Beyond Size: Assessing the Role of Class Dimensions in Software Maintainability

Andrew James Milligan
Department of Computer and Mathematical Sciences
Lewis University
Romeoville, United States

***Abstract*—The primary aim of this study is to investigate the effect of class size on software maintainability. This goal stems from the recognition that maintainability is a crucial attribute of software quality, and the size of classes within a software codebase may have a significant impact on maintainability. The report seeks to explore this relationship to provide data-driven insights that may inform better software design and maintenance strategies.**

To guide this study, two key questions were identified based on the goal: Does the class size correlate with the complexity of the class? And does the class size affect the cohesion of methods in the class? To answer these questions, two metrics will be used from the Chidamber and Kemerer suite: Weighted Methods per Class (WMC) and Lack of Cohesion in Methods (LCOM).

The first metric, Weighted Methods per Class (WMC), measures the total complexity of a class by adding up the complexities of its methods. This metric is particularly relevant for the study because it provides a direct measure of class complexity. As the size of a class (in terms of the number of methods it contains) increases, so does the potential complexity of the class. If larger classes tend to have higher WMC scores, it suggests that they are more complex and potentially more difficult to maintain. WMC was chosen as it directly correlates with class size and provides a quantitative measure of complexity. WMC can investigate whether there is a relationship between class size and complexity, thereby addressing the first question.

The Lack of Cohesion in Methods (LCOM) measures the degree to which methods within a class are related to each other. Classes that perform many unrelated tasks tend to have a higher LCOM score, indicating low cohesion. Such classes might be more difficult to understand, change, and maintain. If larger classes tend to have higher LCOM scores, it suggests that they may be trying to accomplish too many tasks and could benefit from being split into smaller, more cohesive classes. LCOM was an ideal choice for the second metric as it's an excellent indicator of the internal quality of a class and it allows us to assess whether larger classes are more likely to have low cohesion. This metric is directly relevant to the second question about the impact of class size on cohesion.

In this study, it is important to remember that correlation does not imply causation. While WMC and LCOM can provide valuable insights, they should not be used on their own to make direct conclusions about maintainability. They should be interpreted in the broader context of the project, taking into account other potential influencing factors. Having addressed this potential issue, the chosen metrics can allow us to create a meaningful analysis of the effects of class size on software maintainability, which meets the goal of the project.

Moving into the projects that were studied, below a detailed overview of the five Java-based projects selected for the study can be found, focusing on their general description, the project's primary purpose, as well as their attributes in terms of project age, codebase size, number of contributors, and the depth of their revision history. The chosen projects represent a wide range of design patterns and domains, and thus, help us generalize the study's results. Here are the projects:

JFreeChart is a comprehensive open-source library that allows developers to incorporate professional-grade charts in their Java applications. With over 2.6K commits and 6 contributors, this mature project, more than three years old, gives us a substantial codebase to study. Its complexity and the variety of design patterns make it an excellent choice for exploring the correlation between class size and software maintainability.

JavaPoet is a powerful Java API, offering the functionality to generate .java source files. This project is over six years old and shows an extensive collaborative effort with over 800 commits made by 59 contributors. The substantial number of contributors brings in a diversity of coding styles, which can impact class size and maintainability.

The Universal Image Loader library offers an adaptable and potent tool for image loading, caching, and displaying in Android applications. With more than eight years of history, over 1.2K commits, and 43 contributors, this project provides a rich and varied codebase for the analysis. Furthermore, its specificity to Android apps helps diversify the pool of projects.

greenDAO serves as a light, fast, Object-Relational Mapping (ORM) solution for Android, which maps objects to SQLite databases. With over nine years of development, over 800 commits, and 17 contributors, greenDAO offers insights into the maintainability of database-focused Android applications.

Spring Boot facilitates the creation of stand-alone, production-grade, Spring-based applications that are straightforward to run. With an impressive count of 26K commits from 552 contributors over about eight years, Spring Boot offers a significant codebase for the study. The project's widespread use and broad functionality provide a vast array of design patterns for examination.

The selection criteria for these projects were based on several factors: the projects' longevity, the size of their codebase, the number of contributors, and the diversity in design patterns. These factors help ensure a comprehensive and varied analysis. These projects were chosen because they each represent different domains, offer diverse design patterns, and provide substantial codebases, thereby maximizing the breadth of the study and the applicability of its findings.

The selection process specified that the projects have been active for several years. The rationale behind this is twofold. Firstly, older projects usually have undergone more development cycles, including additions, modifications, and bug fixes, which potentially make their codebases more complex. Secondly, mature projects have proven their durability, suggesting that they might embody good practices in terms of software maintainability. As a baseline, all projects chosen for this study have been active for at least three years.

Codebase size is crucial as it directly impacts the breadth of the analysis. Larger codebases offer more data points for studying the correlation between class size and software maintainability. In the selection, all projects were ensured to have a considerable number of lines of code (LOC) and commits, leading to a more robust analysis.

A higher number of contributors usually means more variety in coding styles and design decisions, which can influence class size and the approach to maintainability. The projects were selected with a varied number of contributors, from small teams (like 6 contributors in JFreeChart) to larger and more diverse teams (like 552 contributors in Spring Boot).

The projects were chosen to represent a variety of application types and design patterns. For example, JFreeChart is a graphical library, JavaPoet is a code generation tool, Universal Image Loader and greenDAO are Android-specific libraries, and Spring Boot is a framework for building enterprise-grade applications. This diversity helps to ensure that the study's findings are not biased towards a specific type of application or design pattern.

Ensuring the projects were open source was a critical criterion, as open-source projects allow unrestricted access to their entire codebase and revision history. This transparency was required to carry out a comprehensive analysis.

Last but certainly not least when it came to the criteria for selecting these projects was choosing projects that are currently maintained. Some of the projects may not have had an official release in a while, but every project has been actively committed to within a year. These were given preference to ensure the study's findings remain relevant to current software development practices. Active projects are likely to reflect up-to-date development practices and tools. Careful consideration of this criteria ensured the selected projects could provide more comprehensive insights into the effect of class size on software maintainability in various contexts. A table is provided below for quick reference of some of the criteria of the selected projects:

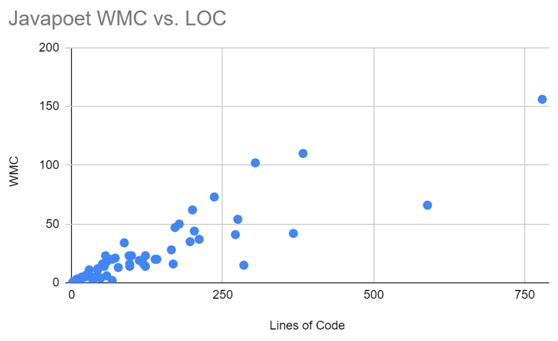
TABLE I. JAVA PROJECT SELECTION CRITERIA

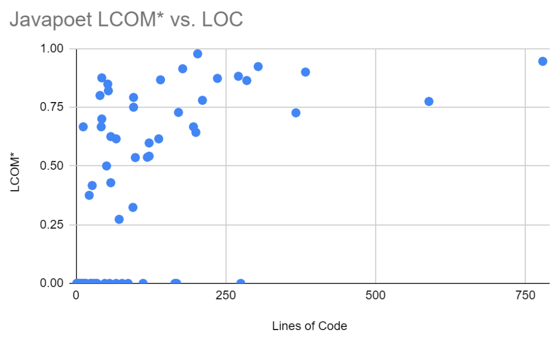
Project Name	Description	Age	Commits	Contributors	Unique Aspect
JFreeChart	A comprehensive open-source library for incorporating professional-grade charts in Java applications.	Over 3 years	2.6K+	6	Variety of design patterns, substantial codebase
JavaPoet	A powerful Java API, offering the functionality to generate .java source files.	Over 6 years	800+	59	Diverse coding styles due to a high number of contributors
Universal Image Loader	An adaptable and potent tool for image loading, caching, and displaying in Android applications.	Over 8 years	1.2K+	43	Rich and varied codebase specific to Android apps
greenDAO	A light, fast, Object-Relational Mapping (ORM) solution for Android, which maps objects to SQLite databases.	Over 9 years	800+	17	Provides insights into the maintainability of database-focused Android applications
Spring Boot	Facilitates the creation of stand-alone, production-grade, Spring-based applications.	About 8 years	26K+	552	Widespread use and broad functionality, vast array of design patterns for examination

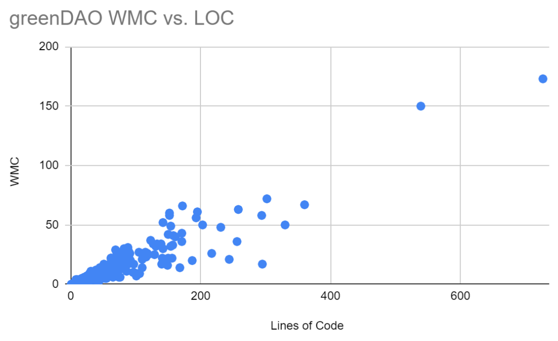
The tool that was used for data collection in this project is CK, a Java software metrics suite developed by Maurício Aniche. This tool calculates various class-level and method-level metrics for any given Java project, making it an excellent tool for this kind of research. CK is open-source, which adds to its transparency and trustworthiness. CK offers several advantages that make it suitable for the study. CK offers a wide array of metrics and measures a total of 13 metrics, including Weighted Methods per Class (WMC) and Lack of Cohesion in Methods (LCOM*), which are directly relevant to the study. CK is easy to set up and use. It is distributed on GitHub and can be cloned and compiled into a .jar file using Apache Maven that can be run from the command line. CK can analyze a Java project directly from its source code, or even a compiled .jar file. CK is also built for performance and can handle large codebases efficiently. This is particularly important given that some of the projects in the study, such as Spring Boot, have a large number of classes. The conducted research focuses on the class level, examining how class size affects maintainability. CK is designed with a similar focus, making it ideal for the needs of the project. The open-source nature of

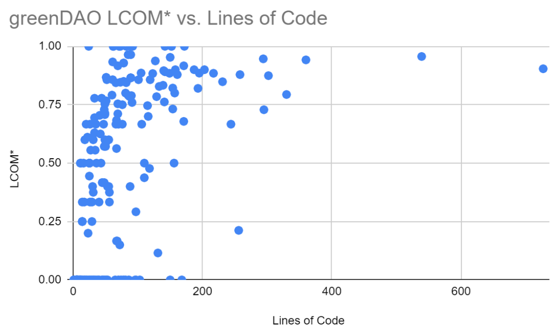
CK also ensures there are no hidden biases or processes in the metric collection, enhancing the credibility of the research. To summarize, CK is a capable tool that provides us with the precise metrics needed to study the correlation between class size and software maintainability. By utilizing CK, meaningful data could be extracted from the selected projects, allowing us to draw effective conclusions.

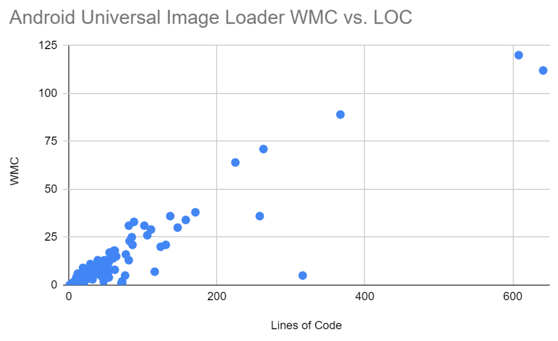
After using the CK program to collect data from each of the five projects, a way to display data in an easy to view and understand format was required. Scatter plots seemed to be an optimal choice for visualizing the data due to their capacity to demonstrate bivariate relationships, allowing clear depiction of how class size (in Lines Of Code) correlates with maintainability metrics (WMC and LCOM*). They efficiently highlight trends, patterns and outliers in the dataset, assisting in discerning whether larger or smaller classes have varying levels of maintainability. Importantly, scatter plots do not presuppose the nature of the relationship between variables, making them suitable for revealing both linear and non-linear patterns. Moreover, they provide an effective means for comparing different types of classes by using varied colors or symbols. This versatility and powerful visual representation capacity of scatter plots makes them an excellent tool for representing the data presented in this report. Following this explanation are the scatter charts generated from the output of the CK tool. Each chart is labeled according to the software and which metric is being measured against the Lines of Code (LOC).

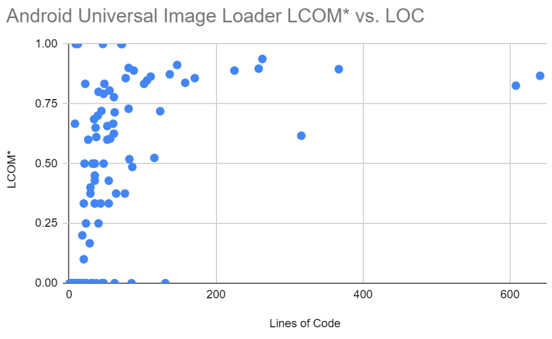


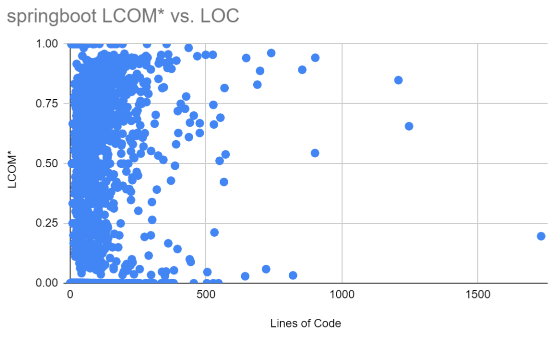


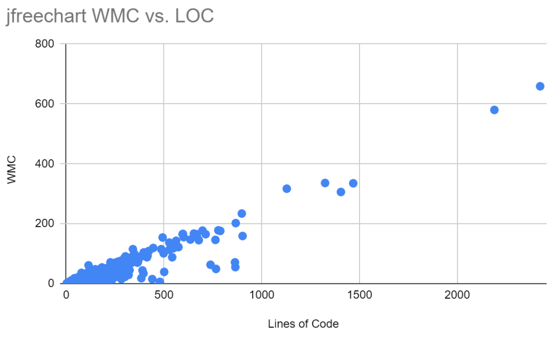


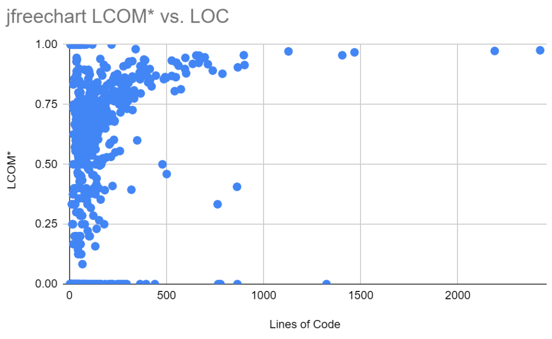


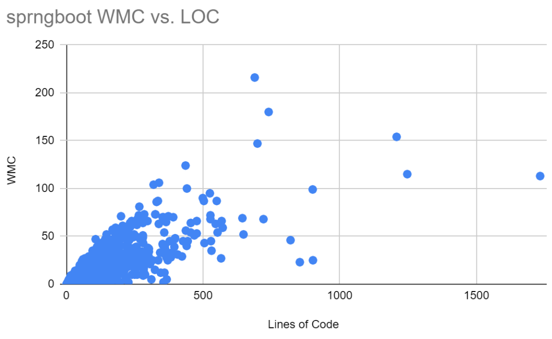












Based on the data from the five projects and in the context of the project goal to study the relationship between the size of a class (measured in lines of code) and software maintainability, some important conclusions can be made.

Firstly, a correlation between larger classes (more lines of code) and higher Weighted Methods per Class (WMC) values was observed. This implies that as the size of the class increases, the complexity of the class, as indicated by WMC, also increases. High complexity is generally regarded as a deterrent to maintainability as it makes the code harder to understand and modify.

In addition, a positive correlation between the size of the class and Lack of Cohesion in Methods (LCOM) was found. Generally, larger classes demonstrated a greater lack of cohesion, which means they tend to do more than one thing and hence violate the Single Responsibility Principle. This can lead to difficulties in maintaining and modifying the software since changes in one area might require changes in a loosely related area.

The data suggests that larger classes might negatively impact the maintainability of the software, due to increased complexity and lack of cohesion. In terms of maintainability, it would be beneficial to focus on creating smaller, more cohesive classes with lower complexity.

However, while these trends are significant, it's important to remember that size, in terms of lines of code, is just one aspect of maintainability. The use of best practices, good documentation, and appropriate testing also play crucial roles in ensuring maintainability. It's recommended to consider these aspects as well when measuring and improving the maintainability of software.

This comprehensive study has provided evidence highlighting the impact of class size, in terms of lines of code, on the maintainability of software across five distinct projects. A notable correlation between larger classes and increased complexity (measured via Weighted Methods per Class) as well as a lack of cohesion (indicated by Lack of Cohesion in Methods) has been identified. The findings suggest that larger classes could potentially hamper the ease of maintenance and modification of software, thereby emphasizing the value of designing smaller, more cohesive, and less complex classes. However, this research also underscores the importance of incorporating other crucial factors such as adherence to best coding practices, effective documentation, and rigorous testing, in the continual pursuit of enhancing software maintainability. Moving forward, it's clear that achieving high maintainability is a multifaceted task that requires managing class size in addition to other practices.

REFERENCES

[1] Maurício Aniche, “Java code metrics calculator (CK)”, 2015, Available in https://github.com/mauricioaniche/ck

[2] Andy Wilkinson, “Spring Boot”, 2023, Available in https://github.com/spring-projects/spring-boot

[3] Markus Junginger, “greenDAO”, 2020, Available in https://github.com/greenrobot/greenDAO

[4] Sergey Tarasevich, “Android Universal Image Loader”, 2015, Available in https://github.com/nostra13/Android-Universal-Image-Loader

[5] Jake Wharton, “javapoet”, 2020, Available in https://github.com/square/javapoet/tags

[6] David Gilbert, “jfreechart”, 2023, Available in https://github.com/jfree/jfreechart/releases/tag/v1.5.4