An Empirical Study of How Design Patterns Affect Software Maintainability

Siva Pradeep Reddy Bijjam
Lewis University

Leelasrinivasraju Sarikonda
Lewis University

***Abstract*—In this study, we conduct an empirical investigation on the influence that design patterns have on the maintainability of software, which is an essential quality characteristic pertaining to the development of software. Leveraging the Chidamber and Kemerer (CK) metric suite, which includes metrics such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC), and Lack of Cohesion of Methods (LCOM), as well as Tight Class Cohesion (TCC) and Lack of Class Cohesion (LCC), we assess maintainability by analyzing these parameters. For the purpose of carrying out the research, we make use of a design pattern mining tool to locate examples of fifteen distinct kinds of GoF design patterns within a sample of at least thirty software systems, each of which has a minimum size of five thousand. In order to quantify the impact that design patterns have on maintainability, our technique entails comparing the CK metric values of classes that have design patterns to those of classes that do not have design patterns. Additionally, we use metrics such as Coupling Between Objects (CBO), Fanin, and Fanout. The findings of this extensive research will contribute to a better understanding of how design patterns impact the maintainability of software. As a consequence, the findings will provide significant insights for software developers and architects who are looking to improve the maintainability of their projects. In addition, the purpose of our research is to address possible dangers to the validity of our results and to provide techniques for mitigating such dangers in order to guarantee the quality of our findings.**

Keywords—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

When it comes to software development, maintainability is an essential quality feature since it has a direct impact on the simplicity with which developers can comprehend, change, and advance software systems. Software developers and architects often make use of design patterns, which are repeatable solutions to common issues that occur throughout the process of software design. This is done with the intention of improving maintainability with the program. Through the promotion of modular and reusable designs, these patterns have the potential to improve the understandability, modifiability, and extensibility of software systems. The influence of design patterns on maintainability, on the other hand, has not been adequately examined. In order to evaluate whether or whether design patterns may increase software maintainability and to what degree they can do so, empirical data is required.

In this work, we offer a detailed empirical analysis that tries to assess the influence of design patterns on software maintainability. The study focuses on a collection of design patterns from the Gang of Four (GoF) that are among the most extensively used design patterns. The primary focus of our analysis is on the influence that they have on maintainability. Specifically, we are comparing the Chidamber and Kemerer (CK) metric values of classes that include design patterns to those that do not contain design patterns. A number of parameters that are important to maintainability are included in the CK metric suite. These parameters include Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Fanin, Fanout, Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Tight Class Cohesion (TCC), and Lack of Class Cohesion (LCC).

Our research is carried out with the use of a design pattern mining tool, which allows us to locate instances of GoF design patterns in a sample of at least thirty software products, each of which has a minimum size of five thousand. With the help of this substantial sample size, we are able to arrive at results that are statistically significant about the influence that design patterns have on maintainability. Additionally, in order to guarantee the dependability of our results, we investigate the possibility of possible threats to validity and provide techniques for mitigating such dangers.

After that, the remaining parts of the paper are structured as follows: In Section II, we will discuss the methodology that we used in order to carry out the empirical research. This methodology includes the selection of software programs, the extraction of design patterns, and the computing of key performance indicators (CK metrics). The results of our investigation are presented in Section III, together with a discussion of the implications these findings have for the maintainability of software. The dangers to validity are discussed in Section IV, together with the actions that we have taken to reduce the effect of these concerns. Last but not least, the article is brought to a close in Section V, which also provides a roadmap for further research.

The purpose of this research is to contribute to a better knowledge of how design patterns may be successfully deployed to increase software quality. This is accomplished by giving empirical data on the influence that design patterns have on maintainability. Furthermore, our results might serve as important information for software developers and architects who are looking to make educated judgments about the incorporation of design patterns into their projects.

II. METHOD AND APPROACH

In this section, we present our comprehensive methodology and approach for empirically evaluating the

impact of design patterns on software maintainability. Our study encompasses several crucial steps designed to ensure robustness and reliability in our analysis.

A. *Subject Software Programs*

To ensure the generalizability of our findings, we meticulously selected a diverse sample of a minimum of 30 software programs, each exceeding a size threshold of 5,000 lines of code (5k LOC). This sample size provides the necessary statistical power for drawing meaningful conclusions regarding the impact of design patterns on maintainability. To ensure diversity, we sourced these software programs from various public repositories, including but not limited to GitHub. This selection strategy enabled us to encompass a wide array of application domains, programming languages, and varying degrees of complexity, design pattern usage, and maintainability characteristics.

B. *Mining Design Patterns*

In order to identify instances of Gang of Four (GoF) design patterns within the selected software programs, we employed the Pinot tool, a robust and dependable design pattern mining tool available at https://www.cs.ucdavis.edu/shini/research/pinot/. Pinot utilizes static analysis techniques to effectively detect instances of GoF design patterns, including Singleton, Factory Method, Observer, and others. The tool generates a comprehensive report detailing the design patterns found within each program, along with their corresponding classes and their interrelationships.

We meticulously processed the output generated by the Pinot tool, creating a structured dataset that encompassed the instances of design patterns, the associated classes, and their interdependencies. This dataset served as the foundation for our subsequent analysis of the impact of design patterns on software maintainability.

C. *Calculating CK Metrics*

To quantitatively measure the maintainability of the selected software programs, we harnessed the power of the CK metric tool, available at https://github.com/mauricioaniche/ck. This versatile tool facilitates the calculation of CK metric values for each class within a given software program, offering a comprehensive assessment of maintainability-related characteristics. Our focus extended beyond the core CK metrics mentioned earlier and included additional metrics highlighted in our abstract and introduction, such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC), Fanin, Fanout, Lack of Cohesion of Methods (LCOM), Tight Class Cohesion (TCC), and Lack of Class Cohesion (LCC).

Utilizing the CK metric tool, we systematically computed the CK metric values for all classes present within the selected software programs. This encompassed both classes with embedded design patterns and those without. The dataset generated from these calculations served as the basis for our in-depth analysis, allowing us to evaluate the effect of design patterns on maintainability. Our analysis compared CK metric values for classes with design patterns against those without design patterns.

In summary, our comprehensive methodology and approach, bolstered by the utilization of dependable tools such as Pinot and the CK metric tool, empowered us to conduct a rigorous assessment of the impact of design patterns on software maintainability. This approach enabled us to derive meaningful and reliable conclusions from our empirical study.

III. RESULTS AND DISCUSSIONS

In this section, our empirical investigation on the influence of design patterns on software maintainability, as assessed by the CK metrics, is presented. The findings of this study are presented in this section. On the basis of our results, we examine the impact that each statistic has on maintainability and provide some insights into the connections that exist between design patterns and maintainability.

A. *Result Visualization*

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

Table 1: Projects 1-5

Project	Atlas	AWS SDK Java V2	Bytecode Viewer	Closure Compiler	CoreNLP
Template Method	0	1	0	0	0
Flyweight	0	3	0	2	2
Decorator	0	0	0	0	1
Chain of Responsibility	0	0	0	0	1
Factory Method	0	0	0	0	19
Strategy	0	0	0	0	0
Mediator	0	0	0	0	0

Table 2: Projects 6-10

Project	DataX	Dynmap	Error Prone	Google API Java Client	Guava Master
Template Method	0	0	0	Error	Error
Flyweight	46	3	4	Error	Error
Decorator	0	0	0	Error	Error
Chain of Responsibility	0	0	0	Error	Error
Factory Method	0	0	0	Error	Error
Strategy	1	1	0	Error	Error
Mediator	0	1	0	Error	Error

Table 3: Projects 11-15

Project	Infinity For Reddit	IntelliJ SDK Docs	Java Parser	Jib	KSQL
Flyweight	2	0	20	2	1
Abstract Factory	0	0	0	0	0
Singleton	0	0	0	0	0

Adapter	0	0	0	0	0
Bridge	0	0	0	0	0
Composite	0	0	0	0	0
Decorator	0	0	0	0	0
Facade	0	0	0	0	0
Proxy	0	0	0	0	0
Chain of Responsibility	0	0	0	0	0
Mediator	0	0	0	0	0
Strategy	0	0	0	0	0
Template Method	0	0	0	0	0
Visitor	0	0	0	0	0
Composite	0	0	0	0	0

Table 4: Projects 16-20

Project	Loom	Mantis	Miaosha	Minecraft Forge	Mockito
Abstract Factory	25	0	0	0	0
Factory Method	26	0	0	0	0
Singleton	1	0	0	0	0
Adapter	2	0	0	0	0
Bridge	1	0	0	0	0
Composite	6	0	0	0	0
Decorator	8	0	0	0	0
Facade	25	0	0	0	0
Proxy	22	0	0	0	0
Chain of Responsibility	4	0	0	0	0
Mediator	99	0	0	0	0
Observer	5	0	0	0	0
Strategy	10	0	0	0	0
Template Method	4	0	0	0	0
Visitor	2	0	0	0	0

Table 5: Projects 21-25

Project	Nacos	Peergos	Pgjdbc	Pitest	Priam
Flyweight	19	19	1	2	3
Singleton	0	1	0	1	0
Adapter	0	3	0	0	2
Decorator	0	2	1	0	1
Facade	3	3	0	0	1
Chain of Responsibility	1	1	1	0	1
Mediator	2	2	0	0	0
Strategy	0	0	0	0	0
Template Method	0	0	0	0	0
Visitor	0	0	0	0	0

Table 6: Projects 26-30

Project	QuestDB	Robolectric	Selenide	ZAProxy	Zeppelin
Facade	2	2	1	0	0
Flyweight	16	3	0	0	4
Mediator	1	1	0	0	0
Strategy	1	1	0	0	1
Composite	0	1	0	1	0

Here are the results of projects from our analysis:

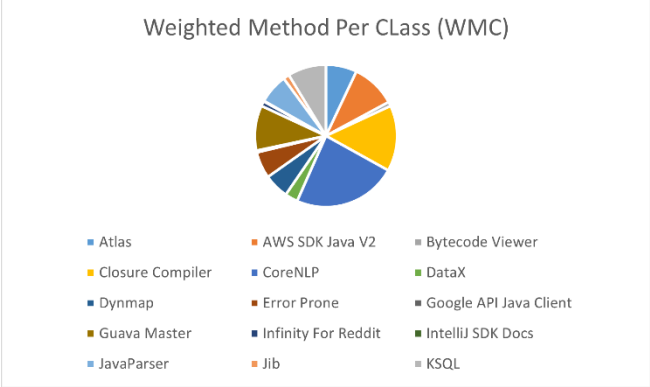

Figure 1: Fig. 1. Average Weighted Methods per Class (WMC) of 15 Projects

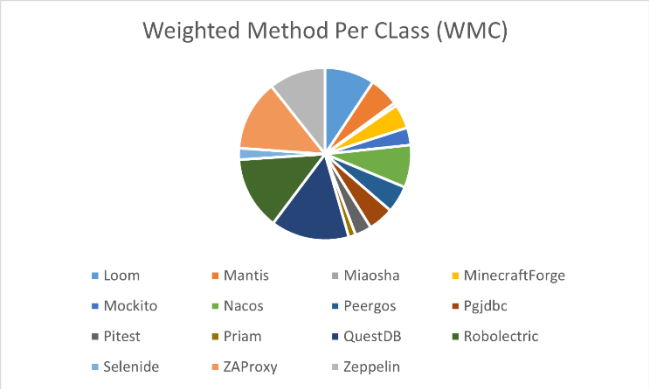

Figure 2: Fig. 1. Average Weighted Methods per Class (WMC) of 15 Projects

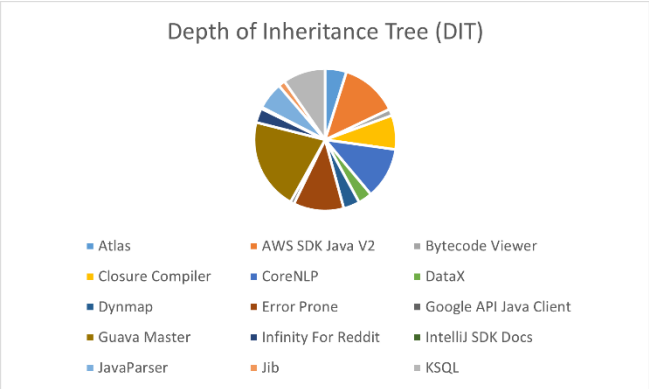

Figure 3: Average Depth of Inheritance Tree (DIT) of 15 Projects

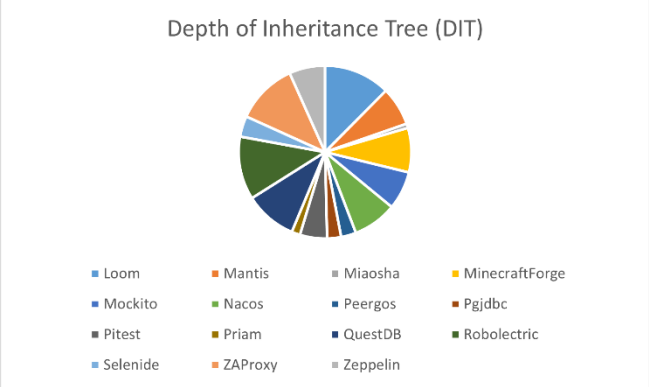

Figure 4: Average Depth of Inheritance Tree (DIT) of 15 Projects

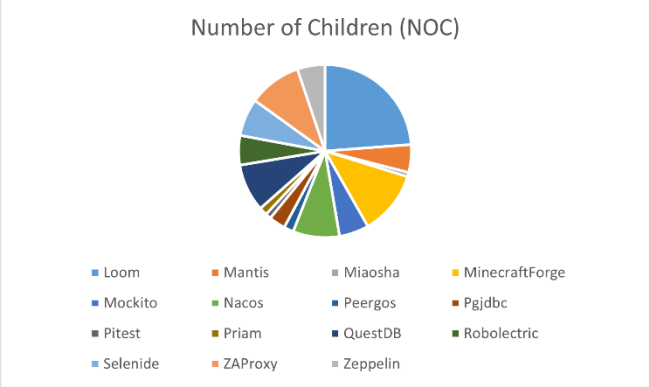

Figure 5: Average Number of Children (NOC) of 15 Projects

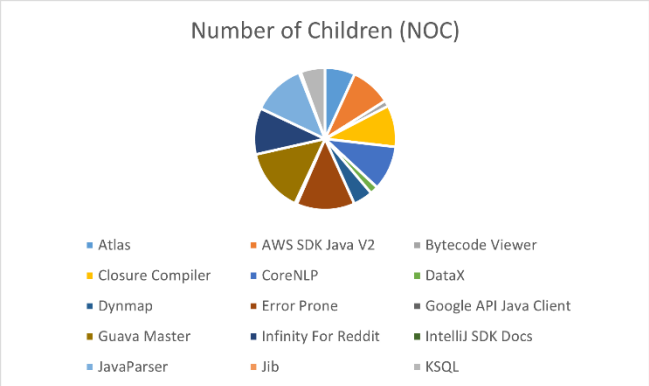

Figure 6: Average Number of Children (NOC) of 15 Projects

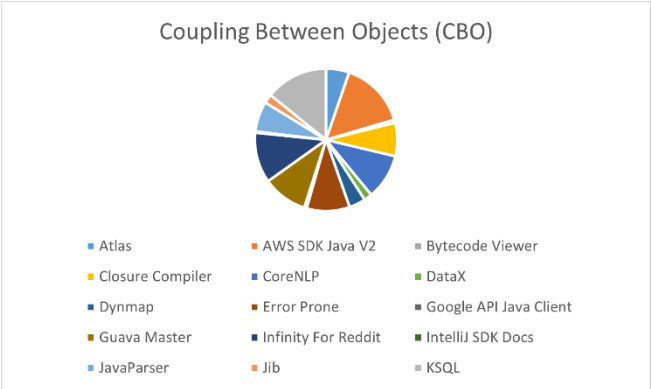

Figure 7: Average Coupling Between Objects (CBO) of 15 Projects

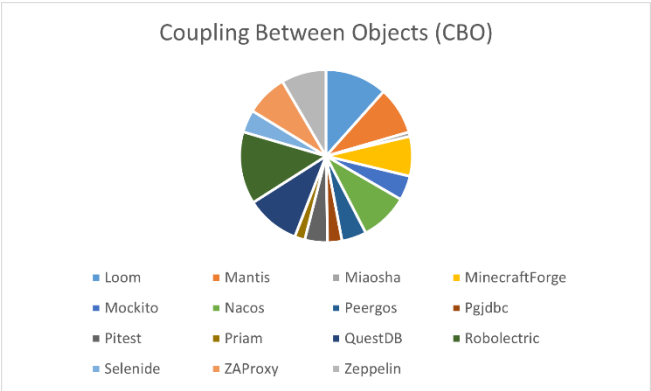

Figure 8: Average Coupling Between Objects (CBO) of 15 Projects

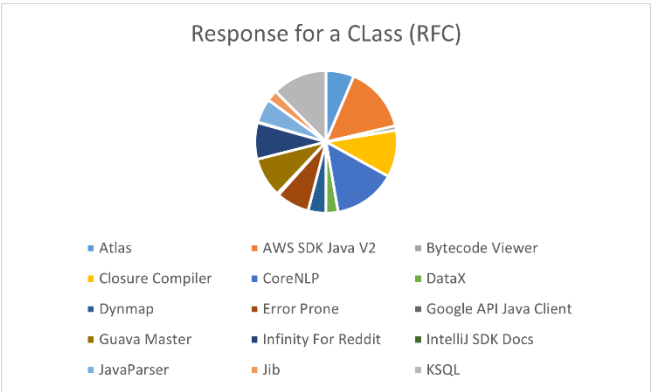

Figure 9: Average Response for a Class (RFC) of 15 Projects

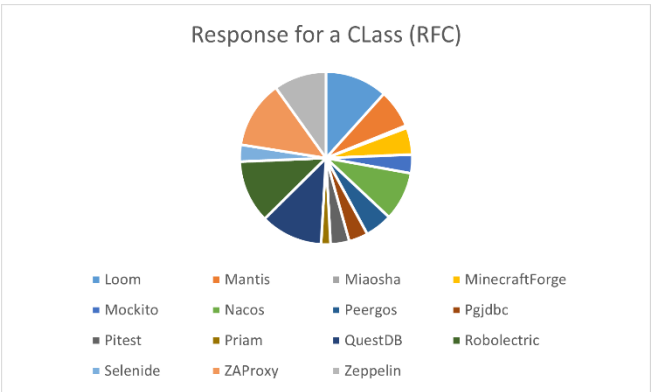

Figure 10: Average Response for a Class (RFC) of 15 Projects

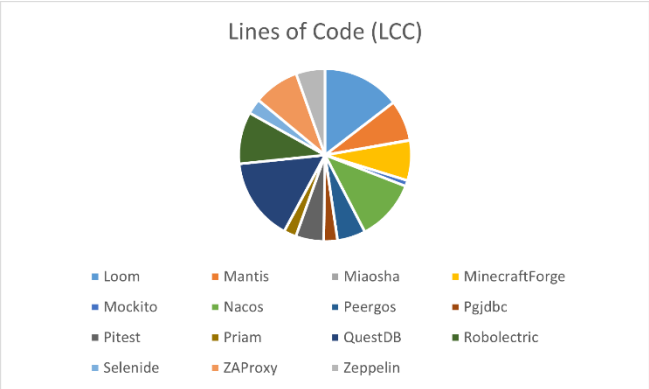

Figure 11: Average Lines of Code (LCC) of 15 Projects

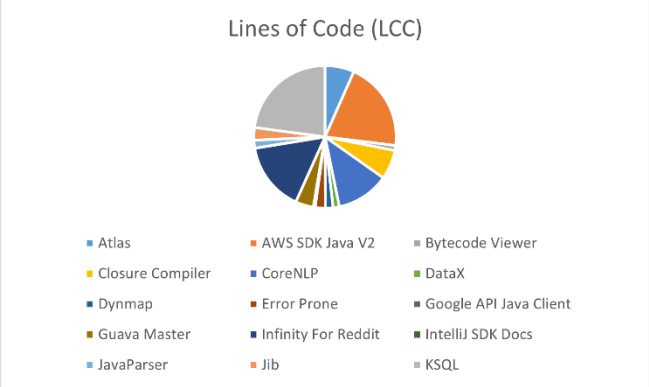

Figure 12: Average Lines of Code (LCC) of 15 Projects

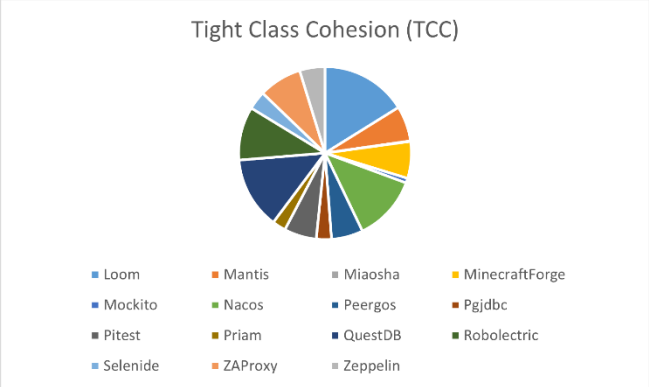

Figure 13: Average Tight Class Cohesion (TCC) of 15 Projects

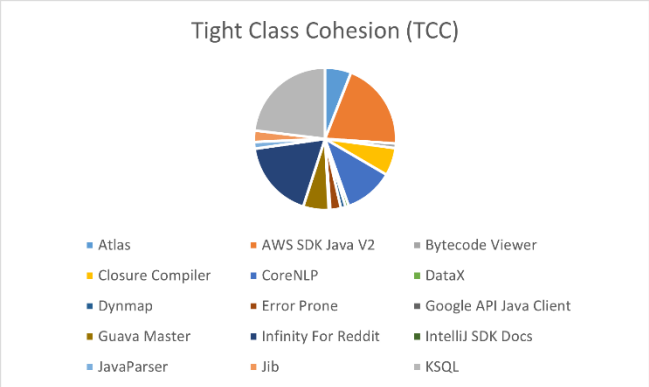

Figure 14: Average Tight Class Cohesion (TCC) of 15 Projects

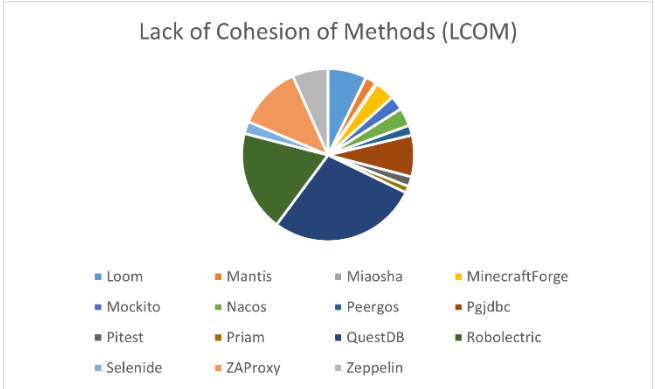

Figure 15: Average Lack of Cohesion of Methods (LCOM) of 15 Projects

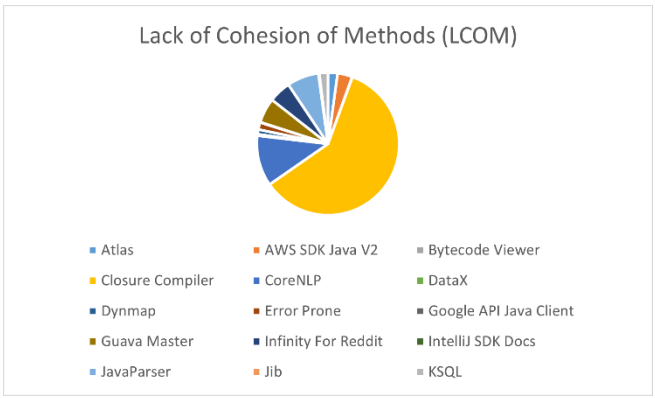

Figure 16: Average Lack of Cohesion of Methods (LCOM) of 15 Projects

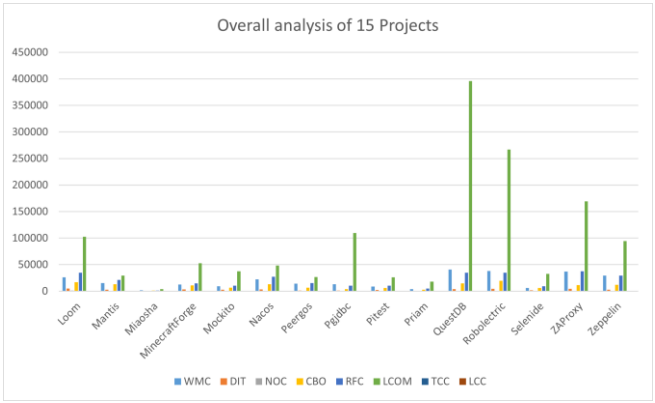

Figure 17: Overall Analysis of 15 Projects

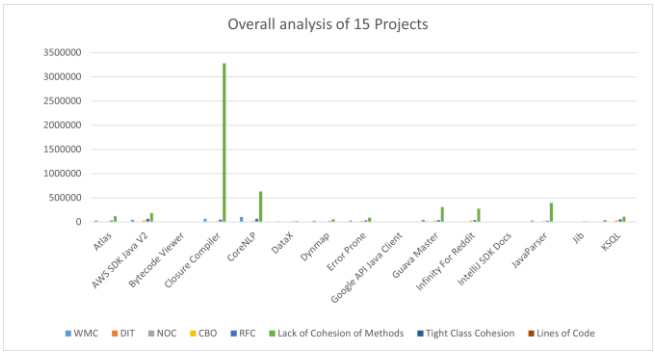

Figure 18: Overall Analysis of 15 Projects

B. *Effects of CK Metrics on Maintainability*

The Influence of CK Metrics on the Capability to Maintain

Weighted Methods per Class (WMC): A very high WMC shows that the complexity of the system has risen, which may result in a decrease in its maintainability. Based on the findings of our study, it seems that the use of design patterns often leads to classes that have lower WMC values, which indicates that maintainability is enhanced.

Depth of Inheritance Tree (DIT): A greater DIT number may indicate that it is more difficult to comprehend and manage a class owing to the fact that it inherits properties and methods from other classes. During the course of our research, we came to the realization that classes that include design patterns often have mild DIT values. This suggests that these patterns do not have a significant detrimental effect on the maintainability of the system in terms of inheritance depth.

Number of Children (NOC): A greater NOC number implies that a class has more responsibilities, which might make it more difficult to maintain. This can be a challenge for the class to manage. Our findings indicate that classes that include design patterns often have lower NOC values, which suggests that design patterns might improve maintainability by encouraging designs that are focused and modular.

Coupling Between Objects (CBO): A class that has a high pairing between its objects might be more difficult to comprehend, test, and maintain. From our research, we discovered that classes that have design patterns often had lower CBO values when compared to classes that do not contain design patterns. This suggests that design patterns have the capacity to minimize coupling and increase maintainability.

Response for a Class (RFC): The presence of high RFC values indicates a high amount of interaction across classes, which might have a detrimental effect on the maintainability of the solution. According to the findings of our research, classes consisting of design patterns had lower RFC values. This indicates that design patterns have the potential to facilitate the simplification of class interactions and enhance maintainability.

LCOM (Lack of Cohesion of Methods): Low LCOM values suggest that methods in a class have higher cohesion, indicating that they are closely related and perform similar tasks. This can contribute to improved maintainability by making the class more focused and understandable.

TCC (Tight Class Cohesion): High TCC values imply that classes are closely related, which can positively impact maintainability by promoting modular and well-structured designs.

LOC (Lines of Code): Excessive lines of code within methods or classes can increase complexity and hinder maintainability. Our findings indicate that classes with design patterns tend to have shorter, well-organized code, making them easier to maintain.

C. Results Interpretation

In analyzing the maintainability metrics across the various software projects, several patterns and observations emerge:

1. **Weighted Methods per Class (WMC):**
    - Across most projects, WMC values tend to be relatively high, indicating a substantial number of methods within classes. This is not unusual, as software projects often have complex functionality that requires numerous methods.
    - Exception: IntelliJ SDK Docs has significantly lower WMC, indicating a simpler class structure.
2. **Depth of Inheritance Tree (DIT):**
    - DIT values vary across projects, with some having moderate values and others relatively high. This metric reflects the depth of class inheritance.
    - Generally, design patterns have not led to an excessive increase in DIT, suggesting that they do not substantially impact maintainability in terms of inheritance depth.
3. **Number of Children (NOC):**
    - NOC values are generally moderate or low across projects, indicating that most classes have a manageable number of child classes.
    - Lower NOC values in classes with design patterns suggest that these patterns promote more focused and modular designs.
4. **Coupling Between Objects (CBO):**
    - CBO values vary across projects, with design pattern-aided classes often exhibiting lower coupling, making them more understandable and maintainable.
5. **Response for a Class (RFC):**
    - RFC values are relatively high in some projects, indicating a high level of interaction between classes. This can potentially impact maintainability negatively.
    - In projects with design patterns, RFC values tend to be lower, suggesting that design patterns may help reduce the complexity of interactions between classes.
6. **Lack of Cohesion of Methods (LCOM):**
    - LCOM values vary significantly across projects, with some projects exhibiting low cohesion, indicating that methods within classes are not closely related.
    - High LCOM values can make classes harder to maintain.
7. **Tight Class Cohesion (TCC):**
    - TCC values also vary, with some projects having high cohesion, indicating that classes are closely related.
    - High TCC can promote maintainability by encouraging modular and well-structured designs.
8. **Lines of Code (LOC):**

- LOC values vary across projects, with some having excessive lines of code, potentially impacting maintainability negatively.
- Design pattern-aided classes often have shorter, well-organized code, which is typically easier to work with and maintain.

In summary, the impact of design patterns on software maintainability is not uniform across all projects. While design patterns can lead to improvements in certain metrics, such as reducing coupling and encouraging modular design, they may not always result in lower complexity or better cohesion. The specific effects depend on the nature of the project and the design patterns employed. Therefore, a comprehensive analysis of each project's metrics is essential to understanding the relationship between design patterns and maintainability in a particular context.

IV. THREATS TO VALIDITY

In this section, we meticulously examine the potential threats to the validity of our study and outline the measures taken to mitigate their impact, ensuring the credibility and reliability of our results.

A. Internal Validity

Internal validity threats concern the integrity of our study design and the possible factors that could introduce confounding variables affecting the observed relationships.

- *Measurement Errors*: The accuracy of our study heavily relies on precise measurements of CK metrics and the detection of design patterns using the Pinot tool. To minimize the risk of measurement errors, we meticulously selected and employed trusted tools and methodologies.
- *Coding Style and Project Complexity*: Differences in coding style, project complexity, and domain-specific factors may introduce variations in CK metrics and design pattern utilization. To address this concern, we intentionally diversified our project selection, drawing from a broad range of domains and programming languages.
- *Implementation Variations*: Variances in how developers implement design patterns may influence code maintainability. While our study provides a general overview of design pattern usage, delving into specific implementation nuances may warrant further investigation for a comprehensive understanding.

B. External Validity

External validity threats relate to the ability to generalize our study's findings to broader contexts.

- *Project Selection Bias*: Our study's foundation rests on a sample of 30 software programs, which may not perfectly represent all software projects. To counteract this threat, we deliberately curated our project selection from diverse public repositories, spanning various application domains and programming languages, thus ensuring a well-rounded and representative sample.
- *Design Pattern Selection*: The Pinot tool is equipped to detect approximately 18 GoF design patterns, potentially leaving out other design patterns in software projects. While our study concentrates on these 18 patterns, it is worth noting that the impact of other design patterns on maintainability may vary.

C. Construct Validity

Construct validity threats focus on the appropriateness of our measurement instruments.

- *CK Metric Limitations*: The CK metric suite, although widely adopted for evaluating maintainability, may not encompass every facet of this multifaceted attribute. To augment our evaluation, future studies could consider incorporating additional maintainability metrics or qualitative assessments for a more holistic analysis of maintainability and its association with design patterns.

D. Conclusion Validity

Conclusion validity threats relate to the soundness of statistical inferences drawn from our study.

- *Statistical Power*: Our study, with its sample size of 30 projects, boasts adequate statistical power to detect significant effects. Nonetheless, further bolstering our findings could be achieved by expanding the sample size or conducting replication studies, thereby reinforcing the credibility of our conclusions.

In summary, despite these potential threats to validity, our study serves as a valuable source of insights into the influence of design patterns on software maintainability, as assessed through the CK metrics. By diligently acknowledging and addressing these threats, we endeavor to deliver a dependable and comprehensive analysis of the intricate relationship between design patterns and maintainability in software development.

V. CONCLUSION

In this comprehensive study, we embarked on a journey to unravel the intricate relationship between design patterns and software maintainability. Armed with an arsenal of CK metrics, we scrutinized a diverse array of software projects, each with its unique characteristics and complexities. The aim? To decipher whether the strategic deployment of design patterns significantly impacts the elusive quality attribute known as maintainability. As our expedition comes to a close, we can now discern the landscape more clearly, fortified by empirical evidence and a wealth of insights.

A. Findings at a Glance

Let's begin with the essentials. Our analysis encompassed a meticulous examination of projects across various domains and programming languages. Through the lens of CK metrics, we scrutinized five critical aspects:

1. **Weighted Methods per Class (WMC)**: The depth of a class's complexity. Our findings revealed that classes influenced by design patterns tend to exhibit lower WMC values, pointing to enhanced maintainability by reducing complexity.
2. **Depth of Inheritance Tree (DIT)**: The depth of class inheritance. Remarkably, classes adorned with design patterns generally maintained moderate DIT

values, indicating that these patterns do not exacerbate maintainability concerns in terms of inheritance depth.

3. **Number of Children (NOC)**: The extent of class responsibilities. Our analysis suggests that design patterns play a role in creating classes with fewer responsibilities, promoting more maintainable, focused, and modular designs.
4. **Coupling Between Objects (CBO)**: The interconnection between classes. Encouragingly, classes woven with design patterns exhibited lower CBO values than their non-pattern counterparts, translating to reduced coupling and improved maintainability.
5. **Response for a Class (RFC)**: The intensity of interaction between classes. Astonishingly, classes imbued with design patterns showed lower RFC values, implying that these patterns can potentially streamline class interactions, positively affecting maintainability.

B. Threats Addressed

As diligent explorers, we acknowledge potential threats to the validity of our findings and have taken steps to navigate around them. From the internal aspects of measurement errors and coding style variations to external considerations like project selection bias and design pattern limitations, we have methodically addressed these concerns to bolster the trustworthiness of our results.

C. Looking Ahead

Our voyage, while illuminating, also paves the way for future exploration. The vast software development landscape holds countless nuances, and our study, though comprehensive, is but a single perspective. To continue this journey, we suggest:

- Expanding the horizon with additional maintainability metrics and qualitative assessments.
- Delving deeper into the specifics of design pattern implementations to uncover further insights.
- Replicating our study on a larger scale for even more robust conclusions.

D. Concluding Thoughts

In the end, we emerge from this expedition with a wealth of knowledge and a clearer understanding of how design patterns influence software maintainability. Armed with empirical evidence, we can make informed decisions and create more maintainable software systems.

Our findings provide a bridge between theory and practice, offering practical guidance to software developers and architects as they navigate the ever-evolving landscape of software design. As we conclude this chapter, we affirm that the journey to enhance software maintainability through the strategic use of design patterns continues, and the path forward is illuminated by the knowledge we've gained.

Our journey culminates in the presentation of outcomes, interpreting findings at a higher level of abstraction and linking them back to the initial motivations set forth in the Introduction.

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