ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Utilizing the Facade Design Pattern in Practical Phone Application Scenario

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

In this paper, we explored the utilization of the Facade design pattern within the realm of software engineering, with a specific focus on its application in a real-world business context for a phone application. We elucidate how the Facade pattern simplifies intricate subsystems by presenting a unified interface, thereby improving both usability and maintainability. The paper includes an in-depth case study that exemplifies the seamless integration of a new service into an existing phone system, highlighting the tangible advantages of employing the Facade pattern in practical situations. Furthermore, the study delves into the influence of design patterns on software maintainability, underscoring their pivotal role in effective software design and architecture.

***Corresponding author**

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

Received: January 08, 2022; **Accepted:** January 10, 2022; **Published:** January 30, 2022

Keywords: Design Patterns, Facade, 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.

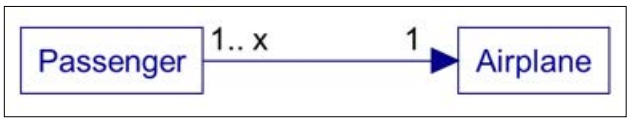

Figure 2: Association Relationship

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).

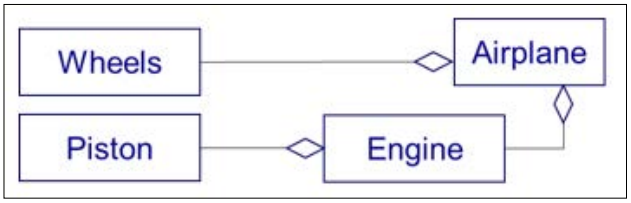

Figure 3: Aggregation Relationship

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 figure 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.

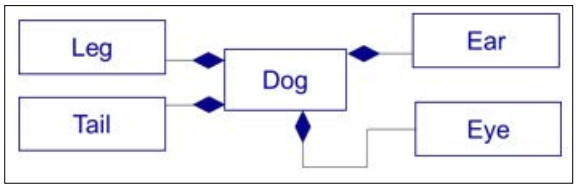

Figure 4: Composition Relationship

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 Figure 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.

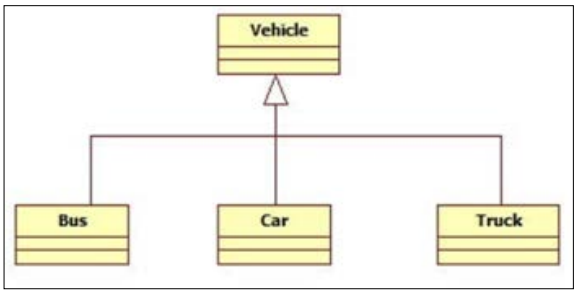

Figure 5: Generalization

Programming Technologies

We will using the basic programming tools to show the implementation of the Facade 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.

Structural Pattern – Facade
Structural design patterns within the realm of software engineering offer effective solutions for addressing design challenges that revolve around the arrangement of classes or objects. These patterns excel in the creation of software systems that are both flexible and amenable to extension. They do so by prescribing methodologies for the assembly of objects and classes, enabling seamless modification of object composition without impinging on their individual implementations.

Moreover, these design patterns advocate for the judicious reuse of pre-existing classes and objects. This practice facilitates the development of software through the harmonization of existing components in diverse configurations. Furthermore, they enhance the overall lucidity and organizational structure of the codebase by delineating explicit relationships and hierarchies among classes and objects.

In addition, structural design patterns effectively encapsulate the intricacies associated with object composition. This encapsulation simplifies the management and upkeep of large-scale software systems. Furthermore, these patterns place emphasis on minimizing the coupling between classes and objects, thereby fostering modularity and maintainability in the codebase...

The Facade Pattern, a structural design patterns in software engineering, offers a solution to a persistent challenge on how to provide a streamlined interface to intricate subsystems composed of classes, interfaces, or objects. This pattern serves as an invaluable tool for hiding the intricate workings of a system, presenting clients with a cohesive, uncomplicated interface. Its biggest objective is to elevate system usability and comprehensibility by diminishing the convolutions that often permeate complex software systems. Key components and characteristics of the Façade pattern are-

Façade
This entity, whether it be a class or an interface, functions as the solitary ingress point to a convoluted subsystem. Its role encompasses the encapsulation of interactions and operations involving myriad classes or objects nested within the subsystem.

Subsystem
The subsystem embodies an assembly of classes, objects, or components, collaboratively engaged in the execution of diverse tasks. Frequently, these classes are intricately interlinked, engendering complex relationships among them.

Client
Within the system's framework, the client assumes the role of the entity that interfaces with the Facade, thereby gaining access to the functionalities embedded within the subsystem.

Phone Application - Use Case
"Rooftop Windows" a window manufacturing local business establishment, install a factory-made window for home and commercial office buildings. The Rooftop Windows is a brick-and-mortar shop, operating from the store, website and a phone system. To allow a seamless customer experience, Rooftop windows has developed a phone application with seven different phone numbers (shown in figure 6) which allows the customers to order on phone, get delivery update, pay on phone, request packaging, order supplies, find out the taxes, estimated warehouse delivery dates.

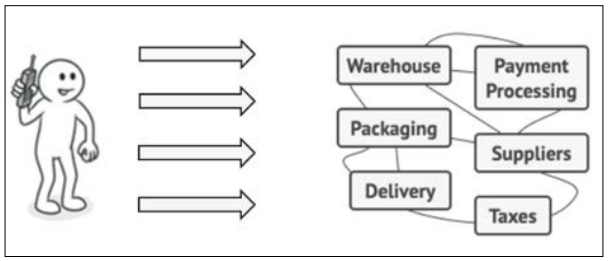

Figure 6: Phone Application

After operating the business for approximately one year and considering the consistent requests from customers to physically inspect the windows before making a purchase, the owner of this local establishment has made the decision to introduce a new service called "Schedule Showroom Appointment." This addition presents a significant challenge for the application developer, as it involves incorporating a new functionality into the existing phone system.

The developer has two options to choose from-

- Create a new phone number dedicated to scheduling showroom appointments.

- Implement a "Facade" design pattern (figure 7) within the phone system, wherein the developer modifies the system to act as an intermediary for all services and departments of the shop. This means that when a customer calls the phone application to schedule a showroom appointment, the system can seamlessly guide them through any of the other seven services or connect them with various departments of the shop.

In essence, the phone application offers a user-friendly voice interface for accessing the ordering system, payment gateways, and the newly introduced appointment scheduling service, ensuring a smooth and convenient experience for customers.

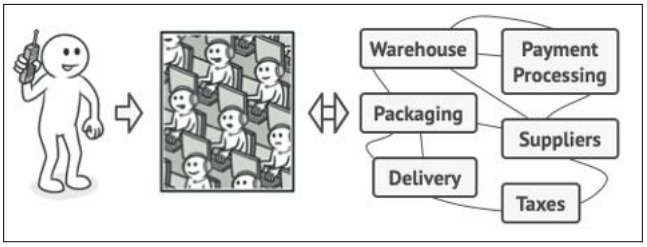

Figure 7: Apply façade to phone system

Design Pattern Application

Design pattern that provides a facade (1) for interfacing with the subsystems (2) and hides the complexity within the subsystems from the client (3) – see figure 8.

1. The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts. The Facade class plays a pivotal role in this design pattern, acting as an intermediary between the client and the subsystem classes. The client interacts with the Facade, which in turn communicates with the various subsystem classes. This interaction is symbolized by the solid lines from the Facade to the subsystem classes. The dashed lines encircling the subsystem classes illustrate the idea that the Facade encapsulates the complexity of the subsystem.
2. The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, your code will have to dive deep into the subsystem's implementation details. Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly. The attributes within the Facade, such as -LinksToSubsystemObjects and -optionalAdditionalFacade, imply that the Facade maintains references to the subsystem objects, and potentially to other facades, thereby facilitating the delegation of client requests. The +subsystemOperation() method represents the unified interface through which the clients can perform operations, where the Facade translates these requests into a set of interactions with the subsystem classes.
3. The **Client** uses the facade instead of calling the subsystem objects directly and would not need to navigate these complexities directly.

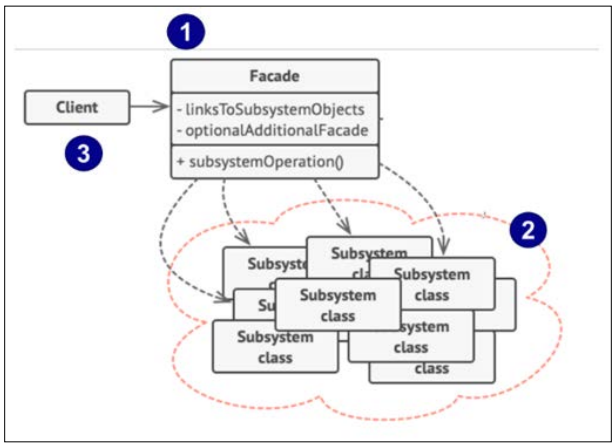

Figure 8: Facade Design Pattern

Code Construction

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

```
using System;
// Define subsystems
public class OrderEntrySystem
{
 public void AcceptOrder()
 {
 Console.WriteLine("Order accepted.");
 }
}
public class PaymentSystem
{
 public void ProcessPayment()
 {
 Console.WriteLine("Payment processed.");
 }
}
public class DeliverySystem
{
 public void ScheduleDelivery()
 {
 Console.WriteLine("Delivery scheduled.");
 }
}

//Add new subsystems called ShowroomAppointmentSystem
public class ShowroomAppointmentSystem
{
 public void ScheduleAppointment()
 {
 Console.WriteLine("Showroom Appointment Scheduling done.");
 }
}
// Define facade
public class Facade
{
 private OrderEntrySystem order;
 private PaymentSystem payment;
 private DeliverySystem delivery;

 //Add new subsystems called ShowroomAppointmentSystem
 private ShowroomAppointmentSystem showroomAppointment;
 public Facade()
 {
```

```
 order = new OrderEntrySystem();
 payment = new PaymentSystem();
 delivery = new DeliverySystem();
 }
 public void AcceptNewOrder()
 {
 order.AcceptOrder();
 }
 public void PayAndScheduleDelivery()
 {
 payment.ProcessPayment();
 delivery.ScheduleDelivery();
 }
 public void ReceiveExistingOrderPayment()
 {
 payment.ProcessPayment();
 }
// Add new method to call ShowroomAppointmentSystem
 public void ScheduleShowroomAppointment()
 {
 ShowroomAppointmentSystem showroomAppointment = new
ShowroomAppointmentSystem();
 showroomAppointment.ScheduleAppointment();
 }
}
public class Program
{
 public static void Main(string[] args)
 {
Facade facade = new Facade();

Console.WriteLine("Select \"A\" for New Order, Select \"B\" for
Payment and Delivery, Select \"C\" for Existing Order Payment,
Select \"S\" for Showroom Appointment Scheduling");
ConsoleKeyInfo cki;
do
{
        cki = Console.ReadKey(true);
        if (cki.Key == ConsoleKey.A)
        {
                facade.AcceptNewOrder();
        }
        else if (cki.Key == ConsoleKey.B)
        {
                facade.PayAndScheduleDelivery();
        }
        else if (cki.Key == ConsoleKey.C)
        {
                facade.ReceiveExistingOrderPayment();
        }
        //Add new else if condition for Showroom Appointment
Scheduling
        else if (cki.Key == ConsoleKey.S)
        {
                facade.ScheduleShowroomAppointment();
        }
        else
        {
                Console.WriteLine("Invalid Input, press esc to
exit !");
        }
} while (cki.Key != ConsoleKey.Escape);
 }
}
```

Design Pattern and Software Maintainability

The original study to evaluate the impact of design patterns on software maintenance was applied by [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 facade 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. 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.