**Evaluating Software Maintainability through CK Metrics: An Empirical Study on Open-Source Java Projects**

***Presented By***

**Vijesh Medikonda - L30085988**

**Sowmya Keerthigari - L30088015**

**Aafrinben Farukbhai Vohra -L30092152**

**Fall 2024 -I**

**Abstract**

This study's goal is to assess software maintainability with an emphasis on open-source Java projects by utilizing the Chidamber and Kemerer (C&K) metrics package. We analyze important metrics that are known to affect maintainability, such as Weighted Methods per Class (WMC) and Coupling Between Object Classes (CBO), using a static analysis on a carefully chosen dataset of Java applications. Through examining these measures in various classes, we investigate the connection between the size of software and its maintainability. Graphs that show our findings draw attention to significant trends and anomalies that may have an impact on long-term maintainability. The study's conclusions have applications for developers looking to improve code quality by providing direction for refactoring and design enhancements. Furthermore, this empirical study contributes to the validation of the use of C&K metrics in software quality evaluation and maintainability forecasting.

*Keywords:* CK Metrics, Weighted Methods per Class (WMC), Coupling Between Object Classes (CBO), software maintainability, Java, static analysis.

**1. Introduction**

One important factor that affects the long-term viability and adaptability of software systems is software maintainability. It shows how quickly a system can be improved, changed, or fixed, all of which have an impact on how much a system will cost over its lifetime. The Chidamber and Kemerer (C&K) metrics are among the many measures available for assessing maintainability, and they have become important markers of software quality. Specifically, Coupling Between Object Classes (CBO) and Weighted Methods per Class (WMC) are well-known as critical metrics that provide information on how maintainable object-oriented systems are.

This empirical study applies these measures to a selection of open-source Java projects with different sizes and levels of complexity to investigate how they affect maintainability. We also investigate how class size affects maintainability, since larger classes are typically more intricate and difficult to change, which might have an impact on the quality of the code over time. We want to find patterns and trends that relate these variables to maintainability through static analysis, with a particular focus on how project size, age, and complexity may lead to a decline in maintainability.

We want to offer useful insights through our analysis of the metrics data obtained, so that software development teams may make well-informed judgments on refactoring tactics and design enhancements that will ultimately lead to increased maintainability and decreased technical debt.

* 1. **Research Aim:**

The purpose of this study is to investigate how class size affects software maintainability in Java-based applications. The particular goals are:

a) **Class Size Evaluation**: To examine how big a class is to its overall maintainability and quantify it in terms of lines of code, or LoC. The fundamental question is whether longer courses require more upkeep over time due to their increased complexity.

b) **Measuring Maintainability using C&K Metrics**: Evaluate maintainability using the Chidamber and Kemerer (C&K) metrics, especially the Coupling Between Object Classes (CBO) and Weighted Methods per Class (WMC). The goal is to ascertain how class size affects these measures, which represent inter-class dependency and complexity.

c) **Effect of Class Size on Maintainability**: Examining how class size affects important aspects of maintainability such as the simplicity of debugging, refactoring, and adding functionality. The goal is to identify potential size thresholds that, when exceeded, make maintainability more difficult while providing developers with useful information.

By addressing these objectives, the study aims to offer evidence-based recommendations for enhancing class design, size management, and structural organization to increase software maintainability.

* 1. **Research Questions**

The following queries will be the main focus of the study:

**1)How do the software maintainability metrics of WMC and CBO relate to class size?**

The purpose of this inquiry is to investigate the relationship between higher coupling (CBO) and greater complexity (WMC) and larger class sizes. It aims to ascertain whether complexity and interdependencies increase disproportionately with class size and whether larger classes are intrinsically harder to maintain.

**2)How does class size impact the ease of modification and refactoring in terms of maintainability?**

This inquiry explores whether larger classes are more vulnerable to maintenance issues including expensive refactoring, difficult debugging, and reduced adaptability because of their greater complexity and coupling. It searches for possible cutoff points at which the size of the class begins to impair maintainability.

The investigation into the relationship between class size, complexity, and maintainability in actual Java programs will be guided by these research topics.

* 1. **Key Metrics for Analysis**

**i)Lines of Code (LoC):** This metric, which counts the executable lines in each class, will be the main way to determine class size. This measure aids in evaluating a class's physical footprint. Large classes can indicate modules with lots of features, but an excessive LoC is typically associated with complexity and greater maintenance costs. LoC acts as a starting point for assessing the class's overall volume, which could cause problems like exceeded code or trouble keeping logical flow.

**ii)Weighted Methods per Class (WMC):** By weighing the quantity and intricacy of each method, the WMC metric assesses the complexity inside a class. The weight of each approach is either derived from counting methods or from its cyclomatic complexity. More complicated classes that are more difficult to manage, test, and comprehend are indicated by higher WMC scores. Because of this, it is a crucial indicator for assessing a class's internal organization and maintainability. Additionally, a high WMC may make it more likely that mistakes may be made when implementing changes.

**iii)Coupling Between Object Classes (CBO):** This technique gauges how reliant a class is on other classes. A class with a greater CBO is said to be heavily linked, which may make maintenance more challenging. Changes in highly connected classes can have a multiplicity of effects on other classes, making alterations riskier. Highly coupled classes typically have more dependencies. This measure is essential for comprehending the interactions between classes and the external complexity. It also sheds light on the possible effects of modifications and how tight coupling affects maintainability.

These metrics will offer a thorough understanding of how complexity and class size affect maintainability, assisting in the identification of patterns that can lead to more difficult-to-adapt or alter code.

**2. Dataset Description and Selection Strategy**

We created the following standards for choosing the Java projects to ensure that they offer insightful studies of how class size affects software maintainability:

**2.1 Selection Strategy of Projects:**

The following criteria were used in the selection process to make sure the selected Java projects provide insightful information about the connection between class size and software maintainability:

a) **Codebase Size Requirement**: Projects need to have a minimum of 20,000 Lines of Code (LoC) in their codebase. This guarantees that the class structures in the software are sufficiently complicated to enable a useful study of maintainability problems. The complex systems that frequently face maintainability issues are reflected in the size of the codebase.

b) **Longevity of the Project**: At least seven years must pass between the projects that are chosen. This criterion provides a clear understanding of how class size affects maintainability over time by ensuring that the software has undergone multiple periods of development, evolution, and maintenance.

c) **Contributor Base**: At least 50 active contributors are required for a project. Variations in coding styles and teamwork brought about by a larger development team can impact the software's maintainability. This guarantees that the study captures the intricacy of relationships across multiple developers.

**2.2 Studied Projects:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S.NO** | **Project Title** | **Number of Developers** | **Age of the project (Years)** | **Codebase size (LOC)** | **Project Overview** |
| 1 | huanghongxun/HMCL | 51 | 7 | |  | | --- | | 23488 | |  | |  | |  | |  | | It is a well-liked, multipurpose, cross-platform Minecraft launcher |
| 2 | OpenHFT/Chronicle-Queue | 58 | 11 | 22164 | It is a microsecond messaging that saves all data to disk |
| 3 | real-logic/simple-binary-encoding | 76 | 11 | |  | | --- | | 24735 | | Message Codec with High Performance: Simple Binary Encoding (SBE) |
| 4 | FasterXML/jackson-core | 59 | 13 | 21767 | Jackson's core component that defines the Streaming API and the fundamental shared abstractions |
| 5 | Discord4J/Discord4J | 70 | 7 | 22690 | Using the official Discord Bot API, Discord4J is a fast, powerful, unbiased, reactive library that makes it simple to create Discord bots for Java, Kotlin, and other JVM languages. |

**2.3 Brief Description of the selected projects**

1. **huanghongxun/HMCL**: An open-source, lightweight launcher for Minecraft is called HMCL (Hello Minecraft! Launcher). It makes it simple for users to maintain several Minecraft versions, setups, and mods. Along with complex features like skin changing, multi-account support, and Java version control, it supports a number of systems. It is intended to serve as a user-friendly and customizable substitute for the official Minecraft launcher.

2. **OpenHFT/Chronicle-Queue (Chronicle-Queue):** A thread-safe, low-latency, high-performance persisted messaging library for microservices, stream processing, and event sourcing is called Chronicle Queue. By publishing to memory-mapped files, which users then read sequentially, it offers an efficient way to store and handle massive volumes of data. Its ultra-low latency, robust messaging and event logging capabilities make it ideal for real-time data processing in financial services.

3. **Simple Binary Encoding (real-logic/simple-binary-encoding)**: A high-performance messaging codec for low-latency trading and financial systems is called Simple Binary Encoding (SBE). It is a component of the Aeron framework, which emphasizes throughput optimization and message size reduction. High-performance communication between computers is necessary for real-time applications, and SBE provides an effective way to encode and decode messages in a binary format.

4. **Jackson Core [FasterXML/jackson-core]**: Jackson Core is the Jackson JSON processor's core library. It offers low-level JSON streaming capabilities, including parsing and writing. Jackson Core is a component of a broader Jackson library suite that facilitates the conversion of JSON data into Java objects and back again. It is now a standard in the Java environment for JSON processing and is frequently used in Java-based applications to interact with JSON data.

5. **Discord4J:** To communicate with the Discord API, use the Java package Discord4J. It enables programmers to create bots and use Java to interface with Discord servers. All Discord API functions, including as channel creation, message sending, event processing, and user interaction, are fully supported by Discord4J. It is extensively utilized in the creation of tools and bots for managing and automating Discord communities.

**3.1 Introduction to the Analysis Tool**

The main analytical tool for this study is the CKJM (Class-level Java Metrics) tool. This free and open-source tool is made to analyze several object-oriented metrics that are important for determining how maintainable Java programs are. The knowledge derived from these metrics is essential for comprehending the architecture and structure of a software system, as these factors have a direct bearing on future development endeavors and maintainability.

**3.2 Tool for Metric Computation**

To calculate an extensive set of metrics based on the Chidamber and Kemerer (C&K) suite, CKJM processes Java bytecode. It computes the following six important object-oriented metrics: Depth of Inheritance Tree (DIT), Number of Children (NOC), Response for a Class (RFC), Coupling Between Objects (CBO), Weighted Methods per Class (WMC), and Lack of Cohesion in Methods (LCOM). We have chosen to concentrate on the WMC and CBO metrics for this study because of their close relationship to software maintainability.

**3.3 Application and Extraction of Data**  
After installing and running CKJM in accordance with protocol, it was able to examine the source code of five chosen Java projects. The tool's output gave us a thorough analysis of the overall complexity and class coupling of each project, which improved our comprehension of its architecture. After that, this data was prepared for additional examination.  
  
**3.4 Important Evaluation Metrics**Two measures from the CKJM collection were found to be very pertinent to this investigation:

* **Weighted Methods per Class (WMC)**: Based on the C&K suite, WMC totals the complexities of all the methods in a class to determine the total complexity of that class. The number of parameters, branching, and interactions with external data types are some examples of elements that can affect how complex a method is. More complicated class structures are generally indicated by higher WMC values, which may translate into more work being required for testing, comprehending, and maintaining the class. WMC is therefore a key predictor of the possible work needed for software maintenance in the future.
* **Coupling Between Objects (CBO):** This crucial statistic, which is part of the C&K suite, measures how dependent a system's classes are on one another. This measure keeps track of the number of other classes that a given class directly interacts with. A class with a high CBO value is thought to have many dependents, which is indicative of a tightly coupled architecture. Because modifications to one class may cause a chain reaction of changes in its dependent classes, this kind of coupling can make maintenance difficult. In addition to reducing system flexibility, high coupling raises the possibility of flaws being introduced during modifications. Therefore, CBO is essential for assessing a software design's overall resilience and maintainability.

**4. Results**

With a particular emphasis on how class size influences program maintainability using measures like Weighted Methods per Class (WMC) and Coupling Between Objects (CBO), the results section seeks to shed light on the study topics that were posed. To address the research questions, it is important to investigate the subsequent crucial domains:

**4.1 Understanding Complexity and Coupling in Class Design: The Relationship Between Software Maintainability Metrics (WMC and CBO) and Class Size**  
Examining the relationship between class size and Weighted Methods per Class (WMC) and Coupling Between Objects (CBO) provides important insights into software maintainability. It has been shown that WMC and CBO indicators typically rise at the same time that class sizes increase. According to this connection, more methods and dependencies are typically encapsulated in larger classes, which naturally results in higher complexity and coupling.

**4.2 Data Summary:**

The following metrics, which show how class size, WMC, and CBO interact, were computed for five Java projects to show this correlation:

We investigated how class size—measured in Lines of Code (LOC)—affects program maintainability using two important metrics: Coupling Between Object Classes (CBO) and Weighted Methods per Class (WMC). The results reveal a consistent trend where an increase in class size correlates with higher complexity and greater inter-class dependencies, as represented by rising WMC and CBO values across the studied Java projects.

|  |  |  |  |
| --- | --- | --- | --- |
| **Project Name** | **Avg CBO** | **Avg WMC** | **Avg LOC** |
| huanghongxun/HMCL | 6.608120869 | 11.17658168 | 58.06515581 |
| OpenHFT/Chronicle-Queue | 5.586916 | 10.52897 | 54.37944 |
| real-logic/simple-binary-encoding | 6.81982 | 24.81982 | 130.8874 |
| FasterXML/Jackson-core | 5.589852 | 25.77167 | 115.5497 |
| Discord4J/Discord4J | 7.336475 | 8.39491 | 36.37889 |

**4.3 Comparison of Avg.CBO and Avg.LOC of five projects:** Class Size vs. CBO: In the same way, the CBO metric showed that higher class sizes have higher dependency levels, which raise inter-class coupling. Projects such as Discord4J demonstrated a modest class size (Avg LOC: 36.37) and a high CBO (Avg CBO: 7.33), demonstrating that even relatively sized classes can create considerable coupling.

**4.4 Comparison of Avg.WMC and Avg.LOC of five projects:** The analysis revealed that the WMC rises with class size. More methods are typically encapsulated in larger classes, which increases computational complexity. To illustrate the intrinsic difficulty of larger codebases, real-logic/simple-binary-encoding had the biggest average class size (Avg LOC: 130.88) and almost the highest WMC (Avg WMC: 24.81).

**4.5 Comparison of Avg.WMC and Avg.LOC and Avg.LOC of five projects:** Projects with larger classes, like real-logic/simple-binary-encoding and FasterXML/Jackson-core, consistently displayed higher levels of complexity and interdependencies when the combined effects of LOC, WMC, and CBO were examined. These findings imply that internal complexity (WMC) and external coupling (CBO) both rise with class sizes, potentially posing problems for long-term maintainability.

**5.0 Conclusion**

**1. Influence of Class Size on Software Maintainability:** The results of the study clearly show that class size has a major impact on how maintainable software systems are. Higher levels of complexity (WMC) and increased coupling (CBO), two important measures of how readily the software may be expanded, altered, or debugged, are consistently linked to larger classes.

**2. Impact of Class Size on Complexity**: Greater functionality is typically encapsulated in larger classes, which results in higher WMC values. It is more difficult to understand, test, and maintain these classes because of their increased complexity. Examples of projects with the highest-class sizes and much higher WMC values are real-logic/simple-binary-encoding. This increase in complexity could lead to longer debugging times, more refactoring work, and a larger chance of error introduction during updates.

**3. Relationship Between Class Size and Coupling:** Larger classes are thought to be more dependent on smaller classes, which raises the degree of coupling, according to the link between CBO and class size. Because of this interconnectedness, system modifications in one class are likely to have an impact on other classes as well, making maintenance more difficult. The two best examples are Discord4J and DBeaver, which show significant coupling, a sign that the tightly coupled class hierarchies make changes more complicated and dangerous.

**4. Consequences for Software Maintenance**The findings support the notion that software maintainability is adversely impacted by greater class sizes. The possibility of higher maintenance costs, less system flexibility, and challenges with future modifications becomes increasingly evident when WMC and CBO values rise with class size. These projects run the potential of developing technical debt over time, which would make maintenance more difficult unless proactive refactoring and the application of modular design principles are implemented.

**5.1 Recommendations**

To counteract the detrimental impact of class size on maintainability, developers must take into account tactics like:

* **Reducing Class Size**: Divide big class sizes into smaller, easier-to-manage groups. Class size and complexity can be reduced with the use of modular design and the Single Responsibility Principle (SRP).
* **Managing Coupling**: Encourage loose coupling through dependency injection or design patterns like facade and adapter, and actively monitor and minimize coupling between classes.
* **Frequent Refactoring**: Restructuring big, complicated classes regularly can assist in preserving the codebase's maintainability, lower technical debt, and increase system resilience.

**5.2 Limitations**

* Because the study only included a small number of Java projects, it might not be representative of all software systems written in different languages or in different areas.
* Other significant metrics that potentially impact maintainability, such as cohesiveness or inheritance depth, were not examined; only WMC and CBO were.
* Runtime behavior and other dynamic elements, which can also affect program maintainability, were not taken into consideration in the analysis.
* Each project is only ever included in a single snapshot by the study, which leaves out the consequences of software evolution over time.

**5.3 Future Research Directions**

* To validate the results on a bigger scale and expand to additional projects from various languages and areas.
* For a more thorough understanding of maintainability, include metrics like cohesion, inheritance depth, and cyclomatic complexity.
* Examine projects over time to observe how maintainability and class size change as a result of refactoring and expansion.
* Evaluate the real-time effects of class size on maintainability and performance by combining runtime data with static analysis.
* Examine how complexity and maintainability are impacted by contemporary development techniques like continuous integration and agile development.

**REFERENCES**

* Chidamber, S. R., & Kemerer, C. F. (1994). *A metrics suite for object-oriented design. IEEE Transactions on Software Engineering*, 20(6), 476-493.
* Aniche, M. (n.d.). CK: Tool for calculating Chidamber and Kemerer Java metrics. Retrieved from <https://github.com/mauricioaniche/ck>
* huanghongxun. (n.d.). \*HMCL: Hello Minecraft Launcher\* [GitHub repository]. GitHub. <https://github.com/huanghongxun/HMCL>
* OpenHFT. (n.d.). \*Chronicle-Queue\* [GitHub repository]. GitHub. <https://github.com/OpenHFT/Chronicle-Queue>
* real-logic. (n.d.). \*Simple Binary Encoding (SBE)\* [GitHub repository]. GitHub. <https://github.com/real-logic/simple-binary-encoding>
* FasterXML. (n.d.). \*Jackson-core\* [GitHub repository]. GitHub. <https://github.com/FasterXML/jackson-core>
* Discord4J. (n.d.). \*Discord4J\* [GitHub repository]. GitHub. <https://github.com/Discord4J/Discord4J>