# Empirical Study on the Impact of Class Size on Software Maintainability in Java Projects

***Presented by***

**Siva Reddy Potlapati -** **L30086362**

**Pavan Kalyan Teluguti – L30088205**

**Spring 2024-II**

**Abstract**

This empirical study uses the Goal-Question-Metric (GQM) paradigm to frame the research strategy as it examines the effect of class size on software maintainability. Maintainability was evaluated using the Chidamber and Kemerer (C&K) metrics suite, namely the Weighted Methods per Class (WMC) and Coupling Between Objects (CBO). These metrics were taken from a carefully chosen dataset of Java projects that fulfilled certain requirements, such minimum project size, age, and developer engagement, using a static analysis tool published on GitHub. To ascertain trends and differences in maintainability across various class sizes, the study examined the values of its metrics. Preliminary results point to the possible need for solutions that lessen maintenance issues in bigger software systems, suggesting a link between greater class sizes and decreased maintainability. The goal of this study is to promote more maintainable software design while also adding to our understanding of best practices in software development.

*Keywords:* software maintainability, class size, Chidamber and Kemerer metrics, static analysis, Java

1. **Introduction:**

An essential aspect of software development is maintainability, which comprises how a software system can be inexpensively and easily updated, modified, or extended. It is important to understand the variables that affect maintenance as software systems become larger and more sophisticated. Previous research has revealed several characteristics that affect software quality, such as system complexity, integration, communication, and most importantly, class size, however, there are empirical findings on impact the specificity of class size in terms of management remains limited and unclear. This work aims to close this gap by applying methodology and empirical data to explore the relationship between class size and managed software.

The methodology for this research is provided by a Goal-Question-Metric (GQM) model that focuses on specific goals that can be used to guide research. The main objective of this study is to investigate how class size affects the maintainability of a software project. The Chidamber and Kemerer (C&K) metrics suite, which provides powerful tools to measure software quality including maintainability, is used to compile this analysis specifically focusing on weighted method per class (WMC) and coupling between objects (CBO), two important measurements of this suite. These metrics were chosen because of their record of relevance to two important maintenance-related dimensions: system complexity and cross-class coordination.

This study aims to provide quantitative evidence on the link between class size and maintainability by using a static analysis tool to extract these metrics from a variety of Java projects that match selection criteria—such as project size, age, and developer engagement. It is anticipated that the results will provide insightful information to project managers and software developers, guiding best practices in software design and development to improve maintainability, especially in bigger software systems.

* 1. **Research Objective**

The purpose of this research is to investigate the connection between software maintainability and class size. The goals are:

a) To gauge class sizes across a broad range of software applications, with an emphasis on Java-based projects. The overall number of lines of code as well as the structural complexity of each class will be considered in this measurement.

b) Utilizing the Chidamber and Kemerer (C&K) metrics suite to evaluate the maintainability of software. The Coupling Between Objects (CBO) and Weighted Methods per Class (WMC) metrics, which show the dependency and complexity of software components, will be the main emphasis of this study.

c) To examine the relationship between class size and software maintainability with the goal of identifying how software classes' logical and physical sizes affect their maintainability. This study will assist in identifying patterns and possible cutoff points where maintainability is negatively impacted by class size.

By addressing these goals, the research hopes to offer practical advice that will help software architects and developers create better maintainable software systems, especially ones that grow and are complex.

* 1. **Research Questions**

This study aims to address the following research questions, which are guided by the research objectives:   
  
a) What is the relationship between class size (measured in Lines of Code) and the complexity and coupling metrics (WMC and CBO) of Java classes?

The objective of this question is to investigate the statistical relationships between the class's physical size and its structural and interaction complexities, as measured by WMC and CBO. It looks for patterns or trends that point to the relationship between a class's internal complexity and interdependencies with other classes and the sheer amount of code that makes up that class.  
  
  
b) What effects do differences in WMC and CBO among various class sizes have on Java software projects' overall maintainability?

This question looks at how variations in WMC and CBO values among small, medium, and big classes lead to maintenance issues to assess the effect of class size on program maintainability. It seeks to ascertain if larger classes—which may have higher WMC and CBO scores—show a measurable decline in maintainability, offering factual data to confirm or contradict widely held beliefs in software engineering methods.

* 1. **Research Metrics:**

The research study will make use of a set of improved metrics in order to successfully address the research questions:

**a)** **Lines of Code (LoC)** for each class will be used to objectively quantify the structural complexity and class size. The main metric for class size will be LoC, but structural complexity will also consider things like the density of logical and control structures in the code. A detailed grasp of the relationship between maintainability and the structural and physical characteristics of class size is made possible by this dual measuring technique.

**b) Weighted Methods per Class (WMC):** Based on the number and complexity of each class's methods, WMC will be utilized to determine how difficult each class is. This measure is essential for comprehending a class's underlying intricacies and estimating the effort for testing and maintenance.

**c)Coupling Between Objects (CBO):** CBO counts the classes on which a certain class directly depends as well as the number of classes on which it depends. This statistic is crucial for assessing how modifications to one class could influence other classes and hence affect the software's maintainability.

1. **Overview of Data Set**  
   This section includes a description of the Java projects that were selected for this empirical study on software maintainability and class size, along with the selection criteria.
   1. **Project Selection Criteria**

To guarantee that the chosen Java projects provide insightful analyses of how class size affects software maintainability, we established the following criteria for choosing programs:

a) **A minimum size requirement**: We especially looked for programs with a minimum of 200,000 Lines of Code (LoC) to make sure the class structures were sufficiently complicated and varied for an insightful analysis. These programs were considered to have a large quantity of code. This cutoff represents the scale and intricacy common to reliable software systems that can exhibit a range of maintainability issues.

b) **Historical Depth:** Programs with an 8-year-old minimum age requirement have to be selected. This criterion guarantees that the projects have seen significant evolution and maintenance, offering a more profound understanding of the long-term consequences of design choices on software maintainability.

c) **Involvement of Developers:** We selected projects with a significant development team, involving at least 200 contributors. Due to varying coding styles and collaboration methods, bigger teams that usually work on more complicated systems may have dynamics that affect maintainability. For this reason, this criterion was designed to reflect those dynamics.

Following these requirements, we were able to choose five Java projects from GitHub that are typical of the class and should offer deep insights on the relationship between class size and software maintainability.

* 1. **Studied Projects:**

Based on the aforementioned standards, the projects listed below were chosen to offer a variety of systems and applications for analysis:

**Table 1: *Summary of Studied Projects***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Program Name** | **Developers** | **Program Age** | **Size (LOC)** | **Description** |
| **Elasticsearch** | 350 | 12 | 990,540 | A highly scalable open-source full-text search and analytics engine. It allows you to store, search, and analyse big volumes of data quickly and in near real-time. |
| **DBeaver** | 204 | 7 | 273,545 | A free and open-source universal database tool for developers and database administrators. It supports all popular databases, providing a powerful graphical interface for databases management, development, and maintenance. |
| **Guava** | 273 | 8 | 388,778 | Google's core libraries for Java-based projects, which include utilities for collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. |
| **Apache Kafka** | 340 | 11 | 142,143 | A distributed streaming platform capable of handling trillions of events a day. It provides high-throughput, fault-tolerant services for collecting and processing real-time data. |
| **Apache Druid** | 370 | 10 | 245,187 | A high-performance real-time analytics database designed for business intelligence (OLAP) queries on event data. |

* 1. **Comprehensive Project Details** Here is a thorough summary of the chosen projects that will help us gain a deeper grasp of each while showcasing its special qualities and applicability to this research:

1. **Elasticsearch**: Based on the Apache Lucene framework, Elasticsearch is a distributed, RESTful, free, open-source search engine. It is renowned for its scalability, speed, distributed architecture, and straightforward REST APIs. It is extensively utilized for search functions, real-time application monitoring, and log and event data processing. This project is a great choice for researching how class size affects maintainability in intricate, high-traffic systems because of its large codebase and active developer community.
2. **DBeaver:** A complete and all-purpose database tool for developers, analysts, and database administrators is called DBeaver. All widely used databases, including Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto, MySQL, PostgreSQL, MariaDB, SQLite, Oracle, DB2, SQL Server, and more, are supported. Examining how tools supporting different databases manage maintainability issues is made easy with DBeaver's broad feature set and support for a variety of databases. These include SQL scripting, data export and import, ER diagrams, and more.
3. **Guava:** A collection of Google's core libraries, Guava offers utilities for concurrency, I/O, hashing, caching, primitives, strings, and more. It also contains novel collection types, such multimap and multiset, as well as immutable collections and a graph library. Because of its widespread usage in Java applications, this project is essential for assessing the maintainability of utility libraries, which form the basis of many other software projects.
4. **Apache Kafka:** Java and Scala are used in the development of Apache Kafka, an open-source stream processing technology contributed to the Apache Software Foundation by LinkedIn. The project's goal is to offer a single, low-latency, high-throughput platform for managing real-time data flows. Because of the great durability and fault tolerance guarantees provided by its design, complex mechanisms that have consequences for maintainability must be examined.
5. **Apache Druid:** Apache Druid is an open-source analytics data store designed for business intelligence (OLAP) queries on large datasets. Druid is commonly used in business intelligence applications to power user-facing analytic applications. Its real-time data ingestion, arbitrary data exploration, and fast data aggregation capabilities highlight significant challenges in maintaining such a dynamic system.
6. **Framework for Measurement**   
     
   With an emphasis on the Chidamber and Kemerer Java Metrics (CKJM) tool, this section describes in detail the metrics and tools utilized to collect and evaluate the data for this study.  
   1. **Overview of the Research Tool**   
        
      The main tool used in this research is the CKJM tool. This open-source program is made to calculate several class-level metrics that are essential for determining how maintainable Java applications are. These measurements are essential for comprehending the software's architecture and design, which have a direct impact on maintenance chores.
   2. **Metrics Calculation Tool**  
      In order to provide an extensive collection of metrics from the Chidamber and Kemerer suite, CKJM evaluates Java bytecode. Six metrics are included in this: Response for a Class (RFC), Number of Children (NOC), Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM). We have chosen to concentrate on WMC and CBO for the purposes of this study because of their important implications for software maintainability.
   3. **Utilization and Extracting Data**   
        
      The utility was downloaded and deployed in accordance with regular processes to deploy CKJM. After that, it was applied to the source code of the chosen five Java projects, yielding results that measured the projects' overall complexity and class coupling. These were formatted outputs.
   4. **Key Metrics for Analysis**

In the context of this investigation, two measures stand out as particularly important:

* **Weighted Methods per Class (WMC):** By summing the complexities of its methods, the WMC metric—which is taken from the Chidamber and Kemerer (C&K) set of software metrics—quantifies the complexity inside a single class. Every method adds to the class's overall complexity score, which considers variables like logical branching, parameter counts, and interactions with other data types. A more complicated class structure is usually indicated by a higher WMC rating. Because knowledge and adjustments of complex classes demand more work and a deeper grasp of many relationships inside the class, they are sometimes more difficult to test and maintain. WMC is hence a crucial criterion for assessing the possible effort and maintainability needed for efficiently managing software classes.
* **Coupling Between Objects (CBO):** A fundamental statistic from the C&K metric package, CBO assesses how dependent classes are on one another in a software system. It keeps track of how many other classes a particular class directly interacts with, either by exchanging data with them or utilizing their data. Strong dependencies on several other classes are suggested by high CBO values, which point to a closely connected system design. Because modifications in one class may need cascade changes across dependent classes, such coupling can make maintenance duties more difficult. Moreover, a system's adaptability may be hampered by excessive coupling, and the likelihood of errors during change may rise. As a result, CBO is an essential metric for determining how amenable a system is, as well as reflecting the interconnection and possible brittleness of a software design.

1. **Results**  
     
   This section summarizes our research on how class size affects the software maintainability of the chosen projects. We analysed the metrics of Coupling Between Objects (CBO) and Weighted Methods per Class (WMC) in relation to class size. The complexity and connection properties of the classes inside these projects are revealed by the analysis, and these factors point to maintainability issues.
   1. **Analysis using Weighted Methods per Class (WMC):** This measure evaluates how complicated the software projects' classes are. The number of methods and the computational logic used in each method serve as indicators of complexity. The average, minimum, and maximum WMC for each project are summarized in the table.

**Table 2:** ***Table of WMC values for 5 projects***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Name of the Project** | **Avg WMC** | **Min WMC** | **Max WMC** |
| 1 | DBeaver | 12.5817 | 0 | 921 |
| 2 | Druid | 9.17096 | 0 | 1022 |
| 3 | Elastic-search | 6.865095 | 0 | 289 |
| 4 | Guava | 6.865095 | 0 | 289 |
| 5 | Apache Kafka | 12.31603 | 0 | 450 |

* 1. **Visual Analysis of Weighted Methods per Class (WMC):**

1. **Bar Chart Representation of Average WMC**: The bar chart shows the variation in WMC values by giving a clear visual comparison of the average class complexity within each project. Taller bars represent projects with higher average WMCs, such as DBeaver and Kafka, suggesting a higher level of complexity that may affect their maintainability.

**Figure1: *Bar chart representing Avg. WMC of five projects.***

A graph with blue bars

Description automatically generated with medium confidence

1. **Trend Analysis of Line Charts for WMC Distribution**: For every project selected, the line graph shows the distribution of WMC values from the lowest (Min WMC) to the largest (Max WMC). It provides a glimpse of the range of complexity seen in the projects.

**Figure 2: *Line chart for Avg.WMC for five projects.***

A graph with lines and text

Description automatically generated

* 1. **Analysis using Weighted Methods per Class (WMC):** For every codebase, the average Coupling Between Objects (CBO) statistic across the examined projects sheds light on the normal inter-class relationships. The degree to which classes within the projects are dependent upon one another is reflected in these average CBO values, and this can have important ramifications for the software systems' flexibility and maintainability. The average CBO details of five projects are summarized in the below table.

**Table 3: *CBO details of five projects***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Name of the Project** | **Avg CBO** | **Min CBO** | **Max CBO** |
| 1 | DBeaver | 7.41628 | 0 | 202 |
| 2 | Druid | 7.64354 | 0 | 1022 |
| 3 | Elastic-search | 3.879981 | 0 | 81 |
| 4 | Guava | 3.879981 | 0 | 81 |
| 5 | Apache Kafka | 7.352838 | 0 | 403 |

**4.4 Visual Analysis of Coupling Between Objects (CBO)**

**a)** **Bar Chart Representation of Average CBO:** The bar chart shows the average Coupling Between Objects (CBO) measure for every project and shows how dependent each class is on the others. Longer bars indicate projects with a higher average CBO, such as Druid, which may have a complex network of class connections that might make maintenance more difficult.

**Figure 3: *Bar chart representation of Avg. CBO of 5 projects***

A graph with blue bars

Description automatically generated with medium confidence

**b) Line Graph Trend Analysis for Avg. CBO:** This provides a dynamic perspective of how class dependencies vary from project to project by plotting the average CBO values across the projects. The graph's peaks, especially for Druid, show where the class coupling is at its highest point and indicate possible locations where the complex coupling may make maintenance more difficult.

**Figure 4: *Line chart representation of Avg. CBO across 5 projects***

A graph with a line

Description automatically generated

* 1. **Analysis of Five Projects Based on Class Size:** Class sizes vary significantly among my five selected projects, as shown in the table.

**Table 4:** ***LOC details of five projects***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Name of the Project** | **Avg LOC** | **Min LOC** | **Max LOC** |
| 1 | DBeaver | 55.0131 | 1 | 4275 |
| 2 | Druid | 51.1236 | 1 | 3215 |
| 3 | Elastic-search | 40.2935 | 1 | 2828 |
| 4 | Guava | 40.2935 | 1 | 2828 |
| 5 | Apache Kafka | 73.0759 | 1 | 3761 |

* 1. **Visual Analysis of Lines of Code (LOC)**

1. **Bar Chart Representation of Average LOC**: A bar chart based on this data would show the average Lines of Code for each project, providing a comparative picture of codebase size. For example, a bar for Kafka, which has the highest average LOC of 73.0759, would rise above the others, showing a substantially bigger average size per class.

**Figure 5: *Bar chart representation of Avg. LOC for the 5 projects***

A graph with blue bars

Description automatically generated

1. **Line Graph Trend Analysis for LOC**: A line graph may show the trend or dispersion of the average LOC across several projects. We can see that Kafka has the largest average size, peaking at 73.0759, which may indicate the broad breadth and usefulness of its classes. The other projects show a lower average LOC, with Elasticsearch and Guava sharing the lowest average LOC value.

**Figure 6: *Line graph representation of Avg. LOC for the 5 projects***

A graph with a line

Description automatically generated

1. **Correlation Analysis of Class Size and Software Maintainability**

The statistics from five Java projects show a consistent pattern: as class size rises, so does its complexity and connection with other classes. This tendency, as seen by the link with LOC, WMC, and CBO measures, implies that bigger classes are more difficult to maintain due to their complex structure and dependencies. Smaller, well-defined classes are thus desirable for improved program maintainability.

1. **Correlation between Class Size and WMC:** The line graph of normalized LOC and WMC metrics across five projects shows a positive correlation: bigger classes have more complexity. Specifically, 'Apache Kafka' has higher values in both measures, indicating more complicated structures that may necessitate more maintenance demands. In contrast, 'Elasticsearch' and 'Guava' exhibit both reduced LOC and WMC, signifying simpler and presumably more maintainable classes.

The present investigation answers the research topic of the link between class size and complexity, confirming that larger classes are more complicated, which may affect program maintainability.

**Figure 7: *Comparison of Average LOC and WMC per Project***

A graph with a line and a green line

Description automatically generated with medium confidence

1. **Correlation between Class Size and CBO:** The line graph depicting normalized Lines of Code (LOC) and Coupling Between Objects (CBO) from five projects demonstrates a common trend: bigger classes frequently exhibit greater degrees of coupling. This trend shows that as classes grow, they have more dependencies, becoming increasingly intertwined with other elements of the software system. For example, 'Apache Kafka' has high scores for both LOC and CBO, indicating a complicated dependency network.

This conclusion is relevant to the study topic of the link between class size and coupling. The observed pattern supports the assumption that higher class sizes are often more coupled, which may increase maintenance problems.

**Figure 8:** ***Comparison of Average LOC and CBO per Project***

A graph with a green line

Description automatically generated

c)**Correlation between CBO, WMC and LOC:** The following line graph compares the normalized average Lines of Code (LOC), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO) for each Java software project. The graph depicts the differences and probable correlations between the three indicators across many projects. We can see that projects with higher LOC typically have higher WMC and CBO, indicating that larger classes are more complicated and coupled.

This graphic and its underlying data can help us answer our research question by demonstrating how changes in class size (as represented by LOC), complexity (as indicated by WMC), and coupling (as indicated by CBO) might impact maintainability. Typically, increased complexity and coupling have a detrimental influence on maintainability. The graph shows that as class size rises, so does complexity and coupling, potentially leading to additional maintenance issues in Java software projects.

**Figure 9: *Comparison of LOC, CBO,WMC for 5 projects***

A graph with green lines

Description automatically generated

1. **Conclusions:**

The study's findings support the theory that greater class sizes correlate with worse maintainability in Java software projects. This is supported by a positive relationship between class size (LOC), complexity (WMC), and coupling (CBO). According to the findings, classes with more lines of code are more complicated and have higher degrees of coupling, both of which could increase maintenance issues.

* 1. **Key Findings**
* A consistent positive association was discovered between LOC, WMC, and CBO in five Java applications.
* Larger classes with greater LOC have more complicated structures, as shown by WMC.
* As class size rises, so does coupling (CBO), suggesting increasingly sophisticated interdependencies within the software system.
* Apache Kafka has high ratings for LOC and CBO, indicating a complex and perhaps less maintainable.

**6.2 Implications**

The study emphasizes the importance of maintainable software design and suggests that bigger class sizes may be an impediment to this goal. It emphasizes the need of taking class size into account during the design and development stages to avoid future maintenance issues. To improve maintainability, project maintainers can consider tactics such as microservices, modular architecture, and breaking down huge classes into smaller portions.

**6.3 Recommendations**

According to the study's conclusions, developers and project managers should

* Embrace approaches that limit class size, such as modular design and the use of design patterns that promote minimal coupling and high cohesion.
* Conduct frequent code reviews and reworking sessions to manage and minimize complexity and coupling as projects progress.
* Use automated static analysis tools on a regular basis to track maintainability trends and make educated design and refactoring decisions.

**6.4 Limitations and Future Research**

The study's results are limited by the breadth of the initiatives examined and the measures used. Future study might broaden these findings by examining a wider range of projects of varying sizes, topics, and development approaches. Further study should look into the effects of other aspects on maintainability, such as testing techniques, documentation quality, and developer experience.

In conclusion, the study adds significant insights to our understanding of software maintainability in relation to class size. It emphasizes the complex nature and interconnection of maintainability features while also providing actual information to help guide best practices in Java program development. The objective is that this study will encourage the development of better maintainable software systems, benefiting both developers and clients.

**References**

* Chidamber, S. R., & Kemerer, C. F. (1994). *A metrics suite for object-oriented design. IEEE Transactions on Software Engineering*, 20(6), 476-493.
* Elasticsearch. Retrieved from <https://github.com/elastic/elasticsearch>
* Apache Software Foundation. (n.d.). Apache Kafka. Retrieved from <https://github.com/apache/kafka>
* Apache Software Foundation. (n.d.). Apache Druid. Retrieved from <https://github.com/apache/druid>
* Google. (n.d.). Guava: Google Core Libraries for Java. Retrieved from <https://github.com/google/guava>
* DBeaver. (n.d.). DBeaver. Retrieved from <https://github.com/dbeaver/dbeaver>
* Aniche, M. (n.d.). CK: Tool for calculating Chidamber and Kemerer Java metrics. Retrieved from <https://github.com/mauricioaniche/ck>