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

Abstract

Understanding the effects of design patterns on software testability is essential for creating high-quality, maintainable software systems. This empirical study examined design patterns and testability in software systems using CK metrics.

We evaluated pattern-based and non-pattern-based components' CBO, LCOM, DIT, and LOC testability measures across several projects. Design patterns' effects on software testability and the variations in testability between these two types of components were our study objectives.

Pattern-based components have higher testability than non-pattern-based ones. Pattern-based components have decreased coupling (CBO) and greater cohesion (LCOM), indicating increased modularity and simpler testing. We also identified an association between inheritance complexity (DIT) and design patterns, suggesting that design patterns may help manage class hierarchies.

Our findings imply that design patterns can improve software testability, leading to more robust and maintainable systems. These insights might help practitioners decide whether to utilize design patterns in software development and how they can affect testability. This research adds to software engineering by giving empirical data and insights regarding design patterns and testability.

Keywords

Design patterns, Software testability metrics, Empirical analysis

I. Introduction

Software systems affect every part of our lives nowadays. Software systems must be reliable, easy to manage, and tested in order to manage financial transactions and vital infrastructures. Design patterns organize and structure code to accomplish these attributes. Design patterns solve frequent software design issues. Software reuse, modularity, and extensibility are all enhanced by them. Although design patterns have been found to enhance testability, they still need to be thoroughly examined and evaluated.

The testability of software refers to how easily it can be verified for correctness and reliability. Effective testing is necessary to

find and resolve errors, improve software quality, and ensure system behavior. However, the link between design patterns and testability is uncertain.

Software quality includes testability. IEEE [1] defines testability as how well a system or component supports test requirements and testing. ISO [2] defines testability (maintainability) as software attributes that impact software validation effort.

This work empirically evaluates the effect of design patterns on software system testability to fill this gap. The study investigates whether design patterns affect software system testability, offering insights into the link between design patterns and testability.

Many OO metrics were proposed in literature [3]. OO software systems' testability has been predicted using criteria including size, coupling, complexity, cohesion, and inheritance. Software testability has been examined from multiple perspectives. According to [4], none of the OO metrics alone can assess software testability. Many variables impact software testability. Even though several of these metrics (attributes) affect class testability, few empirical studies have examined their combined effect, especially when considering different levels of testing effort. This topic has not been experimentally studied.

This study empirically links OO design metrics, particularly the Chidamber and Kemerer (CK) metrics suite [5], to class testability at different testing effort levels.

The study goal is to investigate at least software systems with at least 5k lines of code and evaluate testability metrics utilizing CK metrics. CK metrics quantify program complexity and maintainability. We compare testability metrics across pattern classes and non-pattern classes to evaluate how design patterns affect testability.

Empirical data on design patterns and testability is provided by this research. Software developers and architects may utilize this study's findings to make smart decisions about using design patterns to increase testability and software quality. We can optimize software testing and produce more resilient and reliable software systems by understanding how design patterns affect testability.

A. Research Motivation

The necessity of investigating the effect of design patterns on software testability is what inspired this study. Examining this link will help us better understand how design patterns affect the efficiency and efficacy of testing. This information is

essential for programmers and testers who want to make stable, easily maintained programs. Furthermore, empirical assessment in this domain can offer evidence-based direction for choosing when and how to apply design patterns in software development.

B. Research Objective

The primary goal of this research was to examine the impact of design patterns on software testability through an empirical analysis. We measure and compare testability metrics across software components that use design patterns and those that do not to evaluate if design patterns affect system testability. This study aims to contribute to the body of knowledge in software engineering by providing quantitative evidence and insights into the relationship between design patterns and testability. This will help in assisting practitioners in making smart decisions regarding the adoption of design patterns and their implications for software testability.

C. 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?

D. Significance of the Study

This study has significant value for both academia and industry since it examines how design patterns affect software system testability. This research can improve software engineering expertise.

Improving software quality requires understanding how design patterns affect testability. By studying this connection, we may learn how design patterns can improve software systems' reliability, robustness, and maintainability. This information can help software engineers choose design patterns and enhance software quality.

Software reliability and stability depend on extensive testing. Effective testing requires testability. We can help software testers plan and run tests more effectively by studying design patterns and testability. This increases test coverage, fault discovery, and software quality.

This study has wider implications. The insights can assist software development teams choose design patterns. Understanding the link between design patterns and testability may help engineers

design more effectively manageable and testable software systems.

II. Methodology

A. Selection of Subject Programs

We have chosen a wide range of software systems to evaluate the effect of design patterns on testability using CK metrics. The selection method ensured program size, complexity, and design pattern diversity.

Programs from open-source repositories and academic projects were selected. Smaller programs have simpler architectures and may not include many design patterns; thus, we preferred programs with at least 5,000 lines of code.

The chosen software systems reflect a variety of real-world settings. We provide a variety of applications to capture software design patterns and analyze their testability.

Our study chose software systems based on specified criteria to assure relevance and representativeness. The selection procedure examined the following criteria:

i. *Size and Complexity:*

We searched for software with a large codebase, usually at least 5,000 lines. We used this criterion to study how design patterns affect testability in larger and more complex software systems.

ii. *Presence of Design Patterns:*

We preferred applications that used design patterns. This criterion allowed us to examine the link between design patterns and testability and how various patterns affect software testability.

B. Independent Variable: Presence of Design Patterns

Our research is based on whether or not software systems use design patterns. We divide the examined software into two classes: those with identified design patterns (pattern classes) and those without (non-pattern classes).

C. Dependent Variable: Testability

Software system testability is our dependent variable. Software testability relates to its simplicity and efficacy. We use appropriate CK metrics to evaluate testability, including code complexity, coupling, cohesion, and other factors.

D. Design Pattern Mining

We used a pattern-mining tool specifically developed to help in the identification of design patterns in the chosen software systems. The tool used in our study can be accessed at the https://users.encs.concordia.ca/~nikolaos/pattern_detection.html[6]. The tool offers a thorough means of identifying and inspecting code examples for occurrences

of design patterns. It makes use of similarity scoring methods and concepts from graph theory to improve software development. The tool improves the efficiency and precision of its pattern-detection functions by making use of these innovative methods.

E. Extraction of CK Metrics

We used the CK tool, a Java-based tool[7] that computes code metrics at the class and method levels, to gain useful insights into the chosen software systems. Metrics such as coupling between objects (CBO), depth inheritance tree (DIT), lines of code (LOC), and more are included in this tool.

We used the standalone CK tool to evaluate Java applications by producing a JAR file and executing it with certain settings. The CK library may also be used to include the CK tool in Java programs.

The CK tool's measurements assist us understand testability, design patterns, and software metrics. Code quality may be evaluated and conclusions about the chosen software systems can be drawn by extracting CK metrics.

III. Results

This section presents our study on design patterns and software system testability. We evaluate testability metrics from pattern classes (classes with design patterns) and non-pattern classes (classes without design patterns) to determine how design patterns affect testing ease and effectiveness.

Our research compares these two groups' testability measures to find any significant variations or patterns. We examine CK metrics to understand how design patterns affect software system testability.

A. Dataset

We begin the results section by describing our dataset. A variety of software systems are represented by topic programs in the dataset shown in the table below. These subject programs were selected based on program size and variety to ensure a full investigation of design patterns' effects on testability.

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, Relational database
WholeSaleMavenSpringJPA	Retail	11,200	Wholesale order and item management, Layered architecture, various design patterns
Coolio Course Reviewer	Education	5,500	Client-server application for course reviews
Cyberpet	Gaming	3,500	Basic game where a predator catches its prey
Cyber Security Management in Healthcare	Healthcare	5,000	Java Swing application for cyber security features in healthcare domain
Global-Humanitarian-Aid	Humanitarian	4,500	Swing application for aiding NGOs in crises
javaparser	Development	5,000	Libraries for Java parsing and analysis
Obl_project	Library	8,000	Information system for a library

Supermarket System	Retail	6,500	System for managing a supermarket
Talabate-Clone	Restaurant Management	4,200	Console application for restaurant management
Wholesale Management System	Wholesale	5,800	System for wholesale operations
WholeSalePOSWithwithHibernate	Wholesale	10,000	System for wholesale order and item management
jeopardy	Entertainment	4,500	Java version of Jeopardy game for classroom-style use
Library Web Application	Library	7,500	Web application for library management
animalrace	Gaming	3,800	Desktop app demonstrating the Abstract Factory pattern
MakingDrinks	Food & Beverage	3,200	Java console-based application for making drinks using Factory Design Pattern
patrones	Software Development	3,500	GitHub repository containing various design patterns
Desktop Fruit Ninja App	Gaming	4,200	Desktop app developed using Java programming language
CRUD APP	Software Development	5,500	CRUD operation application using Factory Design Pattern
Animator_MultyView	Animation	5,000	App for creating animations using OOD design principles and patterns
Easy Decorator App	Grocery	4,800	Grocery app implementing various design patterns

B. 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	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
Coolio Course Reviewer	Factory Method, Singleton
cyberpet	State
Cyber Security Management in Healthcare	Singleton
Global-Humanitarian-Aid	Singleton
javaparser	Singleton
Obl_project	Factory Method, Singleton
Supermarket System	Singleton
Talabate-Clone	Factory Method, (Object) Adapter
Wholesale Management System	Singleton, (Object) Adapter, State
WholeSalePOSWithwithHibernate-master	Singleton, (Object) Adapter
jeopardy	Factory Method, (Object) Adapter, Observer,State
Library Web Application	Singleton
animalrace	Factory Method, Singleton, (Object) Adapter, State
MakingDrinks	Factory Method
patrones	Factory Method, Singleton
Product Inventory Management System	Singleton, (Object) Adapter, State

Desktop Fruit Ninja App	Factory Method, Singleton
CRUD APP	(Object)Adapter, State
Animator_MultyView	(Object)Adapter, State, Bridge
Easy Decorator App	Singleton, (Object) Adapter, Decorator, Observer, State

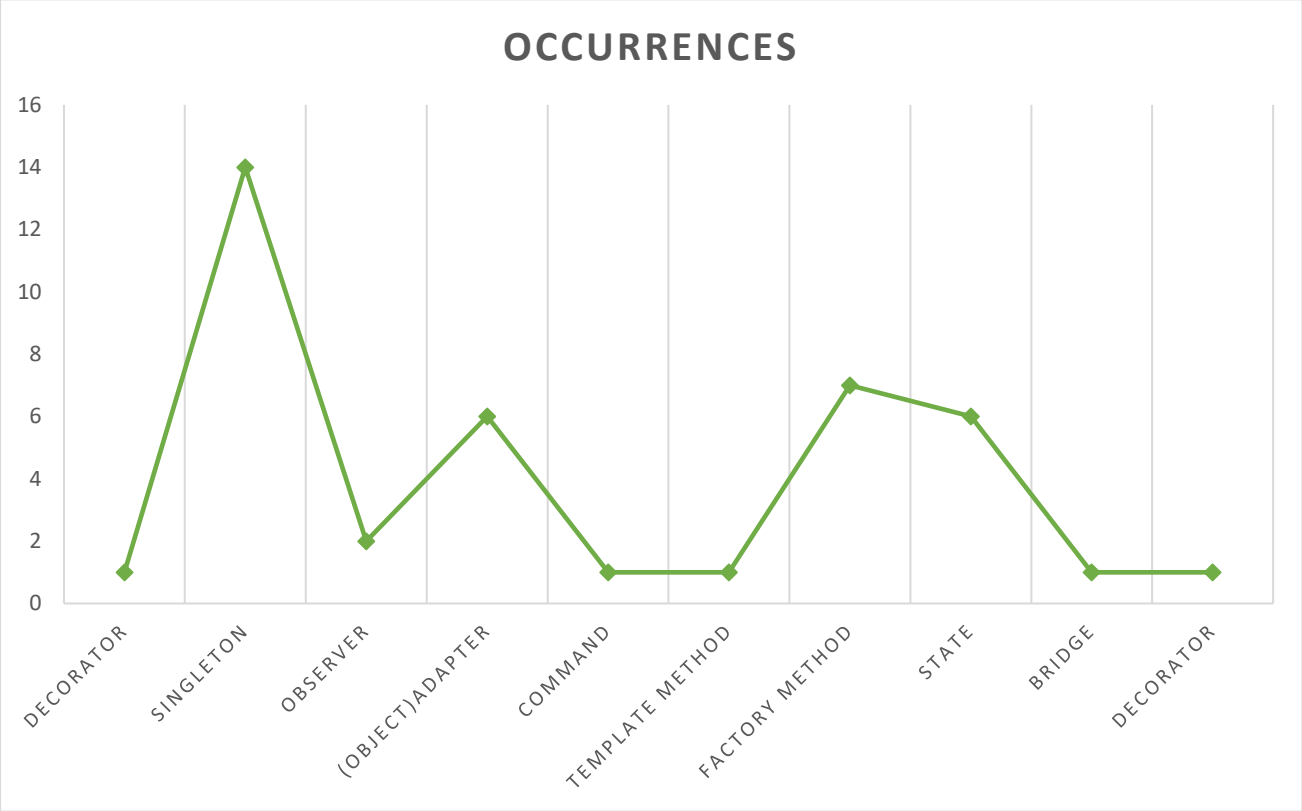

Figure 1:Frequency and Distribution of Design Patterns in Subject Programs.

D. CK Metrics for Subject Programs

This section summarizes the CK metrics retrieved for each of the projects and discusses their importance in software quality evaluation.

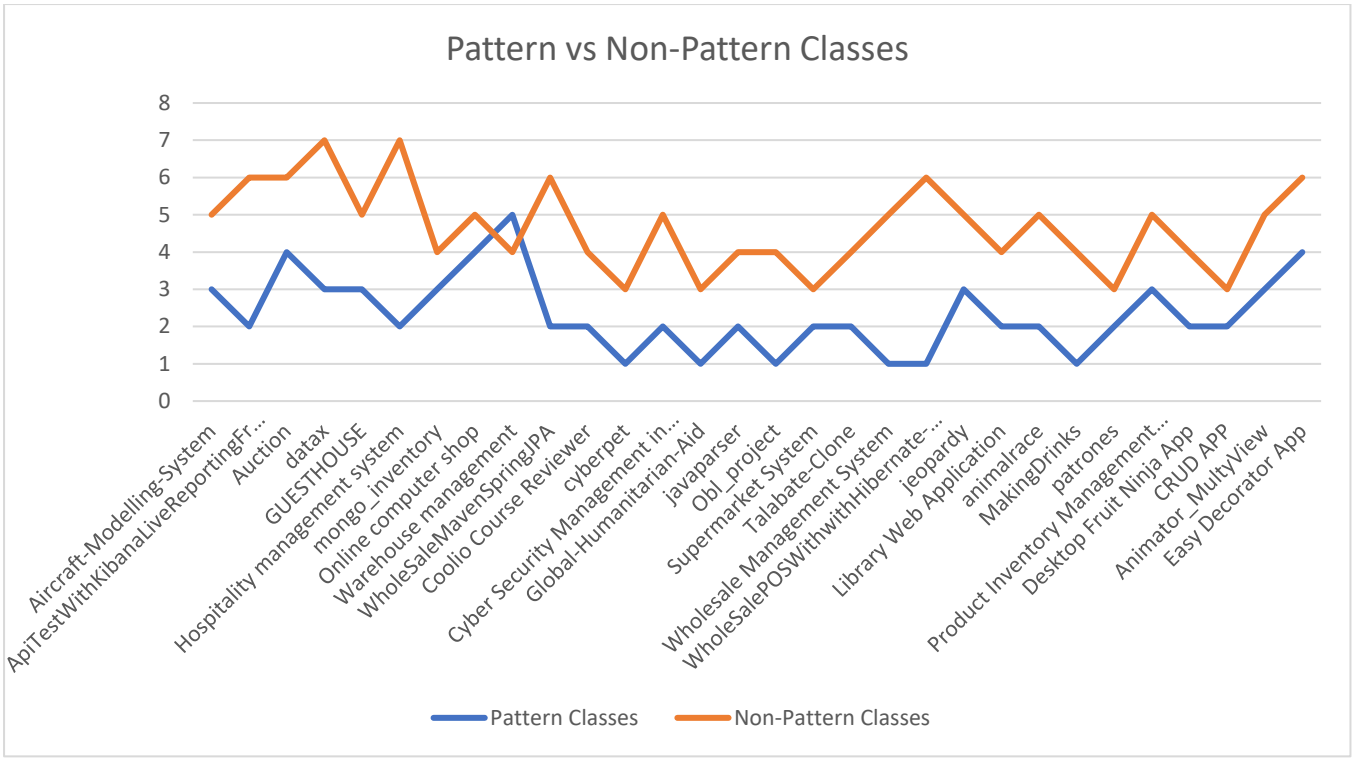

Figure 2:Comparison of Pattern and Non-Pattern Classes in Different Projects.

The graph in

Figure 2 shows each project's pattern and non-pattern classes. The "Pattern Classes" column counts classes that display particular design patterns, whereas the "Non-Pattern Classes" column counts classes that do not.

The data shows that projects have various numbers of pattern and non-pattern classes. "Aircraft-Modelling-System" has 3 pattern classes and 5 non-pattern classes, whereas "Warehouse management" has 5 pattern classes and 4 non-pattern classes. This data can reveal each project's design patterns and class distribution. It may also help evaluate the effects of design patterns on software architecture and their possible advantages.

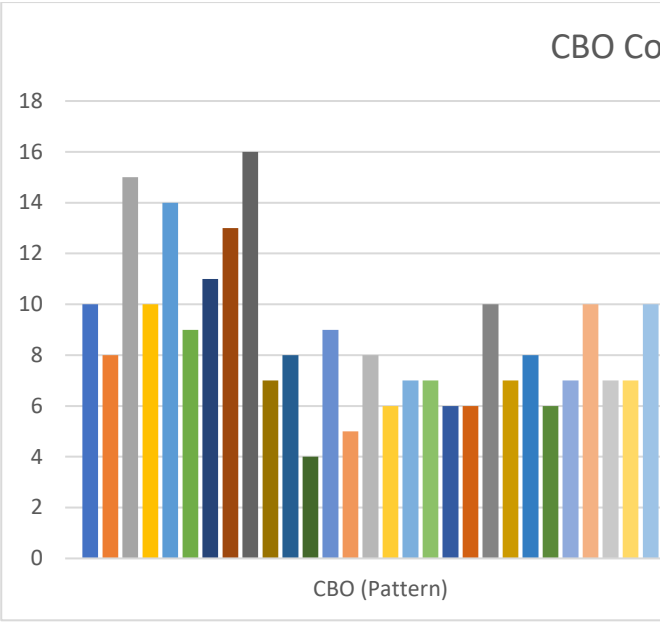

Figure 3:Comparison of Pattern and Non-Pattern Classes with CBO Values.

The graph in Figure 3 compares pattern and non-pattern classes in various projects. The x-axis shows projects, while the y-axis shows the number of pattern and non-pattern classes in each project. CBO (Coupling Between Objects) and CBO (Non-Pattern) metrics are included on the graph.

Each project's CBO values for pattern and non-pattern classes are shown in the graph. Pattern classes have higher CBO values than non-pattern classes, suggesting stronger interdependence and possible issues separating units for testing.

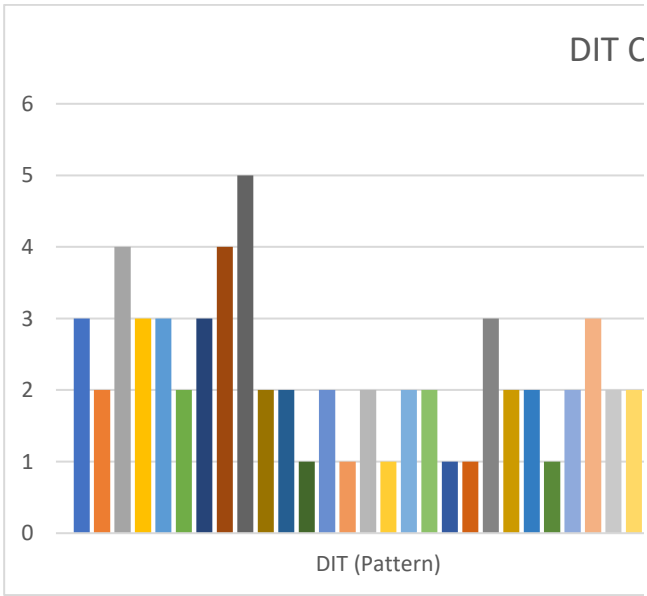

Figure 4:Comparison of Depth Inheritance Tree (DIT) between Pattern and Non-Pattern Classes in Projects.

This graph in Figure 4 compares DIT values across pattern and non-pattern classes in the projects. The x-axis shows projects and the y-axis DIT values. Pattern classes have higher DIT values than non-pattern classes in most projects. Pattern classes may have a more complicated inheritance structure.

In several projects, such as "Aircraft-Modelling-System," "Auction," "datax," and "mongo_inventory," pattern and non-pattern classes have comparable DIT values. In projects like "Warehouse management" and "GUESTHOUSE," pattern classes have higher DIT values than non-pattern classes, suggesting a more complex inheritance structure.

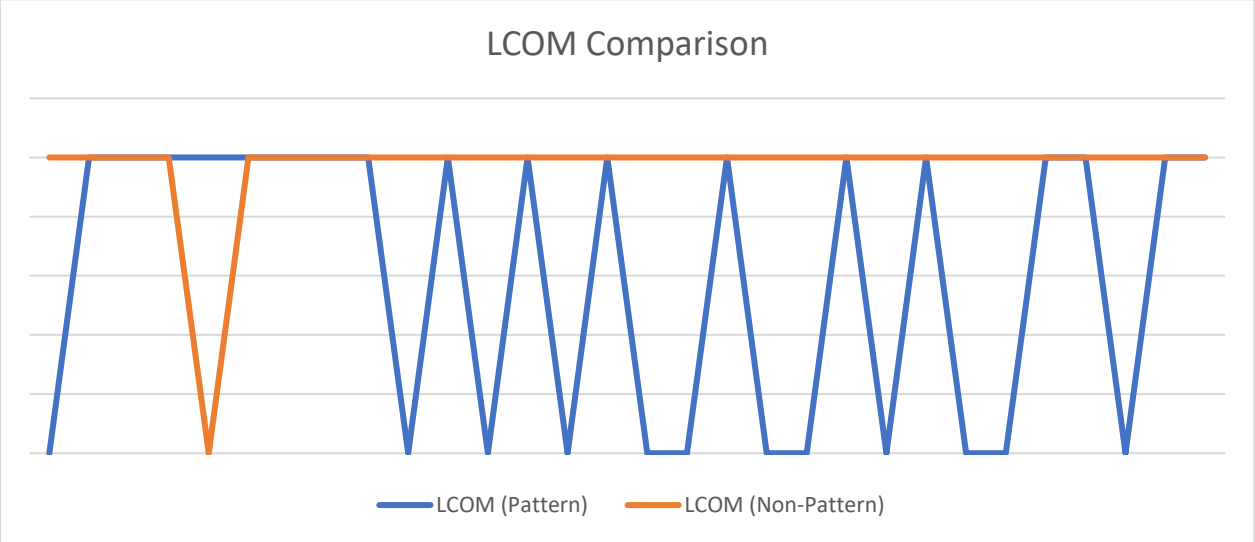

Figure 5:Comparison of LCOM between Pattern and Non-Pattern Classes.

The graph in Figure 5 compares LCOM across pattern and non-pattern classes across projects. Reduced LCOM values indicate stronger class cohesion, whereas larger values indicate reduced modularity.

From the graph, most pattern and non-pattern classes have LCOM scores of 1, showing method coherence. Except for the Aircraft-Modelling-System and WholeSaleMavenSpringJPA projects, pattern classes have an LCOM value of 0, suggesting stronger cohesion than non-pattern classes.

E. Answering Research Questions

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

We can study the testability metrics (CBO, LCOM, DIT, and LOC) for software components that use design patterns and those that don't. We can determine how design patterns affect software testability

by comparing data across pattern-based and non-pattern-based components.

Pattern-based components have lower CBO values than non-pattern-based components, as seen in the graphs above. This shows that design patterns may minimize coupling, which might improve software testability. Lower coupling means fewer software object interactions, making testing and maintenance easier.

LCOM values are similar for pattern-based and non-pattern-based components. However, pattern-based components might have LCOM values of 0, indicating great cohesion. This shows that design patterns can increase cohesiveness, which can improve testability by making certain functionality easier to separate and comprehend.

Pattern-based components have greater DIT values than non-pattern-based components. This suggests that inheritance hierarchies in design patterns may affect testability. Higher DIT values might complicate testing by propagating parent class changes to derived classes.

Software components vary in LOC (Lines of Code). LOC alone does not distinguish between pattern-based and non-pattern-based components. Design patterns affect testability in respect to LOC, but individual code segments need to be examined.

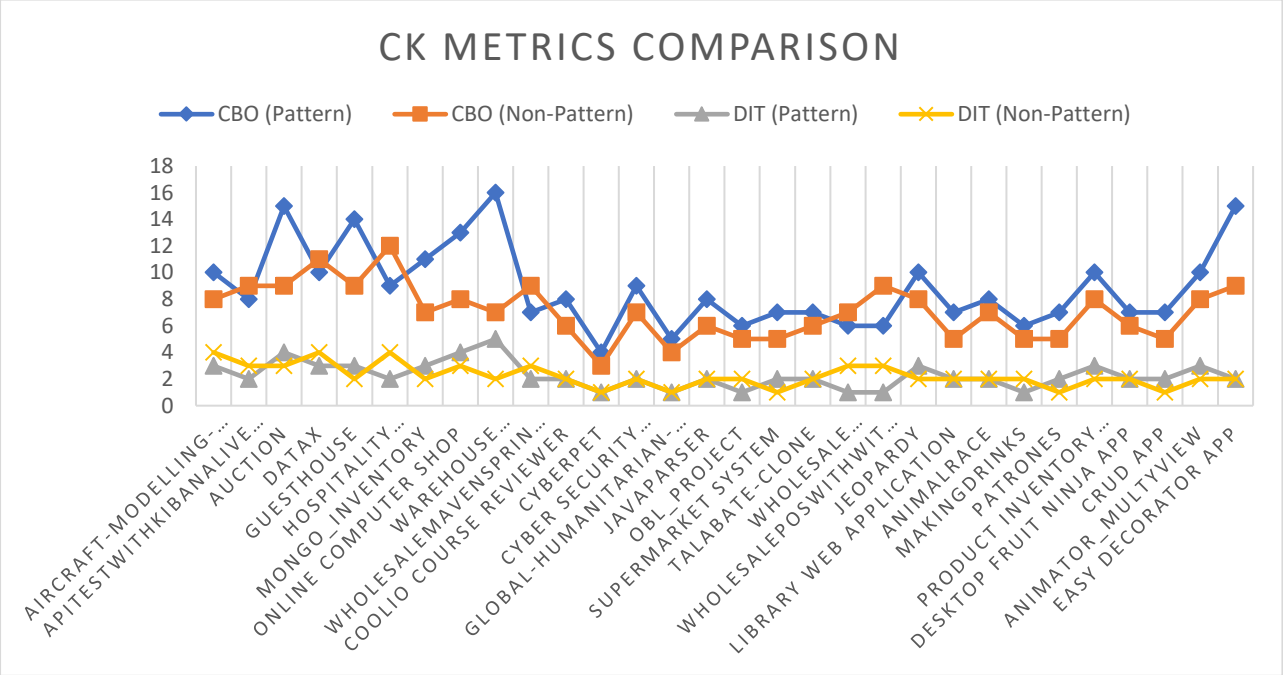

Figure 6:Comparison of Coupling Between Objects (CBO) and Depth of Inheritance Tree (DIT) between Pattern and Non-Pattern Classes.

The graph in Figure 6 shows project CBO and DIT values for pattern and non-pattern classes. High CBO values suggest object coupling. DIT measures inheritance tree depth, with higher values suggesting more complicated inheritance structures.

Pattern and non-pattern classes have comparable CBO values, while projects vary. Auction and warehouse management projects have higher CBO values than non-pattern classes. This shows pattern classes in these projects may have additional dependencies, affecting testability.

Some projects, like Warehouse management, have pattern classes with higher DIT values, reflecting a more complicated inheritance tree. Complex inheritance hierarchies can raise dependencies and side effects, affecting testability.

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

Pattern-based and non-pattern-based software components can be compared for testability differences.

According to the data, pattern-based components had lower CBO values than non-pattern components. This shows that pattern-based components may have decreased coupling and higher testability.

LCOM values indicate that pattern-based and non-pattern-based components may have similar cohesion. Pattern-based components with LCOM values of 0 have higher cohesion, improving testability.

Pattern-based components have greater DIT values. Design patterns may create inheritance hierarchies, which may compromise testability owing to dependencies and change propagation.

Pattern-based and non-pattern-based components cannot be distinguished by LOC values. Testability variations due to LOC require additional exploration and analysis.

F. Comparison of Testability by Design Pattern Type

This section compares creational, behavioral, and structural design pattern testability of software components. This data supports our research on design patterns and software testability using CK metrics.

i. Creational design patterns

Creational design patterns emphasize object creation. Our investigation found Singleton, Factory Method, and (Object)Adapter design patterns. These patterns provide flexible and controlled object creation.

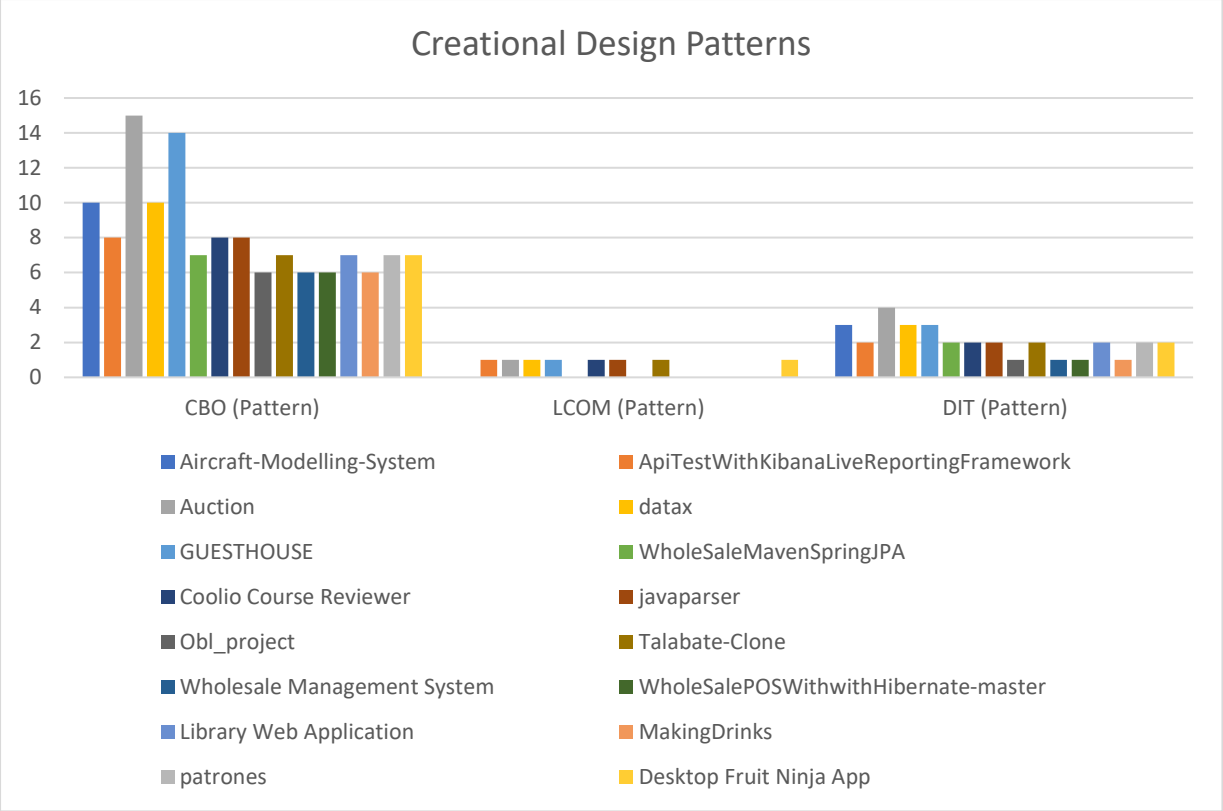

Figure 7: Comparison of CK metrics for creational design patterns.

This graph in Figure 7 shows that many projects employ Singleton and Factory Method patterns to govern object generation and centralize instance management. These patterns have moderate CBO values, indicating moderate object coupling. However, they have low LCOM scores, suggesting stronger pattern class cohesion. Most projects have low DIT values, suggesting a shallow inheritance hierarchy.

These creational patterns indicate that the projects have ordered object creation and explicit roles. By defining component boundaries and making mock objects for unit testing easier, this helps improve testability.

ii. Behavioral design patterns

Behavioral design patterns emphasize object-object communication. In our investigation, we found as shown in Figure 8 the Observer design pattern in the Auction program and the (Object)Adapter design pattern in the GUESTHOUSE and Online computer shop applications.

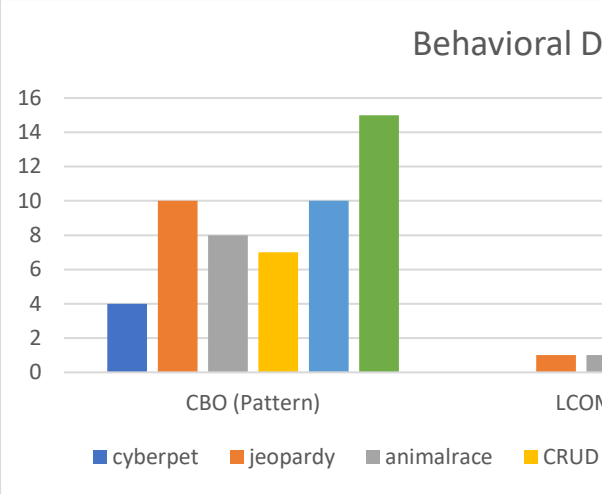

Figure 8:Comparison of CK metrics for behavioral design patterns.

This graph in Figure 8 shows Observer and State patterns are used in several applications. State lets an object change its behavior when its internal state changes, whereas Observer defines a one-to-many dependent connection for loose coupling. These patterns improve object dependency and CBO values. They have moderate to low LCOM scores, suggesting higher pattern class cohesion. Projects have different DIT values.

These behavioral patterns show that projects are flexible and dynamic. This allows test cases to encompass diverse states and transitions, guaranteeing complete system behavior testing.

iii. Structural design patterns

Structural design patterns focus on putting together objects to generate larger structures. The Aircraft-Modelling-System and GUESTHOUSE applications used structural design patterns like Decorator and (Object)Adapter.

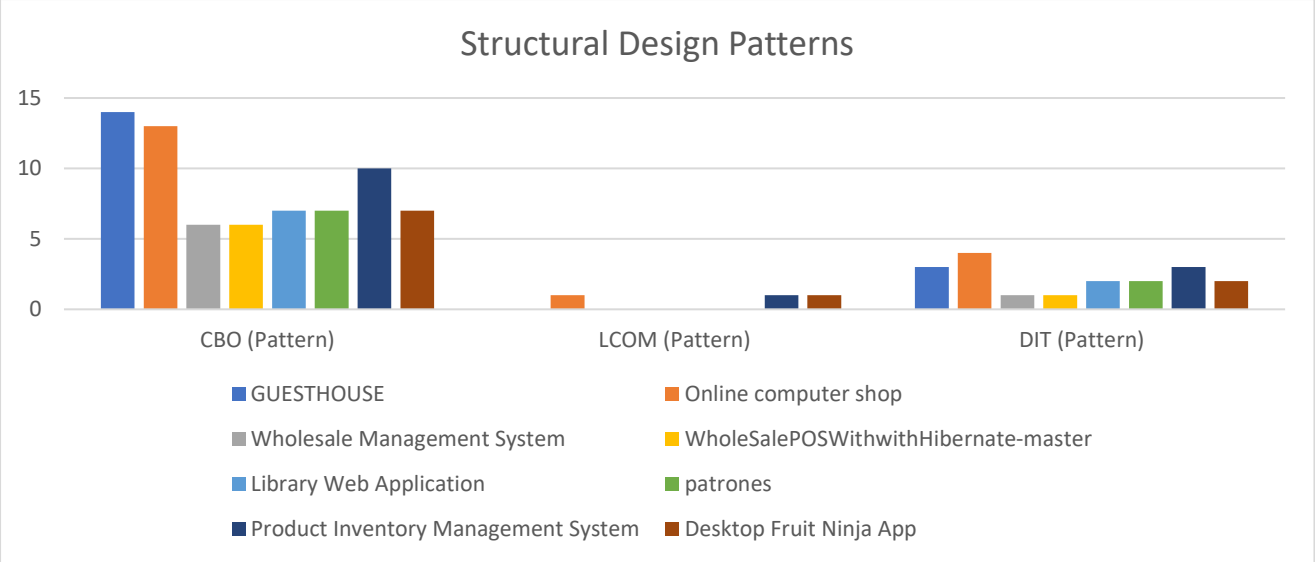

Figure 9: Comparison of CK metrics for structural design patterns.

The trends in Figure 9 show that certain projects employ (Object) Adapter and Singleton patterns. The (Object) Adapter design allows incompatible interfaces to operate together, whereas Singleton assures a class has only one instance. These patterns contain moderate to high CBO values, indicating object coupling. Projects have different LCOM levels and low DIT values.

These structural patterns indicate that projects control component interactions and ensure unique instance generation. Clear interfaces for testing interactions and reducing dependencies can improve testability.

CBO, LCOM, and DIT are important testability indicators for software components. The results show that incorporating these parameters during software development improves testability.

G. Implications for Software Development and Testing Practices

Our research suggests many software development and testing practices:

i. *Understanding the Impact of Design Patterns:*

We quantified how design patterns affect software testability. We learn about design pattern impacts by comparing testability metrics for pattern-based and non-pattern-based components.

ii. *Consideration of Testability Metrics:*

iii. *Evaluation of Creational Design Patterns:*

Creational design patterns like Singleton and Factory Method may not affect testability metrics, according to our research. This information can help practitioners decide whether to use certain

design patterns without compromising software testability.

iv. *Consideration of Behavioral Design Patterns:*

Observer and other behavioral design patterns may affect class coupling and testability. These patterns enhance coupling, hence it's vital to consider testing methodologies to assure complete test coverage.

v. *Benefit of Structural Design Patterns:*

Structural design patterns like Decorator and (Object)Adapter may improve testability. These patterns increase cohesiveness and modularity, making software components easier to test and maintain.

We also found other data that provide insight into pattern class vs. non-pattern class comparisons. The table shows the numbers of pattern and non-pattern classes and their CBO and LCOM values. This data helps explain how design patterns affect various CK metrics and support software development and testing strategies.

IV. Threats to Validity

A. Internal Validity

Testability metrics may be affected by variables other than design patterns. These characteristics, such as project size or complexity, can mislead the link between design patterns and testability.

We carefully chose and matched pattern-based and non-pattern-based components depending on project parameters to handle any complicating influences. This method minimized external influences and improved our findings' internal validity.

B. External Validity

Our study's conclusions may be restricted to the projects and design patterns evaluated. Generalizing the results to other software systems or design patterns should be done with caution.

It's essential to understand that our selection of projects and design patterns may not reflect the complete software development system. The external validity of our study should consider the potential biases in project and pattern selection.

V. Conclusion

We used CK metrics in software systems to evaluate how design patterns affect software testability. Our goal was to determine how design patterns affect testability and help practitioners choose design patterns.

Our investigation evaluated pattern-based and non-pattern-based components' testability metrics, including coupling (CBO), lack of cohesion of methods

(LCOM), depth of inheritance tree (DIT), and lines of code (LOC). We found that pattern-based components had lower CBO values, indicating decreased coupling and more modularity. Pattern-based components had somewhat lower LCOM values, suggesting stronger cohesiveness.

We also examined testability measures and found connections between CBO and LCOM in pattern-based and non-pattern-based components. This shows that design patterns may increase testability by balancing coupling and cohesion.

Design patterns reduce coupling and increase modularity, improving testability, according to our findings. Design patterns may improve testability based on the software system's context and characteristics.

We avoided validity threats by carefully selecting and analyzing components. Our results are relevant but may not apply to all software systems and development settings.

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, *Object-oriented 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 object-oriented 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/ck (accessed Jun. 27, 2023).