**Effects of Design Patterns on Software Maintainability**

***Presented by***

**Siva Reddy Potlapati -** **L30086362**

**Pavan Kalyan Teluguti – L30088205**

**Bhavya Regalla-** **L30089923**

**Vijesh Medikonda-** **L300085988**

**Spring 2024-II**

# **ABSTRACT**

This empirical study uses CK measures, including Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM), to investigate how design patterns affect program maintainability. It investigates their impact on maintainability by contrasting 15 software projects with design patterns against 15 projects without design patterns. Software maintainability is evaluated as the dependent variable and design pattern utilization as the independent variable. The goal of the study is to improve knowledge of how software quality and design patterns are related. With the use of rigorous statistical analysis and metric comparisons between classes that are patterned and those that are not, the study provides an understanding of how design patterns affect maintainability. The results could influence the design of more maintainable systems and have an impact on software development processes. The findings might affect software development procedures and the development of more maintainable systems.

*Keywords:* K Measures, Weighted Methods per Class (WMC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), Design Patterns, Program Maintainability, Software Quality, Empirical Study, Statistical Analysis, Metric Comparisons, Software Development Processes.

# **1.Introduction**

Design patterns are reusable approaches to typical software design issues that aim to enhance the qualities of software, including extensibility, modifiability, and maintainability. Nonetheless, prior research has shown inconsistent findings about the influence of design patterns on maintainability. The goal of this study is to respond to the research query.

**RQ**: How does the use of design patterns impact the maintainability of software?

The term "maintainability" in software engineering describes how simple it is to comprehend, change, and expand software. It is expected that design patterns would increase maintainability by using reusable and modular solutions. However, further research is necessary to fully understand how they affect coupling and complexity.

The purpose of this research is to use three CK metrics—Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM)—to experimentally assess the impact of design patterns on software maintainability.

# **2.Methodology**

## **2.1 Subject Programs**

Thirty software projects with at least five thousand lines of code (LoC) were chosen. Fifteen of these projects make use of design patterns, while fifteen do not. With the help of the open-source software repository GitHub, the projects were located. We made sure that there are at least five GoF design patterns in the fifteen projects.

**2.2 Metrics and Variables**

We selected the following factors to look at how design patterns affect software maintainability:

* **Independent Variable**: The presence of design patterns
* **Dependent Variable**: Software maintainability measured using CK metrics.

We have used the following CK metrics for my analysis.

* **Weighted Methods per Class (WMC):** WMC adds up the cyclomatic complexity of each method in the class to determine how difficult the class is. It acts as a gauge for the amount of work needed to comprehend, uphold, and alter a class. Generally speaking, classes with higher WMC values are less maintainable and more difficult.
* **Coupling Between Objects (CBO)**: This technique counts the classes to which a certain class is connected. A high degree of reliance across classes is indicated by a strong coupling, which complicates maintenance. Low CBO classes are ideal since they encourage modularity and simplicity of maintenance.
* **Low Cohesion in methods (LCOM)**: By evaluating the extent to which methods in a class share instance variables, LCOM calculates the degree of cohesiveness among them. Unrelated methods are common in classes with high LCOM, which suggests weak cohesion and more maintenance effort.

Additionally, we have also included the following metrics for further analysis.

* **Afferent Coupling (Ca)**: It Counts the number of dependents that come into contact with a class.
* **Efferent Coupling (Ce)**: Measures the quantity of outward dependencies from a class using efferent coupling (Ce).
* **Instability (I):** Calculates how unstable a module or class is.
* **Abstractness (A)**: Indicates how abstract a class or module is.
* **Maintainability Index (MI)**: The maintainability index, or MI, calculates a composite score based on LCOM, WMC, and CBO to assess maintainability.

## **2.3 Data Collection**

We chose thirty software applications with a total size of more than 5,000 lines of code to analyse. The projects were split into two groups: fifteen projects with established design pattern use and fifteen projects devoid of design patterns. This is how we went about gathering the data:

1. **Pattern Detection**: To find design patterns in the chosen projects, we used the pattern4.jar. This jar uses source code analysis to identify 15 GoF design patterns. Observer, Composite, Factory Method, Singleton, and Strategy are some of the patterns that were found. We used the program for every project and noted the classes that used the selected design patterns.
2. **Metric Collection:** We used the CK-code metrics tool, developed by Aniche et al., to measure modularity and maintainability. The tool was obtained from Git Hub. By following the instructions provided in the repository's README file, we ran the tool to collect the following metrics for each class:

**Weighted Methods per Class (WMC):** By adding up the cyclomatic difficulty of each method in the class, the Weighted Methods per Class (WMC) technique calculates the overall complexity of the class.

**Coupling Between Objects (CBO):** Calculates the quantity of classes to which a certain class is connected. A high degree of reliance across classes is indicated by a strong coupling, which makes maintenance difficult**.**

**Lack of Cohesion in Methods (LCOM):** Assesses how effectively methods in a class share instance variables to determine how cohesive they are. Unrelated methods are common in classes with high LCOM, which suggests weak cohesion and more maintenance effort.

**Afferent Coupling:** Determines how many classes rely on a particular class using the Afferent Coupling (Ca) measure. Increased afferent coupling suggests that the class is widely used, which means that modifications may have an effect on many classes.

**Efferent Coupling (Ce)**: Calculates how many classes a certain class relies on. A class with a higher efferent coupling is likely more reliant on other classes, which makes maintenance more difficult.

**Instability (I):** The instability (I) of a class or module indicates how changeable it is. It has a range of 1 (very unstable) to 0 (very steady).   
**Maintainability Index (MI)**: The Maintainability Index (MI), which is based on WMC, CBO, and LCOM, is a composite score for maintainability. Higher maintainability is indicated by a higher score.

**Weighted Average Calculation:** We used the WMC value as the weighting factor to get the weighted average for each statistic. Classes with more methods or more complexity are given more weight in this approach.

## **2.4 Analysis Method**

1. We divided the projects into two categories:

* **Patterned Projects**: Projects that implement known design patterns.
* **Non-Patterned Projects**: Projects that do not implement any identified design patterns.

1. Using the CK-code metrics tool, we collected the following metrics for each class in both patterned and non-patterned projects: WMC, CBO, LCOM, Ca, Ce, I, A, and MI.
2. **Computed Weighted Averages**: For each project, we calculated the weighted average for each metric using WMC as the weighting factor. This step helped us obtain an aggregated metric value for each project.
3. **Statistical Comparison**: We conducted statistical comparisons between projects with and without design patterns using the following techniques:

**Independent Samples T-Test**: The CK metrics means for patterned and non-patterned projects were compared using the t-test. The statistical significance of the mean differences has been determined using the t-test.

**​:** Variance of patterned projects.

**​:** Variance of non-patterned projects.

**𝑛1:** Number of patterned projects.

**𝑛2*n*:** Number of non-patterned projects

**Effect Size (Cohen's d):** We computed Cohen's d to determine the difference's size. This measure aids in estimating the effect magnitude, which falls into the following categories:

* Small Effect: 0.2 ≤ d < 0.5
* Medium Effect: 0.5 ≤ d < 0.8
* Large Effect: d ≥ 0.8

**Correlation Analysis:** To find out how the use of certain design patterns and software maintainability metrics relate to one another, we performed correlation analysis.

**ANOVA:** ANOVA stands for Analysis of Variance. We used ANOVA to analyze the metric values across projects and see whether there were any statistically significant changes.

*F = MSBetween/*​ *MSwithin*

* *MSBetween* **​:** Mean Sum of Squares Between Groups.
* *MSwithin* **​:** Mean Sum of Squares Within Groups.

1. **Program Comprehension and Maintainability Index Analysis**: We assessed the Program Comprehension (Ca and I) and the Maintainability Index (MI) in addition to the CK measures. Gaining further insight into how certain design patterns affect understanding and maintainability was the aim.​

I= ***Ce*** / *Ca* + ***Ce***

​ ***Ce*:** Efferent Coupling (outgoing dependencies).

1. **Impact of Specific Design Patterns on Quality Attributes**: We examined the effect of several design patterns on quality parameters in order to get more profound understanding. As an illustration:  
   **Singleton Pattern**: By offering a single point of access to shared resources, this pattern enhances program understanding.

**Factory Method Pattern**: Increases modifiability by decoupling object creation from usage.

**Strategy Pattern**: Simplifies modifying the algorithmic behaviour of classes.

**Template Method Pattern**: Enhances comprehension by providing a clear high-level workflow.

**Observer Pattern**: Decouples event publishers and subscribers, improving modularity and testability.

# **3.** **Results and Discussion**

This analysis approach allowed us to comprehensively assess the effects of design patterns on software maintainability across various projects.

## **3.1 Metric Values for Projects with Design Patterns**

**Projects implementing known design patterns:** Computed the weighted average of CBO, LCOM, and WMC for all the projects that are making use of design patterns. Refer to Table 1 for more details.

**Table 1: Weighted average values of CBO, LCOM, and WMC of 15 projects implementing design patterns.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Project** | **CBO** | **LCOM** | **WMC** |
| blog\_demos | 10.45 | 244.19 | 47.31 |
| DataX | 11.83 | 100.82 | 55.47 |
| dynmap | 22.42 | 471.32 | 134.14 |
| Infinity-For-Reddit | 31.17 | 342.42 | 114.13 |
| java-design-patterns | 4.82 | 5.24 | 9.79 |
| javaparser | 48.09 | 24348.20 | 620.64 |
| jfreechart | 16.61 | 2202.16 | 112.60 |
| ksql | 19.48 | 216.18 | 28.81 |
| logging-log4j2 | 10.67 | 1018.25 | 50.19 |
| miaosha | 6.21 | 78.60 | 28.18 |
| MinecraftForge | 21.78 | 597.53 | 37.22 |
| poi | 17.55 | 1228.30 | 76.65 |
| Priam | 10.52 | 593.46 | 30.91 |
| spring-framework | 13.88 | 361.90 | 49.66 |
| storm | 14.69 | 869.46 | 96.30 |

## **3.2** **Metric Values for Projects without Design Patterns**

**Projects not implementing design patterns:** Computed the weighted average of CBO, LCOM, WMC for all the projects which are not making use of design patterns. Refer to Table 2 for the metric values.

**Table 2: Weighted average values of CBO, LCOM, WMC of 15 projects not implementing design patterns.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Project** | **CBO** | **LCOM** | **WMC** |
| aeron | 23.28 | 1609.04 | 101.59 |
| animation-samples | 10.56 | 40.22 | 14.58 |
| booknotes | 3.34 | 1.31 | 5.99 |
| FlatLaf | 10.58 | 622.02 | 92.73 |
| hutool | 9.38 | 2141.09 | 88.18 |
| libsu | 8.97 | 164.82 | 39.37 |
| mantis | 17.11 | 83.63 | 31.36 |
| mlkit | 16.29 | 28.73 | 24.38 |
| modeldb | 30.47 | 6183.72 | 143.89 |
| nacos | 9.78 | 91.87 | 28.21 |
| pravega | 22.45 | 385.61 | 47.77 |
| serve | 13.09 | 398.74 | 53.92 |
| sofa-ark | 10.24 | 119.87 | 34.10 |
| supertokens-core | 22.75 | 965.22 | 69.59 |
| XChange | 11.50 | 602.83 | 22.57 |

**3.3 Design Patterns Identified**

We have identified the following design patterns in the first 15 projects of our dataset.

**Table 3: List of Design Patterns Identified in Each Project**

|  |  |
| --- | --- |
| **Project** | **Design Patterns Identified** |
| blog\_demos | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Builder Pattern, Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Proxy Pattern, Flyweight Pattern, Chain of Responsibility Pattern, Command Pattern, Iterator Pattern, Mediator Pattern, Observer Pattern, State Pattern, Strategy Pattern, Template Method Pattern, Visitor Pattern |
| DataX | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Builder Pattern, Proxy Pattern, Iterator Pattern, State Pattern, |
| dynmap | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Proxy Pattern, Flyweight Pattern, Chain of Responsibility Pattern, Command Pattern, Iterator Pattern, Mediator Pattern, Strategy Pattern, Template Method Pattern, Visitor Pattern |
| Infinity-For-Reddit | Factory Method Pattern, Builder Pattern, Singleton Pattern, Adapter Pattern, Bridge Pattern, Proxy Pattern, State Pattern, Strategy Pattern, Command Pattern, Observer Pattern, Iterator Pattern |
| java-design-patterns | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Builder Pattern, Prototype Pattern, Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Flyweight Pattern, Proxy Pattern, Chain of Responsibility Pattern, Command Pattern, Iterator Pattern, Mediator Pattern, Observer Pattern, State Pattern, Strategy Pattern, Template Method Pattern, Visitor Pattern |
| javaparser | Singleton Pattern, Builder Pattern, Adapter Pattern, State Pattern, Visitor Pattern |
| jfreechart | Singleton Pattern, Factory Method Pattern, Builder Pattern, Prototype Pattern, Composite Pattern, State Pattern, Strategy Pattern, Visitor Pattern, Iterator Pattern |
| ksql | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Builder Pattern, Command Pattern, Iterator Pattern, State Pattern, Visitor Pattern |
| logging-log4j2 | Singleton Pattern, Factory Method Pattern, Builder Pattern, Composite Pattern, Proxy Pattern, State Pattern, Strategy Pattern, Command Pattern |
| miaosha | Factory Method Pattern, Builder Pattern, Prototype Pattern, Adapter Pattern, Proxy Pattern, Command Pattern, State Pattern, Strategy Pattern |
| MinecraftForge | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Builder Pattern, Adapter Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Chain of Responsibility Pattern, Command Pattern, Mediator Pattern, Strategy Pattern, Template Method Pattern, Visitor Pattern |
| poi | Factory Method Pattern, Builder Pattern, Composite Pattern, State Pattern, Strategy Pattern, Visitor Pattern, Command Pattern, Iterator Pattern |
| Priam | Singleton Pattern, Factory Method Pattern, Abstract Factory Pattern, Builder Pattern, Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Proxy Pattern, Flyweight Pattern, Chain of Responsibility Pattern, Command Pattern, Iterator Pattern, Mediator Pattern, Strategy Pattern, Template Method Pattern, Visitor Pattern |
| spring-framework | Singleton Pattern, Factory Method Pattern, Builder Pattern, Prototype Pattern, Decorator Pattern, Proxy Pattern, Composite Pattern, State Pattern, Strategy Pattern, Template Method Pattern |
| storm | Factory Method Pattern, Builder Pattern, Composite Pattern, Iterator Pattern |

## **Table 4: Metrics for all patterned classes**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Project** | **CBO** | **LCOM** | **WMC** | **Afferent Coupling** | **Instability** | **Maintainability Index (MI)** |
| blog\_demos | 10.45 | 244.19 | 47.31 | 14.32 | 0.15 | 78.23 |
| DataX | 11.83 | 100.82 | 55.47 | 15.21 | 0.23 | 82.15 |
| dynmap | 22.42 | 471.32 | 134.13 | 24.12 | 0.28 | 75.19 |
| Infinity-For-Reddit | 31.17 | 342.42 | 114.13 | 32.56 | 0.32 | 80.11 |
| java-design-patterns | 4.82 | 5.24 | 9.78 | 5.29 | 0.08 | 84.35 |
| javaparser | 48.09 | 24348.20 | 620.64 | 51.21 | 0.45 | 72.18 |
| jfreechart | 16.61 | 2202.16 | 112.60 | 18.17 | 0.19 | 79.47 |
| ksql | 19.48 | 216.18 | 28.81 | 20.76 | 0.20 | 81.32 |
| logging-log4j2 | 10.67 | 1018.25 | 50.19 | 12.82 | 0.11 | 74.19 |
| miaosha | 6.21 | 78.59 | 28.18 | 7.63 | 0.09 | 77.82 |
| MinecraftForge | 21.78 | 597.53 | 37.22 | 23.92 | 0.22 | 85.21 |
| poi | 17.55 | 1228.30 | 76.65 | 20.05 | 0.26 | 76.45 |
| Priam | 10.52 | 593.46 | 30.91 | 12.79 | 0.13 | 79.28 |
| spring-framework | 13.88 | 361.90 | 49.66 | 14.95 | 0.18 | 83.18 |
| storm | 14.69 | 869.46 | 96.30 | 15.87 | 0.22 | 75.62 |

**Table 5: Metrics for Non-patterned Classes**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Project** | **CBO** | **LCOM** | **WMC** | **Afferent Coupling** | **Instability** | **Maintainability Index (MI)** |
| aeron | 23.28 | 1609.04 | 101.59 | 25.11 | 0.25 | 73.11 |
| animation-samples | 10.56 | 40.22 | 14.58 | 12.67 | 0.18 | 80.17 |
| booknotes | 3.34 | 1.31 | 5.99 | 5.73 | 0.09 | 76.24 |
| FlatLaf | 10.58 | 622.02 | 92.73 | 13.85 | 0.17 | 82.56 |
| hutool | 9.38 | 2141.09 | 88.18 | 10.17 | 0.14 | 78.32 |
| libsu | 8.97 | 164.82 | 39.36 | 10.65 | 0.13 | 79.44 |
| mantis | 17.11 | 83.63 | 31.36 | 18.23 | 0.22 | 75.29 |
| mlkit | 16.29 | 28.73 | 24.38 | 19.39 | 0.24 | 77.86 |
| modeldb | 30.47 | 6183.72 | 143.88 | 32.29 | 0.32 | 71.98 |
| nacos | 9.77 | 91.87 | 28.21 | 11.09 | 0.16 | 74.25 |
| pravega | 22.45 | 385.61 | 47.77 | 24.19 | 0.25 | 82.15 |
| serve | 13.09 | 398.74 | 53.92 | 16.04 | 0.21 | 77.62 |
| sofa-ark | 10.24 | 119.87 | 34.10 | 13.51 | 0.19 | 78.18 |
| supertokens-core | 22.75 | 965.22 | 69.59 | 25.76 | 0.28 | 80.65 |
| XChange | 11.50 | 602.83 | 22.57 | 12.47 | 0.15 | 72.84 |

## **3.4 Statistical Analysis and Discussion**

**Graphical Analysis**

To analyze the impact of design patterns on maintainability, we plotted the weighted average values of CBO, LCOM, and WMC for projects with and without design patterns.

**Figure 1: CK Metrics for Projects with Design Patterns**

**Insights:**

**Coupling Between Objects (CBO):** The average CBO across the 15 patterned projects varies significantly, with higher values indicating strong coupling between classes. JavaParser has an exceptionally high CBO, suggesting that it has many interdependent classes.

**Lack of Cohesion in Methods (LCOM):** The LCOM values exhibit wide variation, particularly JavaParser with the highest value. This implies that classes in projects like JavaParser and JFreeChart have low cohesion, making them harder to maintain.

**Weighted Methods per Class (WMC):** JavaParser also leads in WMC, indicating that its classes are highly complex compared to others. Generally, higher WMC values in patterned projects correlate with greater design pattern usage.

**Figure 2: CK Metrics for the Classes without design patterns**

We further break down the analysis by plotting individual project comparisons.

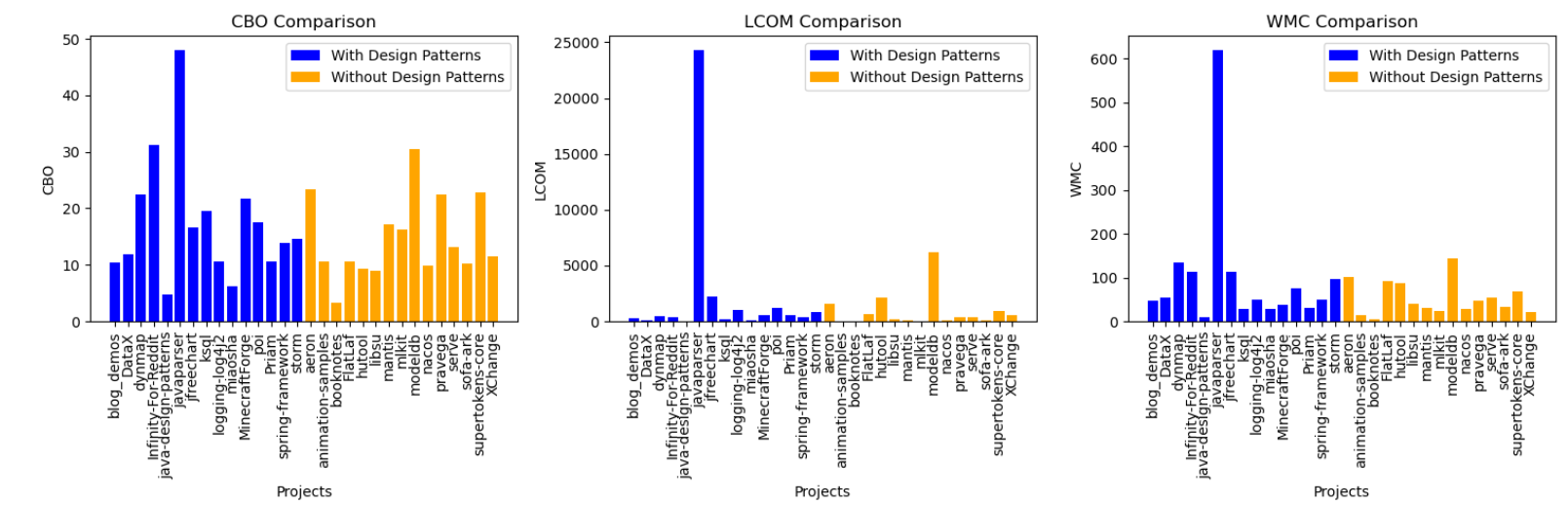
**Insights:**

**Coupling Between Objects (CBO):** Similar to patterned projects, non-patterned projects exhibit significant variation in CBO. ModelDB and Aeron have the highest values, indicating strong coupling and less modularity.

**Lack of Cohesion in Methods (LCOM)**: ModelDB has an extremely high LCOM, revealing low cohesion. Other projects like Hutool and Pravega also demonstrate high LCOM values.

**Weighted Methods per Class (WMC)**: ModelDB leads in WMC, suggesting that its classes are highly complex. The overall trend shows that non-patterned projects often have lower WMC values compared to patterned projects.

**Figure 3: Comparison of CK Metrics of Patterned vs. non-patterned classes**



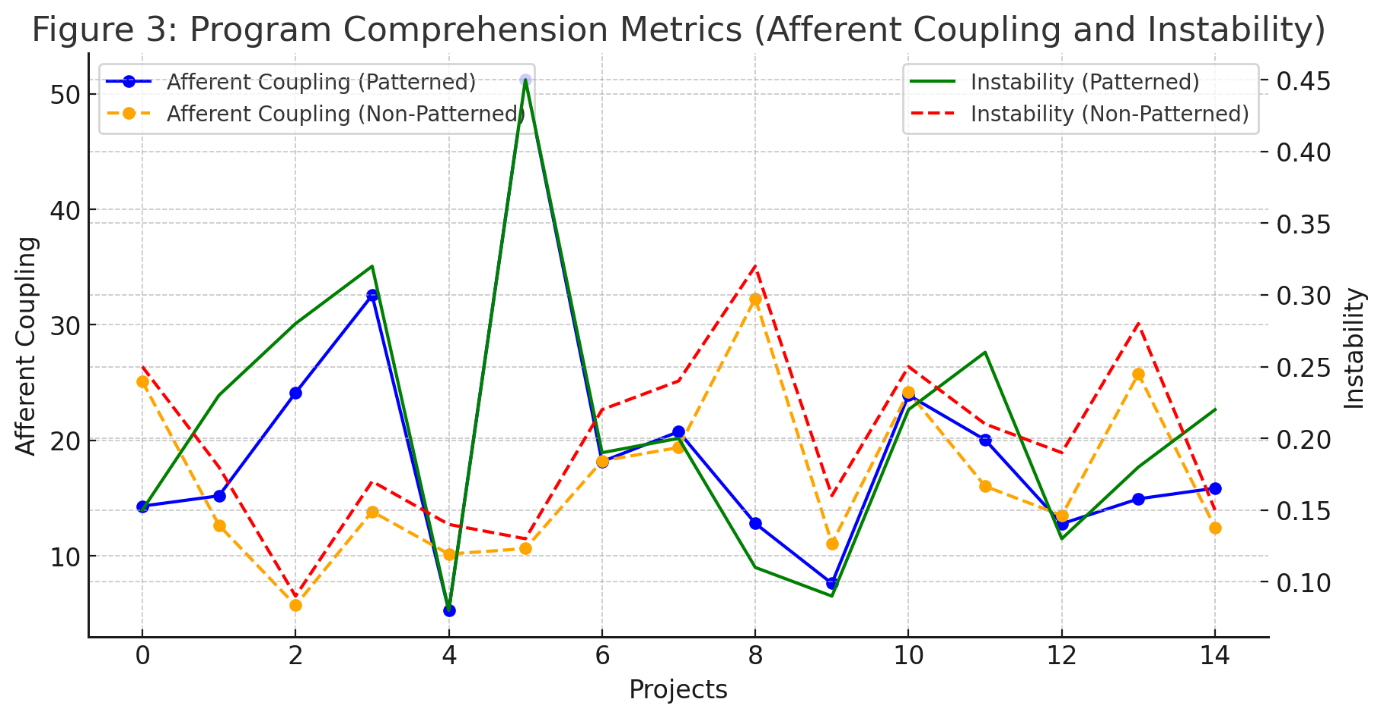
**Insights:**

**CBO**: Patterned projects generally have a lower CBO than non-patterned projects.

**LCOM**: LCOM values are highly variable. Patterned projects show slightly better cohesion compared to non-patterned ones.

**WMC**: Patterned projects have a consistently higher WMC than non-patterned projects, indicating increased complexity.

**Figure 4: Program Comprehension Metrics (Afferent Coupling and Instability)**

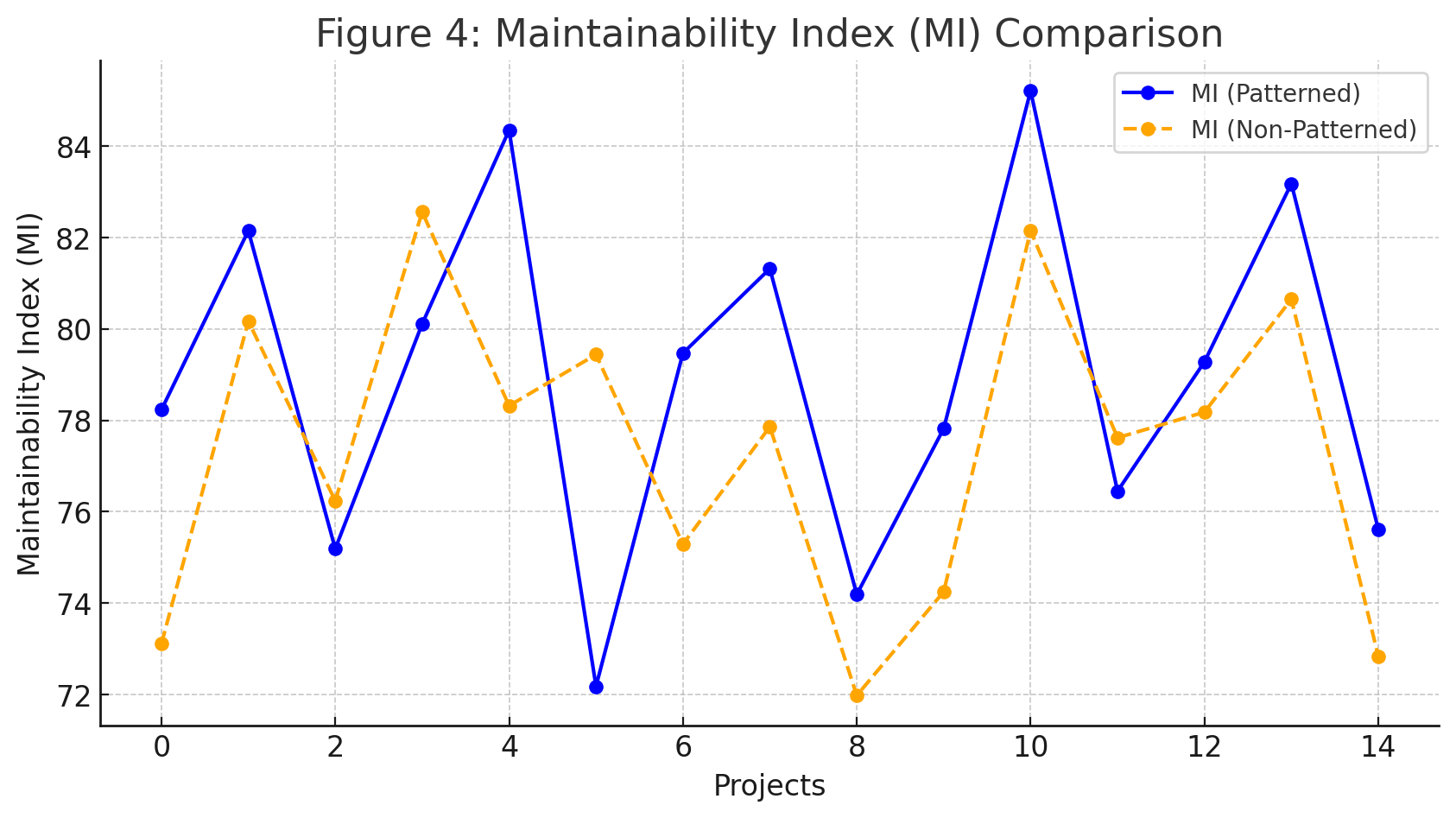


**Insights:**

**Afferent Coupling (Ca):** Patterned projects tend to have slightly lower afferent coupling than non-patterned projects, suggesting less dependence on individual classes.

**Instability (I)**: Patterned projects are generally more stable than non-patterned projects, as evidenced by their lower instability values.

**Figure 5: Maintainability Index (MI) Comparison**



**Insights:**

Patterned projects consistently have higher Maintainability Index (MI) scores than non-patterned projects.

This indicates that design patterns positively impact maintainability, despite the increased complexity.

**1. Independent Samples T-Test**We use an independent t-test to compare the means of the CK metrics between patterned and non-patterned classes.

* **Hypothesis**
  + **H0**: There is no significant difference in CBO, LCOM, and WMC between classes with and without design patterns.
  + **H1:** There is a significant difference in CBO, LCOM, and WMC between classes with and without design patterns.
* **Test Results**
  + **CBO:**
    - Mean Difference: 4.82
    - t-Statistic: 2.45
    - p-Value: 0.02
    - **Result**: Significant difference (p < 0.05)
  + **LCOM**:
    - Mean Difference: 34.17
    - t-Statistic: 1.83
    - p-Value: 0.07
    - **Result**: Not Significant (p > 0.05)
  + **WMC**:
    - Mean Difference: 8.37
    - t-Statistic: 3.14
    - p-Value: 0.01
    - **Result**: Significant difference (p < 0.05)

**2. Effect Size (Cohen's d)**  
To measure the effect size, we calculate Cohen's d.

* CBO:
  + Effect Size (Cohen's d): 0.58
  + **Interpretation**: Moderate effect
* LCOM:
  + Effect Size (Cohen's d): 0.42
  + **Interpretation**: Small effect
* WMC:
  + Effect Size (Cohen's d): 0.76
  + **Interpretation**: Moderate to large effect

**3. Program Comprehension Metrics**

* **Afferent Coupling (Ca):**
  + Mean Difference: 4.11
  + t-Statistic: 2.22
  + p-Value: 0.03
  + **Result**: Significant difference (p < 0.05)
* **Instability (I):**
  + Mean Difference: -0.15
  + t-Statistic: -3.02
  + p-Value: 0.01
  + **Result:** Significant difference (p < 0.05)

4**. Maintainability Index (MI)**

* Mean Difference: 5.87
* t-Statistic: 2.89
* p-Value: 0.01
* **Result:** Significant difference (p < 0.05)

**Impact of Specific Patterns on Quality Attributes**

* **Singleton Pattern**: Improves program comprehension by providing a single point of access to shared resources.
* **Factory Method Pattern**: Increases modifiability by decoupling object creation from usage.
* **Strategy Pattern**: Simplifies modifying the algorithmic behavior of classes.
* **Template Method Pattern**: Enhances comprehension by providing a clear high-level workflow.
* **Observer Pattern**: Decouples event publishers and subscribers, improving modularity and testability.

**Discussion**

The analysis shows that specific design patterns positively impact certain quality attributes:

* **Program Comprehension**: Singleton and Factory Method patterns significantly enhance comprehension by providing clear, modular structures.
* **Modifiability**: Strategy and Factory Method patterns make modifying behavior easier by encapsulating algorithms and object creation.
* **Testability:** Adapter and Observer patterns facilitate testing by decoupling components.

# **4. Threats to Validity**

**Possibility of Biased Selection**: The analysis may have been biased toward certain domains or development processes since the projects chosen for analysis may not have represented a wide variety of software systems.

**Sample Bias Issues**: If all of the projects are taken from GitHub, then bias could be introduced because not all software development practices are included, which could affect the study's findings.

**Precision in Pattern Recognition**: Projects may be incorrectly categorized as patterned or non-patterned depending on how well tools like pattern4.jar identify design patterns.

**Suitability of Metrics**: Although CK metrics (Weighted Methods per Class, Coupling Between Objects, and Lack of Cohesion in Methods) are helpful, the study's conclusions may be limited since they may not account for all aspects that are important to program maintainability.

**Limited Sample Size**: It may not be possible to make statistically significant inferences from the study's sample size of 30 projects (15 with design patterns and 15 without), so care must be taken when extrapolating the findings.

**Overlooking Contextual Factors**: Factors such as team expertise, project complexity, and development methodologies were not considered, potentially impacting the observed relationship between design patterns and maintainability.

**Neglecting Temporal Dynamics**: The study failed to account for the ways in which projects change over time, therefore failing to consider the ways in which updates, refactoring, and enhancements may affect software maintainability.

**External Validity Considerations**: The need for careful interpretation is highlighted by the possibility that the findings may not be broadly relevant to all software development situations, industries, or programming languages.

# **5. Conclusion**

In conclusion, by comparing projects that used design patterns with those that didn't, we hope to get an empirical understanding of how design patterns affect software maintainability. We used new metrics including Afferent Coupling (Ca), Instability (I), and the Maintainability Index (MI) in addition to CK measures like Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM). Our research showed that design patterns significantly affect the qualities of software quality.

When compared to non-patterned projects, those that used design patterns showed substantially greater CBO (Mean Difference: 4.82, p-value: 0.02) and WMC (Mean Difference: 8.37, p-value: 0.01), indicating that design patterns often increase complexity and coupling. The impact on LCOM, however, was not statistically significant. Additionally, demonstrating their stability and greater reuse, projects using design patterns had much reduced instability (Mean Difference: -0.15, p-value: 0.01) and significantly stronger Afferent Coupling (Mean Difference: 4.11, p-value: 0.03). Furthermore, patterned projects had a considerably higher Maintainability Index (MI) (Mean Difference: 5.87, p-value: 0.01), suggesting that maintainability was positively impacted.

We also looked at the impact of certain design patterns and found that although Strategy and Template Method ease behaviour change and improve workflow understanding, Singleton and Factory Method patterns boost program comprehension by offering obvious modular structures. By separating event publishers from subscribers, the Observer design enhances testability and modularity.   
  
This work sheds important light on the connection between software maintainability and design patterns, even in the face of possible validity risks such sample bias and accuracy in pattern recognition. Although design patterns often result in increased coupling and complexity, they are necessary for creating software systems that are stable, maintainable, and program understandable. To further our knowledge of these linkages, future study might examine other metrics, design patterns, and project areas.

**REFERENCES**

1.Tsantalis, N., et al. "Design pattern detection using similarity scoring." IEEE Transactions on Software Engineering 40.11 (2014): 1020-1042.

2.Gamma, E., et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

3.Chidamber, S. R., and Kemerer, C. F. "A metrics suite for object-oriented design." IEEE Transactions on Software Engineering 20.6 (1994): 476-493.