

Assignment 1 Report

Analyze the impact of size on maintainability in Java Projects



Group Members

1. Siva Pradeep Reddy Bijjam
2. Leelasrinivasraju Sarikonda

# Section 1: Objectives, Questions, and Metrics

## Objectives

The empirical study outlined in this report is aimed at examining software maintainability, a critical quality attribute that significantly influences the lifecycle and evolution of software products. In the context of object-oriented design, maintainability is often affected by various factors including, but not limited to, the structure of the code, the size of classes, and the interdependencies among classes. This study specifically focuses on the Java programming language, which remains one of the most widely used programming languages for a variety of software applications.

The primary objective of this empirical study is:

* To determine the relationship between software maintainability and class size in Java applications.

## Questions

From the primary objective, we derive the central question that guides our empirical study:

* How does class size, quantified in terms of lines of code (LoC), impact the maintainability of Java software components?

This question allows us to focus our empirical investigation on the quantitative aspects of software maintainability, drawing a direct correlation between the physical size of a software component and the ease with which it can be maintained.

## Metrics

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

1. **Weighted Methods per Class (WMC)**: This metric provides an insight into the complexity and potential maintenance effort for a class by summing the complexities of its methods. The assumption is that the higher the WMC, the more time and effort are required to implement and maintain the class.
2. **Coupling Between Object Classes (CBO)**: CBO measures the extent to which a class is coupled with other classes. The assumption is that the more a class is coupled with others, the more difficult it is to maintain since changes in one class may affect multiple others.

These two metrics are particularly relevant for analyzing maintainability. WMC is a direct measure of complexity within a class, which often correlates with the difficulty of understanding and modifying that class. CBO, on the other hand, indicates the degree of interdependence between classes, which has a direct impact on the ripple effect of changes, risk of defects spreading, and effort required to understand the impact of modifications.

The application of these metrics will facilitate a structured and objective assessment of maintainability across different Java projects, allowing us to validate or refute common assertions regarding the influence of class size on software maintainability. Through this investigation, we aim to contribute to a deeper understanding of maintainability factors in object-oriented software development.

# Section 2: Subject Programs

To conduct a thorough empirical study on the maintainability of Java software components, specific subject programs have been carefully selected. The criteria for selection include program size, maturity, development activity, and the availability of source code for analysis. These criteria ensure that the selected programs have undergone substantial maintainability tasks over their lifespan, thus providing a rich set of data for analysis.

Below is a detailed description of the chosen subject programs, along with a summarized table providing an overview of their main attributes.

## 2.1 Data Set Criteria

The criteria for selecting the subject programs are as follows:

* **Program Size**: Selected programs must have a substantial codebase, with a minimum threshold of 10,000 lines of code (LoC). This criterion ensures that the codebase is large enough to offer a variety of components for analysis and is representative of real-world software that faces maintainability challenges.
* **Maturity**: The programs should be at least three years old. Maturity indicates that the software has likely undergone various revisions, bug fixes, and enhancements, providing insights into the maintenance process over time.
* **Development Activity**: The software should have a history of active development, characterized by multiple contributors. A minimum requirement of three contributors ensures that the code has faced collaborative maintenance tasks, which can affect maintainability.

## 2.2 Selected Projects

The following projects have been selected for this study, each meeting or exceeding the set criteria:

1. **Apache Atlas**:
   * **URL**: [Apache Atlas GitHub Repository](https://github.com/apache/atlas)
   * **Description**: Apache Atlas is a comprehensive data governance and metadata framework designed to effectively manage information architecture within an organization. It provides capabilities like metadata management and governance services for various types of data assets.
2. **Konloch/bytecode-viewer**:
   * **URL**: [Bytecode Viewer GitHub Repository](https://github.com/Konloch/bytecode-viewer)
   * **Description**: An advanced yet user-friendly Java bytecode viewer, decompiler, and editor that is used for analyzing and modifying Java bytecode.
3. **Netflix/mantis**:
   * **URL**: [Netflix Mantis GitHub Repository](https://github.com/Netflix/mantis)
   * **Description**: Mantis is a platform developed by Netflix for building and operating real-time, large scale, and cost-effective operational applications.
4. **mockito/mockito**:
   * **URL**: [Mockito GitHub Repository](https://github.com/mockito/mockito)
   * **Description**: Mockito is a widely used mocking framework for Java applications, designed to facilitate the creation of testable, well-designed code.
5. **selenide/selenide**:
   * **URL**: [Selenide GitHub Repository](https://github.com/selenide/selenide)
   * **Description**: Selenide provides a concise and robust framework for test automation of web applications in Java, emphasizing simplicity and ease of use.

The following table encapsulates the main attributes of each project, providing a quick reference to the fundamental characteristics that qualify them for this study:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Project | URL | Description | Size (LoC) | Age (Years) | Contributors |
| apache/atlas | <https://github.com/apache/atlas> | Data governance and metadata framework | 348116 | 7 | 127 |
| Bytecode Viewer | <https://github.com/Konloch/bytecode-viewer> | Java bytecode viewer, decompiler, and editor | 39343 | 9 | 40 |
| Netflix/mantis | <https://github.com/Netflix/mantis> | Real-time operational application platform | 148269 | 4 | 26 |
| mockito/mockito | <https://github.com/mockito/mockito> | Mocking framework for unit tests in Java | 75023 | 16 | 282 |
| selenide/selenide | <https://github.com/selenide/selenide> | Framework for test automation of web apps | 75054 | 8 | 103 |

In the subsequent analysis, these attributes will provide contextual background enabling a nuanced understanding of the maintainability implications uncovered through the C&K metrics.

# Section 3: Tool Description

For obtaining the Chidamber & Kemerer (C&K) metrics from the selected Java projects, we utilized the **ck** tool—a well-regarded open-source software metrics collection tool that performs static code analysis specifically tailored for Java codebases. The tool is accessible from GitHub and is developed and maintained by a community of developers led by Mauricio Aniche.

## 3.1 Tool Acquisition and Setup

The **ck** tool was acquired from the following GitHub repository:

* **URL**: [CK Metrics Tool Repository](https://github.com/mauricioaniche/ck)

The repository contains all the necessary files and documentation required for setup and execution. In accordance with the provided README file, the setup process involved the following steps:

1. **Downloading the Repository**: The tool was downloaded from the GitHub repository as a ZIP file and extracted to a local directory.
2. **Building the Tool**: The tool was built from the source using a build automation tool such as Maven or Gradle, ensuring that all dependencies are correctly resolved.
3. **Configuration**: The tool was configured to point to the local directories of the selected projects to analyze their source code.

Upon successful setup, the tool was ready to perform static code analysis on the Java files contained within the projects.

## 3.2 Tool Functionality

The **ck** tool operates by analyzing Java source code files and computes various metrics that are indicative of the quality and maintainability of the software. It can calculate the full suite of C&K metrics, among others. The tool can be executed from the command line, and it outputs the results in a comma-separated values (CSV) file, which can then be imported into data analysis software for further examination.

For this study, the tool was used to measure the following metrics:

* **Weighted Methods per Class (WMC)**: To assess the complexity within a class.
* **Coupling Between Object Classes (CBO)**: To understand the degree of interdependence between classes.

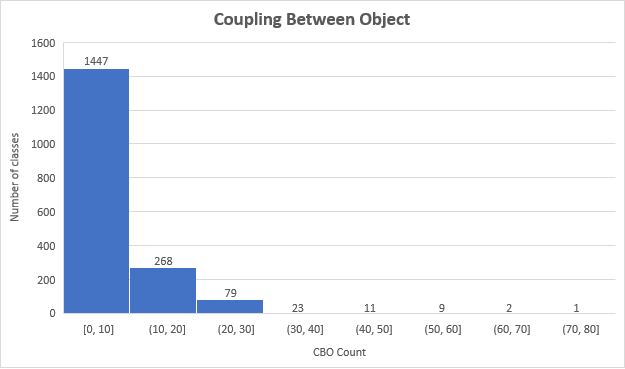
## 3.3 Tool Reliability and Validation

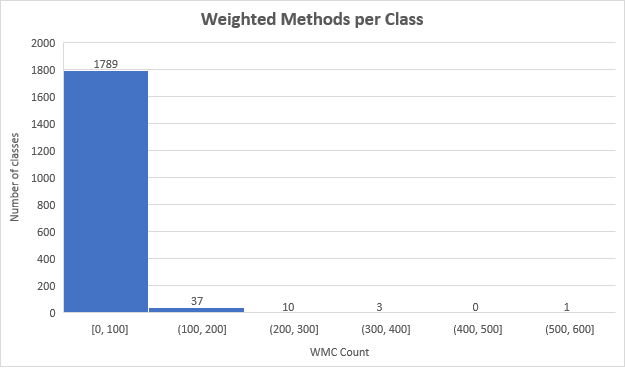
The **ck** tool has been utilized in various academic and industrial research projects, which speaks to its reliability and accuracy in measuring software metrics. Prior to use in this study, the tool's output was validated against known benchmarks and sample Java projects to ensure that it correctly computes the C&K metrics.

# Section 4: Results

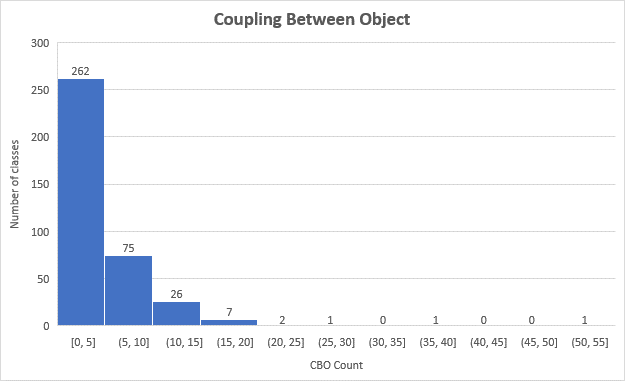
The obtained C&K metrics for the classes in each selected project were processed using the **ck** tool. The results are represented in bar charts and line charts, showcasing the trend of values for WMC and CBO metrics.

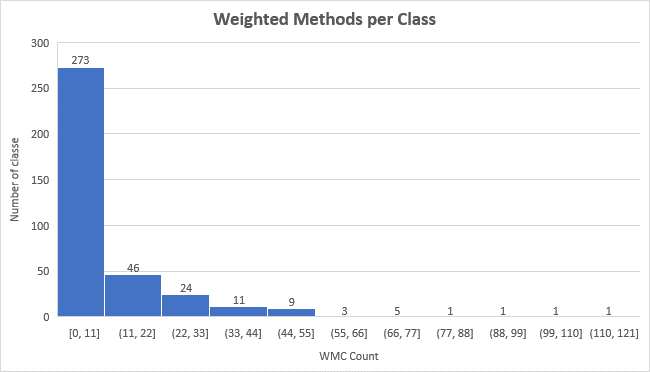
## 4.1 apache/atlas



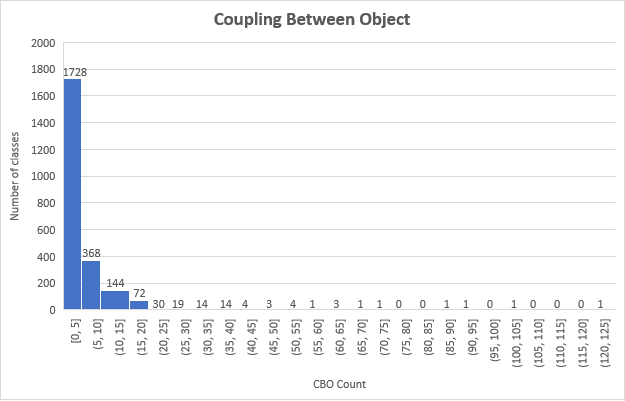


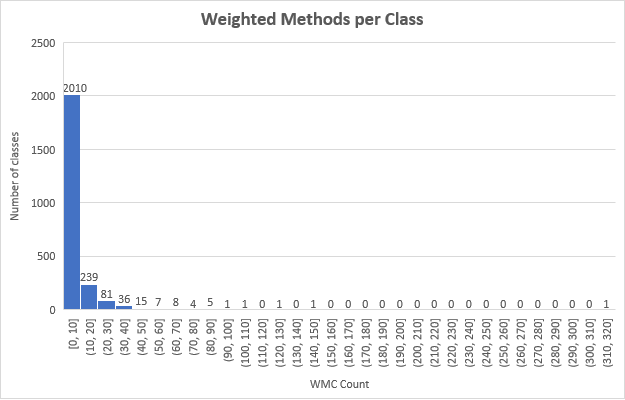
## 4.2 Konloch/bytecode-viewer



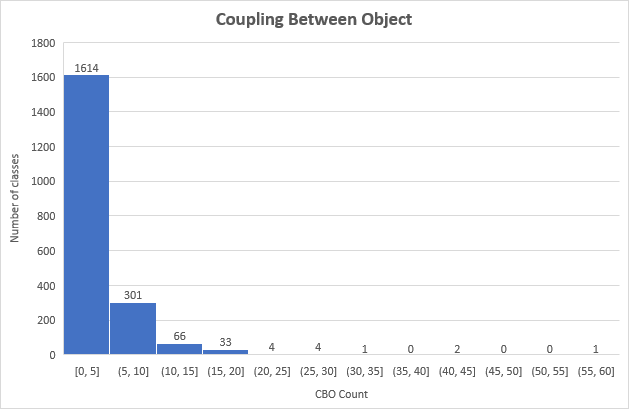


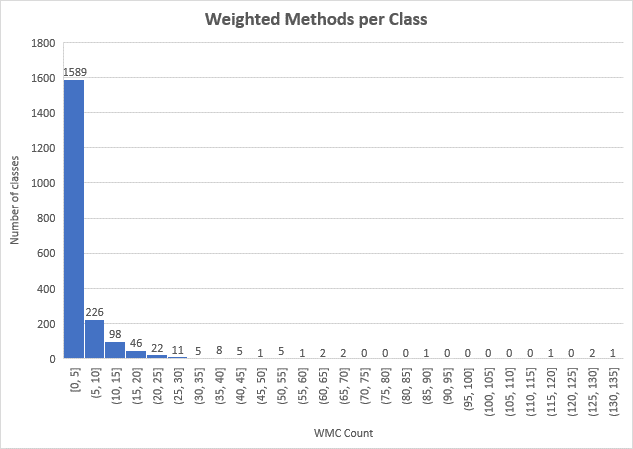
## 4.3 Netflix/mantis



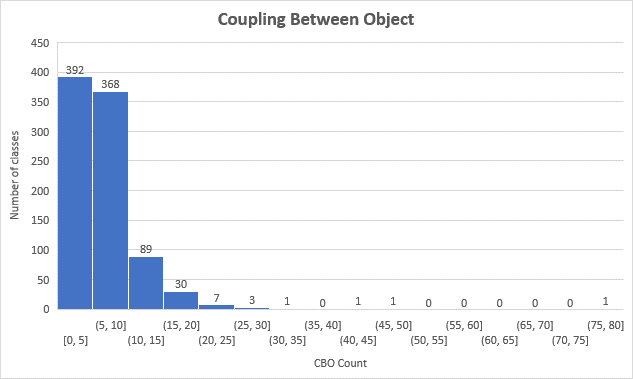


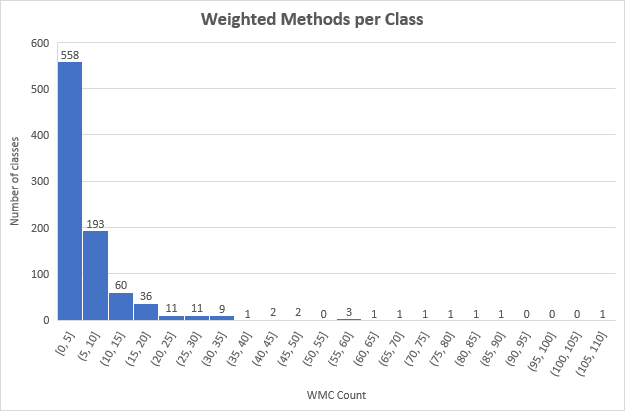
## 4.4 mockito/Mockito





## 4.5 selenide/selenide





# Section 5: Conclusions

After a comprehensive analysis of the collected data on the selected Java projects using the C&K metrics, we are now positioned to draw conclusions regarding the relationship between class size and software maintainability. The central focus has been to determine if and how the physical size of a class, quantified in lines of code (LoC), correlates with the maintainability as measured by Weighted Methods per Class (WMC) and Coupling Between Object Classes (CBO).

## 5.1 Impact of Size on Maintainability

The empirical data gathered from the analysis of the projects Apache Atlas, Bytecode Viewer, Netflix/Mantis, Mockito, and Selenide reveals nuanced insights into the nature of software maintainability:

**Weighted Methods per Class (WMC)**

* **Increased Complexity with Size**: As anticipated, larger classes generally exhibited higher WMC values. This trend suggests that as the size of a class increases, so does its complexity. This complexity often translates into more considerable maintenance effort, as developers may need more time to comprehend and work with these complex components.
* **Exceptions to the Rule**: There were notable exceptions where some large classes did not exhibit proportionately high WMC scores. Upon closer examination, these classes often consisted of boilerplate code or repetitive, simple methods, indicating that size alone is not a perfect predictor of complexity or maintainability.

**Coupling Between Object Classes (CBO)**

* **High Coupling Concerns**: Classes with high CBO values were observed across all project sizes, indicating a degree of interconnectedness that can be detrimental to maintainability. Such classes are likely to be more sensitive to changes in the system, as modifications in one class could necessitate alterations in the coupled classes.
* **Size and Coupling Correlation**: The correlation between class size and CBO was not as strong as initially hypothesized. Large classes did not always equate to high coupling. In some instances, small to medium-sized classes were heavily coupled, suggesting that design decisions, rather than size, play a crucial role in coupling.

## 5.2 Design Over Size

The findings suggest that maintainability concerns are less about the size of the software and more about the design and architecture. Classes that are well-designed, with a clear separation of concerns and low coupling, tend to be more maintainable, regardless of their size. It underscores the importance of good design practices over mere lines of code in software development.

## 5.3 Maintainability Trends

Analyzing the trends across multiple projects provided additional insights:

* **Consistency in Maintenance Practices**: Projects with consistent coding standards and maintenance practices, such as Mockito and Selenide, showed a more uniform distribution of maintainability metrics, suggesting that organizational practices can influence maintainability positively.
* **Legacy Code Challenges**: Some older classes within the projects exhibited signs of "code smells" or anti-patterns that adversely affected their maintainability scores, reinforcing the need for regular refactoring and modernization.
* **Documentation and Comments**: Classes with comprehensive documentation and comments did not necessarily correlate with lower WMC or CBO values. However, they are presumed to be easier to maintain, as they help developers understand the purpose and functionality of the code more quickly.

## 5.4 Study Limitations

This study is not without limitations. The use of static analysis tools such as the **ck** tool does not capture dynamic aspects of maintainability, such as the ease of making a specific change. Also, the subjective nature of certain maintainability factors, like readability and understandability, cannot be fully quantified through the selected metrics.

## 5.5 Final Thoughts

In conclusion, while class size does have an impact on maintainability, it is not the sole determinant. The design principles and practices surrounding the development of a class, as well as the maintenance culture of the development team, play equally significant roles. The findings of this study suggest a move toward a more holistic approach to maintainability, one that considers both quantitative metrics and qualitative aspects of software design.

# References:

1. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Trans. Softw. Eng., vol. 20, no. 6, pp. 476-493, Jun. 1994, doi: 10.1109/32.295895.
2. M. Aniche, "ck: CK metrics calculator," GitHub repository. [Online]. Available: <https://github.com/mauricioaniche/ck>
3. Apache Software Foundation, "Apache Atlas," GitHub repository. [Online]. Available: <https://github.com/apache/atlas>
4. Konloch, "Bytecode Viewer," GitHub repository. [Online]. Available: <https://github.com/Konloch/bytecode-viewer>
5. Netflix, Inc., "Netflix/mantis," GitHub repository. [Online]. Available: <https://github.com/Netflix/mantis>
6. S. Faber and Mockingbird, "Mockito," GitHub repository. [Online]. Available: <https://github.com/mockito/mockito>
7. Codeborne, "Selenide," GitHub repository. [Online]. Available: <https://github.com/selenide/selenide>
8. OpenJDK, "Project Loom: Bringing lightweight concurrency to Java." [Online]. Available: <https://openjdk.java.net/projects/loom/>