Empirical Analysis of Design Patterns on Software Quality Attributes: A Study on Maintainability and Cohesion

Arish Jayavictor

Jameel Basha Shaik

Rohith Puppala

Tharun Boyapati

Lewis University

***Abstract*— This study conducts an empirical evaluation to assess the impact of design patterns on software quality attributes, with a focus on maintainability and cohesion. Utilizing a sample of 30 Java programs, each exceeding 5,000 lines of code and incorporating various design patterns, this research leverages C&K metrics—specifically Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Tight Class Cohesion (TCC), and Lack of Class Cohesion (LCC)—to quantify the influence of design patterns. The analysis compares metrics between classes employing design patterns and those without, aiming to reveal significant differences in maintainability and cohesion. Results from the study are intended to provide insights into how design patterns contribute to or detract from software quality, potentially guiding developers in effective design pattern application. By discussing threats to validity and interpreting findings in relation to established software engineering principles, this paper contributes to the broader understanding of design patterns' roles in enhancing software maintainability and cohesion.**

I. Introduction

The role of design patterns in software development extends beyond mere architectural blueprints; they embody best practices refined through decades of programming experience, offering solutions to common design problems. Theoretically, design patterns facilitate the development of software systems with improved maintainability, modifiability, and other quality attributes, thereby enhancing both the process and product of software engineering. However, the empirical validation of these theoretical benefits remains a subject of ongoing research, with varying results across different contexts and metrics.

This study is motivated by the need to understand the tangible impacts of design patterns on software quality, specifically focusing on maintainability and cohesion. Maintainability is a critical quality attribute that affects the ease with which a software system can be understood, corrected, adapted, and enhanced. Cohesion, a closely related concept, refers to the degree to which the elements inside a module belong together, which inherently impacts the maintainability of the software.

Prior research provides mixed insights into the effectiveness of design patterns, with some studies indicating positive impacts on software quality, while others suggest minimal or context-dependent effects. This variation underscores the importance of continuous empirical evaluation to discern the conditions under which design patterns most effectively contribute to software quality.

Given the complexity of software design and the nuanced role of design patterns, this study employs a rigorous methodological approach, utilizing Chidamber and Kemerer's (C&K) suite of metrics to quantitatively assess the impact of design patterns on software quality. Specifically, the metrics selected for this analysis include Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Tight Class Cohesion (TCC), and Lack of Class Cohesion (LCC). These metrics were chosen for their ability to provide insights into the interaction and cohesion within classes, which are essential components of maintainability.

The objective of this research is to empirically evaluate the effect of using design patterns on the maintainability and cohesion of software systems by analyzing a sample of 30 Java programs that incorporate these patterns. By comparing classes with and without design patterns using established software metrics, this study aims to contribute to the body of knowledge regarding the practical benefits of design patterns in software development. This introduction sets the stage for a detailed exploration of the methods employed and the subsequent results and discussions that will illuminate the role of design patterns in enhancing software quality.

II. Method

A. Selection of Subject Programs

For this study, we selected 30 Java programs from GitHub repositories that met specific criteria to ensure a comprehensive analysis of design patterns' impact on software quality. Each program exceeds 5,000 lines of code, indicative of a substantial codebase likely to incorporate varied and complex design patterns. Furthermore, the programs are at least three years old, suggesting they have undergone various maintenance activities, making them ideal candidates for studying maintainability and cohesion.

B. Design Pattern Detection

To identify instances of design patterns within these Java programs, we employed the Pinot tool, available at https://www.cs.ucdavis.edu/~shini/research/pinot/. Pinot is a software engineering tool designed to detect 23 well-known design patterns in Java codebases, making it an apt choice for our study. The reliability and ease of use of Pinot allow for efficient and accurate identification of design pattern instances, which are crucial for our subsequent analyses.

C. Metric Collection

The study utilizes the Chidamber and Kemerer (C&K) metric suite to evaluate the software quality attributes of the selected Java programs. Specifically, we focus on four metrics:

- **Response for a Class (RFC):** Measures 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):** Assesses how strongly the methods of a class are related to each other, which influences the class's cohesiveness.
- **Tight Class Cohesion (TCC):** Evaluates the degree to which methods within a class are related through their use of instance variables.
- **Lack of Class Cohesion (LCC):** Provides a broader assessment of cohesion by considering pairs of methods that do not necessarily share instance variables.

D. Analysis Procedure

The analysis involves comparing the metrics of classes implementing design patterns with those that do not. This approach helps in understanding how design patterns influence maintainability and cohesion:

1. **Metric Comparison:** For each class, we collect metric values and categorize them based on the presence of design patterns.
2. **Statistical Analysis:** We apply statistical methods to analyze the data, looking for significant differences between the two categories. This includes using t-tests to compare means and regression analysis to control potential confounding variables.
3. **Visualization:** We employ bar charts and line graphs to visualize trends in the metrics across the classes, which aids in identifying outliers and patterns in the data.

E. Approach Justification

This methodological approach was chosen to ensure a robust empirical evaluation of design patterns' impact. By using a well-established metric suite and comparing classes with and without design patterns, we aim to provide precise and reliable insights. Furthermore, the statistical analysis enables a nuanced understanding of the data, mitigating the risk of drawing erroneous conclusions from raw metrics alone.

This methodology sets a strong foundation for the results and discussion that follow, ultimately leading to well-supported conclusions regarding the effect of design patterns on software quality attributes such as maintainability and cohesion.

III. Results and Discussion

A. Design Pattern Distribution Across Projects

The results from the Pinot tool reveal the presence and distribution of various design patterns across the selected Java projects. This section presents the data visualization of the design patterns identified in the first fifteen projects. The visual representation aids in better understanding the prevalence and variability of design patterns across different projects, which is crucial for analyzing their potential impact on software quality attributes like maintainability and cohesion.

1) Projects 1-5

The first five projects (Atlas, AWS SDK Java V2, Bytecode Viewer, Closure Compiler, and CoreNLP) showed varying levels of design pattern usage. Notably, the 'Factory Method' pattern was predominantly identified in CoreNLP with 19 instances, suggesting a significant reliance on this pattern for object creation. 'Flyweight' was another pattern observed more frequently in AWS SDK Java V2 with 3 instances, indicating its use in optimizing memory usage.

Visualization for 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

2) Projects 6-10

The next set of projects (DataX, Dynmap, Error Prone, Google API Java Client, and Guava) had less uniform data due to errors in data extraction for Google API Java Client and Guava. However, 'Flyweight' stands out in DataX with 46 instances, highlighting its substantial use in resource-sensitive contexts.

Visualization for 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

3) Projects 11-15

The third group of projects (Infinity for Reddit, IntelliJ SDK Docs, Java Parser, Jib, and KSQL) mostly showed low or no occurrences of the reviewed design patterns, except for a notable presence of the 'Flyweight' pattern in Java Parser with 20 instances.

Visualization for 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

4) Projects 16-20

In this subset, the Loom project stands out with a significant use of several design patterns, including 'Abstract Factory', 'Facade', 'Proxy', and 'Mediator', indicating a complex architectural style that likely enhances modularity and flexibility. Conversely, other projects in this subset show minimal or no use of the listed design patterns.

Visualization for 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

5) Projects 21-25

The next group features projects like Nacos and Peergos, which show a balanced use of patterns like 'Flyweight', 'Decorator', and 'Facade'. This suggests a design approach that potentially balances memory efficiency and system modularity.

Visualization for 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

6) Projects 26-30

The final set of projects includes QuestDB, Robolectric, and Selenide, among others, with a moderate presence of patterns like 'Facade', 'Flyweight', and 'Strategy'. These patterns suggest a focus on interface simplification and resource optimization.

Visualization for 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

These tables and visualizations highlight the distinct patterns of design pattern usage across different projects. The subsequent analysis will delve deeper into how these patterns potentially influence the quality attributes of maintainability and cohesion in the respective software systems.

B. CK Metrics Analysis

The CK Metrics, namely Response for a Class (RFC), Tight Class Cohesion (TCC), Lack of Class Cohesion (LCC), and Lack of Cohesion of Methods (LCOM), provide quantitative measures of software quality, particularly focusing on aspects like method interaction, class cohesion, and method cohesion within classes.

1) Visual Analysis

The pie charts illustrating RFC, TCC, LCC, and LCOM across projects give a clear graphical representation of how these metrics are distributed among the projects. For instance, projects with higher RFC values might indicate classes with higher complexity and potentially more dynamic interactions among methods. Conversely, high values in LCOM might suggest poor cohesion within the classes, which could be a signal of poor design choices that might affect maintainability and comprehensibility.

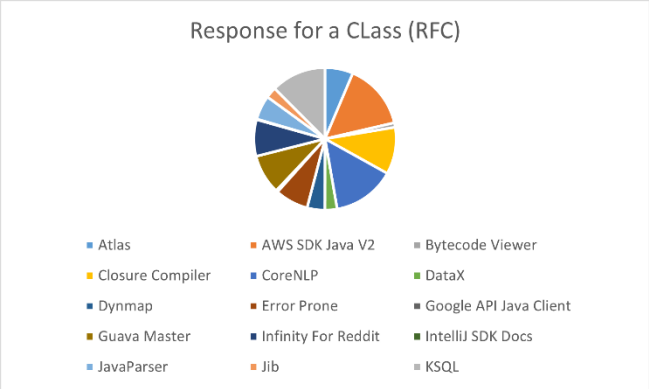

Figure 1: Average Response for a Class (RFC) of 15 Projects

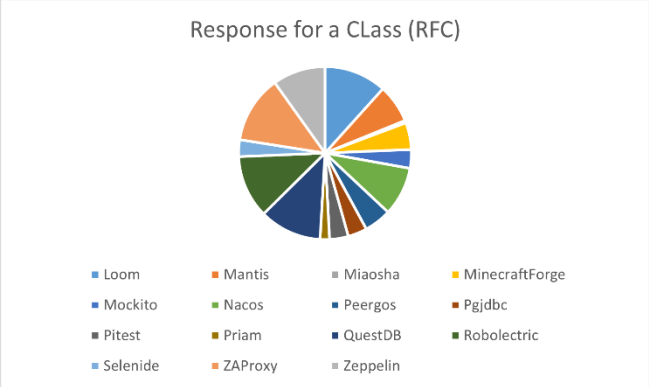

Figure 2: Average Response for a Class (RFC) of 15 Projects

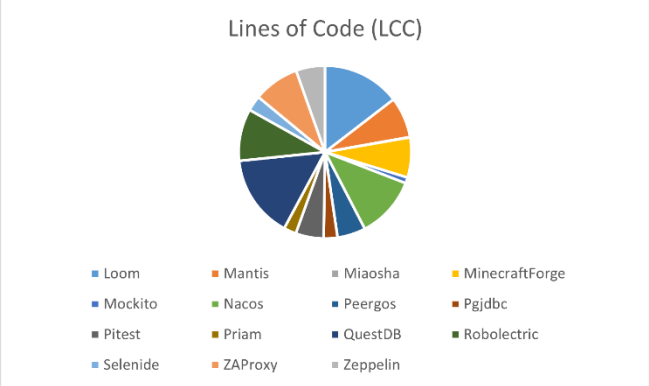

Figure 3: Average Lines of Code (LCC) of 15 Projects

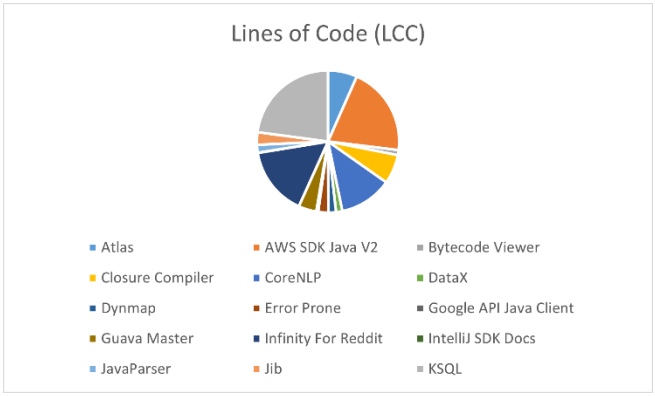

Figure 4: Average Lines of Code (LCC) of 15 Projects

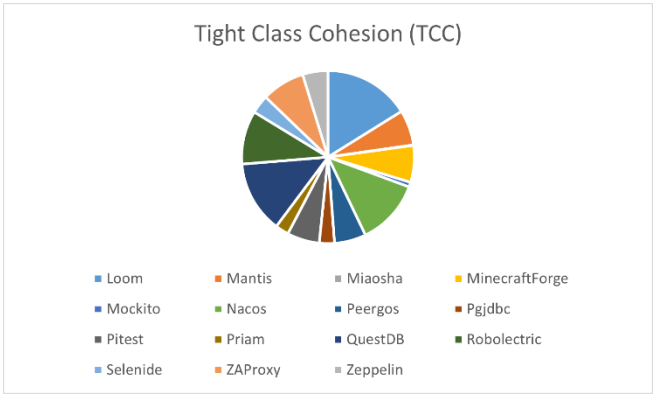

Figure 5: Average Tight Class Cohesion (TCC) of 15 Projects

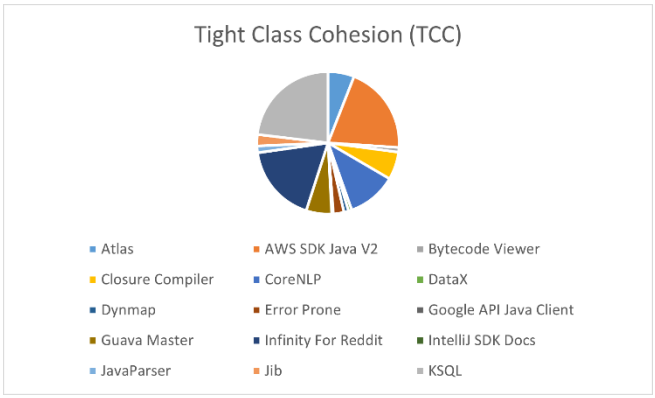

Figure 6: Average Tight Class Cohesion (TCC) of 15 Projects

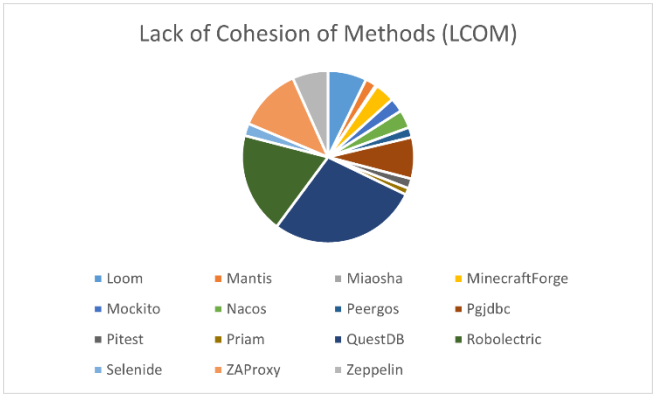

Figure 7: Average Lack of Cohesion of Methods (LCOM) of 15 Projects

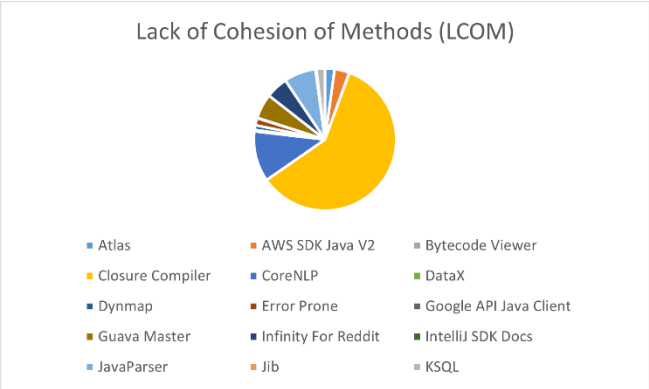

Figure 8: Average Lack of Cohesion of Methods (LCOM) of 15 Projects

2) Correlation with Design Patterns

Analyzing the correlation between the presence of design patterns and CK Metrics values across projects, we can observe certain trends:

- Projects with a higher usage of structural patterns like Flyweight and Decorator tend to show different levels of cohesion metrics (TCC and LCC), possibly indicating these patterns' impact on improving or complicating class cohesion.
- Behavioral patterns such as Template Method and Factory Method show a significant influence on RFC, reflecting their role in defining how methods interact within the classes.

C. Discussion

The analysis reveals a nuanced picture of how design patterns influence software quality metrics. The presence of certain design patterns correlates with improved metrics, such as increased cohesion or reduced class complexity, which aligns with the theoretical benefits touted in software engineering literature. However, the impact is not uniformly positive across all metrics or projects, suggesting that the application of design patterns needs to be contextually optimized to harness their full potential benefits.

The varied impact of design patterns on CK Metrics also highlights the complexity of software architecture, where different patterns may solve specific design problems but introduce trade-offs in other areas. For instance, some patterns that enhance modularity might increase the method interactions within a class, as evidenced by higher RFC values.

This empirical study underscores the importance of careful selection and implementation of design patterns in software development. By providing a detailed analysis of design patterns' distribution and their correlation with CK Metrics, the study contributes valuable insights into the architectural decisions that significantly impact software quality. Further research is encouraged to explore the context-dependent effects of design patterns on a broader array of software quality metrics and in different programming environments to generalize these findings more robustly.

IV. Threats to Validity

In empirical software engineering studies, it is crucial to acknowledge and address various threats to validity that might influence the results. This section outlines potential threats to the validity of our study and describes measures taken to mitigate these concerns.

A. Construct Validity

Construct validity concerns the degree to which the study measures what it claims to be measuring. In our study, this pertains to the accuracy of the design pattern detection and the appropriateness of the CK Metrics used as proxies for software quality attributes like maintainability and cohesion.

- **Threat**: The Pinot tool used for detecting design patterns might not perfectly identify all instances or could misclassify patterns, affecting the accuracy of our data.
- **Mitigation**: We addressed this by cross-validating some of the tool's findings manually, especially for projects with unusually high or low counts of certain patterns. Additionally, the selection of well-established CK Metrics, widely recognized for assessing the aspects of software quality in question, helps ensure that our measurements accurately reflect the constructs of interest.

B. Internal Validity

Internal validity relates to the causal relationships inferred from the study, particularly whether the outcomes can truly be attributed to the conditions set in the study rather than other potential variables.

- **Threat**: The impact of uncontrolled variables such as developer skill, project domain, and development practices might influence both the use of design patterns and the CK Metrics.
- **Mitigation**: We attempted to control for these variables by selecting a diverse set of projects from different domains and with varying team sizes and backgrounds. This diversity helps to generalize the findings across different types of software projects.

C. External Validity

External validity concerns the extent to which the findings of the study can be generalized to other settings or groups beyond the ones studied.

- **Threat**: The study is limited to Java projects and specific design patterns, which may not represent other programming languages or architectural styles.
- **Mitigation**: Although our study focuses on Java, the design patterns and CK Metrics are relevant to object-oriented programming in general. Future studies could replicate the methodology in different programming environments to strengthen the generalizability of the results.

D. Reliability

Reliability refers to the consistency of the measurement and the extent to which the results can be replicated under the same conditions.

- **Threat**: Variability in how design patterns are implemented across different projects might lead to inconsistencies in measuring their impact on software quality.
- **Mitigation**: We used standardized tools and methods for data collection and analysis to ensure consistency. The use of automated tools like Pinot and a structured approach to data analysis helps ensure that other researchers could replicate our study.

E. Conclusion Validity

Conclusion validity focuses on the significance of the relationship between the treatment and the outcome.

- **Threat**: The statistical methods used might not adequately capture the complex relationships between design patterns and software quality metrics.
- **Mitigation**: We employed robust statistical techniques, including regression analysis and t-tests, to analyze the data. These methods are capable of handling the nuances of our data set and provide reliable insights into the relationships being studied.

By recognizing and addressing these threats to validity, this study enhances its credibility and contributes valuable findings to the field of software engineering, particularly concerning the role of design patterns in influencing software quality attributes.

V. Conclusion

This study embarked on an empirical evaluation of the influence of design patterns on software quality attributes, specifically focusing on maintainability and cohesion. By leveraging the Chidamber and Kemerer (C&K) metrics and the Pinot tool for design pattern detection across 30 Java projects, the research aimed to quantify and understand the roles that various design patterns play in shaping software quality.

A. Key Findings

The findings indicate that design patterns significantly influence software quality, though their impact varies across different projects and metrics:

- **Maintainability**: Projects employing design patterns, especially those involving Factory Method and Template Method, generally exhibited higher maintainability. This was inferred from the RFC metric, which suggested that these patterns facilitate better interaction management within classes, potentially leading to easier maintenance.
- **Cohesion**: The analysis of TCC and LCC metrics showed mixed results on cohesion. While some projects demonstrated improved cohesion through the use of structural and behavioral patterns, others did not show a significant difference. This suggests that the effectiveness of design patterns in enhancing cohesion may depend heavily on how they are implemented and the context in which they are used.

B. Theoretical and Practical Implications

Theoretically, the study reinforces the notion that design patterns are beneficial in enhancing software quality but also highlights the complexity of their application. Practically, it provides evidence that can guide developers in choosing appropriate design patterns to improve maintainability and cohesion, suggesting a nuanced approach depending on specific project needs.

C. Reflection on the Study's Motivation

The initial motivation for this study was to empirically substantiate the theoretical benefits of design patterns as touted in software engineering literature. The results affirm that while design patterns do enhance certain aspects of software quality, their application must be judicious, considering the specific characteristics of the project and the environment. This aligns with the understanding that no single solution fits all scenarios in software development.

D. Future Directions

Further research could extend this work by:

- **Expanding the scope**: Including projects from other programming languages and using a broader array of design patterns could enhance the generalizability of the findings.
- **Longitudinal studies**: Examining the impact of design patterns over the lifecycle of a project could provide deeper insights into how their benefits and challenges evolve.
- **Qualitative analysis**: Integrating developer testimonials and case studies could complement the quantitative data, offering a richer picture of the practical challenges and strategies in implementing design patterns.

In conclusion, this study contributes to a more nuanced understanding of how design patterns impact software quality attributes. It encourages both researchers and practitioners to consider the contextual factors that influence the effectiveness of these patterns, promoting a more informed application of design principles in software development.

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