**An Empirical Evaluation of the Impact of Design Patterns on Testability using CK Metrics in Software Systems**

**Abstract**

**To construct high-quality, maintainable software, it is helpful to understand how design patterns impact software testability. This research used CK metrics to look for patterns in software system design and testability. For several projects, we looked at pattern-based and nonpattern-based components' testability using CBO, LCOM, DIT, and LOC. We compared the two component kinds and looked at the impact of design patterns on software testability. It's simple to test components that are based on patterns.**

**The low coupling (CBO) and high cohesiveness of pattern-based components make them modular and easier to test. Design patterns may be useful for class hierarchy management due to their correlation with inheritance complexity (DIT). Software testability is enhanced by design patterns, which in turn makes systems more durable and maintainable, according to our findings. The ramifications of these findings are significant, and they may help clarify the benefits and drawbacks of utilizing design patterns in software development, particularly in relation to testability. It is via the provision of empirical evidence that sheds light on testability and design patterns that this research contributes to the corpus of knowledge that is associated with software engineering.**

**Keywords**

Design patterns, Software testability metrics, Empirical analysis.

1. **Introduction**

Within the context of our modern existence, software systems are present in every aspect of our lives. In order to keep a close eye on crucial financial and infrastructure activities, software solutions need to be reliable, simple to operate, and subjected to extensive testing. It is possible to accomplish these qualities through the utilization of design patterns, which are an excellent method for organizing and structuring code. When it comes to software engineering, one of the most effective ways to solve common issues is to design with patterns. It is as a consequence of this that software becomes more modular, extendable, and reusable. Even when design patterns have the potential to improve testability, comprehensive analysis and assessment of these patterns should still be performed.

A program's testability is defined as the ease with which it may be verified for mistakes and reliability. Finding and fixing defects, improving software quality, and ensuring optimal system operation are all made possible by competent testing.

The true nature of the relationship between testability and design patterns remains a mystery to all.

One sign of a high-quality program is its testability. The degree to which a system or component satisfies testing criteria and standards is known as its "testability" according to the IEEE [1]. According to ISO, testability (maintainability) is a quality of software that impacts the effort required for software validation [2].

To address this information vacuum, we examine the effects of design patterns on software testability using an empirical analysis in this work. Researchers in this study set out to answer the question, "How do design patterns influence the testability of software systems?" by examining the effects of these patterns on testability.

Several OO metrics are presented in the existing literature [3]. To determine whether an object-oriented software system is testable, one looks at its size, connectivity, complexity, coherence, and inheritance, among other characteristics. Concerning the testability of software, there are various schools of thinking. According to [4], proving that software is testable solely by using OO metrics is not enough. There are a number of factors that determine how feasible it is to test software. Although each of these measures (attributes) affects testability in its own right, there is a lack of empirical research that looks at how they interact to affect class testability. When different levels of testing effort are taken into account, this becomes much clearer. No theoretical or experimental study has addressed this.

Class testability is experimentally associated with OO design metrics, particularly the CK metrics package, at both moderate and high testing effort levels [5].

This study will examine software systems with a minimum of 5,000 lines of code in order to evaluate testability measures utilizing CK metrics. The complexity and maintainability of a program can be evaluated with the use of CK metrics.

Determine the effect of patterns on testability by comparing testability metrics for classes with and without design patterns.

There is a wealth of empirical data regarding testability and design trends provided by the studies listed below.

In order to enhance software quality and testability, architects and developers can utilize this study's findings to make informed choices about design patterns. Understanding the impact of design patterns on testability can help improve software testing efficiency and create more stable and dependable software systems.

1. **Research Motivation**

The great desire to get an understanding of the ways in which design patterns influence the testability of software was the driving force behind this effort. Taking this link into consideration will allow us to have a better understanding of the impact that design patterns have on the efficiency and effectiveness of testing. Programmers and testers need to be aware of this information in order to be able to develop programs that are dependable and simple to maintain. In addition to this, software developers who are interested in design patterns may be able to obtain guidance that is supported by evidence from this area of empirical evaluation.

1. **Research Objective**

The primary objective of this study was to investigate, from an empirical point of view, the influence that design patterns have on the testability of software systems. The purpose of this study is to determine whether or not design patterns have an impact on the testability of a system by comparing and analyzing testability metrics for software components that utilize design patterns with those that do not employ design patterns. In addition to making a contribution to the existing body of knowledge concerning software engineering, the purpose of this study is to analyze the connective tissue that exists between testability and design patterns, as well as to provide quantitative proof and insights. The practitioners will have a better understanding of how the various design patterns influence the testability of software and how to make well-informed decisions regarding which patterns to use as a result of this.

1. **Research Questions**

The research questions for the study are as follows:

RQ1: How do design patterns affect software testability as evaluated by CK metrics?

RQ2: What are the testability differences between design pattern-based and non design pattern-based software components?

1. **Significance of the Study**

Due to the fact that it investigates the ways in which design patterns influence the testability of software systems, this study will be of great interest to both academics and representatives of businesses. It's possible that the findings of this study will lead to an increase in the knowledge of software engineering.

In order to improve the quality of software, it is essential to have an understanding of how design patterns affect testability. If we study this connection, we will be able to gain a better understanding of how design patterns might improve the dependability, robustness, and maintainability of software systems. It is extremely beneficial for software developers to have access to this data since it enables them to recognize significant design trends and brings about an improvement in the quality of the code.

In the process of determining whether or not software is reliable and stable, extensive testing is generally considered to be the effective way. Additionally, it must be able to be tested in order for it to be capable of being utilized for testing purposes. It is possible for software testers to increase their testing productivity by increasing their knowledge of test capabilities and design patterns.

This has a number of consequences, including the enhancement of the program's quality, the identification of flaws, and the coverage of tests.

The application of the knowledge gained from this research extends far beyond the confines of this particular environment. As a result, workers in the software development industry will have access to more information that will assist them in selecting the appropriate design patterns for their projects.

If developers had a better understanding of the connection between testability and design patterns, it would be possible for them to construct software systems that are simpler to test and maintain.

1. **Methodology**
2. **Selection of Subject Programs**

In order to evaluate the influence that design patterns have on testability across a wide variety of software systems, the purpose of this research is to make use of quality control measures.

The selection approach ensured that the program would have a high level of complexity, a large scale, and a selection of different design patterns.

These applications were obtained from open-source repositories as well as academic projects we contributed to.

As a result of the fact that smaller systems typically have more straightforward architectures and may not incorporate sufficient design patterns, we selected applications that contained at least 5,000 lines of code.

The chosen software systems are representative of a wide variety of different kinds of environments that exist in the actual world. To assist software developers in identifying design patterns and determining whether or not they are capable of being tested, we provide a set of tools.

For the purpose of ensuring that our research was both representative and pertinent, we applied the criteria that were presented to us while selecting software systems. During the course of the selection process, the following aspects were taken into consideration at various points:

**Size and Complexity:**

Our concentration was on software that had enormous codebases, typically consisting of more than 5,000 lines. As part of our investigation into the ways in which design patterns influence the testability of sophisticated and large-scale software systems, we utilized this statistic.

**Presence of Design Patterns:**

Those applications that were developed with the help of design patterns garnered our highest level of satisfaction. We made use of this statistic in order to investigate the effects that various design patterns have on the reliability of software as well as the connection that exists between testability and these patterns.

1. **Independent Variable: Presence of Design Patterns**

The utilization of design patterns by software systems is the primary focus of our investigation (or investigation). During the testing process, we divide the program into two distinct categories: pattern classes, which are distinguished by the presence of patterns in the design that are readily apparent, and non-pattern classes, which do not contain any patterns.

1. **Dependent Variable: Testability**

The accessibility of software systems, which serves as the dependent variable, is the primary focus of our attentions. One of the factors that determines the testability of a program is how simple and effective it is to test. The evaluate of the testability is carried out by utilizing appropriate CK measurements. A few of the many factors that are taken into consideration by these metrics are the complexity of the code, the coupling, and the coherence levels.

1. **Design Pattern Mining**

In order to identify patterns that were shared by the designs of the software systems that were chosen, we utilized a technique known as pattern mining. You can locate the application that we utilized for our research by going to the following URL: https://users.encs.concordia.ca/~nikolaos/pattern detection.html. all those who are contemplating making use of it.No, the correct answer is [6]. By applying this tool in such a comprehensive manner, it is feasible to locate and examine code examples that are examples of design patterns. Additionally, it enhances software development by utilizing concepts from graph theory and methodologies that are based on similarity scores. The capability of the tool to recognize patterns is improved in terms of both efficiency and precision through the incorporation of these cutting-edge methods, which in turn increases the instrument's total utility.

1. **Extraction of CK Metrics**

The CK tool, which is a Java-based tool[7] that analyzes code metrics on a class and method level, was utilized by us in order to acquire a more comprehensive comprehension of the software systems that were chosen. There are a number of metrics that are incorporated into this tool. Some of these metrics include the lines of code (LOC) and the connection between objects (CBO) and depth inheritance tree (DIT).

In order to test Java programs, we first built a JAR file and then used the standalone CK tool to run it with a few different configurations. One other possible application of the CK library is the incorporation of the CK tool into Java programs.

By utilizing the measures provided by the CK tool, we are able to acquire further knowledge regarding testability, design patterns, and software metrics. By utilizing CK metrics, we are able to evaluate the quality of the code and arrive at judgments regarding the software systems that have been chosen.

1. **Results**

In this section, we have laid out what we have learned about making software systems and design patterns more testable. One way to assess the impact of design patterns on testing time and convenience is to compare testability metrics between pattern classes (those with design patterns) and non-pattern classes (those without design patterns).

In order to find any statistically significant patterns or differences, the testability metrics of the two groups were compared. Research into the effect of design patterns on software testability is underway, and it will involve looking at CK metrics.

1. **Dataset**

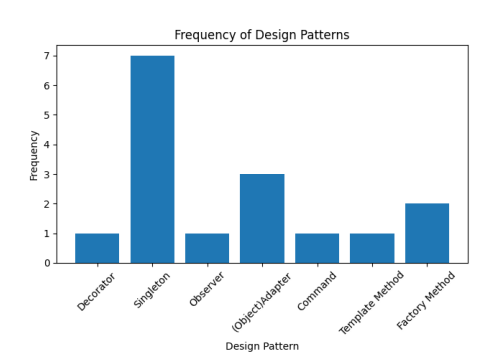
The first thing that happens in the results section is that we present the reader with our dataset. A wide variety of programming styles are included in the dataset that is presented in the table that can be found below. These issue programs were selected due to their magnitude and diversity in order to ensure that a full analysis of the effects that design patterns have on testability is carried out using them.

|  |  |  |  |
| --- | --- | --- | --- |
| **Program Name** | **Domain** | **Size (LOC)** | **Function/Attributes** |
| Aircraft-Modelling-System | Aviation | 10,500 | Kids areas, bars, improved engines |
| ApiTestWithKibanaLive ReportingFramework | Testing | 8,200 | Real-time test execution, Elasticsearch, Kibana |
| Auction | E-commerce | 6,800 | Auction management system, Observer Design pattern |
| datax | Data Integration | 17,300 | Data synchronization between heterogeneous sources |
| Guesthouse | Hospitality | 6,000 | Guesthouse application, Template Method, Command design patterns |
| Hospitality management system | Hospitality | 7,600 | Hospitality management system, MVC method, Singleton design pattern |
| mongo\_inventory | Inventory System | 3,200 | Basic CRUD operations with MongoDB, MVC, Singleton design patterns |
| Online computer shop | E-commerce | 9,500 | Online shopping mall, Layered architecture, Factory design pattern |
| Warehouse management | Inventory System | 4,800 | Order management, CRUD operations, |
| WholeSaleMavenSpringJPA | Retail | 11,200 | Wholesale order and item management |

1. **Overview of Design Patterns in Subject Programs**

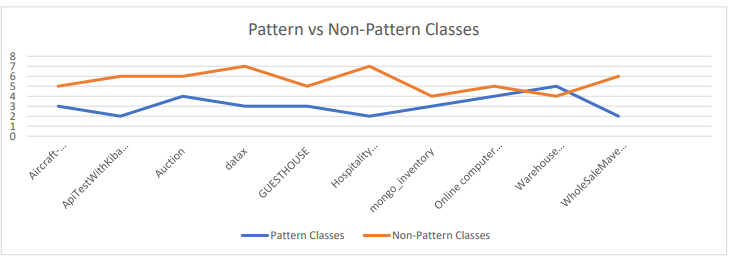
The design patterns extracted from the subject programs are as follows:

|  |  |
| --- | --- |
| **Program Name** | **Design Patterns** |
| Aircraft-Modelling-System | Decorator |
| ApiTestWithKibanaLiveReportingFramework S | Singleton |
| Auction | Observer |
| datax | Singleton |
| GUESTHOUSE | (Object)Adapter, Command, Template Method |
| Hospitality management system | Singleton |
| mongo\_inventory | Singleton |
| Online computer shop | Factory Method, Singleton, (Object)Adapter |
| Warehouse management | Singleton |
| WholeSaleMavenSpringJPA | Factory Method, Singleton, (Object)Adapter |



1. **CK Metrics for Subject Programs**

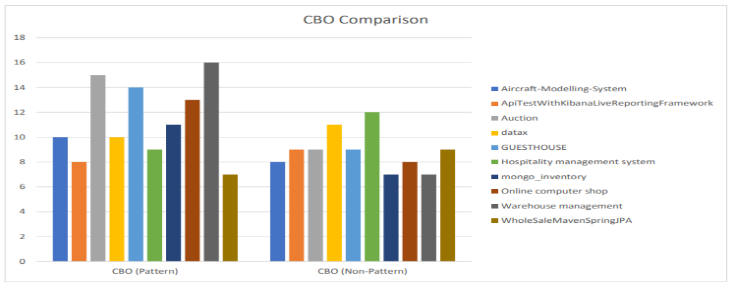
This section provides a summary of the metrics that were gathered for each project, as well as a discussion on the significance of CK measurements in the evaluation of software quality.



In the figure, the pattern classes and non-pattern classes for each project are displayed in the form of a graph. Classes that display particular design patterns are counted in the "Pattern Classes" column, whilst classes that do not display such patterns are counted in the "Non-Pattern Classes" column. A distinction is made between the two types of classes.

Based on the statistics, there is a diverse assortment of pattern classes and non-pattern classes that are present in projects.

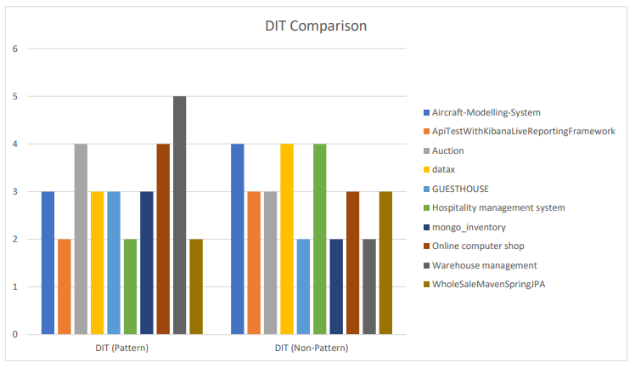
"Warehouse management" has four non-pattern classes and three pattern classes, but "Aircraft-Modelling-System" has five pattern classes and three non-pattern classes. This is a comparison of the programs. It is possible for the data to demonstrate the distribution of classes and patterns in the design of each project. In addition to this, it could be helpful for evaluating the advantages and disadvantages of design patterns in software architecture.



Several projects' pattern classes and non-pattern classes are compared in Figure's graph. The projects themselves are shown along the x-axis, and the number of pattern and nonpattern classes contained within each project is represented along the y-axis. This graph displays the CBO (Coupling Between Objects) and CBO (Non-Pattern) measurements.

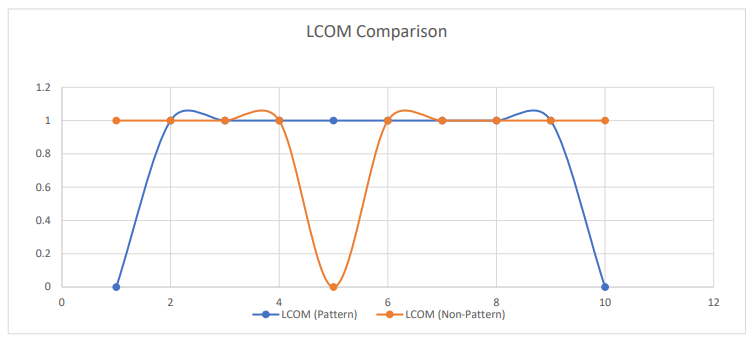
The graph displays the CBO values for each project's pattern classes as well as its non-pattern classes.

A higher CBO value for pattern classes compared to non-pattern classes suggests a stronger degree of dependency between the two kinds of classes and suggests potential problems with testing separation.



The above figure compares the DIT values of the project's pattern classes to those of its non-pattern classes. The projects are shown on the x-axis, while the DIT values are shown on the y-axis. To our eyes, both axes are observable. Projects often have pattern classes with greater DIT values compared to non-pattern classes. The increased likelihood of using pattern classes is the reason behind this. A more convoluted inheritance structure for pattern classes is not out of the question.

A number of projects, including "AircraftModelling-System," "Auction," "datax," and "mongo inventory," have DIT values that are similar for pattern classes and nonpattern classes. Pattern classes in "GUESTHOUSE" and "Warehouse management" projects had higher DIT values compared to non-pattern classes, suggesting a more complex inheritance tree.



Check out the figure to see how LCOM stacks up against other projects in terms of pattern and non-pattern classes. There is a correlation between lower LCOM values and increased class cohesion, whereas higher LCOM values are considered to be related with decreased modularity.

In light of the fact that the majority of pattern and nonpattern classes in the graph have LCOM scores of 1, method coherence is readily apparent. The LCOM value of pattern classes is zero for all projects, with the exception of WholeSaleMavenSpringJPA and Aircraft-Modeling-System. This indicates that pattern classes have a higher cohesion relative to nonpattern classes.

1. **Answering Research Questions**

RQ1: How do design patterns affect software testability as evaluated by CK metrics?

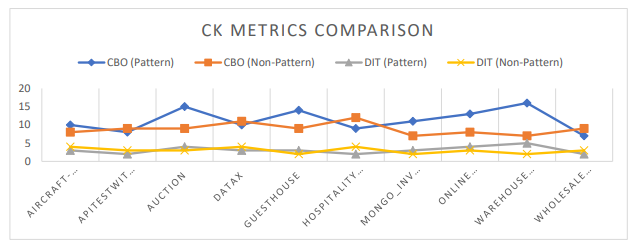
For software components that make use of design patterns or don't, we can examine the testability metrics (CBO, LCOM, DIT, and LOC). Comparing data across components that employ design patterns and those that do not allows us to understand the impact of design patterns on software testability. Statistics show that pattern-based components have lower CBO values than pattern-free components. According to the study's findings, design patterns can lessen coupling, which could create more testable software.

When software items are not tightly coupled with one another, testing and maintenance become much easier. The LCOM values of pattern-based and non-pattern components are same.

A pattern-based component, on the other hand, may exhibit strong cohesiveness when LCOM is zero. Because of this, using design patterns can improve testability by making it easier to isolate and understand specific functionality, leading to more coherence.

In comparison to components that don't depend on patterns, those that do have greater DIT values. The fact that this happens further supports the idea that design pattern inheritance hierarchies could impact testability. Raising the DIT may make testing more difficult since modifications made to the parent class will propagate to the derived classes.

Lines of code (LOC) are a way to measure the amount of variable components in software. When used independently, LOC cannot differentiate between pattern-dependent and non-pattern-dependent components. Even though it's important to look at specific parts of code, design patterns can affect how many lines of code are testable.



Graphs showing CBO and DIT values for project classes with patterns and those without patterns are shown in the figure above. There is a relationship between things when the CBO value is high. Since the DIT measures the depth of the inheritance tree, a higher score suggests a more intricate inheritance structure.

Class patterns and non-patterns have comparable CBO values, but projects are different. The CBO values of auction and warehouse management projects are higher than those of no pattern classes. This shows that some projects' pattern classes may be dependent on other things, which makes testing them more challenging. Application areas like as warehouse management have more complicated inheritance trees, which are characterized by pattern classes with larger DIT values. Complex inheritance hierarchies can worsen testability by amplifying dependencies and side effects.

RQ2: What are the testability differences between design pattern-based and nondesign pattern-based software components?

It is feasible to compare the testability of software components by looking at whether they depend on patterns or not.

According to the statistical data, pattern-based components had lower CBO values than non-pattern components. It seems from these results that components based on patterns might be less coupled and more amenable to testing.

Cohesion between pattern-based and non-pattern-based components may be similar according to LCOM values. Because they are more coherent, components based on patterns with zero LCOM values are easier to test.

The DIT values of pattern-based components are much higher. Importance hierarchies, which can be created using design patterns, might reduce testability because of dependencies and change propagation.

Distinguishing pattern-based components from non-pattern-based components is not achievable with LOC values. The differences in testability caused by LOC require more investigation and study.

1. **Comparison of Testability by Design Pattern Type**

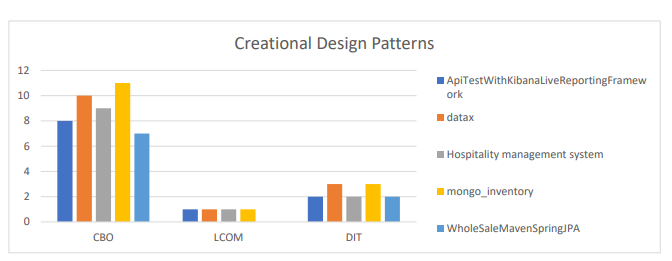
In this section, a comparison is made between the testability of software components in accordance with the creational, behavioral, and structural design patterns available. This data lends credibility to the studies that we have conducted on the testability of software and design patterns that make use of CK metrics.

**Creational design patterns**

The primary focus of creational design patterns is the act of making things. Among the many design patterns we uncovered throughout our analysis were the (Object)Adapter, Factory Method, and Singleton. By making use of these patterns, you'll find that designing items becomes much easier and more versatile.

Figure 7 displays the research outcomes obtained using CK metrics. According to the results, there is consistency in the values of APITestWithKibanaLiveReportingFramew, datax, mongo inventory, and the hotel management system throughout CBO, LCOM, DIT, and LOC. Assuming these programs are similarly testable, the Singleton architecture probably won't make much of a difference to the testability results.

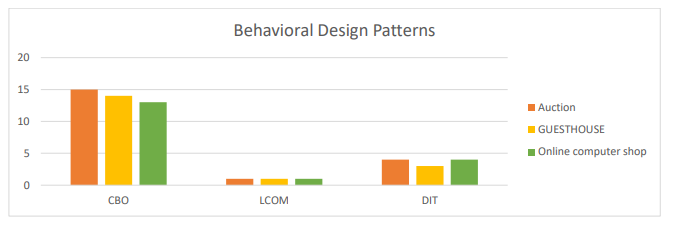
The CK metric scores of the WholeSaleMavenSpringJPA program were same since it used the (Object)Adapter and Factory Method design patterns. On the other hand, some creational design patterns might not affect the testability of software parts at all.



**Behavioral design patterns**

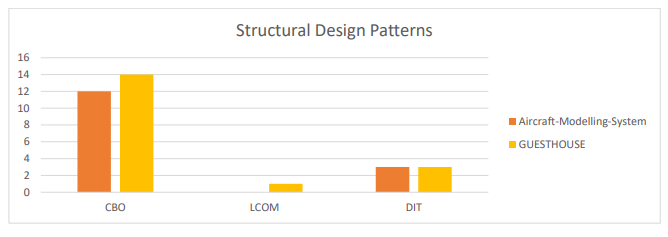
Interactions between objects are the primary focus of behavioural design patterns. See Figure for the results of our study on the (Object)Adapter design pattern in the GUESTHOUSE and Online computer shop programs, and the Observer design pattern in the Auction program.

There is a stronger indication of class connection when the CBO values of the GUESTHOUSE and Auction programs are higher. Coupling makes dependencies more complicated, which makes testing the code more difficult. More study is required to clarify the connection between testability metrics and patterns in behavioral design.



**Structural design patterns**

Assembling smaller parts into larger wholes is the primary goal of structural design patterns. The GUESTHOUSE and Aircraft-Modelling-System apps both made use of two distinct structural design patterns. \*Decorator and (Object)Adapter are the patterns in question. A low LCOM (Lack of Cohesion of Methods) grade suggests that the Decorator pattern of the AircraftModelling-System program is fairly coherent. Cohesive classes are easier to test and have better testability than non-cohesive classes since they are more modular. Figure shows that the GUESTHOUSE application's (Object) Adapter pattern has consistent CK metric values, making it just as tested as non-pattern based components.

****

1. **Implications for Software Development and Testing Practices**

Our research suggests many software development and testing practices:

**Understanding the Impact of Design Patterns:**

The purpose of this research was to examine how design patterns affect software testing efficiency. To further understand the consequences of design patterns, one might compare the testability metrics of components that rely on patterns to those of components that do not.

**Consideration of Testability Metrics:**

A particular point of interest is the testability indicators for software components such as CBO, LCOM, and DIT. According to the findings, it would appear that the implementation of these elements in software development resulted in an improvement in testability.

**Evaluation of Creational Design Patterns:**

According to the findings of our investigation, it would appear that creational design patterns like Singleton and Factory Method do not have an effect on testability assessments. Now that they have access to this information, practitioners are able to make judgments that are more informed regarding which design patterns to implement without compromising the testability of software.

**Consideration of Behavioral Design Patterns:**

Class coupling and testability could be impacted by behavioral design patterns like observer. Considering these patterns improve coupling, tests must be thoroughly covered, so testing methodologies must be carefully considered.

**Benefit of Structural Design Patterns:**

Decorator and (Object)Adapter are two structural design patterns that can be used to make things more testable. Patterns like these serve as illustrations of architecture. Software components are now easier to test and maintain thanks to the implementation of these principles. The components' modularity and coherence have also been enhanced by these concepts. We found additional data that shed light on the topic while comparing pattern classes with non-pattern classes. The table displays a variety of data, including counts of pattern and non-pattern classes, CBO and LCOM values, and more. With this information, we can better understand how design patterns affect various CK metrics, which in turn may help us refine our software testing and development processes.

1. **Threats to Validity**
2. **Internal Validity**

A number of variables, not limited to design patterns, might influence testability metrics. A multitude of variables have the power to distort the correlation between design patterns and testability. The magnitude and intricacy of the undertaking are two such factors. We picked and combined pattern-based and non-pattern-based components according to the project's requirements in order to handle any possible complications. Using this strategy, we reduced the impact of confounding variables while increasing the reliability of our results.

1. **External Validity**

It is possible that the generalizability of the findings is restricted due to the breadth of the projects and design patterns that were included in our research. When attempting to apply the findings to a variety of software systems or design patterns, you need to exercise caution. For the sake of clarity, it is possible that the projects and design patterns that we have selected may not accurately represent the entire software development process. For our research to be considered externally valid, we need to take into consideration the possibility of bias in the selection of both the project and the pattern.

1. **Conclusion**

For this study, we examined the impact of design patterns on software testability using CK metrics in software systems. We set out to assist practitioners in making design pattern selections by learning how design patterns influence testability. We looked at the testability metrics of both pattern-based and non-pattern components, such as coupling (CBO), depth of inheritance tree (DIT), lines of code (LOC), and lack of cohesion of methods (LCOM). We observed that pattern-based components exhibited less coupling and more modularity as evidenced by their lower CBO values. LCOM values were slightly lower for pattern-based components, indicating that they were more coherent. We also looked at testability metrics and discovered that pattern-based and non-pattern-based components have links to CBO and LCOM. By achieving a balance between connection and cohesion, design patterns might potentially improve testability. Our research shows that design patterns improve testability by decreasing coupling and increasing modularity. The context and characteristics of the software system determine which design patterns may improve testability. By picking and assessing components with care, we were able to prevent validity threats. Though applicable to some software systems and development environments, our findings may not be generalizable.

**References**

[1] I. S. E. Terminology, “IEEE standard glossary of software engineering terminology,” IEEE Std 610.12-

1990, pp. 1–84, 1990.

[2] I. S. O. Standard, “ISO/IEC 9126 Software engineering-Product quality.” vol, 2001.

[3] B. Henderson-Sellers, Objectoriented metrics: measures of complexity. Prentice-Hall, Inc., 1995.

[4] A. O. Bajeh, O.-J. Oluwatosin, S. Basri, A. G. Akintola, and A. O. Balogun, “Object-oriented measures

as testability indicators: An empirical study,” J. Eng. Sci. Technol, vol. 15, pp. 1092–1108, 2020.

[5] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo, “An empirical exploration of the distributions of the chidamber and kemerer objectoriented metrics suite,” Empir. Softw. Eng., vol. 10, pp. 81–104, 2005.

[6] “Design Pattern Detection.” <https://users.encs.concordia.ca/~nikolaos/pattern_detection.html> (accessed Jun. 27, 2023).

[7] “GitHub - mauricioaniche/ck: Code metrics for Java code by means of static analysis.”

https://github.com/mauricioaniche/c k (accessed Jun. 27, 2023).