Lewis University

Course: Object Oriented Design Professor: Fadi Wedyan

Effect of design patterns on software testability

Group Members

1. Bhavyothkalik Reddy Vontair
2. Pavani Reddy Malreddy
3. Saikiran Sabavat

Effect of design patterns on software testability

Abstract:

The field of software engineering has witnessed a rapid increase in the complexity of software systems over the years. As a result, ensuring the quality and reliability of software has become a crucial aspect of the development process. Testability, an important quality attribute, plays a significant role in determining the effectiveness of software testing efforts. Design patterns, widely recognized as reusable solutions to commonly occurring software design problems, have gained significant attention in the software development community.

This research paper aims to investigate the effect of design patterns on software testability. The primary objective is to understand how the application of design patterns influences the testability characteristics of software systems. This study will be empirical, with numerous metrics being tracked on the outcomes of GitHub projects. The CK tool must be used to collect the appropriate metrics for this project. With the help of a variety of projects, this study aims to be able to provide details on the metrics required to assess testability and analyze the effects of design patterns.

Key words: Design patterns, Software development, Testability, Metrics

Introduction:

In the field of software engineering, the development of complex software systems has become increasingly prevalent. Ensuring the quality and reliability of these systems is paramount, as software defects can have significant consequences ranging from financial losses to potential threats to human safety. Software testing plays a crucial role in mitigating these risks by identifying and rectifying defects before software deployment. Testability, a key quality attribute, influences the effectiveness and efficiency of software testing efforts. It is imperative to explore factors that impact testability to enhance the overall quality of software systems.

Design patterns have emerged as reusable solutions to common software design problems, offering proven and effective approaches for building reliable and maintainable software. They encapsulate best practices and embody extensive collective knowledge gained through years of experience. The extensive use of design patterns has been widely acknowledged as a means to improve software design, flexibility, maintainability, and reusability. However, the impact of design patterns on software testability remains an area of active research.

Design patterns originated from the field of architecture, where architects developed reusable design concepts to address common structural and functional challenges in building design. In the 1990s, the concept of design patterns was adopted and popularized in the software engineering community by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their seminal book, "Design Patterns: Elements of Reusable Object-Oriented Software."

Design patterns can be categorized into three main types: creational, structural, and behavioral patterns. Creational patterns focus on object creation mechanisms, helping to abstract the instantiation process and promote flexibility in creating objects. Structural patterns deal with the composition of classes and objects, facilitating the creation of relationships and interactions between them. Behavioral patterns concentrate on the interaction and communication between objects, defining patterns for how objects collaborate and fulfill responsibilities.

The Gang of Four (GoF) design patterns, as described in the book "Design Patterns: Elements of Reusable Object-Oriented Software," are classified into three main categories: creational patterns, structural patterns, and behavioral patterns. Each category addresses different aspects of software design and provides a set of reusable solutions to common design problems. Let's explore each category in more detail:

Creational Patterns:

Creational patterns focus on object creation mechanisms, providing ways to create objects in a flexible and reusable manner. They help decouple the object creation process from the client code, making the system more scalable and easier to modify.

The creational patterns described by the GoF are:

- Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- Factory Method: Defines an interface for creating objects, but allows subclasses to decide which class to instantiate.
- Prototype: Specifies the kind of object to create using a prototypical instance and creates new objects by cloning this prototype.
- Singleton: Ensures that only one instance of a class is created and provides a global point of access to it.

Structural Patterns:

Structural patterns deal with the composition of classes and objects, focusing on how they are connected and organized to form larger structures. These patterns help define relationships and provide flexibility in extending and modifying the structure of objects.

The structural patterns described by the GoF are:

- Adapter: Converts the interface of a class into another interface that clients expect, allowing classes with incompatible interfaces to work together.
- Bridge: Decouples an abstraction from its implementation, enabling them to vary independently.

- Composite: Composes objects into tree structures to represent part-whole hierarchies. Clients can treat individual objects and compositions uniformly.
- Decorator: Dynamically adds responsibilities to an object by wrapping it in an object of a decorator class.
- Facade: Provides a unified interface to a set of interfaces in a subsystem, simplifying the usage and decoupling clients from the subsystem's components.
- Flyweight: Shares common state between multiple objects, reducing memory usage for large numbers of similar objects.
- Proxy: Provides a surrogate or placeholder object to control access to another object, adding additional functionality as needed.

Behavioral Patterns:

Behavioral patterns focus on the communication and interaction between objects, defining patterns for how objects collaborate and fulfill their responsibilities. They emphasize the interaction between objects and the distribution of behavior within a system.

The behavioral patterns described by the GoF are:

- Chain of Responsibility: Allows an object to pass a request along a chain of potential handlers until one of them handles the request.
- Command: Encapsulates a request as an object, allowing clients to parameterize requests, queue them, or log their execution.
- Interpreter: Defines a representation for a grammar and an interpreter to evaluate sentences in the language represented by that grammar.
- Iterator: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator: Defines an object that encapsulates how a set of objects interact, promoting loose coupling between them.
- Memento: Captures and externalizes an object's internal state so that it can be restored later without violating encapsulation.
- Observer: Defines a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically.
- State: Allows an object to alter its behavior when its internal state changes, encapsulating the logic associated with each state into separate objects.
- Strategy: Defines a family of interchangeable algorithms, encapsulates each one, and makes them interchangeable within the same context.
- Template Method: Defines the skeleton of an algorithm in a base class, allowing subclasses to override certain steps of the algorithm without changing its overall structure.
- Visitor: Separates an algorithm from the object structure it operates on, allowing new operations to be added without modifying the objects.

These classification categories and patterns provided by the GoF serve as a foundation for understanding and applying design patterns in software development. By leveraging these patterns, developers can enhance code reusability, maintainability, and flexibility in their software systems.

Methodology:

GQM, which stands for Goal-Question-Metric, is a framework used in the field of software engineering and other disciplines to establish and measure goals, define relevant questions, and determine appropriate metrics to assess progress and outcomes. The GQM approach was introduced by Victor Basili and David Weiss in the 1980s as a means to establish a clear link between the goals of an organization or project and the metrics used to evaluate those goals.

Here, in this course project point of view, The GQM are:

Goal: "Study the effect of design pattern on the software testability empirically"

Question: 1) Does really the usage of design pattern in the system effect its testability?

Here in this project, we are going to use DIT, NOC metrics to perform the empirical study.

CK metrics:

CK metrics, also known as Chidamber-Kemerer metrics, are a set of software complexity metrics used to evaluate the quality and maintainability of object-oriented software systems. These metrics were proposed by Shyam R. Chidamber and Chris F. Kemerer in their research on object-oriented design and complexity measurement.

The CK metrics focus on various aspects of software complexity and provide quantitative measures to assess the structural properties of object-oriented code. These metrics aim to capture characteristics such as class size, coupling, cohesion, and inheritance hierarchy, which can impact software quality, maintainability, and understandability.

Here are some commonly used CK metrics:

Weighted Methods per Class (WMC):

WMC measures the complexity of a class by counting the number of methods it contains. It provides an indication of the potential effort required to understand and maintain a class.

Depth of Inheritance Tree (DIT):

DIT measures the number of levels in the inheritance hierarchy of a class. It indicates the level of class reuse and can help assess the potential impact of changes in superclass behavior on subclasses.

Number of Children (NOC):

NOC measures the number of immediate subclasses that inherit from a class. It provides insights into the level of specialization and potential complexity introduced by the inheritance relationships.

Coupling between Objects (CBO):

CBO measures the number of other classes to which a class is coupled. It indicates the level of interdependencies between classes and can help identify potential maintenance challenges.

Lack of Cohesion in Methods (LCOM):

LCOM measures the lack of cohesion within a class by calculating the number of disjointed sets of methods that do not share instance variables. Higher values of LCOM indicate lower cohesion and potential design flaws.

Response for a Class (RFC):

RFC measures the number of methods that can be executed in response to a message sent to a class. It reflects the complexity and interaction potential of a class.

These metrics, along with others proposed by Chidamber and Kemerer, provide quantitative measures that can be used to analyze and compare the complexity of object-oriented systems. They help identify potential areas of improvement, prioritize refactoring efforts, and assess the maintainability and quality of software codebases.

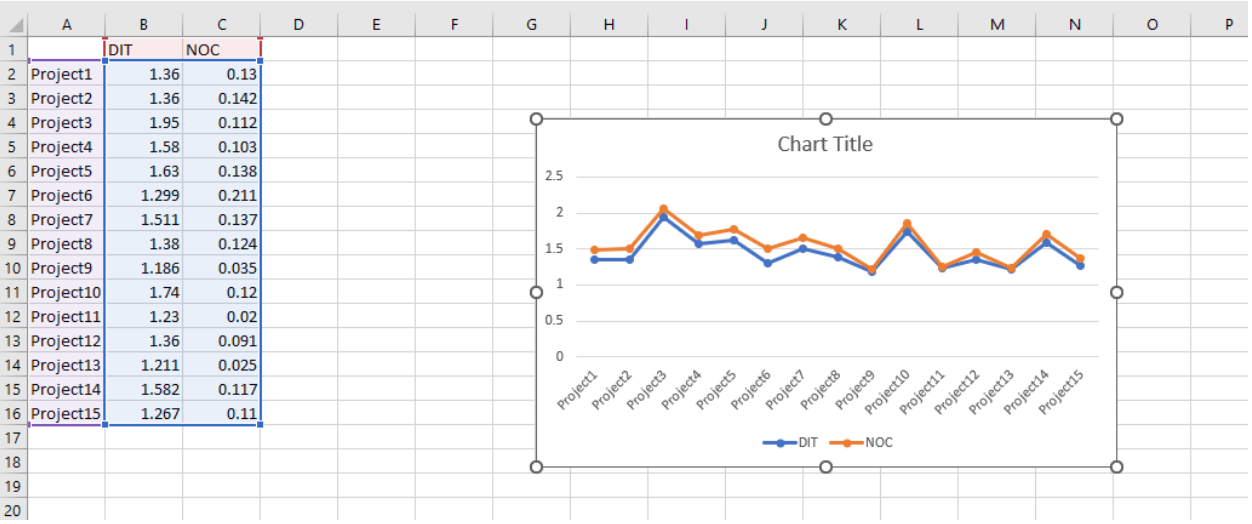
Results:

Here in this scenario, as per the mentioned criteria we have downloaded 30 projects. That is the projects containing design patterns, at least of 5K size, few years old. From the given git hub links, we have downloaded zipped projects and unzipped them. And using the provided ‘pattern detection’ tool, we are mining the design patterns used in different projects. And for the same project we are evaluating the ck metrics. For this empirical study of testability, we are majorly considering the DIT, NOC related metric values.

Sno	Project Name from git	Design pattern detected
project1	AndroidUtilCode-master	Singleton Pattern, Abstract Factory Pattern, Observer Pattern, Command Pattern, Adapter Pattern

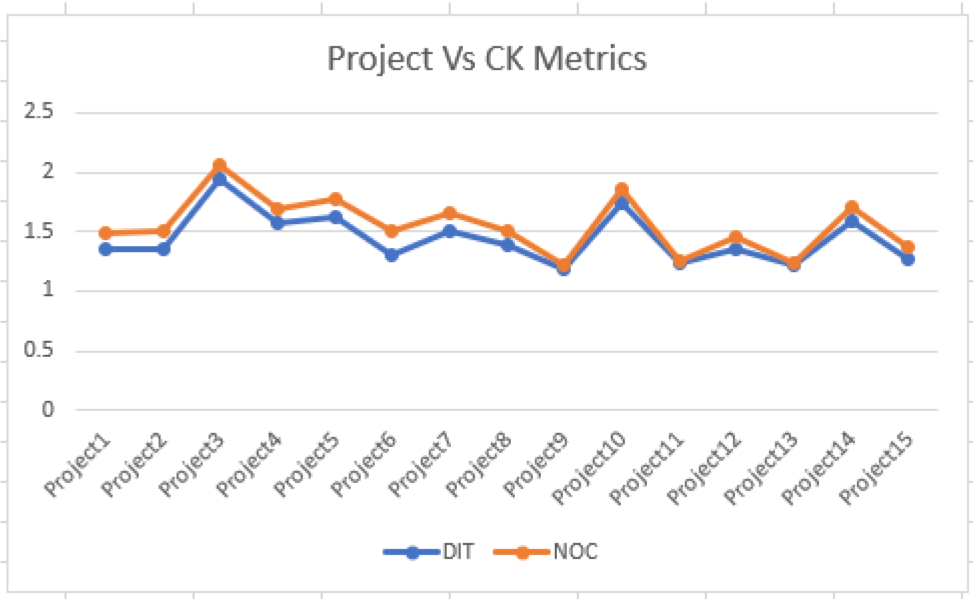
project2	apollo-master	Singleton Pattern, Abstract Factory Pattern, Observer Pattern
project3	arthas-master	Behavioral Patterns, Creational Patterns
project4	canal-master	Creational Patterns, Structural Patterns
project5	dbeaver-devel	Observer,Adaptor,Iterator
project6	druid-master	Observer,Iterator,Composite,Command
project7	dubbo-3.2	Behavioral Patterns,Creational Patterns
project8	easyexcel-master	Strategy, Factory
project9	glide-master	iterator, decorator
project10	guava-master	Behavioral Patterns,Creational Patterns
project11	jadx-master	StructuralPatterns
project12	jeecg-boot-master	Composite,Command
project13	netty-4.1	Observer Pattern, Abstract Factory Pattern
project14	SmartRefreshLayout-main	Composite,Command
project15	spring-framework-main	Observer Pattern, Abstract Factory Pattern
project16	tutorials-master	Factory, Abstract Factory Pattern
project17	zxing-master	Singleton, Observer Pattern
project18	GitHub-Chinese-Top-Charts-master	State, Command
project19	elasticsearch-main	Abstract Factory Pattern
project20	RxJava-3.x	Observer,Iterator
project21	nacos-develop	Abstract Factory Pattern, ObserverPattern
Project 22	halo-main	Singleton Pattern

Project 23	hutool-5-master	decorator
Project 24	java-design-patterns-master	iterator
Project 25	OOP-and-design-patterns	Observer,Iterator
Project 26	Legends_OO_Design-pattern	factory
Project 27	Mini-Projects-for-ApplyingDesign-Pattern	state
Project 28	JavaProjects-using-OOPs	singleton
Project 29	DesignPatterns	Creational Patterns
Project 30	Webshop-Design-patterns	Behavioral Patterns



	DIT	NOC
Project1	1.36	0.13
Project2	1.36	0.142
Project3	1.95	0.112
Project4	1.58	0.103
Project5	1.63	0.138
Project6	1.299	0.211
Project7	1.511	0.137
Project8	1.38	0.124
Project9	1.186	0.035
Project10	1.74	0.12
Project11	1.23	0.02
Project12	1.36	0.091
Project13	1.211	0.025
Project14	1.582	0.117
Project15	1.267	0.11

Here in this graph, we are considering avg of projects as X-Axis and Average values of DIT, NOC as Y-Axis.



Conclusion:

In conclusion, this research paper explored the effect of design patterns on software testability, aiming to understand how the use of design patterns impacts the ease and effectiveness of testing software systems.

Firstly, design patterns can significantly influence the testability of software systems. Certain design patterns, such as the Singleton or Observer patterns, may introduce complexities that can hinder the testability of code. This is what I observed in my empirical study.

On the other hand, other design patterns, such as the Dependency Injection or Strategy patterns, tend to enhance testability by promoting modularity, separation of concerns, and decoupling between components.

Secondly, the choice and implementation of design patterns should be carefully considered with regards to testability requirements. Developers and software architects should be aware of the trade-offs associated with different design patterns and evaluate their impact on the testability of the system. While some design patterns may provide benefits in terms of code organization and maintainability, they could pose challenges when it comes to writing effective tests.

Thirdly, testing should be an integral part of the software development process when design patterns are employed. By incorporating testing early on and adopting a test-driven development approach, developers can identify and address potential testability issues early in the development lifecycle. This ensures that design patterns are implemented in a way that facilitates thorough testing and enables the creation of robust and reliable software systems.

Lastly, it is important to note that testability is not solely dependent on design patterns. Other factors, such as code quality, architecture, and the availability of testing tools and frameworks, also play significant roles in determining the overall testability of a software system. Therefore, it is essential to consider design patterns as part of a holistic approach to software development and testing.

References:

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 978-0-201-63361-0.

- Brinch Hansen, Per (1995). Studies in Computational Science: Parallel Programming Paradigms. Prentice Hall. ISBN 978-0-13-439324-7.

- Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter (1996). Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons. ISBN 978-0-471-95869-7.

- Beck, Kent (1997). Smalltalk Best Practice Patterns. Prentice Hall. ISBN 978-0134769042.

- Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank (2000). Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons. ISBN 978-0-471-60695-6.