ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Observer Design Pattern Applied on Real Life Store Use Case

Nilesh D Kulkarni[1]* and Saurav Bansal[2]

[1]Sr. Director – Enterprise Architecture, Fortune Brands Home & Security,USA

[2]Sr. Manager - Digital Applications, , Fortune Brands Home & Security,USA

ABSTRACT

The paper explores the Observer Design Pattern in the context of a software system. It discusses the significance of design patterns in software engineering, particularly for object-oriented design, emphasizing their role in creating flexible, elegant, and reusable systems. The Observer pattern is specifically examined for its effectiveness in distributed event handling systems, highlighting its utility in decoupling components and facilitating communication between objects. A real-life scenario involving a customer and a store is used to illustrate the application of this pattern, showcasing how it optimizes customer experience and resource management in a retail context. The document also delves into technical aspects like .NET framework, UML basics, and provides C# code examples to demonstrate the practical implementation of the Observer pattern in a software ordering system.

***Corresponding author**

Nilesh D Kulkarni, Sr. Director – Enterprise Architecture, Fortune Brands Home & Security, USA.

Received: April 07, 2022; **Accepted:** April 15, 2022; **Published:** April 25, 2022

Keywords: Design Patterns, Observer, Object, .Net, Software Maintainability

Introduction

The importance of design experience is widely recognized. How often have you encountered a familiar problem during design, sensing that you've tackled something similar in the past, yet struggling to recall the specifics of where and how it was resolved? If you were able to recall the nuances of that past challenge and the strategy you employed to overcome it, you could leverage that previous experience instead of having to re-explore the solution from scratch.

A design pattern represents a universally recognized solution, widely observed in various cases, that effectively addresses a specific problem in a context that may not be predefined. It offers a highly efficient approach to developing object-oriented software that is not only flexible and elegant but also reusable. The utilization of design patterns facilitates the reuse of successful designs and architectural models. By translating proven technologies and methodologies into design patterns, they become more easily accessible to developers building new systems.

Design patterns guide developers in selecting design options that enhance the reusability of a system, while steering clear of choices that could hinder it. Moreover, design patterns can significantly enhance the documentation and maintenance of existing systems by providing a clear and explicit description of class and object interactions, along with their fundamental purposes. In essence, design patterns empower designers to achieve a more effective design more swiftly.

Typically, a design method comprises a set of synthetic notations usually graphical and a set of rules that govern how and when we use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate the design. Each pattern describes a problem which occurs over and over again in the environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice [1].

Design patterns describe problems that occur repeatedly, and describe the core of the solution to that problem, in such a way that the solution can be used many times in different contexts and applications. A good design should always be independent of the technology and the design should help both experience and the novice designer to recognize situation in which these designs can be used and reused.

Eric gamma at el in their book Design Patterns, discussed total 23 design patterns clarified by two criteria figure 1. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the purpose of object creation. Structural pattern deals with the composition of classes or objects. Behavioral pattern characterizes the ways in which classes or objects interact and distribute responsibility [2]. The second criteria called scope, specifies whether the pattern applies primarily to the class or to the object.

Scope	Purpose		
	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figure 1: Design Patterns

UML Basics

The first versions of UML were created by "Three Amigos" - Grady Booch at el defines "The Unified Modeling Language (UML), is a standardized visual language for specifying, constructing, and documenting the artifacts of software systems. It provides a set of diagrams and notations to represent various aspects of software design and architecture, allowing software engineers to communicate, visualize, and model complex systems effectively."

Three Types of Relations Between the Classes

Association Relationship

When classes are connected together conceptually, that connection is called an association. As shown in the Figure 2, let's examine the association between passenger and airplane. A passenger can sit in an airplane or multiple passengers can sit in an airplane.

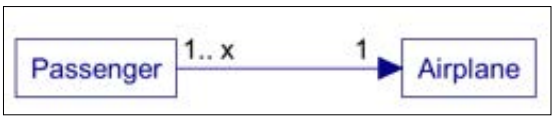

Figure 2: Association Relationship

Aggregation Relationship

This is a special type of relationship, used to model situations where one class (the whole) contains or is composed of other classes or objects (the parts), and the parts have a lifecycle that is independent of the whole. As shown in the figure 3., next examine the aggregation relationship, an engine (whole) can have many Pistons (parts) similarly an airplane (whole) can have multiple engines (parts) as well as an airplane can have multiple wheels (parts).

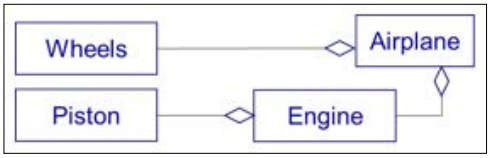

Figure 3: Aggregation Relationship

Composition Relationship

A composition is a strong type of aggregation where each component in the composite can belong to just one whole. As shown in fig 4., a dog can have a tail, four legs, two ears, and two eyes, but eyes, legs, tail, and ears cannot exist on its own.

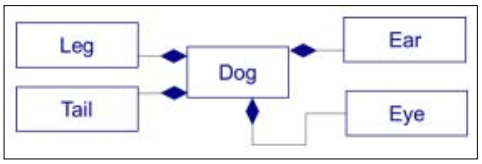

Figure 4: Composition Relationship

Inheritance / Generalization

In this relationship one class (the child class or subclass) can inherit attributes and operations from another (the parent class or superclass). The generalization allows for polymorphism. In generalization, a child is substitutable for parent. That is anywhere the parent appears, the child may appear. The reverse isn't true [3]. As shown in the Fig 5, signifies that "Bus," "Car," and "Truck" inherit from "Vehicle." They are expected to share common characteristics or behaviors that are defined in "Vehicle." For instance, if "Vehicle" has attributes like 'number of wheels' and 'fuel type' and operations like 'start engine ()', then "Bus," "Car," and "Truck" would inherit these operations and attributes.

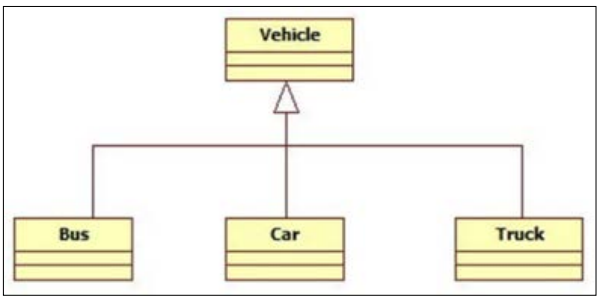

Figure 5: Generization

Interface

An interface is a set of operation that specifies some aspect of classes behavior, and it's set of operation class presents to other classes [3]. As shown in figure 6., the "Electric System" is considered an interface between the light bulb and the light switch. The "Electric System" serves as a contract between the light bulb and the light switch, stipulating that when the switch is turned on, the bulb should light up. Interfaces are used to decouple the implementation and the abstract design, allowing for changes in implementation without affecting the system that uses the interface. Similarly, the light switch and bulb are decoupled from each other, you could replace either the bulb or the switch without needing to change the other, as long as they both adhere to the same electrical system standards. Interface also allows different classes to be treated through a single interface type, the electric system could work with any device that conforms to its standards, not just a light bulb. This could include a fan, a heater, or any other electric device that can be turned on or off.

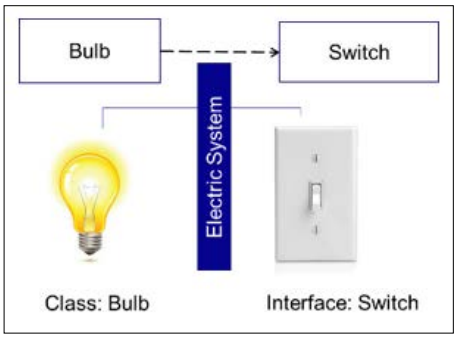

Figure 6: Interface Representation

Programming Technologies
We will using the basic programming tools to show the implementation of the Observer design pattern.

.NET Framework
The .NET Framework, is a software development framework designed and supported by Microsoft. It provides a controlled environment for developing and running applications on Windows. Few features listed below

Windows-Specific
The .NET Framework is designed to work on Windows operating systems.

Base Class Library (BCL)
It includes a large class library known as the Framework Class Library (FCL), providing user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications.

Common Language Runtime (CLR)
Programs written for the .NET Framework execute in a software environment named the Common Language Runtime, which provides services such as security, memory management, and exception handling.

Languages
The .NET Framework supports multiple programming languages, such as C#, VB.NET, and F#.

CLI
Console programming refers to the process of writing software applications that interact with the user through a text-based interface. These applications run in a console or a command-line interface (CLI), where the user inputs text commands and the program provide output in text form.

Visual Studio Code (VS Code) for .NET Development
Visual Studio Code is a lightweight, open-source, and cross-platform code editor developed by Microsoft. It's not specific to any one programming language or framework. With the help of extensions, it can support a wide variety of languages and frameworks, including those of the .NET ecosystem. Few features listed below

Cross-Platform
VS Code runs on Windows, Linux, and macOS.

Extensions
The C# extension by Omni Sharp adds support for .NET development, including features like IntelliSense, debugging, project file navigation, and run tasks.

Lightweight Editor
VS Code is designed to be a fast and lightweight editor, with a smaller footprint than a full IDE like Visual Studio.

Integrated Terminal
Developers can use the integrated terminal to execute .NET CLI commands, enabling them to create, build, run, and test .NET applications.

Git Integration
VS Code has built-in Git support, which is essential for modern software development workflows.

Language Features
VS Code with the C# extension supports advanced language features like code refactoring, unit testing, and code snippets for .NET.

Behavioral Pattern
Behavioral design patterns are a set of design patterns in software engineering that focus on the interaction and communication between different objects and classes in a system. They help in defining how objects collaborate and communicate with each other to achieve a specific behavior or functionality. Behavioral design patterns primarily deal with the delegation of responsibilities among objects and how they interact to accomplish tasks.

Common Behavioral Design Patterns
Strategy Pattern
The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows to select an algorithm or behavior at runtime without altering the client code that uses it. This pattern is useful for providing multiple ways to accomplish a task.

Observer Pattern
This pattern defines a one-to-many relationship between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. It's commonly used in implementing distributed event handling systems.

Command Pattern
The command pattern encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of requests. It also provides support for undoable operations.

Chain of Responsibility Pattern
In this pattern, a request is passed along a chain of handlers. Each handler decides either to process the request or pass it to the next handler in the chain. It's commonly used in implementing event-driven systems like event handling in GUI-Graphical User Interface applications.

State Pattern
The state pattern allows an object to alter its behavior when its internal state changes. It represents various states of an object as separate classes and delegates the state specific behavior to these classes. This pattern is useful when there is an object that needs to change its behavior dynamically based on its internal state.

Behavioral Pattern – Observer
Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. In this pattern, when one object (known as the subject) changes its state, all its dependents (known as observers) are automatically notified and updated. This allows the observers to react to changes in the subject's state without the need for the subject to have direct knowledge of its observers.

Here are the key components and participants in the observer Pattern (figure 7)-
Publisher
Issues events of interest to other objects. These events occur

when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list. When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.

Subscriber
An interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.

Concrete Subscribers
Perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes. Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

Client
Creates publisher and subscriber objects separately and then registers subscribers for publisher updates.
The Observer pattern is widely used in software development to implement distributed event handling systems, decouple components in a system, and facilitate communication between objects in a flexible and loosely coupled way. It promotes the principle of "loose coupling," where the subject and observers are not tightly bound to each other, making the system more maintainable and extensible.

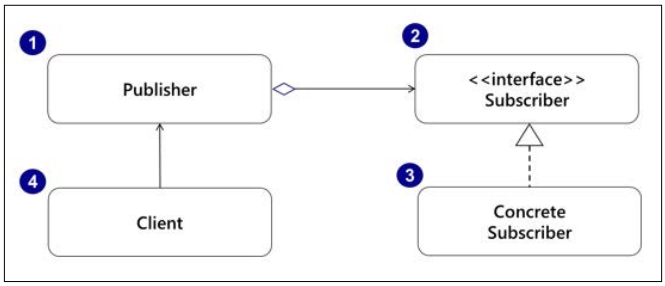

Figure 7: Observer Basic Construct

Real Life Use Case
Imagine a scenario (fig 8)where we have two types of objects: a "Customer" and a "Store." The Customer is eagerly waiting for a product (VR Glasses) which is currently out of stock, and it is expected to become available in the Store very soon.

In this situation, a customer could potentially visit the Store daily to check if the desired product is in stock. However, during the time when the product is still enroute to the Store, most of these visits would be futile and a waste of the Customer's time.

On the other hand, the Store could take the approach of sending numerous emails to all its customers every time a new product becomes available. While this would save some Customers from making unnecessary trips to the Store, but it could also be seen as spam and would inconvenience other Customers who have no interest in the specific product.

To resolve this conflict, the Observer design pattern can be applied-

Subject (Store)
The Store serves as the subject in this scenario. It maintains information about product availability and keeps a list of registered Customers who want to be notified when the desired product arrives.

Observer (Customer)
Each Customer interested in the new VR Glasses becomes an observer. They provide their contact details and preferences to the Store, indicating their interest in receiving notifications about this specific product.

Concrete Subject (Specific Store Location)
In a real-world implementation, there may be multiple Store locations. Each store becomes a concrete subject, managing its product availability and associated observers.

Concrete Observer (Individual Customers)
Each Customer who wishes to be notified about the VR Glasses is a concrete observer. They specify how they want to be notified, whether through email, SMS, or other means.

When the new VR Glasses arrives at the Store, the Store, acting as the subject, checks its list of registered Customers (observers) and sends targeted notifications only to those who have expressed interest in the product. This ensures that Customers are informed about product availability without the need for constant visits, and the Store avoids unnecessary notifications to uninterested Customers, thus optimizing resources and providing a better customer experience.

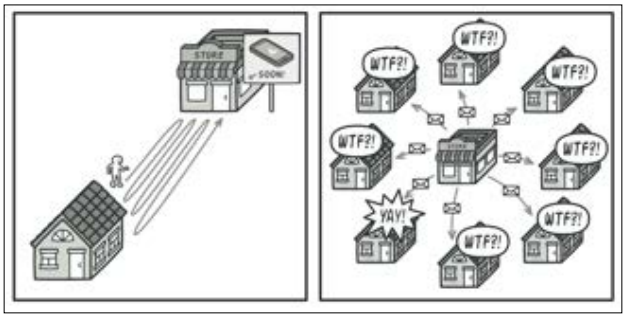

Figure 8: Observer Candidate

Observer Construction
The figure 9. Components applicable based on the observer pattern is explain below -

Publisher (Subject - Store)
- The Publisher class represents the Store, which is the subject in this context. It has a list of 'subscribers', which corresponds to the customers who wish to be notified. The 'mainState' attribute represents the current state of product availability in the store.
- The subscribe(Subscriber s) method allows a new customer to register for notifications.
- The unsubscribe(Subscriber s) method allows a customer to unregister from notifications.
- The notifySubscribers() method is invoked to notify all registered customers about the availability of the product. This is usually done via a loop, as indicated in the pseudo-code foreach (s in subscribers) s.update(this).

- The mainBusinessLogic() method represents the operations the Store may perform, which could include updating the product availability state (mainState).

Subscriber (Observer - Customer)
- The Subscriber interface represents the Customer in this context. Each customer interested in receiving updates about VR glasses implements this interface.
- The update(context) method is a contract that all observers must implement, defining how they will be notified of changes in the Publisher's state.

Concrete Subscribers (Concrete Observer - Individual Customers)
- This represents specific customers. Each customer will implement the update(context) method to receive updates according to their notification preferences (e.g., email, SMS).

Client (Concrete Subject - Specific Store Location)
- The Client represents a particular store location that manages its own stock and customer notifications.
- It's responsible for creating instances of ConcreteSubscribers (customers) and registering them with the 'Publisher' (the store) for updates.

The figure 9 shows the Client section where an instance of ConcreteSubscriber is created and then subscribed to the Publisher. This reflects the action of a customer visiting a specific store location, providing their contact details and preferences, and then being added to the store's notification list for when the new VR Glasses become available.

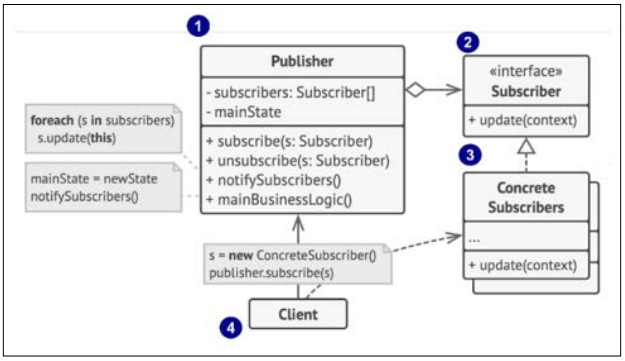

Figure 9: Applying Observer Design Pattern

In the context of the Observer design pattern, the Publisher (Store) does not need to know the specifics of the Subscribers (Customers), just that they implement the Subscriber interface and can be notified through the 'update' method. When the mainState changes (new VR Glasses arrive), the notifySubscribers method is called, which in turn calls the update method on each Subscriber (Customer), passing along the relevant information (context). Each ConcreteSubscriber (Individual Customer) has their own implementation of update (context), which handles the notification based on their specified preference (e.g., email, SMS).

In summary, the Store maintains a list of Customers who want to be notified about the VR Glasses. When the product becomes available, each Store location (acting as a ConcreteSubject) notifies its registered Customers (ConcreteObservers) using their preferred notification method. Each Customer (ConcreteObserver) has a method to update their state or take action based on the notification from the Store (ConcreteSubject) using their preferred notification method. Each Customer (ConcreteObserver) has a method to update their state or take action based on the notification from the Store (ConcreteSubject).

Code Construction

The representation of the code using C# , Visual Studio Code and .Net Framework shown below-

```
using System;
using System.Collections.Generic;

// Observer interface
public interface ICustomer
{
    void Update(ProductAvailabilityInfo availabilityInfo);
}
// Concrete Observer
public class Customer : ICustomer
{
    public string Name { get; set; }
    public string NotificationPreference { get; set; }

    public Customer(string name, string notificationPreference)
    {
        Name = name;
        NotificationPreference = notificationPreference;
    }
    public void Update(ProductAvailabilityInfo availabilityInfo)
    {
                Console.WriteLine($"{Name}, the product '{availabilityInfo.ProductName}' is now available. Notifying via {NotificationPreference}.");
        // Here you would add the logic for sending the notification based on the customer's preference.
    }
}
// Subject interface
public interface IStore
{
    void Subscribe(ICustomer customer);
    void Unsubscribe(ICustomer customer);
    void NotifySubscribers();
}
// Concrete Subject
public class Store : IStore
{
   private List<ICustomer> _customers = new List<ICustomer>();
    private ProductAvailabilityInfo _availabilityInfo;

     public void SetProductAvailability(ProductAvailabilityInfo availabilityInfo)
    {
        _availabilityInfo = availabilityInfo;
        NotifySubscribers();
    }
    public void Subscribe(ICustomer customer)
    {
        _customers.Add(customer);
    }
    public void Unsubscribe(ICustomer customer)
    {
        _customers.Remove(customer);
    }
    public void NotifySubscribers()
    {
```

```
        foreach (var customer in _customers)
        {
            customer.Update(_availabilityInfo);
        }
    }
}

// Information about the product availability
public class ProductAvailabilityInfo
{
    public string ProductName { get; set; }
    public bool IsAvailable { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        // Create store
        var store = new Store();

        // Create customers
        var customer1 = new Customer("John Doe", "Email");
        var customer2 = new Customer("Jane Smith", "SMS");

        // Subscribe customers to the store for notifications
        store.Subscribe(customer1);
        store.Subscribe(customer2);

        // Product availability information
        var availabilityInfo = new ProductAvailabilityInfo
        {
                                ProductName = "VR Glasses",
                                IsAvailable = true
        };
        // Notify customers about product availability
        store.SetProductAvailability(availabilityInfo);
        // Unsubscribe a customer if they no longer wish to receive
notifications
        store.Unsubscribe(customer1);
        // Change product availability and notify again
       availabilityInfo.IsAvailable = false; // Example: the product
is sold out
        store.SetProductAvailability(availabilityInfo);
         Console.ReadLine(); // Wait for user input before closing
the console window
        }
}
```

Design Pattern and Software Maintainability

The original study to evaluate the impact of design patterns on software maintenance was applied by Prechelt et al [4]. They conducted an experiment call PatMain by comparing the maintainability of two implementations of an application, one using a design pattern and the other using a simple alternative. They used four different subject systems in the same programming language. They addressed five patterns - decorator, composite, abstract factory, observer and visitor. The researchers measure the time and correctness of the given maintenance task for professional participants. They found that it was useful to use a design pattern but in case where simple solution is preferred, it is good to follow the software engineer common sense about whether to use a pattern or not, and in case of uncertainty it is better to use a pattern as a default approach.

Conclusion

A design pattern is a generalized reusable solution two commonly occurring problem in a software design. It can be defined as a description or template for how to solve a problem that can be used in many different situations [5]. In this paper, we aim to demonstrate the practical application of the observer design pattern in a specific use case. Design patterns serve as invaluable communication tools and expedite the design process. They empower solution providers to focus on solving the business problem while promoting reusability in the design.

Reusability extends not only to individual components but also to the entire design process, from problem-solving to the final solution. The ability to apply patterns that offer repeatable solutions is well worth the time invested in learning them. There are promising results indicating that the utilization of design patterns enhances quality and contributes to maintainability. The proportion of source code lines involved in design patterns within a system shows a strong correlation with maintainability. However, it's important to note that these findings represent just a small step in the empirical analysis of software quality concerning design patterns. Design patterns should facilitate the reuse of software architecture across different application domains and promote the reuse of flexible components.

References

1. A Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fiksdahl-King I, et al. (1977) A Pattern Language. Oxford University Press, New York.
2. Gamma H (1995) Design Patterns Elements of Reusable Object-Oriented Software https://www.cs.uni.edu/~wallingf/teaching/062/sessions/support/pattern-examples.pdf.
3. Schmuller J (1999) Sams Teach Yourself Uml in 24 Hours https://nibmehub.com/opac-service/pdf/read/Sams%20teach%20yourself%20UML%20in%2024%20hours%20by%20Joseph%20Schmuller%20-A.pdf.
4. Prechelt L, Unger B, Tichy WF, Brossler P, Votta LG (2001) A controlled experiment in maintenance: comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering 27: 1134-1144.
5. Zhang C, Budgen D (2012) What Do We Know about the Effectiveness of Software Design Patterns? IEEE Transactions on Software Engineering 38: 1213-1231.