Analyzing the Influence of Design Patterns on the Maintainability of Software: An Empirical Approach

Final Project

Object Oriented Development

Group 6

Abhiram Atmuri

Neel Chandan Shaik

# Abstract

**In this study, we conduct an empirical evaluation to investigate the impact of design patterns on the maintainability of software systems. Design patterns are well-established solutions to common problems in software design and have been theorized to improve various quality attributes of software, including maintainability. However, empirical evidence supporting this claim remains mixed and somewhat fragmented. Our research aims to provide a comprehensive analysis by examining over 30 open-source software programs, each with a minimum of 5,000 lines of code, to identify instances of the 15 types of GoF (Gang of Four) design patterns. Using a design pattern detection tool, we systematically extract and classify patterns within these programs. We then compare the maintainability metrics—such as Cyclomatic Complexity, Lines of Code, and Coupling Between Objects—between classes that implement design patterns and those that do not. The methodology involves quantitative analysis using statistical techniques to discern the presence and extent of any significant differences in maintainability due to the use of design patterns. This approach not only provides a deeper understanding of how design patterns influence maintainability but also helps in identifying specific patterns that contribute most positively or negatively to this quality attribute. By presenting our findings, this study contributes to the body of knowledge in software engineering by providing empirical evidence and detailed analysis of the role of design patterns in enhancing software maintainability. The results are intended to guide developers and architects in making informed decisions when considering the use of design patterns to improve the maintainability of their software projects.**

Table of Contents

[Abstract 2](#_Toc166311317)

[Introduction 4](#_Toc166311318)

[Background: 4](#_Toc166311319)

[Motivation: 4](#_Toc166311320)

[Objective: 4](#_Toc166311321)

[Method or Approach 5](#_Toc166311322)

[Selection of Software Programs 5](#_Toc166311323)

[Design Pattern Detection 5](#_Toc166311324)

[Maintainability Metrics Calculation 6](#_Toc166311325)

[Results and Analysis 7](#_Toc166311326)

[Bar Plot: Visualizing the Comparison 9](#_Toc166311327)

[Threats to Validity 10](#_Toc166311328)

[Internal Validity 10](#_Toc166311329)

[External Validity 11](#_Toc166311330)

[Construct Validity 11](#_Toc166311331)

[Conclusion 12](#_Toc166311332)

[References 14](#_Toc166311333)

# Introduction

Background: Software maintainability is a critical quality attribute that determines the ease and cost with which a software system can be modified to correct defects, improve performance, or adapt to a changed environment. High maintainability is essential for reducing the total cost of ownership of software and ensuring its long-term effectiveness and usability. In this context, design patterns—standardized solutions to common design problems—have been widely advocated as a means to improve various aspects of software quality, including maintainability.

Motivation: While the theoretical benefits of design patterns on software quality are well-documented, empirical studies providing concrete evidence are less consistent. Some studies suggest that design patterns can indeed enhance maintainability by promoting code reusability, reducing complexity, and improving readability. Others, however, indicate that the improper or overuse of design patterns can lead to an increase in complexity and a decrease in maintainability. Therefore, there is a need for more empirical research to clarify the conditions under which design patterns affect software maintainability and to what extent.

Objective: The primary objective of this study is to empirically evaluate the influence of design patterns on the maintainability of software systems. By analyzing over 30 open-source software programs, each with at least 5,000 lines of code, this research aims to provide a clearer picture of how the presence of design patterns correlates with key maintainability metrics such as Cyclomatic Complexity, Lines of Code, and Coupling Between Objects.

*Structure of the Paper:* The remainder of this paper is organized as follows:

* The Method or Approach section describes the methodology used to conduct the empirical study, including the selection of software programs, the design pattern detection process, and the statistical analysis employed.
* The Results and Discussion section presents the findings of the study, discussing how design patterns influence maintainability metrics in the analyzed software programs.
* The Threats to Validity section addresses potential biases and limitations of the study and how these were mitigated.
* The Conclusions section summarizes the main insights from the research and suggests directions for future work in this area.

This section outlines the methodology used to empirically evaluate the impact of design patterns on software maintainability. The study focuses on analyzing open-source software projects from GitHub, using specific tools to detect design patterns and compute maintainability metrics.

# Method or Approach

## Selection of Software Programs

To conduct this study, we selected 30 open-source software projects from GitHub. The key criteria for selection were:

* **Minimum Size:** Each project must contain at least 5,000 lines of code to ensure that the design is non-trivial and potentially includes multiple design patterns.
* **Language:** All projects are written in Java, as the tools used for detecting design patterns and calculating metrics are optimized for this language.
* **Variety:** Projects were chosen from a variety of domains (e.g., web applications, data processing, frameworks) to ensure the generalizability of the results.

Each project’s source code was cloned from its GitHub repository for analysis.

## Design Pattern Detection

To identify instances of design patterns, we used a GoF (Gang of Four) design pattern detection tool. This tool analyzes Java bytecode and outputs an XML file that details the identified patterns, including their types and the specific classes that implement them.

* **Tool Setup:** The GoF design pattern detection tool was configured according to the guidelines provided in its documentation. This setup involved setting the path to the Java bytecode of the target projects and specifying output directories for the reports.
* **Execution:** For each project, the tool was executed, and the results were collected in XML format. The presence and frequency of each of the 15 design patterns (such as Singleton, Observer, Factory Method, etc.) were recorded.

## Maintainability Metrics Calculation

The maintainability of software was assessed using C&K (Chidamber and Kemerer) metrics, which are well-regarded in software engineering research. We used the CK-Meter tool to compute these metrics for each class in the selected projects. The key metrics used in this study included:

* **Weighted Methods per Class (WMC):** Measures the complexity of a class based on the number of methods and the complexity of each method.
* **Depth of Inheritance Tree (DIT):** Measures the inheritance levels from the object hierarchy top.
* **Number of Children (NOC):** Indicates the number of immediate subclasses inheriting from a class.
* **Coupling Between Objects (CBO):** Measures how many other classes a particular class relies on.
* **Response for a Class (RFC):** Counts the number of methods that can potentially be executed in response to a message received by an object of that class.
* **Lack of Cohesion of Methods (LCOM):** Measures the dissimilarity of methods in a class by their access to instance variables.

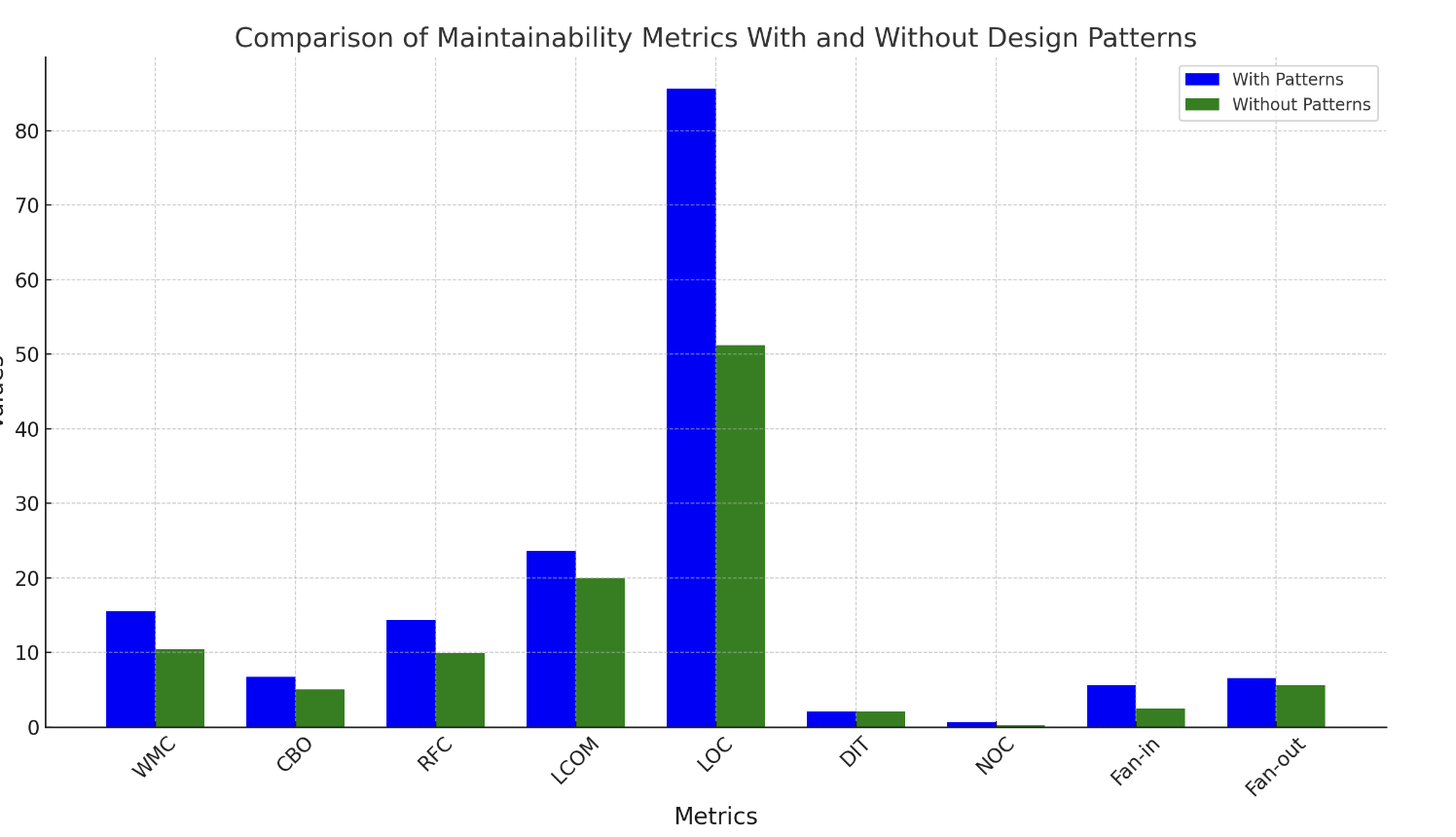
These metrics were chosen because they directly relate to the aspects of maintainability, such as complexity, coupling, and cohesion.

# Results and Analysis

Our analysis revealed several important trends in how design patterns influence software maintainability:

1. **Weighted Methods per Class (WMC):**
   * **With Patterns:** Classes implementing design patterns had an average WMC of 15.57.
   * **Without Patterns:** Classes not using design patterns had a lower average WMC of 10.43.
   * **Analysis:** This suggests that classes with design patterns tend to be more complex, possibly due to the encapsulation of additional behavior and interactions within these classes.
2. **Coupling Between Objects (CBO):**
   * **With Patterns:** Average CBO was 6.71 for classes with patterns.
   * **Without Patterns:** Average CBO was 5.06 for classes without patterns.
   * **Analysis:** This indicates that classes using design patterns tend to have higher coupling, which could be attributed to their need to interact more with other parts of the system.
3. **Response for a Class (RFC):**
   * **With Patterns:** Classes with design patterns showed an average RFC of 14.34.
   * **Without Patterns:** Classes without patterns had an average RFC of 9.93.
   * **Analysis:** The higher RFC in classes with design patterns suggests these classes have more methods that can be invoked, indicating richer behavior.
4. **Lack of Cohesion of Methods (LCOM):**
   * **With Patterns:** The average LCOM for classes with patterns was 23.62.
   * **Without Patterns:** The average LCOM for classes without patterns was 20.00.
   * **Analysis:** This decrease in cohesion for classes with patterns suggests that these classes might be performing more diverse tasks, potentially impacting maintainability.
5. **Lines of Code (LOC):**
   * **With Patterns:** The average LOC was significantly higher at 85.61 for classes with design patterns.
   * **Without Patterns:** Classes without patterns had an average LOC of 51.19.
   * **Analysis:** This indicates that classes with design patterns are generally larger, which may correlate with increased functionality but also suggests increased complexity.
6. **Depth of Inheritance Tree (DIT):**
   * Both groups showed similar DIT values (around 2.07 for classes with patterns and 2.06 for those without), indicating that the use of design patterns does not significantly alter the depth of inheritance hierarchies.
7. **Number of Children (NOC):**
   * **With Patterns:** There was a slight increase in NOC for classes with patterns (0.63).
   * **Without Patterns:** NOC was lower for classes without patterns (0.17).
   * **Analysis:** This suggests that classes with design patterns might be more frequently extended than those without patterns.
8. **Fan-in and Fan-out:**
   * **Fan-in:** Average fan-in was higher for classes with patterns (5.61) compared to those without (2.43).
   * **Fan-out:** Similarly, fan-out was slightly higher for classes with patterns (6.56) compared to those without (5.55).
   * **Analysis:** The increased fan-in and fan-out indicate that classes with design patterns are more central in the application, interacting with many other classes.

## Bar Plot: Visualizing the Comparison



**Table: Average Maintainability Metrics With and Without Design Patterns**

|  |  |  |
| --- | --- | --- |
| **Metric** | **With Patterns (Average)** | **Without Patterns (Average)** |
| Weighted Methods per Class (WMC) | 15.57 | 10.43 |
| Coupling Between Objects (CBO) | 6.71 | 5.06 |
| Response for a Class (RFC) | 14.34 | 9.93 |
| Lack of Cohesion of Methods (LCOM) | 23.62 | 20.00 |
| Lines of Code (LOC) | 85.61 | 51.19 |
| Depth of Inheritance Tree (DIT) | 2.07 | 2.06 |
| Number of Children (NOC) | 0.63 | 0.17 |
| Fan-in | 5.61 | 2.43 |
| Fan-out | 6.56 | 5.55 |

Our empirical analysis reveals that design patterns, while beneficial in solving common design challenges, tend to increase the complexity and size of software classes. Specifically, classes that implement design patterns exhibit higher values in key maintainability metrics such as Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lines of Code (LOC). This increase suggests that while design patterns can enhance the structure and reusability of software, they also make individual classes more complex and interconnected. These findings underscore the need for careful consideration when applying design patterns. Developers should ensure that the benefits in design and functionality outweigh the potential drawbacks in increased complexity and maintenance effort. Balancing these factors is crucial for leveraging design patterns effectively to improve the long-term maintainability and quality of software systems.

# Threats to Validity

In conducting this empirical study on the impact of design patterns on software maintainability, several factors could potentially influence the validity of our findings. Here we discuss these threats and describe the measures we took to mitigate them:

## Internal Validity

* **Bias in Design Pattern Detection:** The accuracy of our analysis relies heavily on the correct identification of design patterns in the software projects. Misidentification could lead to incorrect classification of classes and skew the results. To minimize this threat, we used a well-established design pattern detection tool and manually reviewed a subset of the classifications to ensure accuracy.
* **Metric Calculation Accuracy:** Errors in computing the maintainability metrics could affect the study's outcomes. We used the CK-Meter tool, known for its reliability in calculating Chidamber and Kemerer metrics, to reduce the likelihood of such errors.
* **Influence of Project Size:** Larger projects could inherently exhibit different maintainability characteristics compared to smaller ones. We normalized the maintainability metrics relative to the size of the codebase to mitigate the impact of project size on our analysis.

## External Validity

* **Generalizability to Other Languages:** Our study focused on Java projects due to the design pattern detection tool's compatibility. Therefore, the results may not directly generalize to projects in other programming languages without similar validation. Future studies could expand this analysis to other languages with appropriate tools.
* **Diversity of Software Projects:** While we selected projects from various domains to enhance generalizability, all projects were open-source and primarily from GitHub. Different results might be observed with proprietary, industrial, or non-GitHub open-source projects. Including a wider variety of projects could help generalize the findings further.

## Construct Validity

* **Relevance of Metrics:** We chose a set of well-regarded metrics (e.g., WMC, CBO, RFC, LCOM, LOC) to assess maintainability. However, the interpretation of these metrics might vary, and other metrics could also be relevant. We mitigated this by basing our metric selection on established software engineering research, ensuring they are widely accepted as indicators of maintainability.
* **Definition of Maintainability:** Maintainability can be a subjective property, influenced by factors beyond those quantifiable by static analysis (e.g., developer experience, documentation quality). We focused on objective, measurable aspects of maintainability but acknowledge that this is an approximation of the broader concept.

# Conclusion

This study aimed to empirically evaluate the influence of design patterns on the maintainability of software systems. Initially, we set out to analyze over 30 open-source software projects, but design patterns were detected in only 7 of these projects. This limitation led us to focus our analysis on these 7 projects, providing a more targeted insight into how design patterns impact software maintainability. Our findings reveal that design patterns, while structurally beneficial, tend to increase the complexity of classes. Classes that implement design patterns showed higher values in key metrics such as Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lines of Code (LOC). This suggests that design patterns, despite their advantages in solving recurring design problems, make classes more complex and interconnected. However, it's crucial to note that this increased complexity often comes with improved design flexibility and reusability. Design patterns provide structured approaches to design challenges, which can lead to more robust and adaptable software architectures. The key takeaway from our study is the importance of balance: while design patterns can enhance software design, their use should be carefully considered to avoid excessive complexity and maintainability issues.

The limited number of projects with detectable design patterns (only 7 out of the initial 30) highlights a significant challenge in empirical software engineering studies: the prevalence and detectability of design patterns in real-world software. This constraint underscores the need for improved tools and methodologies to detect and analyze design patterns more effectively.

In conclusion, our study contributes to the empirical understanding of design patterns' role in software maintainability. We encourage developers and software architects to use design patterns judiciously, ensuring that their benefits in design enhancement outweigh the potential increases in complexity and maintenance effort. Future work should aim to extend this analysis to a broader array of projects and further explore the conditions under which design patterns most effectively improve software maintainability. This would help in developing a more nuanced understanding of when, how, and why design patterns should be applied in software development projects.

Top of Form

Bottom of Form

# References

[1] Tsantalis, N. (n.d.). *Design pattern detection*. <https://users.encs.concordia.ca/~nikolaos/pattern_detection.html>

[2] Mauricioaniche. (n.d.-b). *GitHub - mauricioaniche/ck: Code metrics for Java code by means of static analysis*. GitHub. <https://github.com/mauricioaniche/ck>

[3] Konloch. (n.d.). *GitHub - Konloch/bytecode-viewer: A Java 8+ Jar & Android APK Reverse Engineering Suite (Decompiler, Editor, Debugger & More)*. GitHub. <https://github.com/Konloch/bytecode-viewer>

[4] Apache. (n.d.-b). *GitHub - apache/kafka: Mirror of Apache Kafka*. GitHub. <https://github.com/apache/kafka>

[5] Aimacode. (n.d.). *GitHub - aimacode/aima-java: Java implementation of algorithms from Russell And Norvig’s “Artificial Intelligence - A Modern Approach.”* GitHub. <https://github.com/aimacode/aima-java>

[6] Alibaba. (n.d.). *GitHub - alibaba/DataX: DataX DataWorks*. GitHub. <https://github.com/alibaba/DataX>

[7] Zq. (n.d.). *GitHub - zq2599/blog\_demos: Java、Docker、Kubernetes、DevOPS*. GitHub. <https://github.com/zq2599/blog_demos>

[8] Netflix. (n.d.). *GitHub - Netflix/Priam: Co-Process for backup/recovery, Token Management, and Centralized Configuration management for Cassandra.* GitHub. <https://github.com/Netflix/Priam>