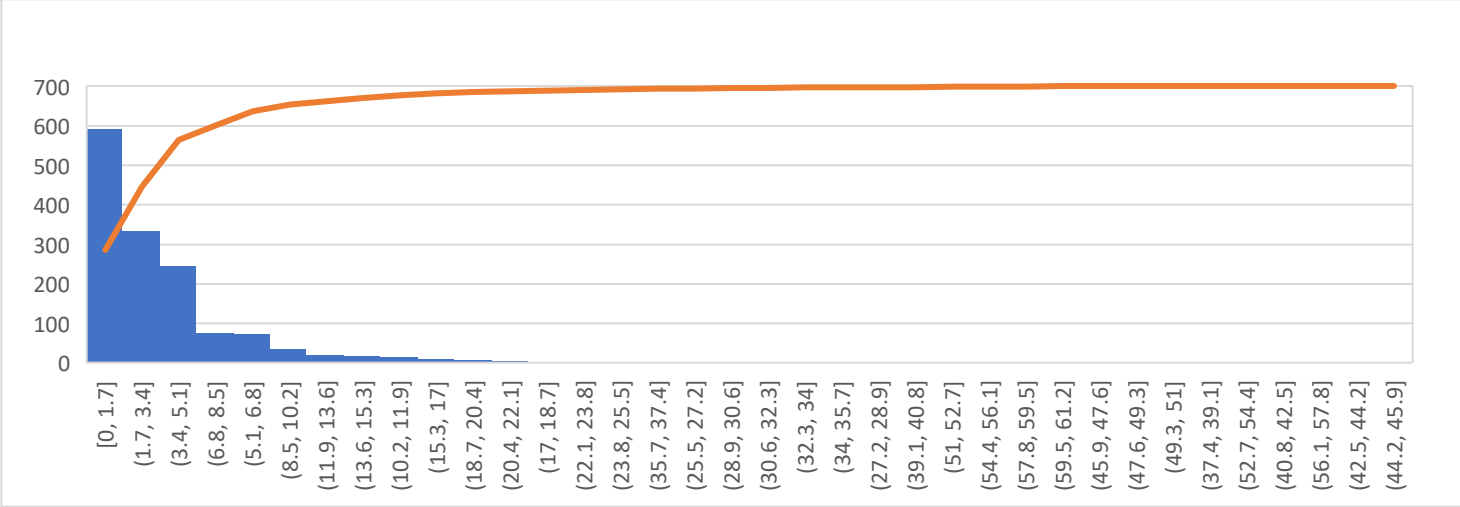



*Fig. 2 CBO HIstogram*

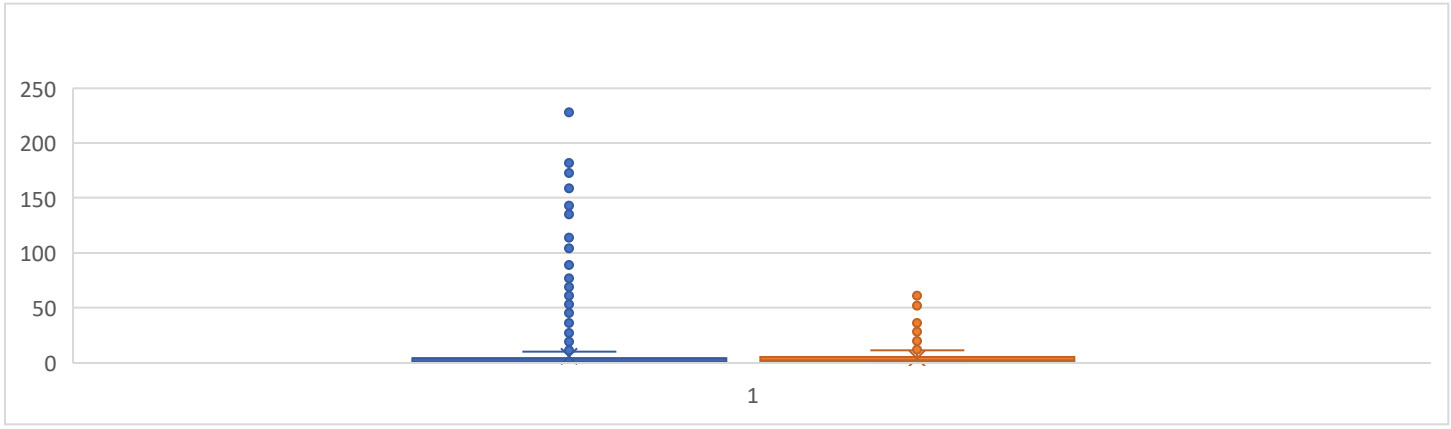



*Fig. 3 WMC & CBO Boxplot*

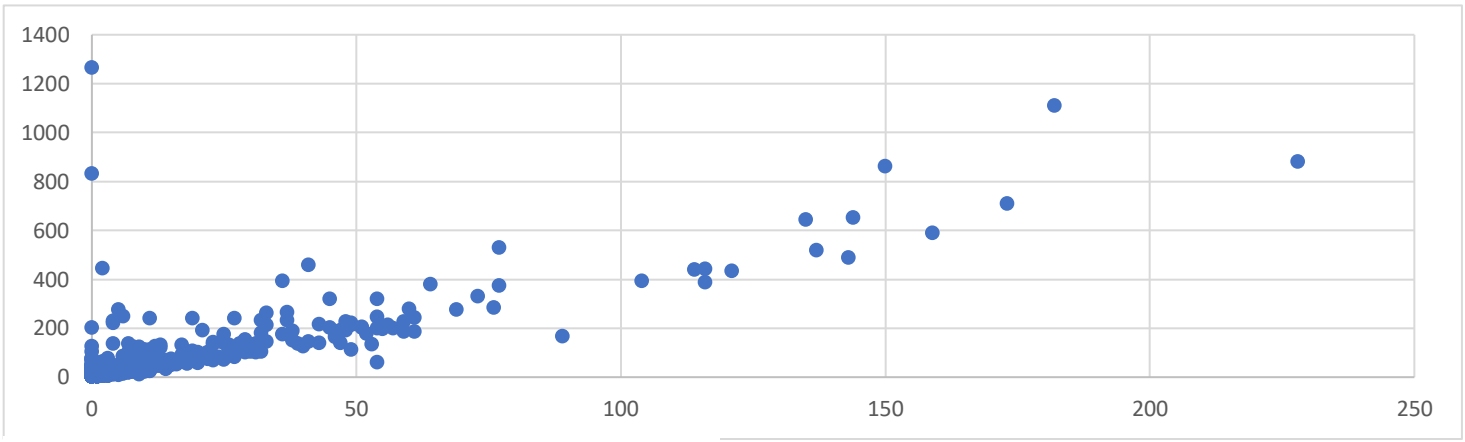



*Fig. 4 WMC vs LOC Scatter plot*

### 4.3 Conductor Project:

**WMC Histogram:**
The WMC histogram for the Conductor project shows the distribution of class complexity. The data suggests that a majority of the classes have moderate complexity, while a few have significantly higher WMC values, which could require additional attention during maintenance.

**CBO Histogram:**
The CBO histogram reflects how classes in the Conductor project are coupled. Most classes have moderate coupling, though there are several with higher CBO values, indicating potential areas where high dependency might affect maintainability.

**WMC vs LOC Scatter Plot:**
The scatter plot for Conductor shows the relationship between class size and complexity. Similar to other projects, larger classes tend to have higher WMC values, which may indicate that larger classes are more complex and could present challenges for maintainability.

**WMC & CBO Boxplot:**
The boxplot compares WMC and CBO values in Conductor, highlighting the range of complexity and coupling across the project. Outliers suggest that certain classes are exceptionally complex or highly coupled, which could affect the maintainability of the system.

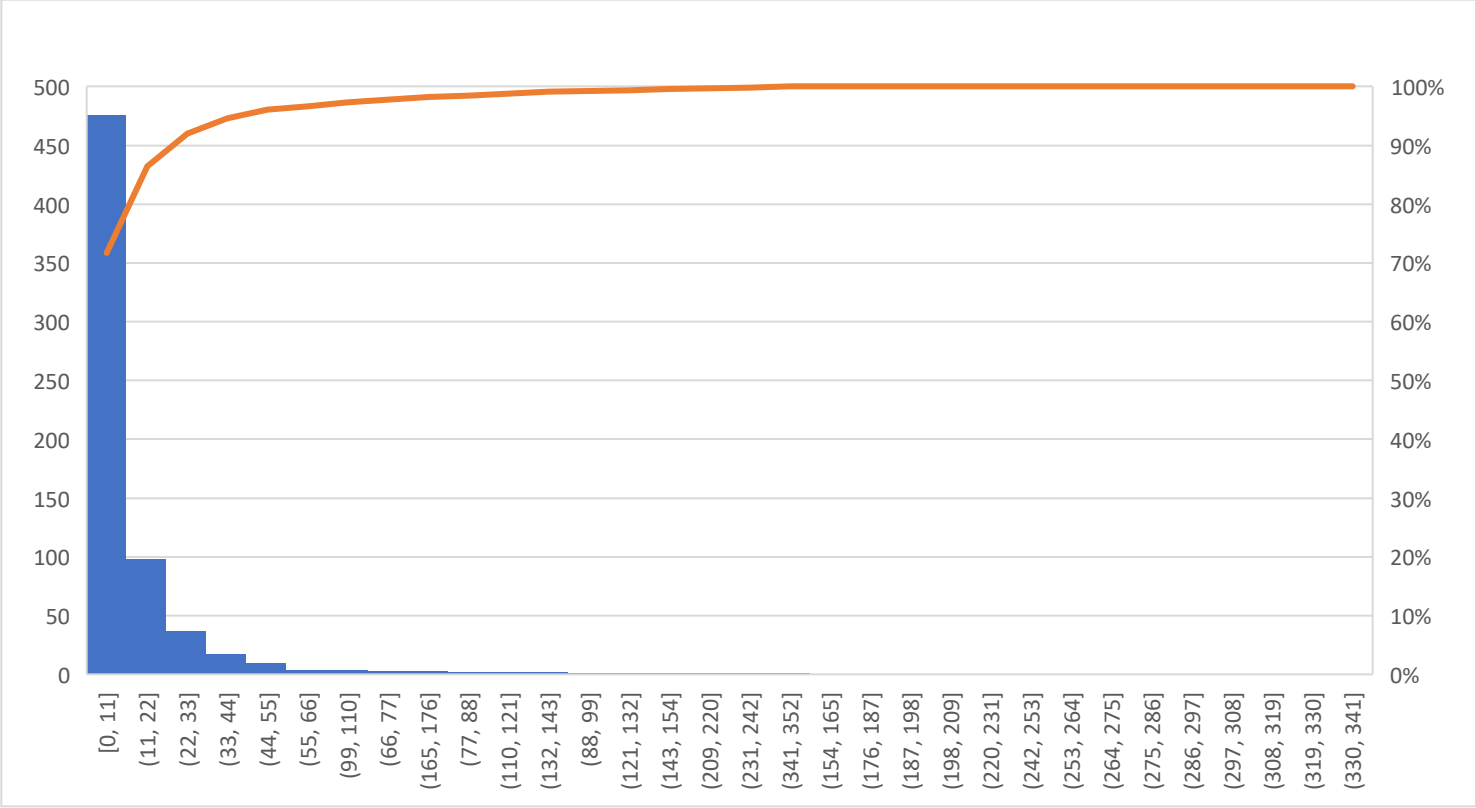



*Fig. 1 WMC Histogram*

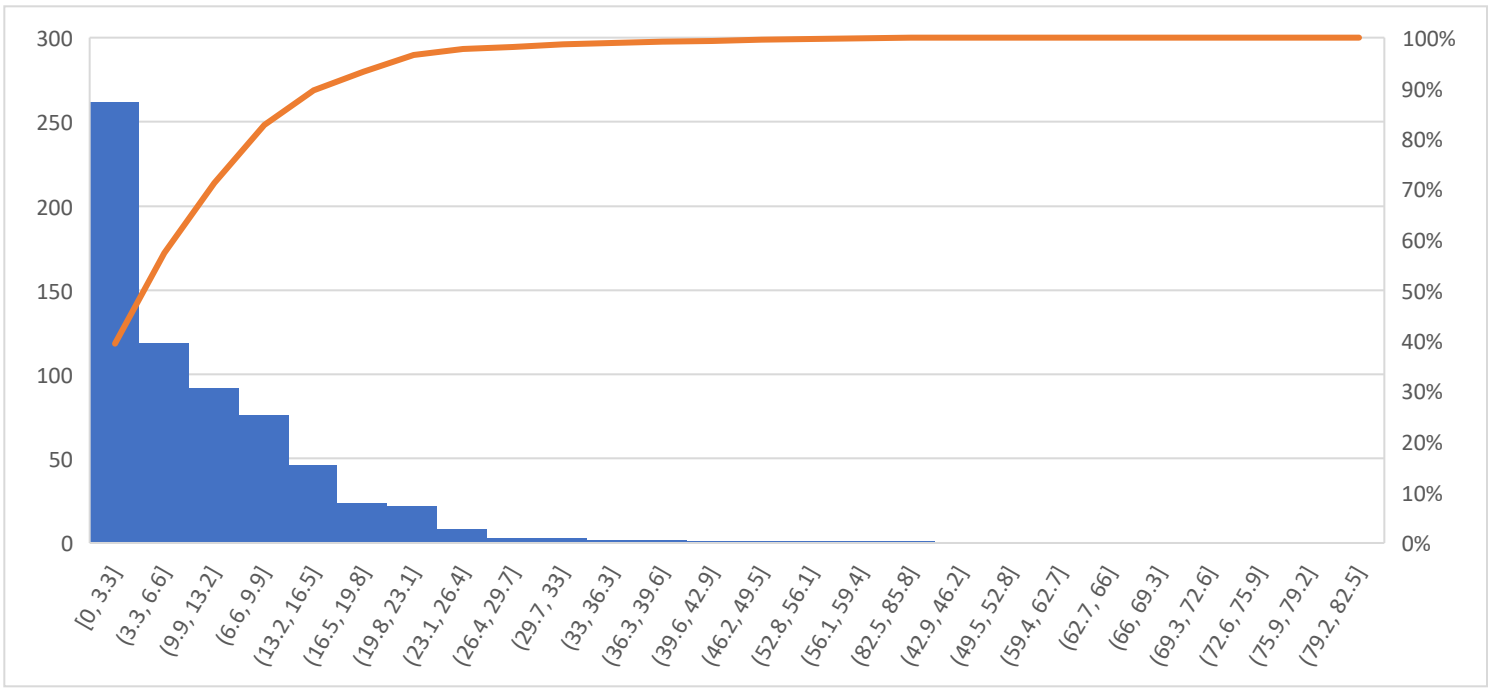



*Fig. 2 CBO HIstogram*

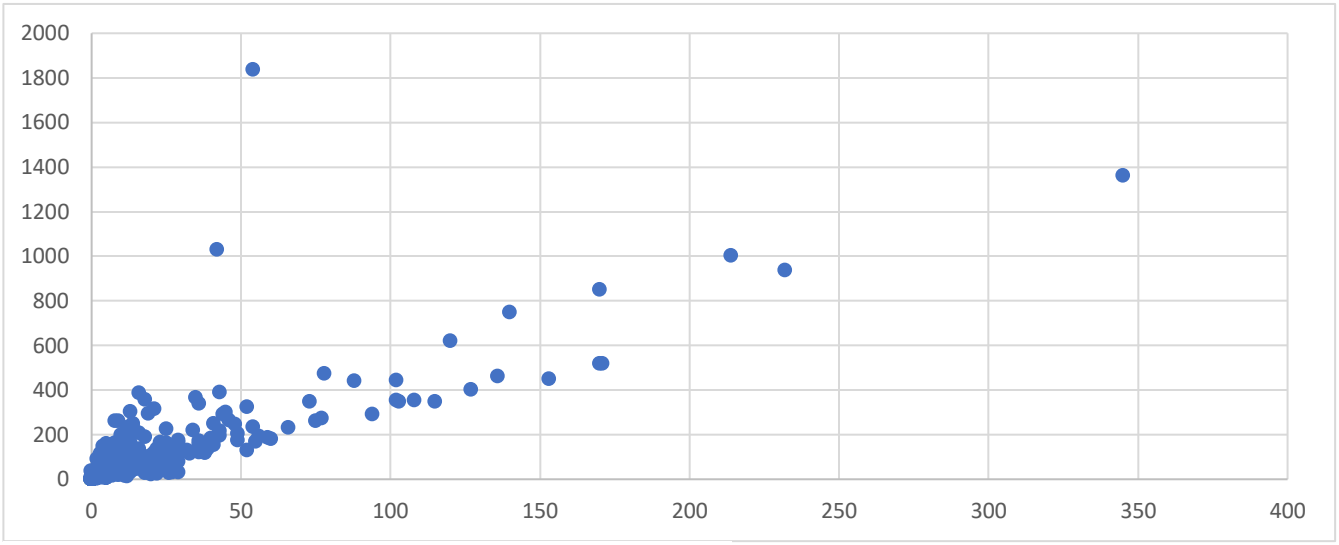



*Fig. 3 WMC vs LOC Scatter plot*

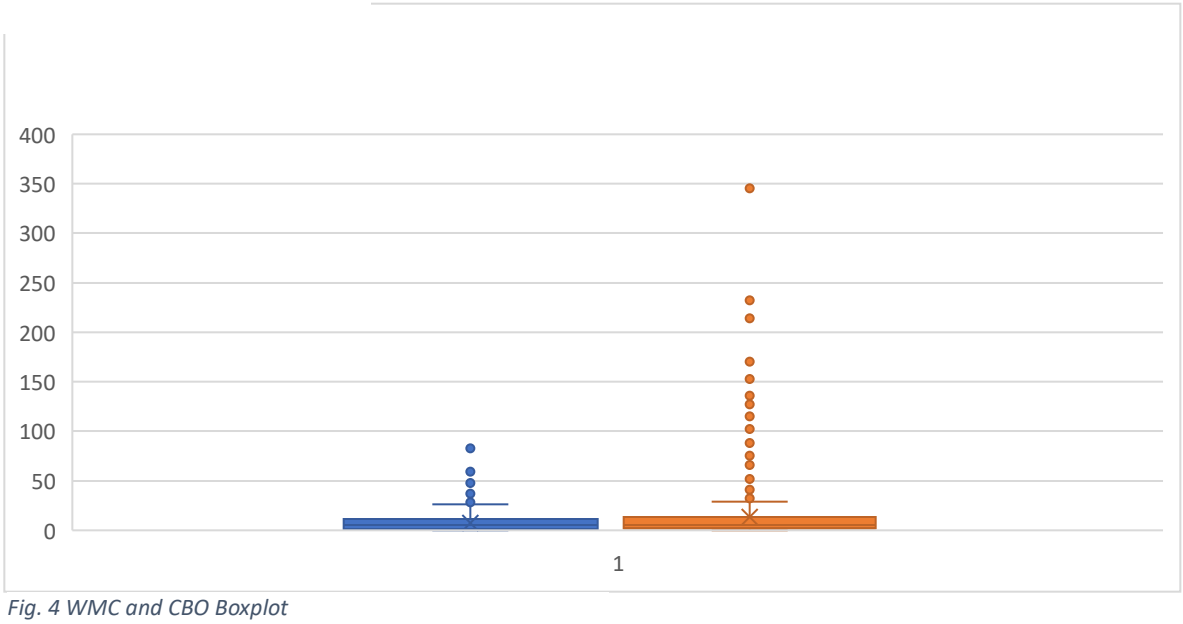



*Fig. 4 WMC and CBO Boxplot*

## 4.4 Spring Security Project:

**WMC Histogram:**
This histogram shows the distribution of WMC values in the Spring Security project. Most classes have moderate WMC values, though a few outliers exhibit higher complexity, which may indicate areas that require extra maintenance effort.

**CBO Histogram:**
The CBO histogram for Spring Security reflects a distribution where most classes have moderate coupling, but there are some classes with high coupling, indicating potential maintainability challenges due to class dependencies.

**WMC vs LOC Scatter Plot:**
The scatter plot illustrates the relationship between class size and complexity. The trend shows that larger classes tend to have higher WMC values, implying that they are more complex, which could affect their maintainability.

**WMC & CBO Boxplot:**
The boxplot highlights the range of WMC and CBO values in Spring Security. The presence of outliers with high complexity or coupling suggests that these classes may pose challenges for maintainability.

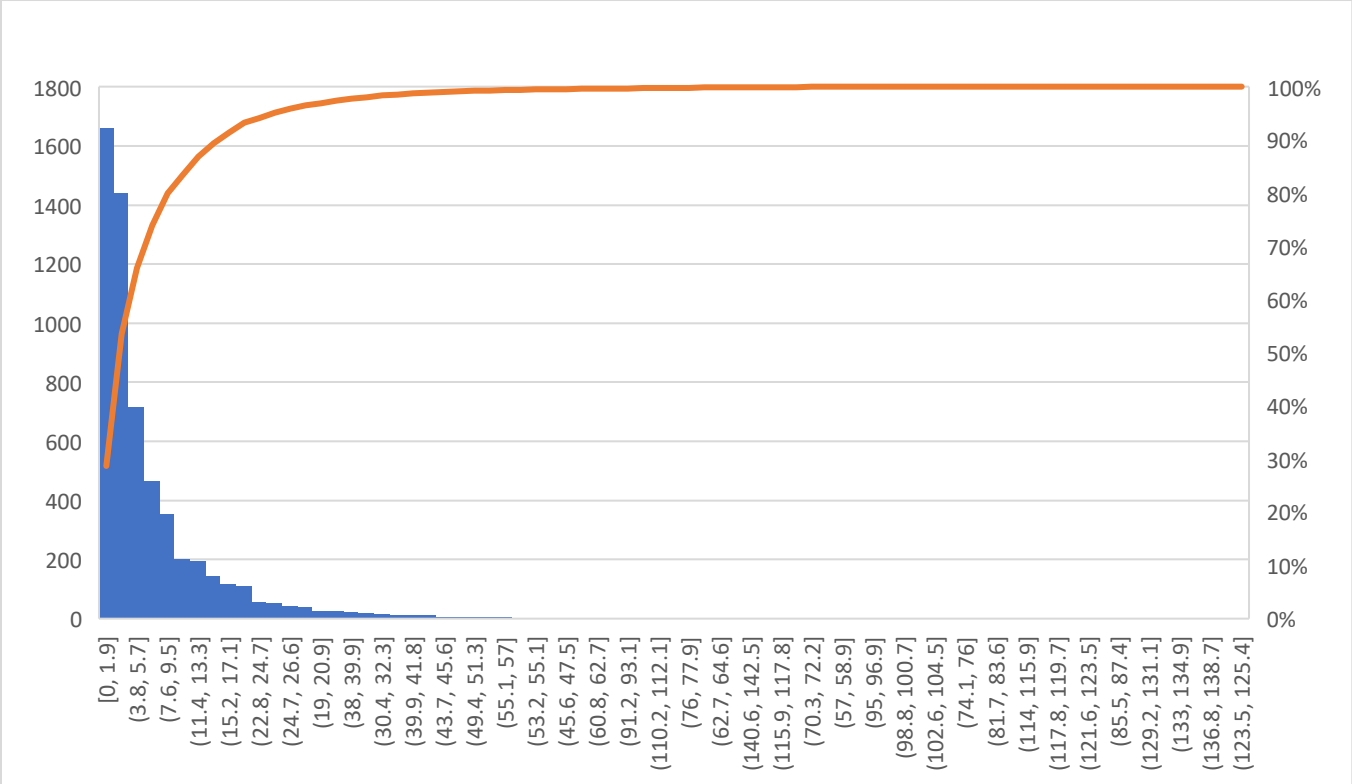



*Fig. 1 WMC Histogram*

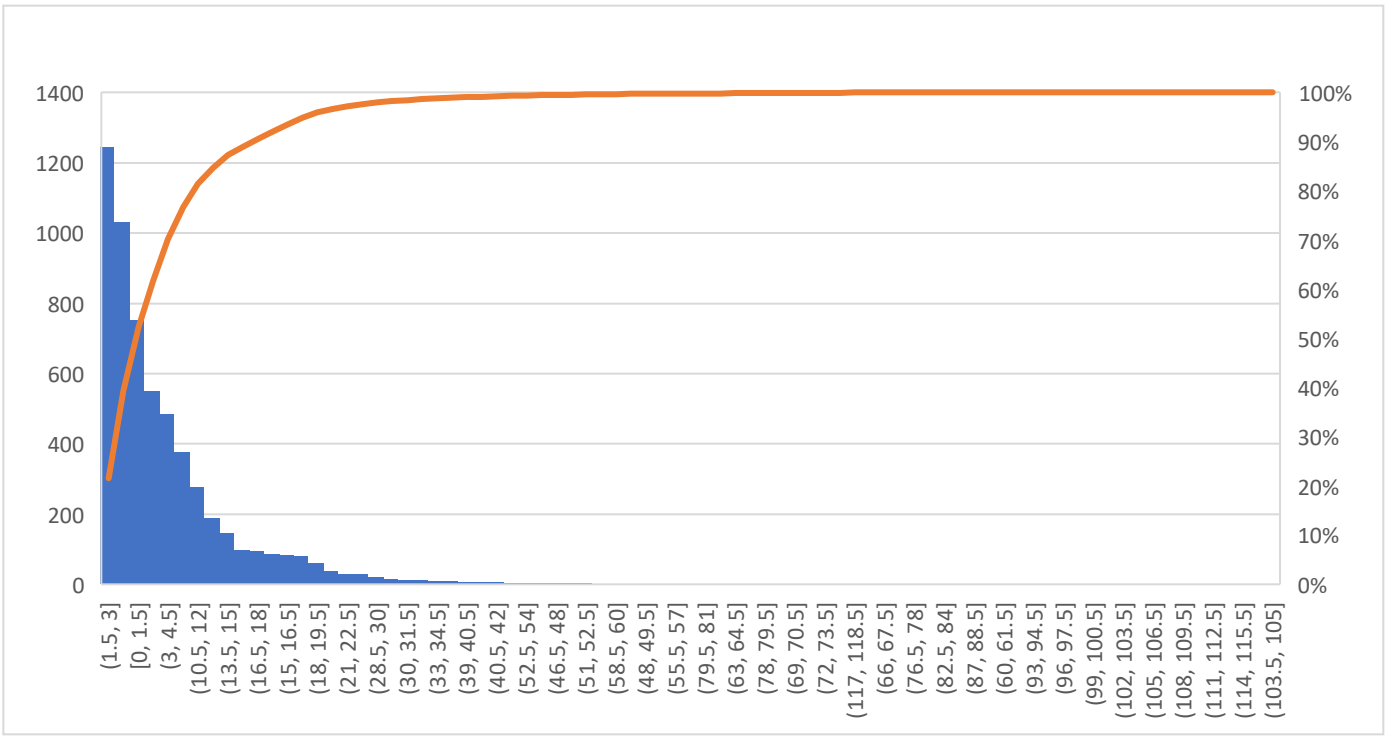



*Fig. 2 CBO HIstogram*

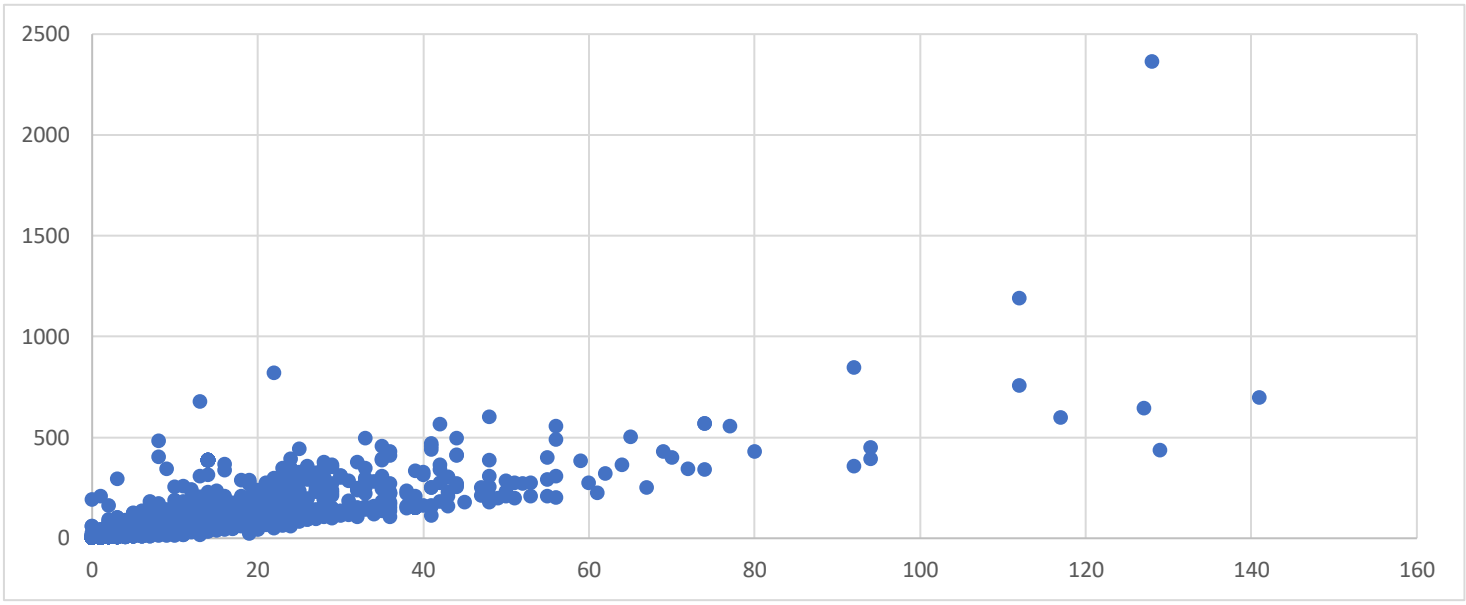



*Fig. 3 WMC vs LOC scatter plot*

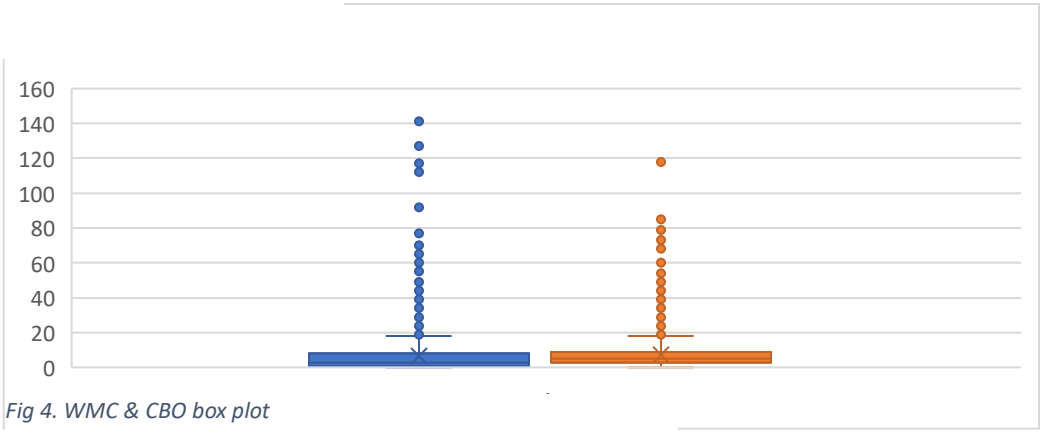



*Fig 4. WMC & CBO box plot*

## 4.5 Error-prone Project:

**WMC Histogram:**
The histogram for WMC values in Error-prone shows that most classes have moderate complexity, with a few classes having higher WMC values, which might require more maintenance effort due to their complexity.

**CBO Histogram:**
The CBO histogram highlights the coupling between classes in Error-prone. While most classes exhibit low to moderate coupling, there are a few classes with higher coupling, potentially complicating their maintenance.

**WMC vs LOC Scatter Plot:**
The scatter plot shows the relationship between class size and complexity in Error-prone. Similar to other projects, larger classes often have higher WMC values, suggesting that as class size increases, so does complexity.

**WMC & CBO Boxplot:**
The boxplot contrasts the complexity (WMC) and coupling (CBO) of classes in Error-prone. Outliers with exceptionally high WMC or CBO values could indicate areas of the codebase that are harder to maintain due to their complexity or dependencies.

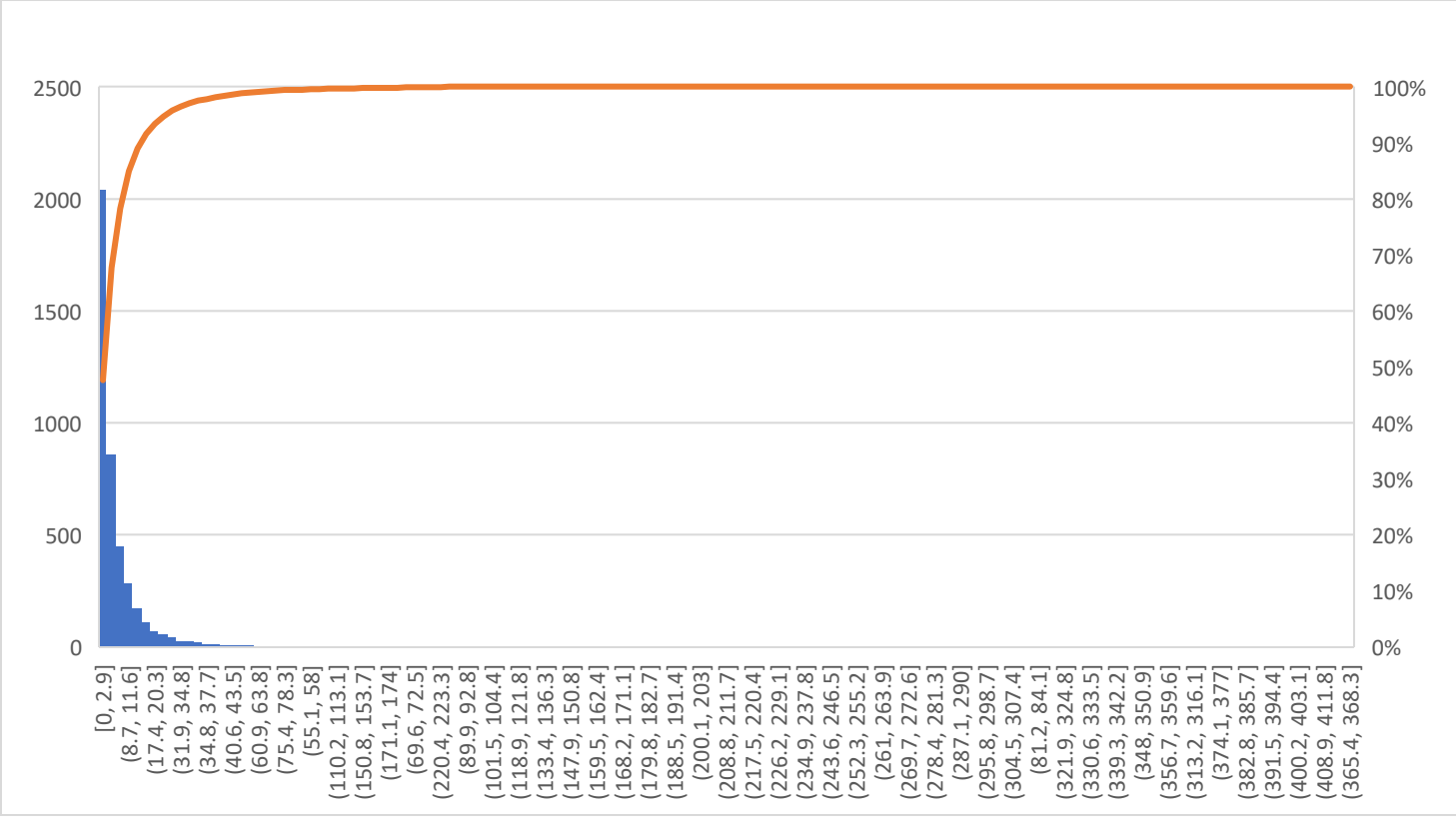



*Fig. 1 WMC Histogram*

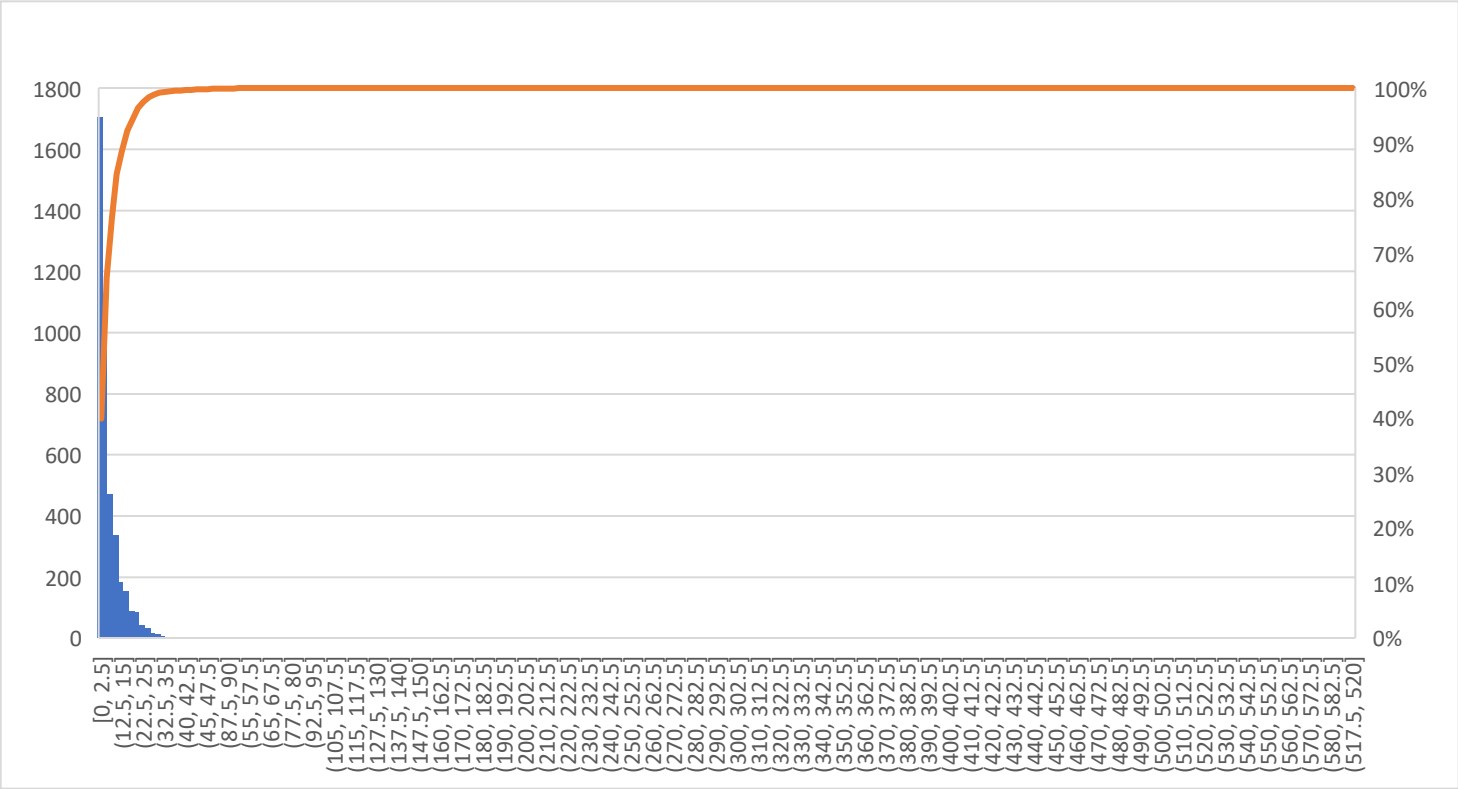



*Fig. 2 CBO HIstogram*

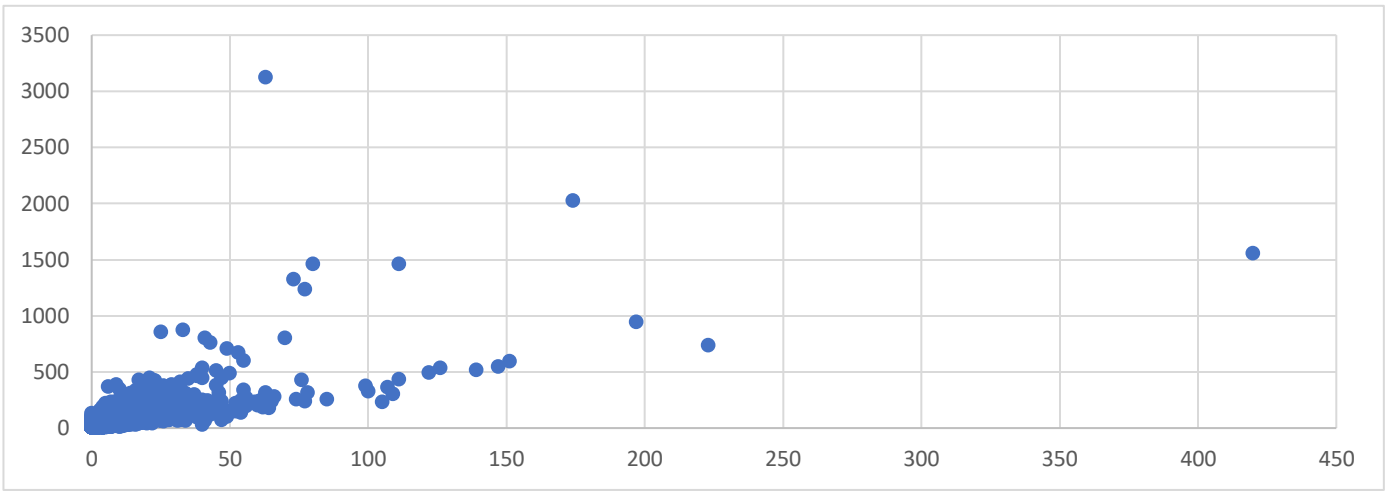



*Fig. 3 WMC vs LOC Scatter*

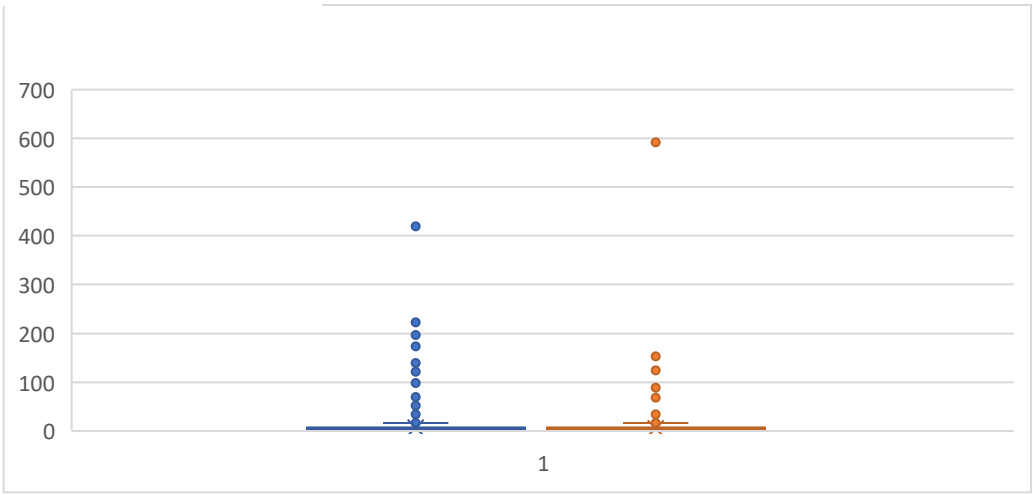



*Fig. 4 WMC and COB box plot*

## 4. Conclusions

The empirical study conducted on the selected Java projects using the C&K metrics has provided valuable insights into their maintainability. By analyzing metrics such as Weighted Methods per Class (WMC) and Depth of Inheritance Tree (DIT), we observed trends that highlight potential areas for improvement. For instance, classes with higher WMC values may indicate complex and potentially less maintainable code, while deeper inheritance trees (higher DIT values) could suggest increased difficulty in understanding and maintaining the codebase. These findings underscore the importance of regular code reviews and refactoring to enhance maintainability.

In conclusion, the analysis of five Java projects using CK metrics (WMC, CBO, and LOC) provides insights into the maintainability of each codebase. Projects such as Conductor and Litho show higher complexity and coupling, indicated by their higher average WMC and CBO values, which suggests that their classes are more challenging to understand and maintain. The positive correlation between class size (LOC) and complexity (WMC) observed across most projects implies that larger classes tend to be more difficult to maintain, highlighting the need for proper size management and refactoring practices.

VirtualXposed and Spring Security, on the other hand, exhibit relatively lower average WMC and CBO values, indicating better modularity and lower complexity, which positively impacts maintainability. The presence of outliers in WMC and CBO metrics across all projects suggests that certain classes are much more complex or tightly coupled than others, which could become maintenance hotspots. Addressing these outliers by simplifying complex classes or reducing coupling could significantly improve the overall maintainability of these projects.

Ultimately, this study highlights the importance of managing class complexity and coupling to ensure software maintainability, with refactoring efforts focusing on large, highly complex, or highly coupled classes. By keeping complexity and dependencies in check, software projects can be made easier to maintain, reducing the effort and cost associated with future modifications.

# References


[1] CK-Code Metrics Tool

https://github.com/mauricioaniche/ck

Project Repositories:

[2] Netflix Conductor

https://github.com/Netflix/conductor

[3] Facebook Litho

https://github.com/facebook/litho

[4] Google Error Prone

https://github.com/google/error-prone

[5] Spring Security

https://github.com/spring-projects/spring-security

[6] VirtualXposed

https://github.com/android-hacker/VirtualXposed