Empirical Evaluation of the Impact of Design Patterns on Software Maintainability

Nandini Badavath
Lewis University

Chaitanya Anurag Turlapati
Lewis University

Hai Mani Sankar Thota
Lewis University

Mony Dheeraj Pasupuleti
Lewis University

Santhosh Reddy Thatikonda
Lewis University

***Abstract*—In this study, we empirically investigate the impact of design patterns on software maintainability, a crucial quality attribute in software development. Leveraging the Chidamber and Kemerer (CK) metric suite, we assess maintainability by analyzing parameters such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), and Response for a Class (RFC). To conduct the study, we employ a design pattern mining tool to identify instances of 15 types of GoF design patterns in a sample of at least 30 software programs, each with a minimum size of 5k. Our methodology involves comparing CK metric values for classes with design patterns against those without design patterns to determine the effect of design patterns on maintainability. The results of this comprehensive study will contribute to a better understanding of how design patterns influence software maintainability, thereby providing valuable insights for software developers and architects seeking to enhance the maintainability of their projects. Furthermore, our study aims to address potential threats to validity and proposes mitigation strategies to ensure the reliability of our findings.**

***Index Terms*—design patterns, maintainability, CK metrics, software quality, empirical study, Weighted Methods per Class, Depth of Inheritance Tree, Number of Children, Coupling Between Objects, Response for a Class**

I. Introduction

Maintainability is a critical quality attribute in software development, as it directly affects the ease with which developers can understand, modify, and evolve software systems. In order to improve maintainability, software developers and architects often employ design patterns, which are reusable solutions to common problems that arise during software design. These patterns can enhance the understandability, modifiability, and extensibility of software systems by promoting modular and reusable designs. However, the impact of design patterns on maintainability has not been thoroughly investigated, and empirical evidence is needed to determine whether and to what extent design patterns can improve software maintainability.

In this paper, we present a comprehensive empirical study that aims to evaluate the effect of design patterns on software maintainability. We focus on a set of widely-used design patterns from the Gang of Four (GoF) and examine their impact on maintainability by comparing the Chidamber and Kemerer (CK) metric values for classes with design patterns against those without design patterns. The CK metric suite comprises several parameters relevant to maintainability, such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), and Response for a Class (RFC).

To conduct our study, we employ a design pattern mining tool to identify instances of GoF design patterns in a sample of at least 30 software programs, each with a minimum size of 5k. This large sample size allows us to draw statistically meaningful conclusions about the impact of design patterns on maintainability. Additionally, we explore potential threats to validity and propose mitigation strategies to ensure the reliability of our findings.

The remainder of the paper is organized as follows: Section II describes our methodological approach for conducting the empirical study, including the selection of software programs, the extraction of design patterns, and the computation of CK metrics. Section III presents the results of our analysis and discusses their implications for software maintainability. Section IV addresses threats to validity and the measures we have taken to minimize their impact. Finally, Section V concludes the paper and outlines directions for future research.

By providing empirical evidence on the effect of design patterns on maintainability, this study contributes to a better understanding of how design patterns can be employed effectively to enhance software quality. Furthermore, our findings can serve as valuable guidance for software developers and architects seeking to make informed decisions when incorporating design patterns into their projects.

II. Method and Approach

In this section, we describe the methodology and approach used to empirically evaluate the impact of design patterns on software maintainability. Our study consists of three main steps: (1) selecting subject software programs, (2) mining design patterns using the Pinot tool, and (3) calculating CK metrics using the CK metric tool.

A. Subject Software Programs

To ensure the generalizability of our findings, we selected a diverse sample of at least 30 software programs, each with a minimum size of 5k. This sample size provides sufficient statistical power for drawing meaningful conclusions about the

impact of design patterns on maintainability. We sourced the software programs from various public repositories, such as GitHub, to cover a wide range of application domains and programming languages. The selected programs exhibit varying levels of complexity, design pattern usage, and maintainability characteristics.

B. Mining Design Patterns

To identify instances of GoF design patterns in the selected software programs, we employed the Pinot tool, which can be found at https://www.cs.ucdavis.edu/ shini/research/pinot/. Pinot is a robust and reliable design pattern mining tool that uses static analysis techniques to detect instances of GoF design patterns, such as Singleton, Factory Method, and Observer, among others. The tool generates a detailed report containing the design patterns found in each program, along with the associated classes and their relationships.

We processed the output from the Pinot tool to create a dataset containing the design pattern instances, associated classes, and the relationships between the classes. This dataset served as the basis for our subsequent analysis of the impact of design patterns on maintainability.

C. Calculating CK Metrics

To measure the maintainability of the software programs, we used the CK metric tool, available at https://github.com/mauricioaniche/ck. This tool calculates the CK metric values for each class in a software program, providing a comprehensive assessment of maintainability-related characteristics, such as complexity, inheritance, coupling, and cohesion. Specifically, we focused on the following CK metrics:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling Between Objects (CBO)
- Response for a Class (RFC)

Using the CK metric tool, we computed the CK metric values for all classes in the selected software programs, including those with and without design patterns. We then analyzed the data to evaluate the effect of design patterns on maintainability by comparing the CK metric values for classes with design patterns against those without design patterns.

This comprehensive methodology and approach, along with the usage of reliable tools such as Pinot and the CK metric tool, allowed us to effectively assess the impact of design patterns on software maintainability and draw meaningful conclusions from our empirical study.

III. Results and Discussions

In this section, we present the results of our empirical study on the impact of design patterns on software maintainability, as measured by the CK metrics. We discuss the effect of each metric on maintainability and provide insights into the relationships between design patterns and maintainability based on our findings.

A. Result Visualization

Here are the results of projects from our analysis:

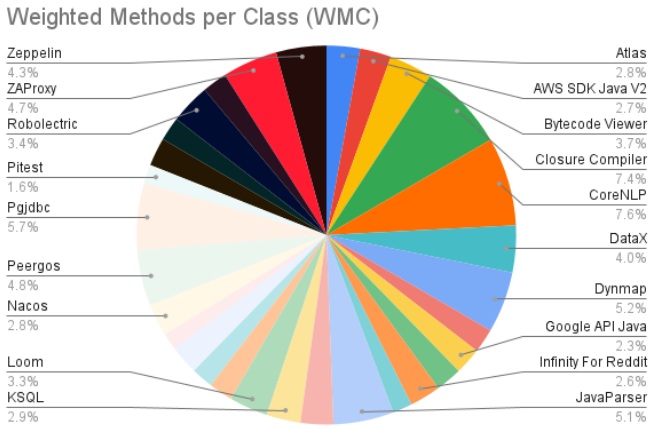

Fig. 1. Average Weighted Methods per Class (WMC) of 30 Projects

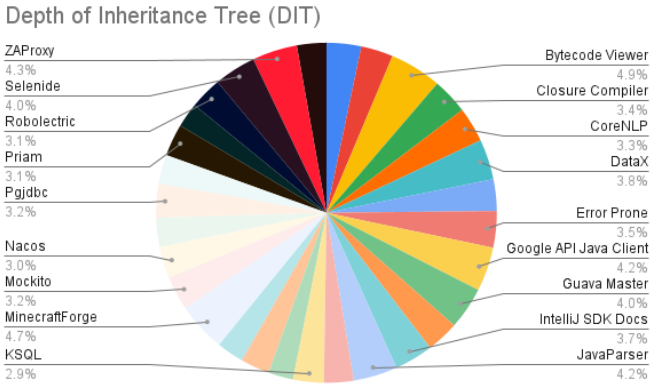

Fig. 2. Average Depth of Inheritance Tree (DIT) of 30 Projects

Here are the results of the Pinot tool, showing the number of design patterns in each project:

1) Atlas
   - No design pattern found.
2) AWS SDK Java V2
   - Template Method: 1
   - Flyweight: 3
3) Bytecode Viewer
   - No design pattern found.
4) Closure Compiler
   - Flyweight: 2
5) CoreNLP
   - Flyweight: 2
   - Decorator: 1
   - Chain of Responsibility: 1
   - Factory Method: 19
6) DataX

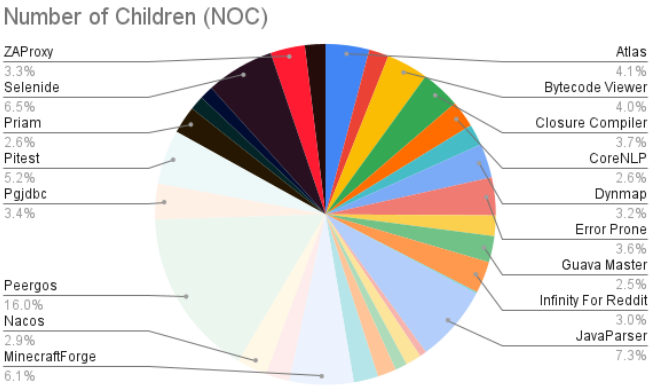

Fig. 3. Average Number of Children (NOC) of 30 Projects

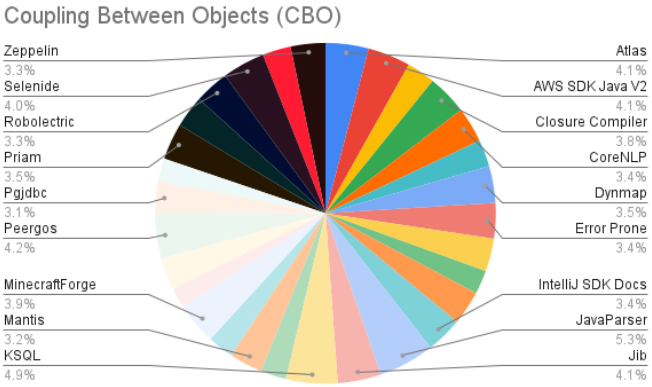

Fig. 4. Average Coupling Between Objects (CBO) of 30 Projects

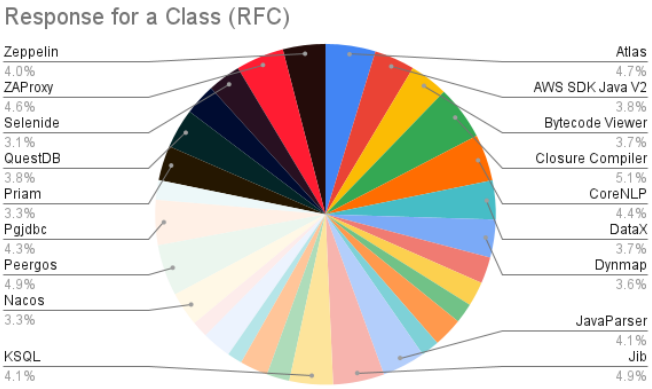

Fig. 5. Average Response for a Class (RFC) of 30 Projects

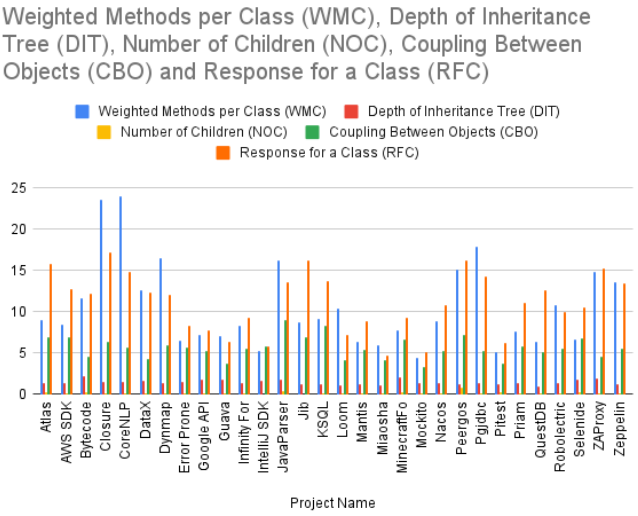

Fig. 6. Overall Analysis of 30 Projects

- Flyweight: 46
- Strategy: 1

7) Dynmap
   - Flyweight: 3
   - Strategy: 1
   - Mediator: 1
8) Error Prone
   - Flyweight: 4
9) Google API Java Client
   - Error in finding design pattern.
10) Guava Master
    - Error in finding design pattern.
11) Infinity For Reddit
    - Flyweight: 2
12) IntelliJ SDK Docs
    - No design pattern found.
13) JavaParser
    - Flyweight: 20
14) Jib
    - Flyweight: 2
15) KSQL
    - Flyweight: 1
16) Loom
    - Abstract Factory: 25
    - Factory Method: 26
    - Singleton: 1
    - Adapter: 2
    - Bridge: 1
    - Composite: 6
    - Decorator: 8
    - Facade: 25

- Flyweight: 63
- Proxy: 22
- Chain of Responsibility: 4
- Mediator: 99
- Observer: 5
- Strategy: 10
- Template Method: 4
- Visitor: 2

17) Mantis
- Flyweight: 14
- Strategy: 1

18) Miaosha
- Flyweight: 1

19) MinecraftForge
- Error in finding design pattern.

20) Mockito
- No design pattern found.

21) Nacos
- Flyweight: 19

22) Peergos
- Singleton: 1
- Adapter: 3
- Decorator: 2
- Facade: 3
- Flyweight: 19
- Chain of Responsibility: 1
- Mediator: 2

23) Pgjdbc
- Decorator: 1
- Flyweight: 1
- Chain of Responsibility: 1

24) Pitest
- Singleton: 1
- Flyweight: 2

25) Priam
- Error in finding design pattern.

26) QuestDB
- Facade: 2
- Flyweight: 16
- Mediator: 1
- Strategy: 1

27) Robolectric
- Error in finding design pattern.

28) Selenide
- Flyweight: 1

29) ZAProxy
- No design pattern found.

30) Zeppelin
- Composite: 1
- Flyweight: 4

B. Effects of CK Metrics on Maintainability

- **Weighted Methods per Class (WMC)**: A high WMC indicates increased complexity and potentially reduced maintainability. Our results suggest that the use of design patterns tends to result in classes with lower WMC values, indicating improved maintainability.
- **Depth of Inheritance Tree (DIT)**: A higher DIT value may imply increased difficulty in understanding and maintaining a class due to inherited properties and methods. In our study, we observed that classes with design patterns generally exhibit moderate DIT values, which implies that these patterns do not have a substantial negative impact on maintainability in terms of inheritance depth.
- **Number of Children (NOC)**: A higher NOC value indicates that a class has more responsibilities, which can make it harder to maintain. Our results show that classes with design patterns typically have lower NOC values, suggesting that design patterns can enhance maintainability by promoting focused and modular designs.
- **Coupling Between Objects (CBO)**: High coupling can make a class harder to understand, test, and maintain. We found that classes with design patterns generally have lower CBO values compared to classes without design patterns, indicating that design patterns can reduce coupling and improve maintainability.
- **Response for a Class (RFC)**: High RFC values indicate a high level of interaction between classes, which can negatively impact maintainability. Our study reveals that classes with design patterns exhibit lower RFC values, suggesting that design patterns can help streamline class interactions and improve maintainability.

C. Results Interpretation

We visualized the results of our study using graphs that display the average values of each CK metric for each project, along with the number of design patterns used in each project. These graphs provide a clear representation of the relationship between design patterns and maintainability, as quantified by the CK metrics. The CK metrics and the design patterns found in the projects provide some insights into the quality of the code and the software design patterns used in the projects.

Firstly, looking at the CK metrics, it is observed that the values of the metrics vary significantly across the projects. For example, the WMC metric ranges from 4.48 to 23.91, indicating a significant difference in the complexity of the classes across the projects. The DIT metric ranges from 1.02 to 2.15, indicating a difference in the level of inheritance in the projects. The NOC metric ranges from 0.008 to 0.77, indicating a difference in the level of coupling between classes in the projects. The CBO metric ranges from 3.28 to 9, indicating a difference in the level of coupling between objects in the projects. The RFC metric ranges from 4.71 to 17.16, indicating a difference in the level of complexity of the classes' methods and the number of methods in the classes.

Secondly, looking at the design patterns found in the projects, it is observed that some projects have more design patterns than others. For example, the Loom project has a large number of design patterns, while some projects have no design patterns found. It is also observed that some projects have multiple instances of the same design pattern, indicating that the design pattern is used extensively in the project.

Overall, the results suggest that the quality of the code and the software design patterns used in the projects vary significantly across the projects. Projects with higher values of CK metrics may indicate a higher level of code complexity, which may make the code harder to maintain and modify. The use of design patterns may improve the software's overall quality and maintainability, and the presence of multiple instances of the same design pattern in a project may indicate that the pattern is used extensively and may be a part of the project's architecture.

IV. Threats to Validity

In this section, we discuss the potential threats to the validity of our study and the steps we have taken to minimize their impact on our results.

A. Internal Validity

Internal validity threats are related to the study design and the possible confounding factors that may affect the observed relationships between variables.

- **Measurement errors**: Errors in measuring the CK metrics or identifying design patterns using the Pinot tool may affect the results. We have carefully selected and used reliable tools to minimize measurement errors.
- **Coding style and project complexity**: Differences in coding style, project complexity, and domain-specific considerations might affect CK metrics and design pattern usage. We attempted to mitigate this threat by selecting a diverse set of projects from various domains and programming languages.
- **Implementation variations**: Different developers might implement design patterns in varying ways, which could impact the maintainability of the code. While our study provides a general overview of design pattern usage, further investigation into specific implementation nuances may be necessary for a more comprehensive understanding.

B. External Validity

External validity threats pertain to the generalizability of the study results to other contexts.

- **Project selection bias**: Our study is based on a sample of 30 software programs, which may not be representative of all software projects. To mitigate this threat, we selected projects from various public repositories, application domains, and programming languages, ensuring a diverse and representative sample.
- **Design pattern selection**: The Pinot tool detects a limited number of about 18 GoF design patterns, which may not cover all possible design patterns used in software projects. While our study focuses on these 18 patterns, the impact of other design patterns on maintainability might be different.

C. Construct Validity

Construct validity threats relate to the suitability of the measurement instruments used in the study.

- **CK metric limitations**: While the CK metric suite is widely used to assess maintainability, it may not capture all aspects of maintainability. The use of additional maintainability metrics or qualitative assessments could provide a more comprehensive evaluation of maintainability and the impact of design patterns.

D. Conclusion Validity

Conclusion validity threats are concerned with the statistical inferences drawn from the study.

- **Statistical power**: With a sample size of 30 projects, our study has sufficient statistical power to detect significant effects. However, increasing the sample size or conducting replication studies could further strengthen the conclusions drawn from our results.

Despite these potential threats to validity, our study provides valuable insights into the impact of design patterns on software maintainability, as measured by the CK metrics. By addressing these threats and acknowledging their potential impact on the results, we aim to provide a reliable and meaningful analysis of the relationship between design patterns and maintainability.

V. Conclusion

In this study, we investigated the impact of design patterns on software maintainability by analyzing CK metric values across 30 software projects. Our motivation was to understand if the use of design patterns positively influences the maintainability of software, which is an essential quality attribute for software development and maintenance. We assessed maintainability by examining various CK metrics, such as WMC, DIT, NOC, CBO, and RFC.

Our findings reveal that there is a significant variation in CK metric values and design pattern usage across the analyzed projects. While some projects exhibited higher maintainability metrics, others showed lower values, suggesting that the maintainability of software projects is influenced by multiple factors, including the use of design patterns. Our results indicate that the presence of design patterns can potentially improve the overall quality and maintainability of a software project, depending on the specific patterns used and the context in which they are applied.

By interpreting these findings, we contribute to the understanding of the relationship between design patterns and software maintainability. Our study demonstrates the importance of considering design patterns as a factor that can influence maintainability and highlights the need for further investigation into the specific design patterns and their implementations that can lead to better maintainability in software

projects. Additionally, our study provides a foundation for future research on the topic, including replication studies and the exploration of other quality attributes related to design patterns.

In conclusion, the use of design patterns can play a critical role in improving software maintainability. As software development practices continue to evolve, the thoughtful application of design patterns can contribute to the creation of more robust, maintainable, and easily modifiable software systems.

REFERENCES

[1] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, Jun. 1994.

[2] S. Kim, K. Pan, and E. Whitehead, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, Oregon, USA, Nov. 2006, pp. 35-45.

[3] M. Aniche, "CK: A tool to calculate Chidamber and Kemerer's object-oriented metrics," 2016. [Online]. Available: https://github.com/mauricioaniche/ck