**An Empirical Analysis of Software Systems' Testability Impacted by Design Patterns Using CK Metrics**

**Abstract**

**If you want to build high-quality, easily-maintained software, you should educate yourself on how design patterns affect software testability. Patterns in software system design and testability were identified in this study using CK metrics. Through the use of metrics such as CBO, LCOM, DIT, and LOC, we were able to evaluate the testability of pattern-based and nonpattern-based components utilized in various projects. We contrasted the two component kinds and performed an analysis on how design patterns affect software testability. It is easy to test components that are based on patterns.**

**Modular and test-friendly design are hallmarks of pattern-based components. The components exhibit strong cohesiveness and low coupling (CBO). Because of their connection to inheritance complexity (DIT), design patterns might be helpful for regulating class hierarchies. System longevity and ease of maintenance are both improved by applying design principles to software testability, as we have discovered. A better understanding of the pros and cons of using design patterns in software development, especially with regard to testability, can be gained from these results, which have far-reaching implications. By providing actual evidence that clarifies the ideas of testability and design patterns, this work adds to the current software engineering body of knowledge.**

**Keywords**

Design patterns, Software testability metrics, Empirical analysis.

1. **Introduction**

For the modern human, software systems permeate every facet of daily life. Reliable, user-friendly, and thoroughly tested software solutions are necessary for monitoring critical infrastructure and financial processes. When it comes to organizing and structuring code, design patterns are your best bet for getting certain traits. One of the most efficient ways to fix common software engineering problems is to use design patterns. This ultimately results in software that is easier to modify, scale, and reuse. Thorough research and analysis of design patterns is necessary regardless of whether they might enhance testability.

A program is considered testable if it can be easily and correctly tested for errors. Effective testing increases software quality, speeds up mistake detection and correction, and guarantees the system runs at peak efficiency. We still don't know how testability relates to specific design patterns.

A software that is easy to test is an excellent example of its kind. The term "testability" describes how well a system or component meets the requirements for testing [1].One program property that affects the amount of work needed for software validation is testability, which is also called maintainability, according to ISO [2].

By performing an empirical study on the effect of design patterns on software testability, this work intends to fill the knowledge gap. This study sought to answer the following question: "How do design patterns influence the testability of software systems?" by investigating how design patterns affect software testability.

Several OO metrics are presented in the existing literature [3]. A number of criteria, including size, connectivity, complexity, coherence, and inheritance, are considered when assessing the testability of an OO software system. Different people have different ideas about how to make sure software can be tested [4]. To prove that software is testable, object-oriented metrics alone aren't enough. The practicality of software testing is affected by multiple elements. While it's true that testability is affected by each of these measurements (attributes) independently, empirical research on how they interact to establish class testability is scarce. This is made much clearer when varying degrees of testing effort are considered. For the time being, no theoretical nor experimental studies have tackled this problem.

At both low and high levels of testing effort, object-oriented (OO) design metrics, which include the CK metrics package, correlate with class testability [5].

The purpose of this research is to assess testability using CK metrics in software systems that contain at least 5,000 lines of code. A program's complexity and maintainability can be assessed with the help of CK metrics.

By comparing testability metrics between classes that use patterns and those that don't, one can understand how design patterns affect testability.

A plethora of empirical evidence about testability and design trends is provided in the following papers.

The results of this study can help architects and developers improve software quality and testability, which in turn can help them make better judgments when it comes to design patterns. Software testing can be enhanced and more reliable software systems can be developed through research into the effects of design patterns on testability.

1. **Research Motivation**

Motivating this endeavor was a strong desire to comprehend how design patterns affect software testability. We can learn more about how design patterns affect testing efficacy and efficiency by clicking on this link. In order to create trustworthy and manageable software, programmers and testers must have access to this data. Software engineers may also benefit from evidence-based recommendations for design patterns that could be found in this branch of empirical evaluation.

1. **Research Objective**

The fundamental goal of this research was to examine the effect of design patterns on software testability in an unbiased manner. This study aims to determine the impact of design patterns on system testability by comparing testability measures for software components that use design patterns and those that do not. The study has three main goals:

* 1. Adding to the software engineering body of knowledge.
  2. Exploring the relationships between testability and design patterns.
  3. Offering quantitative evidence and insights.

Practitioners will be better able to understand how different design patterns affect software testability, which will aid in making educated decisions about pattern adoption.

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**

This study will pique the interest of students and corporate executives alike because it delves into the ways design patterns impact the testability of software systems. Findings from this inquiry may lead to the possible acquisition of further information about software engineering.

Having a strong grasp of how design patterns impact testability is crucial if you aim to enhance software quality. We can learn more about how design patterns can make software systems more reliable, robust, and maintainable if we investigate this link. Because it allows them to spot major design trends, software engineers greatly benefit from having access to this data, which in turn leads to better code quality.

There is universal agreement that rigorous testing is the best way to find out if software is stable and trustworthy. Additionally, for it to serve as a testing instrument, it needs to be testable. Improve your efficiency and effectiveness as a software tester by learning more about test capabilities and design patterns.

Improving the program's quality, finding bugs, and covering tests are only a few of the many outcomes of this.

The implications of this discovery go much beyond the scope of this specific framework. Employees in the software development industry will thus have more resources at their fingertips to help them choose the best design patterns for their projects.

Building software systems that are easier to test and maintain could be achievable if developers had a clearer grasp of the relationship between testability and design patterns.

**Methodology**

1. **Selection of Subject Programs**

Studying how design patterns affect the testability of various software systems is the goal of this research, which employs quality control approaches.

The very nature of the selection process ensured that the final product would be massive in size, packed with numerous design patterns, and exceedingly complex.

These apps were culled from a broad range of academic projects and open-source repositories.

Applications were considered for inclusion if they had 5,000 lines of code or more. The reason behind this is that smaller systems often have simpler architectures and might not utilize enough design patterns.

We chose these software systems because they are all indicative of real-world scenarios that can take many forms. By utilizing our tools, software engineers can improve their ability to spot design patterns and assess how testable they are.

Using the provided criteria, we chose software systems with the intention of conducting representative and relevant study. Several of the following factors were considered at different stages of the selection process:

**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:**

Apps that made use of design patterns during their development were more satisfying to us than those that did not. Through the utilization of this statistic, we investigated the connection between testability and various design patterns, as well as the consequences that these patterns have on the dependability of software.

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

This study mainly aims to examine how software systems employ design patterns. To implement our testing strategy, we first sort the code into two groups: pattern classes, which are identified by obvious design patterns, and non-pattern classes, which do not include any such patterns.

1. **Dependent Variable: Testability**

The accessibility of software systems is a very important variable for us, and it is the dependent variable in this study. The simplicity and efficacy of a program's testing is one measure of its testability. In order to assess the testability, suitable CK measures are employed. These metrics consider a plethora of parameters, including, but not limited to, the complexity of the code, the degree of connectivity, and the degree of coherence.

1. **Design Pattern Mining**

We employed a pattern mining technique to discover commonalities in the selected software systems' designs. Whoever is thinking about using it can find the app we used for our research at this URL: https://users.encs.concordia.ca/~nikolaos/pattern detection.html. No, the correct answer is [6]. You can find and understand code examples that are good representations of design patterns if you use this tool thoroughly. Applying ideas from graph theory and approaches centered on similarity scores can lead to even better software development. Instruments that incorporate these state-of-the-art technologies tend to have better pattern recognition capabilities, which in turn leads to increased overall utility.

1. **Extraction of CK Metrics**

We used the CK tool, a Java-based tool that examines code metrics at the class and method levels, to gain a better understanding of the selected software systems [7]. Several distinct metrics are employed by this device. When discussing measures in this context, terminology like "lines of code," "connections between objects," and "depth inheritance tree" all fall into this category.

In the first stage of testing Java apps with different configurations, we ran the JAR file we had prepared using the standalone CK tool. Incorporating the CK tool into Java projects is another potential use of the CK library.

The CK tool provides measurements that can be used to gain a better understanding of testing, design patterns, and software metrics. The quality of the code and our ability to assess the selected software systems are both enhanced by the use of CK metrics.

1. **Results**

In what follows, we detail our research on the most effective methods for making software systems and design patterns more testable. One approach to investigating the effect of design patterns on testing efficiency is to compare testability metrics between classes with and without design patterns.

To find out if there was any noteworthy difference or pattern, we compared the two groups' testability metrics. The impact of design patterns on the testable software's capabilities is currently the subject of an inquiry into the analysis of CK metrics.

1. **Dataset**

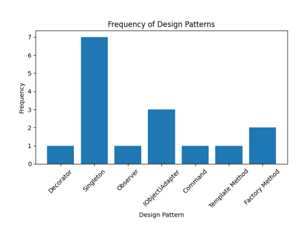
In the results section, we commence by acquainting the reader with our dataset. The dataset presented in the table below encompasses a wide variety of programming methods. In order to ensure a thorough examination of the impact of design patterns on testability, these specific programs were selected based on their varying sizes and diversity.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

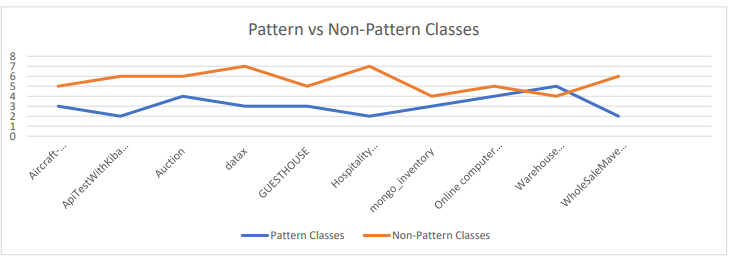
1. **Overview of Design Patterns in Subject Programs**

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**

As part of this section, we will discuss the significance of CK measurements in determining the quality of software and provide a summary of all the metrics that are relevant to the project.

The graphic shows a graphical representation of the pattern classes and non-pattern classes for every project. The "Non-Pattern Classes" column counts classes that do not exhibit any particular design pattern, while the "Pattern Classes" column counts classes that do display such patterns. Both kinds of courses are differentiated from one another.

Projects contain a wide variety of pattern classes and non-pattern classes, according to the data.

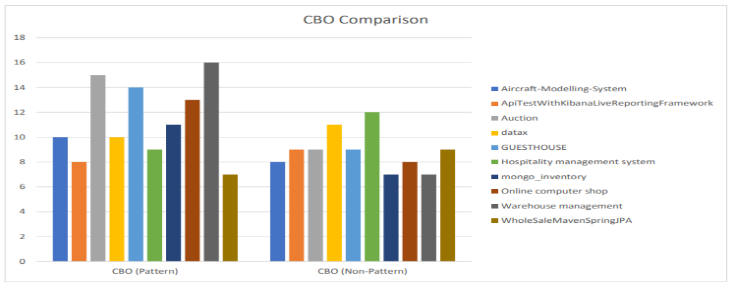
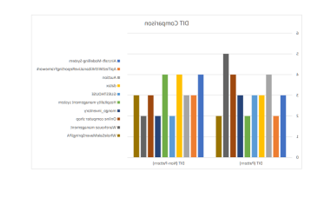
While "Aircraft-Modelling-System" contains five pattern classes and three non-pattern classes, "Warehouse management" only has three pattern classes. Here we will compare the programs. The data may show how each project's design makes use of certain classes and patterns. Furthermore, it may be useful for weighing the pros and cons of design patterns in software architecture

Figure is a graphical representation of a comparison between pattern classes and non-pattern classes across various projects. Projects are shown on the x-axis, and the number of pattern and nonpattern classes in each project is shown on the y-axis. Coupled Between Objects (CBO) and Non-Pattern CBO (NPBO) measures are shown in this graph.

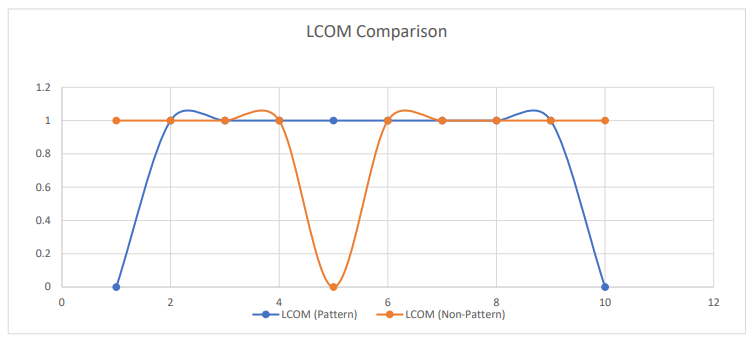
The CBO values for each project's pattern classes and non-pattern classes are shown in the graph.

If pattern classes have a higher CBO score than non-pattern classes, it means that the two kinds of classes are highly dependent on one other and that testing will likely be difficult to achieve separation for the pattern classes.



The image above illustrates the distinction between pattern classes and non-pattern classes based on their DIT values. The x-axis represents the projects, while the y-axis represents the DIT values. Both axes are observable to the unaided eye. Pattern classes in projects generally have a greater DIT value compared to non-pattern classes. This is because pattern classes are increasingly prevalent. Pattern classes may potentially exhibit a more intricate inheritance hierarchy.

Several projects, including "AircraftModelling-System," "Auction," "datax," and "mongo inventory," have similar DIT values for both pattern classes and nonpattern classes. The "GUESTHOUSE" and "Warehouse management" projects had a more intricate inheritance tree because to the higher DIT values of pattern classes compared to non-pattern classes.



Observe the comparison between LCOM and other projects in relation to the presence of pattern and non-pattern classes in the figure. Smaller LCOM numbers indicate a higher level of class cohesiveness, while larger LCOM values indicate a lower level of modularity.

The method coherence is evident since the majority of the graph's pattern and nonpattern classes have LCOM scores of 1. Except for WholeSaleMavenSpringJPA and Aircraft-Modeling-System, all projects have an LCOM value of 0 for pattern classes. It may be deduced from this that pattern classes exhibit more cohesion compared to nonpattern classes.

1. **Answering Research Questions**

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

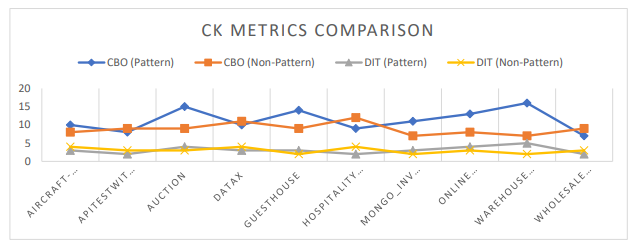
To find out if software parts use design patterns, you can use the testability metrics, which include CBO, LCOM, DIT, and LOC. Analyzing and comparing data from components that use design patterns with those that do not is a good way to understand how design patterns affect software testability. Relative to pattern-free components, pattern-based components had lower CBO values, according to the statistical analysis. Use of design patterns may improve software testability by reducing coupling, according to the research.

When software parts are not tightly linked, testing and maintenance are much easier. All components, pattern-based or non-pattern, have the same LCOM value.

At the same time, a pattern-based component may show strong cohesion when LCOM= 0. By making it easier to isolate and understand individual functions, design patterns can improve testability and coherence.

Typically, components that rely on patterns will have deeper inheritance trees (DIT) than those that don't. Given this, it's possible that testability may be affected by design pattern inheritance hierarchies. Raising the DIT might make testing more difficult since modifications to the parent class would have an automatic ripple effect on all descendant classes.

Looking at the lines of code (LOC) is one way to gauge how extensive the software's variable components are. The use of LOC alone does not allow for the separation of pattern-dependent components from non-pattern-dependent ones. The importance of testing specific sections of code is irrelevant when it comes to design patterns, which can affect the number of tested lines of code.



The CBO and DIT values for pattern-containing and pattern-free project classes, respectively, are shown in the aforementioned diagrams. There is a significant association between variables when their correlation is high. An increasingly high DIT score, which measures the depth of the tree, indicates a more complex inheritance pattern.

Class patterns and projects are distinct entities, even though their CBO values are the same. When compared to pattern-free classes, auction and warehouse management programs have greater CBO values. As a result, depending on additional factors can make testing pattern classes in some projects more challenging. Application domains like as warehouse management often have more complex inheritance structures, as indicated by pattern classes with increased Depth of Inheritance Tree (DIT) values. The complexity of inheritance trees can make tests less reliable since they increase the likelihood of side effects and dependencies.

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

It is possible to evaluate the testability of software components by looking at how much they rely on patterns.

The results showed that compared to non-pattern components, pattern-based components had lower CBO values. These results suggest that components based on patterns can be easier to evaluate and have a lesser association.

According to the LCOM findings, pattern-based and non-pattern-based components may have comparable levels of cohesiveness. It is easier to test components with zero LCOM values since they are more cohesive.

The Depth of Inheritance Tree (DIT) values of pattern-built components are much higher. Importance hierarchies, which can be defined with design patterns, might impede testability because of dependencies and the propagation of changes.

Differentiating pattern-based components from non-pattern-based components using LOC values is not feasible. To resolve the variations in testability induced by LOC, additional research and analysis are needed.

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

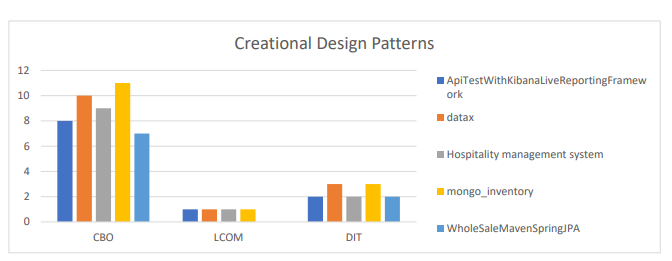
In this study, we assess the testability of software components by examining the structural, behavioral, and creational design patterns that are currently available. This data supports our research on the testability of software and design patterns using CK metrics.

**Creational design patterns**

The process of creating something is fundamental to creational design patterns. During our inquiry, we encountered several design patterns, such as the (Object)Adapter, Factory Method, and Singleton. Utilizing these patterns will significantly simplify and enhance the adaptability of item creation.

Figure 7 displays the findings obtained from the experiment employing CK measurements. The organization's management system, consisting of the CBO, LCOM, DIT, and LOC levels, maintained consistent values for APITestWithKibanaLiveReportingFramew, datax, mongo inventory, and the system as a whole. Assuming that these programs have similar testability, it is improbable that the Singleton architecture will have a substantial effect on the testability outcomes.

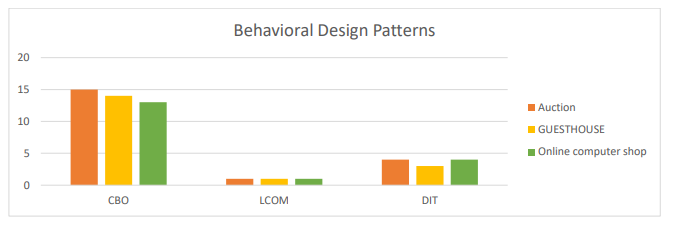
The WholeSaleMavenSpringJPA software achieved equal CK metric ratings because to its utilization of the (Object)Adapter and Factory Method design patterns. Nevertheless, it is conceivable that specific creational design patterns may not affect the testability of software components at all.



**Behavioral design patterns**

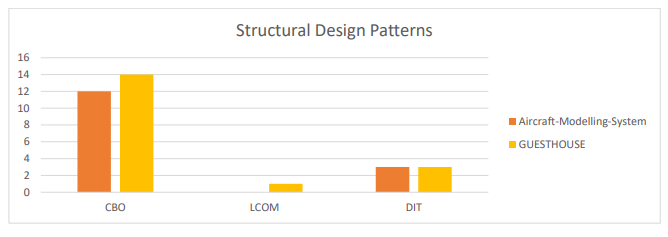
Behavioral design patterns primarily focus on the interactions between objects. To access our research on the (Object)Adapter pattern in the GUESTHOUSE and Online computer shop programs, as well as the Observer pattern in the Auction program, please consult the Figure.

A rise in the CBO values of the GUESTHOUSE and Auction programs is linked to a more pronounced indication of class association. Code testing is increasingly difficult because of the presence of coupling, which adds complexity to the dependencies. Further investigation is required to have a deeper understanding of the correlation between testability measurements and behavioral design patterns.



**Structural design patterns**

The primary goal of structural design patterns is to integrate smaller components into larger entities. The GUESTHOUSE and Aircraft-Modelling-System apps employed two distinct structural design patterns.The patterns in question are the Decorator pattern and the (Object) Adapter pattern. The Decorator pattern in the AircraftModelling-System software seems to be well-organized, as evidenced by its low LCOM (Lack of Cohesion of Methods) ranking. Classes that exhibit high cohesion are characterized by modularity, making them more amenable to testing and increasing their testability compared to classes with low cohesion. The (Object) Adapter pattern in the GUESTHOUSE application exhibits consistent CK metric values, as depicted in the Figure. This implies that it undergoes the same level of testing as components that do not utilize patterns.

****

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

Many different approaches to software development and testing have been identified by our research:

**Analyzing the Influence of Design Patterns:**

The researchers set out to find out how design patterns affect software testing productivity. You may learn more about design patterns and their effects by comparing the testability metrics of components that use patterns with components that don't.

**Consideration of Testability Metrics:**

The testability indicators for software components, such as CBO, LCOM, and DIT, are one area of particular emphasis. The results suggest that these enhancements improved software testability when included during development.

**Evaluation of Creational Design Patterns:**

Creational design patterns like Singleton and Factory Method do not appear to have any effect on testability evaluations, according to our research. The availability of this data allows practitioners to make more educated decisions on the design patterns to use in order to maintain software testability.

**Consideration of Behavioral Design Patterns:**

Class coupling and testability are two areas that behavioral design patterns like the observer pattern might influence. Careful evaluation of testing methodologies is necessary to ensure sufficient coverage of tests in order to improve coupling.

**Benefit of Structural Design Patterns:**

Using structural design patterns like Object Adapter and Decorator improves the system's testability, which is a huge plus. Such patterns serve as visual depictions of architectural style. Software components can now be more easily tested and maintained thanks to the application of these concepts. Thanks to these methods, the parts are now more cohesive and modular. We found more data that shed light on the situation while comparing pattern classes with non-pattern classes. There is a wide variety of statistics presented in the table, including counts of pattern and non-pattern classes, values for CBO and LCOM, and more. We can learn more about how design patterns affect various CK metrics by examining this data. Our software testing and development processes can be improved using this knowledge.

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

Design patterns are just one of many aspects that might affect testability metrics. Several factors can influence the link between testability and design patterns. The scope and complexity of the project are two examples of these characteristics. We pick and combined pattern-based and non-pattern components with care to meet the project's requirements and overcome any obstacles that may have arisen. We improved the credibility of our results and reduced the impact of confounding variables by using this method.

1. **External Validity**

The diverse range of projects and design patterns employed in our study may restrict the generalizability of the findings to different settings. Exercise caution while attempting to modify the outcomes to suit various software systems or design patterns. The selected projects and design patterns may not provide a comprehensive representation of the software development process, but they have been included for the purpose of enhancing clarity. It is essential for our study to be considered externally trustworthy to carefully consider the potential for bias in the selection of the project and the pattern.

1. **Conclusion**

The effect of different design patterns on software testability was investigated in this study using CK metrics. Our goal was to help practitioners understand how design patterns affect testability so they may make better pattern decisions. Using testability metrics like lack of cohesion of methods (LCOM), lines of code (LOC), depth of inheritance tree (DIT), and coupling (CBO), we compared pattern-based and non-pattern components. Higher modularity and reduced coupling are indicated by lower CBO values for components based on patterns. Components based on patterns had slightly lower LCOM values, suggesting more cohesiveness. Further analysis of testability metrics revealed a correlation between pattern-based and non-pattern-based components with CBO and LCOM. Patterns in design can help make tests more reliable by balancing connection and cohesiveness. By reducing coupling and increasing modularity, design patterns improve testability, according to our research. It is the software system's context and attributes that decide which design patterns could increase testability. All possible dangers to validity were eliminated through meticulous component selection and evaluation. Our results may only be relevant to the specific software systems and development environments that we examined.

# **References**

|  |  |
| --- | --- |
| [1] | "Design Pattern Detection," 27 Jun 2023. [Online]. Available: https://users.encs.concordia.ca/~nikolaos/pattern\_detection.html. |
| [2] | I. S. O. Standard, "ISO/IEC 9126 Software engineering-Product quality," 2001. |
| [3] | B. Henderson-Sellers, Objectoriented metrics: measures of complexity, Prentice-Hall, Inc., 1995. |
| [4] | J. E. S. Technol, "Object-oriented measures as testability indicators: An empirical study," *J. Eng. Sci. Technol,* vol. 15, p. 1092–1108, 2020. |
| [5] | W. P. S. D. P. Z. a. B. R. G. Succi, "An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite," *Empir. Softw. Eng,* vol. vol. 10, p. 81–104, 2005. |
| [6] | I. S. E. Terminology, "IEEE standard glossary of software engineering terminology," IEEE Std 610.12-1990, 1990. |
| [7] | "GitHub - mauricioaniche/ck: Code metrics for Java code by means of static analysis," [Online]. Available: https://github.com/mauricioaniche/ck. |