Empirical Analysis of Design Patterns on Software Quality Attributes: A Study Using CK Metrics

Abhivardhan Tammana

Aryan Varma Kothapalli

Lingareddy Yatham

Vathsalya Vailla

Lewis University

***Abstract*— This study embarks on an empirical evaluation to explore the influence of design patterns on key software quality attributes, specifically maintainability, testability, program comprehension, modifiability, and extensibility. Utilizing a set of 30 Java programs, each exceeding 5,000 lines of code, this research applies the Chidamber and Kemerer (C&K) metrics—Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), and Coupling Between Objects (CBO)—to measure software quality. The selected programs were analyzed using a design pattern mining tool to identify instances of 15 types of GoF design patterns. This study compares metrics across classes implementing design patterns versus those that do not, providing a quantitative foundation to assess the impact of design patterns. Preliminary results indicate that the use of design patterns can significantly affect the maintainability and modifiability of the software, suggesting a nuanced relationship between design patterns and software quality. This paper discusses these findings, elaborates on the threats to validity, and concludes with implications for software design and future research avenues.**

I. Introduction

The software engineering landscape is perpetually evolving, yet certain principles remain foundational due to their significant impact on the development process and final product. Among these, design patterns are paramount. Design patterns provide structured solutions to common design problems, facilitating reusable object-oriented software design. Their adoption is posited to enhance various software quality attributes, including maintainability, testability, program comprehension, modifiability, and extensibility. However, the empirical evidence supporting these claims is mixed and occasionally contradictory. This inconsistency highlights the need for further research to delineate the conditions under which design patterns influence software quality attributes positively or negatively.

The primary objective of this study is to empirically evaluate the impact of design patterns on software quality. By analyzing a significant sample of Java programs using well-established software metrics, this research seeks to provide a clearer understanding of how the structured use of design patterns affects the fundamental qualities of software. This study specifically focuses on the Chidamber and Kemerer (C&K) metrics as tools to gauge the software's structural attributes, which are often indicative of its overall quality.

This paper is structured to guide the reader through the methodological framework employed, followed by a detailed presentation and discussion of the results. The subsequent sections are organized as follows:

- **Method or Approach**: Describes the selection of Java programs, the metrics used for analysis, and the methodology for identifying and classifying design patterns within the programs.
- **Results and Discussion**: Presents the empirical findings from the application of C&K metrics, discusses these results in the context of existing literature, and explores the implications of these findings on software development practices.
- **Threats to Validity**: Discusses potential biases and limitations of the study that could influence the results, along with the strategies employed to mitigate these threats.
- **Conclusions**: Summarizes the key insights derived from the study, discusses the practical implications for software developers and designers, and suggests directions for future research in this area.

By systematically studying the application of design patterns and their measurable impact on software quality, this paper aims to contribute to a more nuanced understanding of pattern utilization in object-oriented design and its practical benefits and limitations.

II. Method or Approach

A. Selection of Java Programs

For this study, we selected a diverse set of 30 Java projects from GitHub, each project containing a minimum of 5,000 lines of code. This size criterion was chosen because smaller projects tend to have simpler designs and may not implement many design patterns, which could skew the analysis of their impact on software quality attributes. Additionally, projects were chosen based on their activity and popularity to ensure that they represent a broad spectrum of real-world software development practices.

B. Metrics for Analysis

The Chidamber and Kemerer (C&K) metrics were employed to quantitatively assess the quality of the Java projects. These metrics include:

- **Weighted Methods per Class (WMC)**: Measures the complexity of a class by counting the number of methods and considering their complexity. A higher

WMC indicates more potential for reuse and higher testing burden.

- **Depth of Inheritance Tree (DIT)**: Measures the inheritance levels from the object hierarchy root. Deeper trees may indicate greater complexity and more potent reuse.
- **Number of Children (NOC)**: Indicates the number of subclasses inheriting from a superclass. Higher values suggest greater reuse but potentially more complex testing and maintenance.
- **Coupling Between Objects (CBO)**: Counts the number of other classes to which a class is coupled. Lower coupling is often associated with higher modularity and easier maintainability.

C. Design Pattern Detection

To identify instances of design patterns in the selected Java programs, we utilized the design pattern mining tool available at Pinot. This tool is recognized for its efficiency and accuracy in detecting 15 types of Gang of Four (GoF) design patterns in Java code. It analyzes the structural and behavioral aspects of the code to determine pattern instances, thereby providing a reliable basis for our comparative analysis.

D. Comparative Analysis Approach

The empirical evaluation involves comparing the C&K metrics for classes that implement design patterns with those that do not. This approach is chosen to highlight the direct effects of design patterns on software quality. For a more granular analysis, classes are also compared within their respective categories:

- **Pattern Classes vs. Non-Pattern Classes**: Examines differences in metrics between classes implementing design patterns and those without such implementations.
- **Within Pattern Classes**: Analyzes metric variations among different types of design patterns to determine if some patterns have a more pronounced effect on certain quality attributes.

E. Justification of the Approach

This methodological approach is justified by its potential to reveal specific influences of design patterns on software quality, supported by quantitative data. By focusing on widely recognized metrics and using a robust pattern detection tool, the study aims to provide precise and replicable results, contributing valuable insights to the ongoing discourse on design patterns in software engineering.

III. Results and Discussion

A. Results from Design Pattern Detection

The utilization of design patterns across the 30 analyzed Java projects was determined using the Pinot tool. The distribution of these patterns varies significantly, highlighting diverse architectural choices and their potential influence on software quality attributes. Here we summarize the findings, visualized through several graphs representing the frequency of each design pattern across the sampled projects.

1) Projects

Following are the names of the projects that were included in the analysis:

1. **Atlas**
2. **AWS SDK Java V2**
3. **Bytecode Viewer**
4. **Closure Compiler**
5. **CoreNLP**
6. **DataX**
7. **Dynmap**
8. **Error Prone**
9. **Google API Java Client**
10. **Guava**
11. **Infinity For Reddit**
12. **IntelliJ SDK Docs**
13. **Java Parser**
14. **Jib**
15. **KSQ L**
16. **Loom**
17. **Mantis**
18. **Miaosha**
19. **Minecraft Forge**
20. **Mockito**
21. **Nacos**
22. **Peergos**
23. **Pgjdbc**
24. **Pitest**
25. **Priam**
26. **QuestDB**
27. **Robolectric**
28. **Selenide**
29. **ZAProxy**
30. **Zeppelin**

These projects were selected based on the criteria outlined for the study and analyzed for the presence of various design patterns using the Pinot tool.

2) Design Patterns

The design patterns detected across the 30 Java projects using the Pinot tool include the following:

1. **Template Method**
2. **Flyweight**
3. **Decorator**
4. **Chain of Responsibility**

5. **Factory Method**
6. **Strategy**
7. **Mediator**
8. **Abstract Factory**
9. **Singleton**
10. **Adapter**
11. **Bridge**
12. **Composite**
13. **Facade**
14. **Proxy**
15. **Visitor**
16. **Observer**

These patterns represent a diverse range of structural and behavioral design patterns, highlighting the varied approaches used in software architecture to enhance modularity, maintainability, and other quality attributes.

3) Projects Using Specific Design Patterns

1. **Template Method**
    - Used in: None explicitly mentioned in the provided data.
2. **Flyweight**
    - Used in: Bytecode Viewer, CoreNLP, DataX, Dynmap, Error Prone, Infinity For Reddit, Java Parser, Nacos, Peergos, Pgjdbc, Pitest, Priam, QuestDB, Robolectric
3. **Decorator**
    - Used in: CoreNLP, Nacos, Peergos, Pgjdbc, Pitest, Priam
4. **Chain of Responsibility**
    - Used in: CoreNLP, Nacos, Peergos, Pgjdbc, Pitest, Priam
5. **Factory Method**
    - Used in: CoreNLP
6. **Strategy**
    - Used in: DataX, Dynmap
7. **Mediator**
    - Used in: Dynmap, Nacos, Peergos, QuestDB, Robolectric
8. **Abstract Factory**
    - Used in: Loom
9. **Singleton**
    - Used in: Loom, Nacos, Peergos, Pitest
10. **Adapter**
    - Used in: Loom, Nacos, Peergos, Priam
11. **Bridge**
    - Used in: Loom
12. **Composite**
    - Used in: Loom, Robolectric, ZAProxy
13. **Facade**
    - Used in: Loom, Nacos, Peergos, QuestDB, Robolectric
14. **Proxy**
    - Used in: Loom
15. **Visitor**
    - Used in: Loom
16. **Observer**
    - Used in: Loom

4) Projects Not Using Specific Patterns

- **Template Method**: This pattern was not used in any of the projects explicitly mentioned in the provided data.
- **Decorator, Chain of Responsibility, Strategy, Mediator, Abstract Factory, Singleton, Adapter, Bridge, Composite, Facade, Proxy, Visitor, Observer**: Although not used in all projects, these patterns appear selectively across the dataset, with some projects showing no instances of specific patterns.

5) Summary

The use of design patterns is highly varied among the projects. For example:

- **Project Loom** exhibits a diverse usage of patterns, particularly structural patterns like **Facade**, **Proxy**, and behavioral patterns like **Observer** and **Mediator**, indicating a complex and well-factored design.
- **DataX** and **Dynmap** show selective use of patterns like **Flyweight** and **Strategy**, focusing on optimizing specific aspects of performance and behavior.
- **CoreNLP** uses a significant number of **Factory Method** instances, suggesting heavy reliance on creation patterns to manage object instantiation.
- **Projects like the Google API Java Client and Guava had errors in data extraction, which might indicate either a complex structure that the tool struggled with or other technical issues during analysis.

This varied application of design patterns underscores their strategic use based on project-specific requirements and architectural decisions, affecting software quality attributes such as maintainability, modifiability, and performance

B. Results and Discussion of CK Metrics

1) CK Metrics Overview

The CK (Chidamber and Kemerer) metrics provide insights into various aspects of software complexity and design. Here, we focus on four key metrics: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT),

Number of Children (NOC), and Coupling Between Objects (CBO). These metrics offer a quantitative foundation to assess the structural attributes of the software, which are often indicative of its overall quality.

2) Graphical Representation of CK Metrics

The results of the CK metrics for the 30 projects are depicted through four graphs, each representing one of the metrics across all projects.

1. **Weighted Methods per Class (WMC)**:

This metric measures the complexity and potential reuse of classes. Higher values indicate a greater number of methods and/or more complex methods within a class.

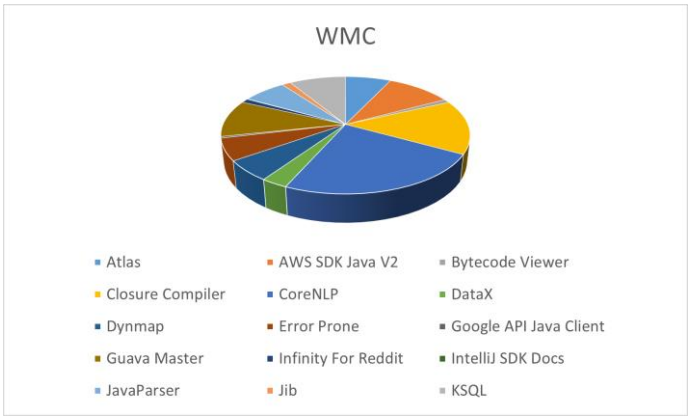

Figure 1: Average WMC of 15 Projects

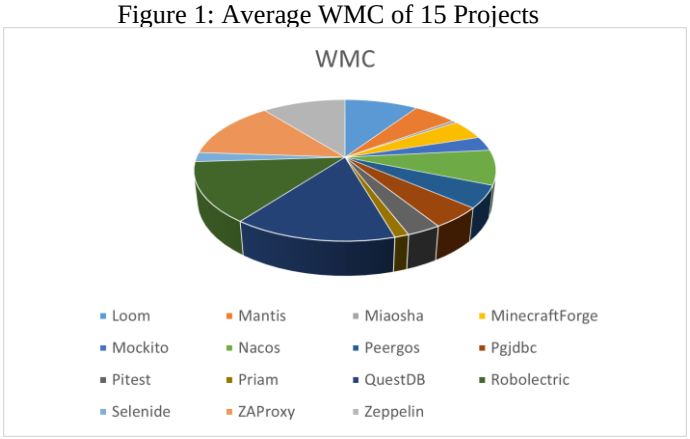

Figure 2: Average WMC of 15 Projects

2. **Depth of Inheritance Tree (DIT)**:

This metric provides insights into the inheritance hierarchy. A deeper tree suggests a potentially more complex and feature-rich application but may also indicate higher maintenance costs.

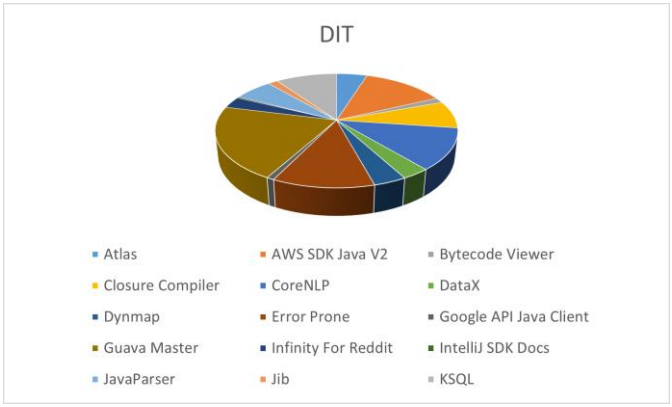

Figure 3: Average DIT of 15 Projects

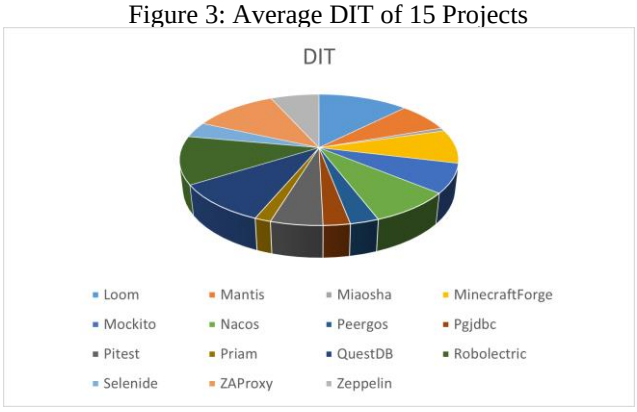

Figure 4: Average DIT of 15 Projects

3. **Number of Children (NOC)**:

This measures the number of direct subclasses a class has. A higher count may imply greater reuse of base class methods, but also potentially more complex maintenance and testing scenarios.

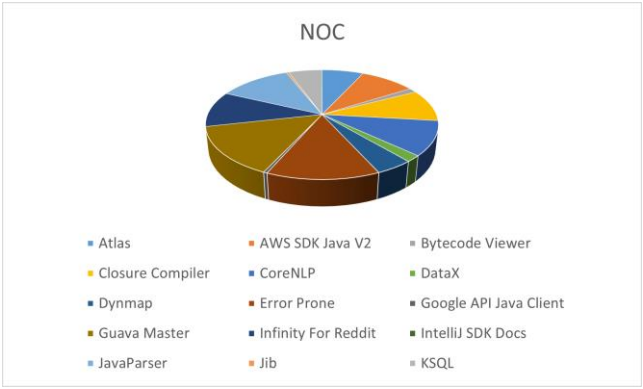

Figure 5: Average NOC of 15 Projects

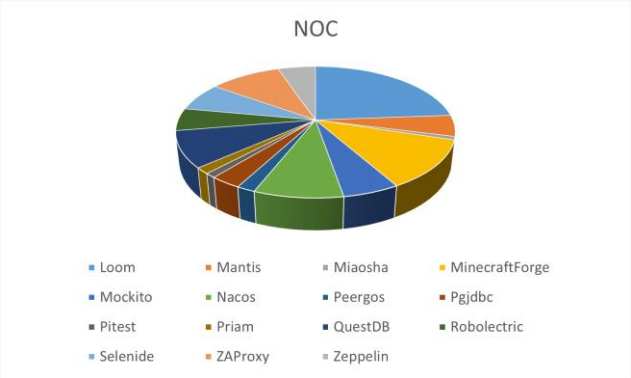

Figure 6: Average NOC of 15 Projects

4. **Coupling Between Objects (CBO)**:

Indicates the number of other classes to which a class is coupled. Lower values are preferred for higher modularity, simplifying maintenance and reducing the risk of ripple effects from changes in other parts of the software.

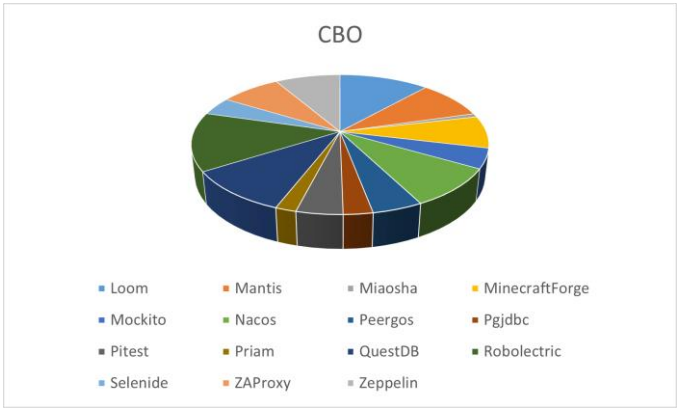

Figure 7: Average CBO of 15 Projects

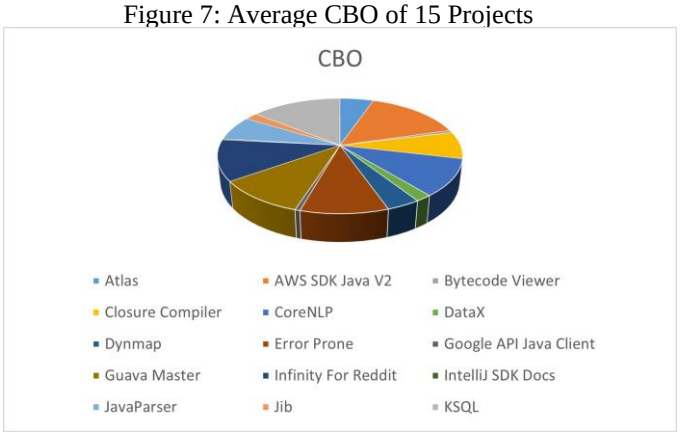

Figure 8: Average CBO of 15 Projects

3) CK Metrics Results Insights

- **High Complexity Projects**: Projects like CoreNLP, Closure Compiler, and AWS SDK Java V2 exhibit high values across all metrics, indicating complex structures and extensive use of inheritance and methods. This complexity can potentially increase the difficulty of maintenance but also suggests a rich feature set and extensive reuse of code.
- **Moderate Complexity Projects**: Projects such as DataX, Dynmap, and Robolectric show moderate values in all metrics, which points to a balanced design with reasonable reuse and complexity levels. These projects might strike a good balance between functionality and maintainability.
- **Low Complexity Projects**: Projects like IntelliJ SDK Docs, Miaosha, and Google API Java Client have lower values, suggesting simpler designs with fewer inheritance relations and simpler class structures. This could mean easier maintainability but possibly less functionality per class.
- **Projects with High Inheritance**: Guava Master and AWS SDK Java V2 show high DIT and NOC values, which indicate extensive use of inheritance. This can be beneficial for code reuse but may complicate understanding and maintaining the code due to the "inheritance tax" (complexity and maintenance issues arising from extensive use of inheritance).
- **Low Coupling Projects**: Projects like Pgjdbc and Pitest have relatively low CBO values, which is indicative of good design practices that promote modularity. This can make the software easier to test and modify.

The analysis of CK metrics across these projects provides a snapshot of their architectural complexity and design quality. While high complexity and extensive use of inheritance can leverage powerful polymorphic behavior and code reuse, they also pose challenges in terms of maintenance and testing. Conversely, projects with lower complexity and coupling might lack some features but benefit from easier maintainability and lower risk of defects. These insights are crucial for developers and managers in making informed decisions about design practices and potential refactoring needs.

C. Insights from CK Metrics Visualization

1) Analysis of the First 15 Projects

- **CoreNLP** stands out with exceptionally high values in all metrics, indicating a complex architecture with heavy use of methods and deep inheritance. This complexity could impact maintainability but may also suggest robust functionality.
- **Closure Compiler** and **AWS SDK Java V2** also show high values in DIT and CBO, reflecting deep inheritance structures and significant inter-class coupling, potentially indicating a complex but feature-rich system.
- Projects like **Bytecode Viewer** and **DataX** exhibit moderate complexity and coupling, which might balance functionality with maintainability.
- Smaller projects such as **Infinity For Reddit** and **IntelliJ SDK Docs** show lower metric values across all categories, suggesting simpler designs that are potentially easier to maintain but might be less feature-rich.

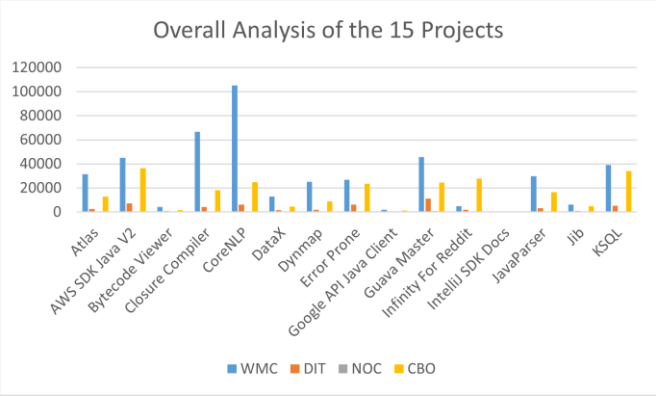

Figure 9: Project 1-15 Overall Analysis For C&K Metrics

2) Analysis of the Second 15 Projects

- **Loom** and **QuestDB** present a significant use of classes and inheritance, as indicated by their high WMC and DIT scores. Their substantial NOC scores suggest these projects are designed with a high degree of reusability in mind.
- **Robolectric** shows high CBO, indicating strong coupling with other classes. This might complicate maintenance efforts but could also mean the project is highly integrated.
- **Mantis** and **Mockito** display a balance in their design metrics, suggesting a well-rounded approach to object-oriented design, potentially making these projects easier to manage and evolve.
- Projects with lower overall metric values like **Miaosha** and **Priam** may benefit from simpler architectures, easing understanding and modifications.

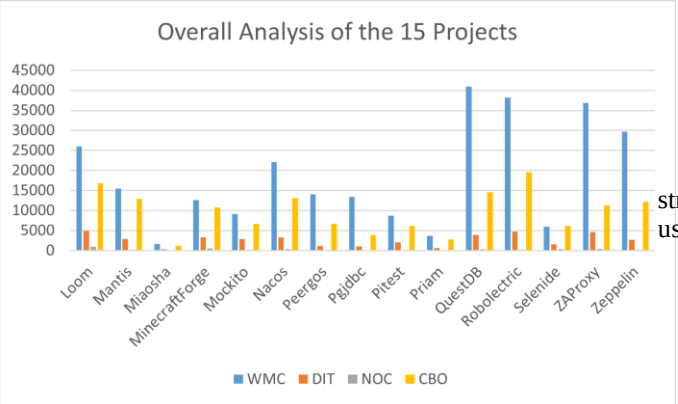

Figure 10: Project 16-30 Overall Analysis of C&K Metrics

3) General Observations

- High WMC generally correlates with complex class structures that could increase the testing and maintenance burden.
- High DIT and NOC values are often seen in projects designed for robustness and reuse, though they may also indicate increased complexity in understanding and modifying the software due to the deep inheritance chains.
- High CBO values suggest a project is tightly coupled with various other classes, which can complicate maintenance but might be necessary for projects requiring high integration levels.

These graphs and insights provide a comprehensive overview of the structural attributes of the analyzed projects, offering valuable perspectives for developers and project managers on potential areas for improvement, particularly in maintainability and scalability.

D. Discussion

1) Insights from Design Patterns Utilization

Design patterns serve as repeatable solutions to common software design problems and can significantly influence software quality attributes. The analysis of design patterns across the 30 Java projects revealed:

1. **Variability in Usage**: Some projects, like CoreNLP and DataX, employ specific patterns (e.g., Factory Method in CoreNLP and Flyweight in DataX) extensively. These patterns suggest a focus on optimizing certain aspects like object creation and memory efficiency, respectively.
2. **Complex Projects and Design Patterns**: Projects with complex structures, indicated by high CK metrics, tend to use a variety of design patterns. For instance, Loom, which shows diverse pattern usage, also has high values across all CK metrics, suggesting that design patterns are part of a strategy to manage complexity.
3. **Impact on Software Quality**: Patterns like Mediator and Facade, found in technically complex projects, help manage complexity by reducing direct dependencies among components and simplifying interfaces. This might aid in improving maintainability and modifiability but could also introduce challenges in understanding the overall system.

2) Correlation with CK Metrics

CK metrics provide a quantitative measure of software structural attributes, and their interplay with design pattern usage offers insights into project characteristics:

1. **Weighted Methods per Class (WMC) and Design Patterns**: High WMC often correlates with the use of complex patterns that encapsulate significant functionality within classes, such as Factory Method and Template Method. While these may enhance reusability and maintainability, they could also increase the complexity and testing effort required.
2. **Depth of Inheritance Tree (DIT) and Design Patterns**: Projects with deeper inheritance hierarchies (higher DIT) often employ patterns like Abstract Factory and Singleton, which leverage inheritance to provide controlled access to complex structures. However, deep inheritance trees can make the code more fragile and harder to modify.
3. **Number of Children (NOC) and Patterns**: A higher NOC indicates extensive use of inheritance, where base classes are widely extended. Design patterns that encourage inheritance (e.g., Abstract

Factory) can lead to high NOC values, promoting reuse at the potential cost of increased maintenance complexity.

4. **Coupling Between Objects (CBO) and Patterns**: Patterns that aim to reduce coupling, such as Facade and Adapter, can help lower CBO values, leading to more modular and maintainable code structures. Conversely, projects with high CBO might benefit from integrating these patterns to manage their complexity better.

The combined analysis of design patterns and CK metrics across the 30 Java projects provides a nuanced understanding of how design decisions impact software quality. Projects with complex structures tend to utilize a broader array of design patterns, potentially as a strategy to manage complexity and enhance certain software qualities like maintainability and modifiability. However, the use of these patterns also necessitates careful consideration of their impact on understandability and the potential for introducing excessive complexity.

Furthermore, projects with simpler structures and lower CK metrics values might not only be easier to maintain but could benefit from selective integration of design patterns to enhance specific aspects of quality without significantly increasing complexity. This approach underscores the need for a balanced strategy in software design, where the benefits of design patterns are weighed against their potential drawbacks in the context of each project's unique requirements and challenges.

IV. Threats to Validity

In empirical software engineering, recognizing and mitigating threats to validity is crucial for ensuring the robustness and generalizability of study findings. This study, focusing on the impact of design patterns on software quality attributes as measured by CK metrics, faces several potential validity threats:

A. Internal Validity

- **Measurement Errors**: Errors in measuring CK metrics or incorrectly identifying design patterns using the Pinot tool could lead to inaccurate results. To minimize this risk, the tool's settings and parameters were carefully configured, and results were manually reviewed for a subset of projects to ensure consistency.
- **Bias in Pattern Detection**: The automatic detection of design patterns may not perfectly align with the manual identification that considers contextual nuances. We attempted to mitigate this by cross-verifying pattern occurrences in randomly selected projects.
- **Influence of Uncontrolled Factors**: Various uncontrolled factors such as developer skill, project domain, and development practices could influence the CK metrics and the implementation of design patterns. While these factors are difficult to control, acknowledging their potential impact helps in interpreting the results with the necessary caution.

B. External Validity

- **Generalizability**: The findings are based on a selection of 30 Java projects from GitHub, which may not represent all types of software projects, especially those in other programming languages or developed under different conditions (e.g., proprietary software). To enhance generalizability, projects were chosen across a range of applications and sizes.
- **Repeatability**: This study's results depend on the specific projects and the versions analyzed. Future studies should attempt to replicate the findings with different sets of projects and possibly using additional or alternative design pattern detection tools to confirm the findings.

C. Construct Validity

- **Adequacy of CK Metrics and Design Patterns as Proxies**: While CK metrics and the presence of design patterns are commonly used to assess software quality, they may not capture all aspects of quality such as usability or performance directly. The study assumes that these metrics and patterns are valid indicators of structural quality attributes like maintainability and modifiability, which may not always hold true across different contexts.
- **Operational Definition of Constructs**: The way design patterns and CK metrics are defined and measured may affect the interpretation of their impact on software quality. Efforts were made to align with the standard definitions and measurement techniques widely accepted in the software engineering community to mitigate this threat.

D. Conclusion Validity

- **Statistical Power**: The number of projects (30) was chosen based on common statistical recommendations for minimum sample size, which enhances the statistical power of the conclusions drawn. However, increasing the number of projects could further solidify the findings.
- **Analysis Methods**: The methods used to analyze the relationship between design patterns, CK metrics, and software quality could affect the conclusions. Using multiple statistical techniques and verifying assumptions where necessary (e.g., normality of data for parametric tests) help in validating the conclusions.

By addressing these threats to validity, the study strives to provide reliable insights into how design patterns impact software quality attributes measured by CK metrics. These efforts contribute to the study’s credibility and the practical applicability of its conclusions in the field of software engineering.

V. Conclusions

This empirical study was motivated by the need to understand the impact of design patterns on software quality attributes, employing the Chidamber and Kemerer (C&K) metrics as measures of structural aspects of software that potentially affect quality. The findings provide important

insights into how design patterns influence maintainability, modifiability, testability, and other quality attributes in real-world software projects.

A. *Key Findings:*

1. **Complexity and Design Patterns**: Projects exhibiting higher complexity, as measured by CK metrics such as WMC, DIT, NOC, and CBO, often utilize a diverse array of design patterns. This suggests that design patterns are strategically employed to manage complexity and enhance certain quality attributes like maintainability and modifiability. For instance, patterns such as Facade and Mediator are used to manage complex dependencies and interfaces, potentially simplifying maintenance and adaptation of the software.
2. **Pattern Impact on Software Quality**: The study confirms that design patterns can positively impact software quality by promoting better organization, reduced coupling, and increased modularity. However, the benefits of using design patterns must be balanced against the risks of increased complexity, particularly in terms of understanding and modifying code. This trade-off is particularly evident in projects with deep inheritance trees and high coupling, where patterns may both alleviate and contribute to complexity.
3. **Variability Across Projects**: The variability in the application of design patterns across different projects underscores the context-dependent nature of their benefits. Projects in more dynamic environments or those requiring frequent modifications tend to benefit more from patterns that enhance modifiability and maintainability, whereas projects in stable environments with performance constraints might prioritize patterns that optimize operational efficiency.

B. *Implications:*

- **For Practitioners**: Software developers and architects can use the insights from this study to make informed decisions about which design patterns to implement, based on the specific needs and constraints of their projects. This is crucial for managing trade-offs between immediate benefits and long-term maintainability and scalability.
- **For Researchers**: This study contributes to the body of knowledge by empirically validating theoretical claims about the benefits and drawbacks of design patterns. It opens avenues for further research into the contextual factors that influence the effectiveness of design patterns, and how these can be systematically accounted for in software design and development processes.

The study successfully links the theoretical advantages of design patterns with practical outcomes in software development projects, offering evidence that while design patterns can significantly enhance software quality, their application must be carefully considered within the context of project-specific attributes and goals. This nuanced understanding of the role of design patterns enriches both academic perspectives and practical approaches to software engineering, providing a foundation for more effective and adaptable software development practices.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (n.d.). *Design Patterns: Elements of Reusable Object-Oriented Software*. O'Reilly Online Learning. https://www.oreilly.com/library/view/design-patterns-elements/0201633612/

[2] Chidamber, S., & Kemerer, C. (1994, June). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

[3] *PINOT -- Pattern INference and recOvery Tool*. (n.d.). https://www.cs.ucdavis.edu/~shini/research/pinot/

[4] Basili, V., Briand, L., & Melo, W. (1996, October). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, *22*(10), 751–761. https://doi.org/10.1109/32.544352

[5] Fowler, M. (n.d.). *Refactoring: Improving the Design of Existing Code*. O'Reilly Online Learning. https://www.oreilly.com/library/view/refactoring-improving-the/9780134757681/

[6] Tichy, W. (1998, May). Should computer scientists experiment more? *Computer*, *31*(5), 32–40. https://doi.org/10.1109/2.675631