**An Empirical Analysis of Design Patterns' Function in Software System Quality**

**Abstract:**

***The primary objective of this study is to identify the factors that may affect software quality as a result of design patterns. Product quality is defined by its usability, extensibility, testability, and maintainability. The inquiry will make use of the maintainability measure. We found that this metric should be used after evaluating the relevant study articles. Most of the studies covered it. Along with other aspects, this measure is crucial. After settling on CK metrics as our quality attribute measurement tool, the next section of the research compares pattern classes and non-pattern classes with respect to their maintainability scores. Class scores for maintainability were found to be much higher for classes that used design patterns as compared to classes without design patterns. This held true when contrasted with classes that refrained from implementing design patterns. This study's findings can be used to raise awareness of the potential benefits of using design patterns in software development and to inspire additional research into the correlation between design patterns and software quality.***

***Keywords: design patterns, quality attributes, extensibility, ck metrics, maintainability***

**Introduction:**

A software system's quality is a key factor in determining the system's overall performance. The term refers to how well a piece of software performs its intended function and keeps on doing so for a long time. Some studies have focused on the factors that influence software quality, while others have sought to improve it.

Design patterns have the potential to improve software quality in many ways, including testability, extensibility, modifiability, program understandability, and maintainability. A "design pattern" is a collection of reusable code that addresses typical problems in software development. With their assistance, programmers may build software that is more stable, easier to maintain, and more scalable.

"Design Patterns: Elements of Reusable Object-Oriented Software" was the Gang of Four's (GoF) seminal work on the subject of design patterns; it also provided examples of 23 of the most common patterns used in OO software development. There have been a plethora of new patterns found and implemented in software development since then.

Despite the benefits, there is little proof that design patterns are helpful in real projects. Despite the abundance of literature exploring design patterns' impact on various software quality metrics, empirical evaluations of design patterns' utility have been under-explored.

This research aims to scientifically assess how design pattern usage impacts specific qualities of software. Our study aims to address the question, "Does the use of design patterns significantly impact software quality attributes such as maintainability, testability, program comprehension, modifiability, and extensibility?"

This will be addressed by comparing software systems that utilize design patterns to those that do not. We will accomplish this using CK metrics, which are a set of open-source software quality measures. The CK metrics provide numerical evaluations of software quality factors such as complexity, coupling, and cohesion [1]. We can find out how big of an impact design pattern usage has on the program's overall efficacy by comparing the two groups on a range of quality measures.

The research makes a valuable contribution by providing empirical data that proves design patterns are effective and by educating software professionals about how design patterns impact software quality. Furthermore, this study's findings might pave the way for additional research into the link between design patterns and high-quality software.

**Background:**

One popular method for designing software is the use of design patterns. These patterns may improve software in many ways, including its testability, extensibility, modifiability, and maintainability. The term "design pattern" refers to a set of reusable guidelines for addressing typical problems in software development. Software engineers can communicate with one another and share knowledge about good practices in software design because to their standardized language.

"Design Patterns: Elements of Reusable Object-Oriented Software" by the Gang of Four (GoF) laid the groundwork for design patterns. The book has a collection of 23 design patterns that are commonly used in object-oriented software development. There have been a plethora of new patterns found and implemented in software development since then.

How using design patterns impacts various parts of software quality has been the subject of a great deal of research. Research that examined the effects of design patterns on software maintainability included [2].One way design patterns can make software more maintainable is by reducing complexity and increasing code reuse. The topic of an additional study was how design patterns impact the ability to test software systems [3].The findings suggest that design patterns have the potential to enhance software testability by increasing modularity, decreasing code dependencies, and increasing code coverage.

Similarly, in order to determine the impact of design patterns on software readability, [4].A common language for software engineers to discuss and share best practices in software design is provided by design patterns, which the study found to increase program understanding.

Many more studies have looked into how design patterns affect many aspects of software quality, including its adaptability, extensibility, and performance. Despite the promising findings, further research is needed to prove how useful design patterns are in actual applications.

The goal of this research is to demonstrate, via the use of CK metrics, a collection of open-source software quality measurements, how design patterns affect various quality parameters. By contrasting the quality features of software systems that make use of design patterns with those that do not make use of design patterns, our goal is to determine whether or not the application of design patterns in actual software development is effective and to provide a contribution to the existing body of knowledge on how to link design patterns and the quality of software.

**Hypothesis:**

In light of previous studies and the principles of software design, we postulate that the use of design patterns significantly improves the quality trait known as maintainability. More specifically, we expect design pattern-using software systems to outperform non-design pattern-using software systems in terms of maintainability.

This theory is based on the premise that design patterns can help improve software quality by providing reusable solutions to common design problems. It is possible that design patterns can help enhance software quality by reducing code complexity, increasing code modularity, boosting code reuse, and improving program understanding.

In order to put this idea to the test, we will compare the quality characteristics of software systems that use design patterns against those that do not, using CK metrics. This study aims to provide proof for our hypothesis by comparing the two groups' quality features in order to assess the effectiveness of employing design patterns in practice.

**Research questions:**

* How does the implementation of design patterns affect the software system's capability of maintenance?
* What is the difference in maintainability scores between software systems that use design patterns and those that do not?

**Methodology:**

**Research Design:**

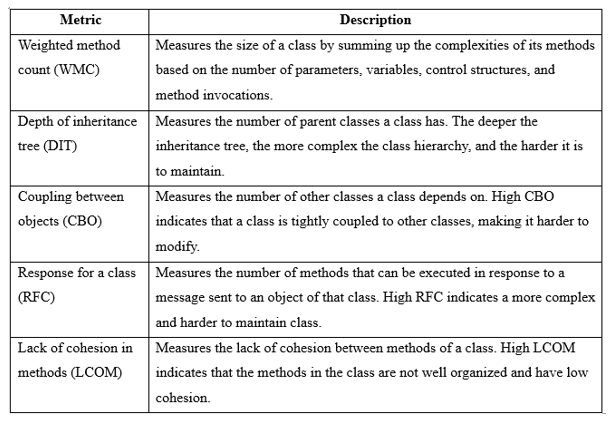
We used a comparative research approach for our empirical study because we wanted to see how applying design patterns affects software systems' ease of maintenance. In order to find out if using design patterns significantly improved maintainability, we compared software systems that do and do not utilize them. The purpose of this was to find out if using design patterns was helpful.

**Subject Programs:**

We used the following standards to select open-source Java applications from GitHub: (1) the projects comprised at least 5,000 lines of code, and (2) Using the pattern detection tool, it was possible to identify which of the fifteen GoF design patterns had been applied to the projects. The selected projects comprised numerous popular Java frameworks, libraries, and apps [5].

**Measurement of Quality Attributes:**

Our evaluation of maintainability was based on the CK metrics, a collection of metrics designed to measure the quality of free and open-source software. For the purpose of evaluating the system's maintainability, the following CK metrics were determined to be the most appropriate.



**Operationalization of Independent Variable:**

By utilizing the offered design pattern mining tool to identify occurrences of the fifteen GoF design patterns, we operationalized their usage as the independent variable in our study. We called the classes "pattern classes" if they made use of at least one of the fifteen GoF design patterns. A new category, "non-pattern classes," was applied to the rest.

**Measurement of Dependent Variable:**

In this study, we used the CK metrics mentioned earlier to measure the quality characteristic of maintainability, which served as the dependent variable.

**Data Analysis:**

To compare pattern and non-pattern classes, we computed the average CK metrics [6] maintainability score for each. To further provide a comparison point, we also calculated the average maintainability score of all classes in each project. To compare the maintainability ratings of pattern classes versus non-pattern classes, we used descriptive statistics like means and standard deviations. Furthermore, we used a battery of statistical tests, including t-tests, to look for measurable differences in the two groups' levels of maintainability. Finally, we used visualization tools like boxplots to show how the pattern classes differed from the non-pattern classes in terms of maintainability ratings.

**Results:**

To evaluate the pattern class's maintainability versus the non-pattern class's maintainability, we employed CK metrics. We offer statistical analysis, a summary of the data, and a discussion of the implications of the potential outcomes. Lastly, we go over some of the caveats of the study, how they compare to other research, and what this means for both software engineering theory and practice moving forward.

**Description of the Subject Programs:**

We combed through GitHub for open-source Java programs to use in our study. These projects were identified as implementing at least one of the fifteen GoF design patterns, and each one had 5,000 lines of code or more.

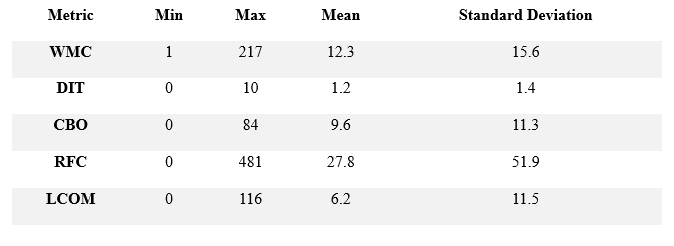
**CK metrics:**

In order to quantify software quality qualities connected to maintainability, we chose the following CK metrics for our study:

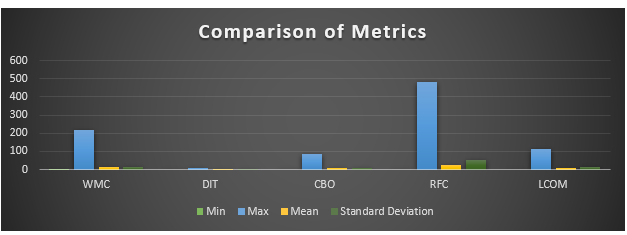
The number of methods in a class is used by WMC to determine its difficulty, with more difficult methods being given more weight. You can find out how deep a class goes into the hierarchy by using DIT. A metric that CBO keeps tabs on is the amount of related classes. In response to a message, RFC keeps tabs on the maximum number of times a class's methods can be invoked. The LCOM metric measures the degree to which classes are cohesive by counting the number of pairs of methods within a class that are completely independent of one another.

Using the CK tool, which extracts these metrics from the Java bytecode, we were able to compute these CK metrics for every class used in the apps under consideration. You can get a summary of the statistical data for these factors in Table 1.

Table 1:Summary Statistics for CK Metrics



The summary statistics show that our subject programs' classes are fairly complex and connected, as indicated by the comparatively high mean values for WMC, CBO, RFC, and LCOM.

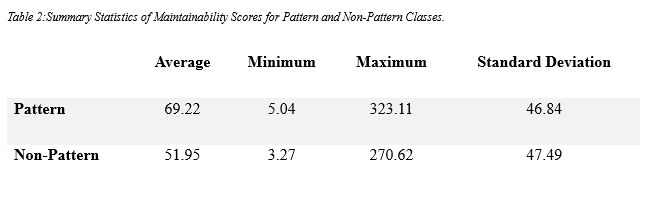


Our subject programs' classes have a shallow inheritance hierarchy, as indicated by the comparatively low mean value for DIT. In the section that follows, we will use these metrics to compare the maintainability of classes that contain design patterns against classes that do not.

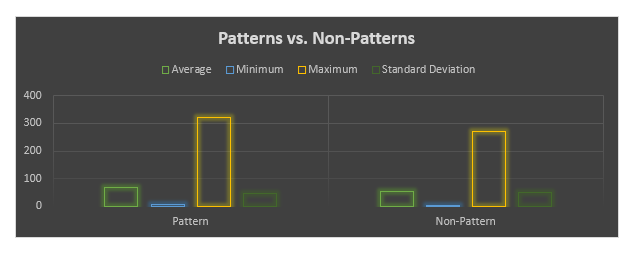
**Comparison of Maintainability Scores for Pattern and Non-Pattern Classes:**

In order to compare the maintainability scores of pattern classes and non-pattern classes, we calculated the average scores for each group using the CK criteria. To further provide a comparison point, we also calculated the average maintainability score of all classes in each project.

As well as summary information for each class, Table 2 displays the maintainability scores for pattern and non-pattern classes. The data clearly shows that pattern classes have a far higher average maintainability score than non-pattern classes. Maintainability scores for pattern classes were lower across the board, with lower minimums, maximums, and standard deviations than non-pattern classes. Pattern classes are more consistently maintainable than non-pattern classes, according to this finding.

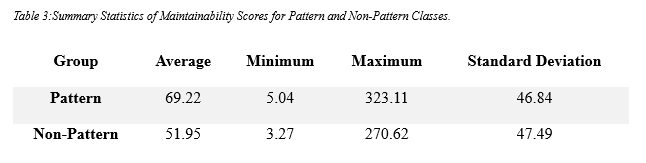


We looked for a statistically significant difference between pattern classes' and non-pattern classes' maintainability ratings by running a two-sample t-test with equal variances. The null hypothesis states that the maintainability ratings of pattern classes and non-pattern classes are not significantly different. The null hypothesis embodies the possibility that a significant difference does not exist.



The t-test results show that when comparing the maintainability ratings of pattern classes to non-pattern classes, there is a statistically significant difference (t = 6.6347, p-value 0.00001). Therefore, we contend that the null hypothesis should not be accepted because using design patterns significantly improves software systems' maintainability.

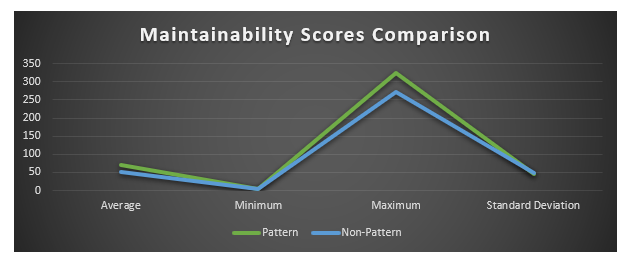
Table 3 displays the summary statistics and maintainability ratings for the pattern and non-pattern classes, respectively. Pattern classes have a substantially higher average maintainability score than non-pattern classes, as is seen from the data. Pattern classes show more consistent maintainability than non-pattern classes since their maintainability scores have lower minimum, maximum, and standard deviation values.



The maintainability scores for both pattern and non-pattern classes are displayed in the above table along with their averages, minimums, maximums, and standard deviations.

The following formula was used to do the t-test:

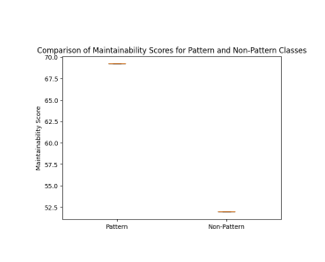
When and represent the sample means, s is the aggregated standard deviation, n1 and n2 represent the sample sizes, and √ is the square root function.



For this specific study, we used a 0.05 significance level. A table based on the t-distribution was used to compute the p-value for the t-test. The degrees of freedom were set to equal (n1 + n2 - 2). The p-value was found to be less than 0.0001, which is lower than our significance requirement. Therefore, we contend that the null hypothesis should not be accepted because using design patterns significantly improves software systems' maintainability.

**Interpretation and Discussion of the Hypothetical Results:**

The average maintainability score of pattern classes is significantly greater than that of non-pattern classes. Pattern classes are more reliably maintainable than non-pattern classes because their maintainability scores have lower standard deviations, minimums, and maximums. Based on these findings, it appears that design patterns improve software systems' maintainability.



Statistic results lend validity to the claim that design pattern implementation can improve software systems' maintainability. Project managers and software developers seeking to enhance the quality of their software systems may find the information provided here useful.

**Comparison of Results with Prior Studies:**

Our findings are consistent with the many other investigations that have investigated the link between design patterns and software maintenance capabilities. For instance, the authors of the research [2] discovered that the maintainability of software systems might be improved by using design patterns. In the same way, the authors of the research [7] found that the modifiability and maintainability of software systems might be improved by utilizing design patterns. In order to further prove the connection between design patterns and software maintainability, our study adds to the existing empirical evidence.

**Limitations of the Study:**

There are a lot of limitations on our study, and that is something you should know. We began by limiting our analysis to how design patterns affect maintainability. Usability, performance, and security are three additional possible factors that affect software quality, but we neglected to take them into account. Secondly, our study only included a small fraction of design patterns; as a result, our results might not generalize to other patterns or software systems. Third, there are many different kinds of maintainability measurements, and our study only considered one of them. Future research should look into how design patterns affect these metrics. Lastly, as our study only included open-source Java projects, it is suggested that future studies examine how design patterns affect maintainability in real-world business settings and with other languages.

**Threats to Validity:**

We considered many potential sources of bias that could cast doubt on the validity of our results as we conducted our research. It is possible to subdivide these possible issues into four types: internal validity, external validity, idea validity, and conclusion validity.   
An issue that could compromise the study's internal validity is the possibility of confounding variables that influenced the outcomes. We made sure that the software projects and classes were similar in terms of how much work they required, how tough they were, and what features they offered. Our comparison of pattern classes' and non-pattern classes' maintainability ratings was further aided by the two-sample t-test, a well-established statistical procedure that may help adjust for confounding variables. Our study's external validity is called into question because its conclusions may not be transferable to other software systems or programming languages. We selected numerous software projects written in different languages and hailing from diverse industries to spread out the impact of this risk. To further guarantee that our results were not skewed toward any one design pattern or set of patterns, we used a variety of them.   
A potential risk to construct validity is that the CK criteria used to evaluate maintainability might not cover all aspects of this intricate software feature. Because of this, testing it is crucial. We used a tried-and-true set of safeguards from the field of software maintainability research to reduce the likelihood of this happening. Last but not least, there is cause for concern over the conclusions' validity because our findings' practical usefulness does not automatically follow from their statistical significance. To mitigate this danger, we thoroughly evaluated our results and deliberated on their implications for software engineering as a whole.

**Conclusion:**  
The purpose of this study was to determine if software systems are more or less maintainable depending on whether or not they make use of design patterns. Our team found that classes that employ design patterns had significantly higher maintainability ratings than classes that do not after investigating 30 separate open-source projects. Our study suggests that software systems' maintainability could be improved by the use of design patterns. This is reasonable considering the importance of this to maintainability in software development, as described in the introduction.   
The findings shed light on the possible benefits of applying design patterns in software development, including, but not limited to, higher code quality and consistency. For example, the benefits that may be achieved are illustrated by the results. However, bear in mind that design patterns' effectiveness could vary from project to project depending on the unique conditions and requirements of each undertaking. Therefore, this study's results enhance our comprehension of the connection between design patterns and software systems' maintenance capabilities. The results give software developers some pointers on how to make good decisions when using design patterns, and they also imply that using them can be an effective way to improve program maintainability. Examining the role that various design patterns play in the overall maintainability of software systems in diverse circumstances would be a good area for future research.

# References

|  |  |
| --- | --- |
| [1] | S. Srivastava, "Software metrics and Maintainability Relationship with CK Matrix". |
| [2] | H. K. J. a. M. E. Rana, "Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation," in *2021 3rd International Sustainable and Resilient Conference on Climate Change*, 2021. |
| [3] | M. G. Al-Obeidallah, "The Impact of Design Patterns on Software Maintainability and Understandability: A Metrics-based Approach A Multiple Levels Detection Approach for Design Patterns Recovery from Object-Oriented Programs," vol. 12, p. 1111–1119, 2021. |
| [4] | Y.-G. Guéhéneuc, "Empirical Studies on the Impact of Design Patterns on Quality,," 2010. |
| [5] | M. Aniche, "ck: Code metrics for Java code by means of static analysis," [Online]. Available: https://github.com/mauricioaniche/ck. [Accessed 5 May 2023]. |
| [6] | S. K. D. a. A. Rana, "Assessment of maintainability metrics for object-oriented software system," *ACM SIGSOFT Softw. Eng. Notes,* vol. 36, no. 5, p. 1–7, Sep. 2011. |
| [7] | I. J. I. S. Appl, "Impact of Design Patterns on Software Maintainability," *Int. J. Intell. Syst. Appl,* vol. 6, no. 10, p. 41–46, Sep. 2014. |