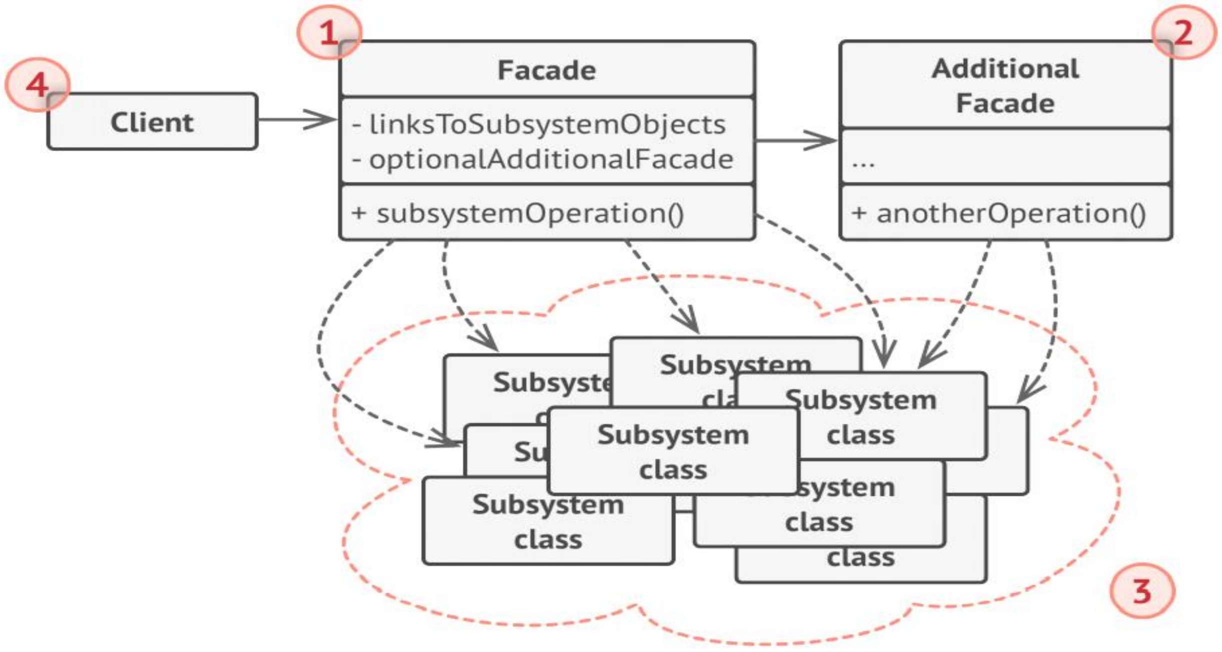
Facade Pattern Implementation Lab Task 1

Introduction & Concept

In this lab task we will learn how to implement the Facade pattern using C#.Net

- Facade provides a simplified interface to a set of interfaces in a subsystem.
- Defines higher level interface makes subsystem easier to use.
- Facade is a simplified interface that performs many other actions behind the scenes.
- Hides complexities of the subsystem from the client and supports loose coupling.
- With this pattern, you can emphasize the abstraction and hide the complex details by exposing a simple interface.
- Makes the code more structured, readable, easy to maintain.
- Very frequently used structural design pattern
- The following UML diagram illustrates the structure of facade pattern.

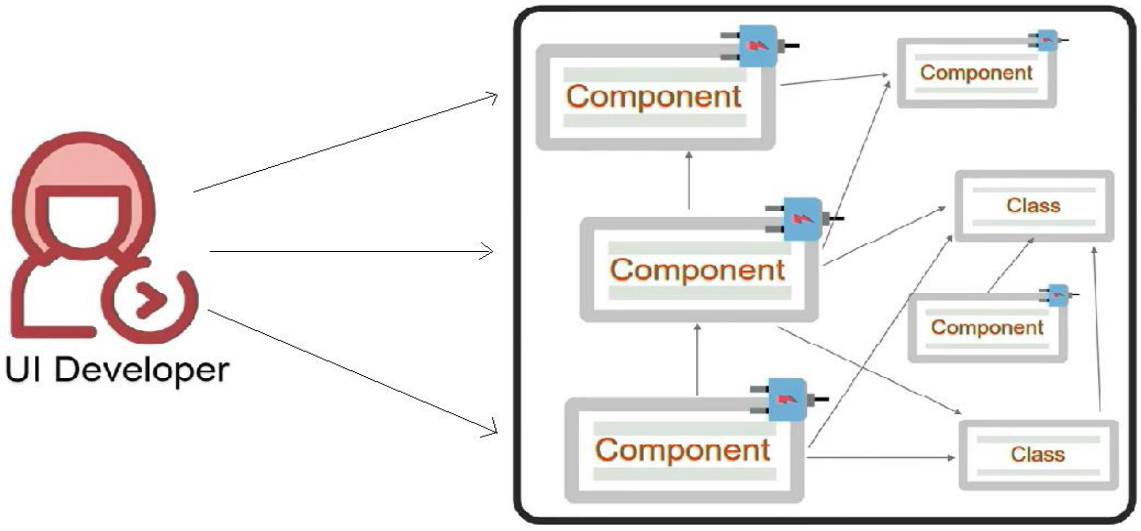


The Problem

In the diagram below the client “UI developer” have access to each of the component in the subsystem. It means that client might be using a subset of the functionality that is exposed through the components. So, direct access to components is the problem and requires to be fixed. In fact, the problem situation depicts the facade design pattern implementation scenario.

In the diagram below notice that UI Developer has access to each of the component.

Client Access Without Facade Implementation



Implementation without facade

1. Create a console application in visual studio and name it FacadeApp.
2. Create a folder and name it BasicFacadeImplementation
3. Create ***Compoent1.CS*** file in this folder and write the following code in it:

```
/// <summary>
/// Component One functionality
/// </summary>
public class Component1
{
    public void Method1()
    {
        Console.WriteLine("Method 1 of the component 1");
    }
    public void Method2()
    {
        Console.WriteLine("Method 2 of the component 1");
    }
    public void Method3()
    {
        Console.WriteLine("Method 3 of the component 1");
    }
    public void Method4()
    {
        Console.WriteLine("Method 4 of the component 1");
    }
}
```

The above created class simulates the functionality of the component. For the simple understanding we created four methods but in a real business use case a component can have a more complex functionality.

In the next step we will create another component like the above one.

4. Create **Component2.*cs*** file in this folder and write the following code in it:

```
/// <summary>
/// Component Two functionality
/// </summary>
public class Component2
```

```
    {
        public void Method1()
        {
            Console.WriteLine("Method 1 of the component 2");
        }
        public void Method2()
        {
            Console.WriteLine("Method 2 of the component 2");
        }
        public void Method3()
        {
            Console.WriteLine("Method 3 of the component 2");
        }
        public void Method4()
        {
            Console.WriteLine("Method 4 of the component 2");
        }
    }
```

We have created another component as similar as the *component1*. This component also has the four methods. The purpose of this simple functionality is to increase the level of understanding by focusing on the concept rather than the business logic.

Next, we will demonstrate how to use the functionality of these components in the client code.

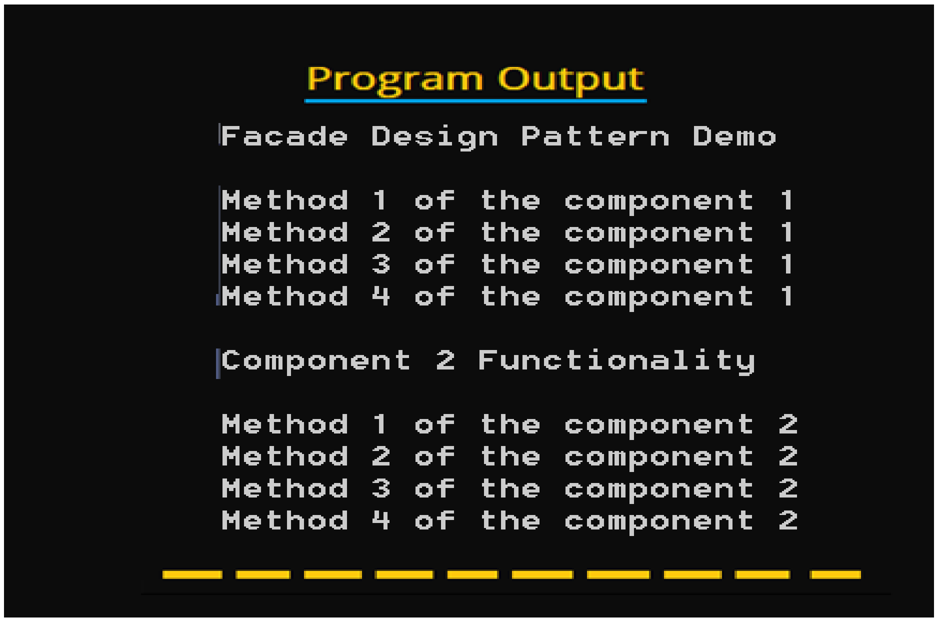
5. Create `ClientDemo.cs` file in this folder and write the following code in it:

```
    public static class ClientDemo
    {
        public static void Main(string[] args)
        {
            Component1 com1 = new Component1();
            com1.Method1();
            com1.Method2();
            com1.Method3();
            com1.Method4();
            Console.WriteLine("Component2 functionality...");
            Component2 com2 = new Component2();
            com2.Method1();
            com2.Method2();
            com2.Method3();
            com2.Method4();
```

```
            Console.ReadKey();
        }

    }
```

Program Output



```
Program Output

Facade Design Pattern Demo

Method 1 of the component 1
Method 2 of the component 1
Method 3 of the component 1
Method 4 of the component 1

Component 2 Functionality

Method 1 of the component 2
Method 2 of the component 2
Method 3 of the component 2
Method 4 of the component 2
```

The above created client has access to all the functionality of the components. It exactly depicts the scenario shown above in the diagram under the heading “Client Access Without facade implementation”.

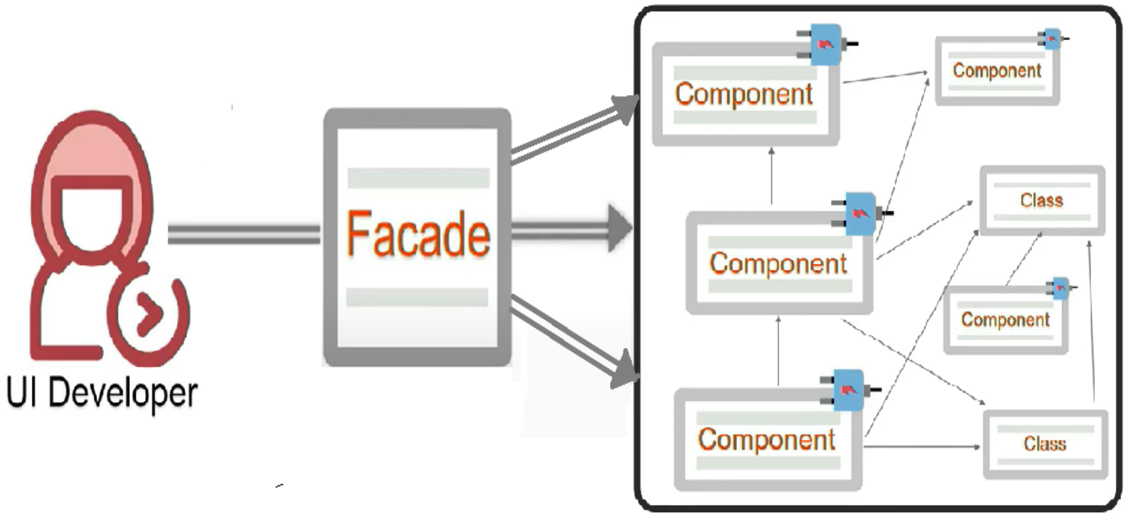
For the demonstration purpose we have called all the methods but assume if some of the functionality of the component is never going to be used by the client, in that case providing access to that functionality is useless and it violates the software engineering design principles.
To provide the solution to the above demonstrated software design problem we are going to demonstrate the implementation of facade design pattern.

The Solution

Now, we have enough understanding of the problem context and usually this scenario is best use case for the facade design pattern.

So next, we will show you how to resolve the above demonstrated problem by implementing the facade design pattern exactly as shown in the diagram below.

Client Access After Facade Implementation



6. Create **IFacade.cs** file in this folder and write the following code in it:

```
/// <summary>
/// A Simplified Façade Interface
/// </summary>
public interface IFacade
{
    //Simplified interface
    void Execute();
}
```

In above interface, we declared a single method ***Execute()*** that returns nothing. In this interface you can simplify abstraction for the client. Basically, it is the place where you can declare restricted functionality to the subsystem the client indirectly interacts with.

Next, we are going to create concrete implementation for the ***IFacade*** abstraction.

7. Create ***Facade.cs*** file in this folder and write the following code in it:

```
/// <summary>
/// Facade Concrete implementation
/// </summary>
public class Facade : IFacade
{
    Component1 com1 { get; set; } = new Component1();
    Component2 com2 { get; set; } = new Component2();
    public void Execute()
    {
        ExecuteComponent1();
        Console.WriteLine("Component 2 Functionality");
        ExecuteComponent2();
    }
    void ExecuteComponent1()
    {
        com1.Method1();
        com1.Method2();
        com1.Method3();
        com1.Method4();
    }
    void ExecuteComponent2()
    {
        com2.Method1();
```

```
            com2.Method2();
            com2.Method3();
            com2.Method4();
        }
    }
```

The above ***Facade*** class implements the ***IFacade*** interface. It provides implementation for its ***Execute()*** method and composes both components ***Component1*** and ***Component2***. The ***Execute()*** method delegates the job to the actual components using the object references *com1* and *com2.*
Simply say, we have created a mechanism through which the client is restricted to access the subsystem.

Next, we are going to demonstrate the usage of the Facade Implementation by creating the following client.

8. Create ***FacadeClientDemo.cs*** file in this folder and write the following code in it:

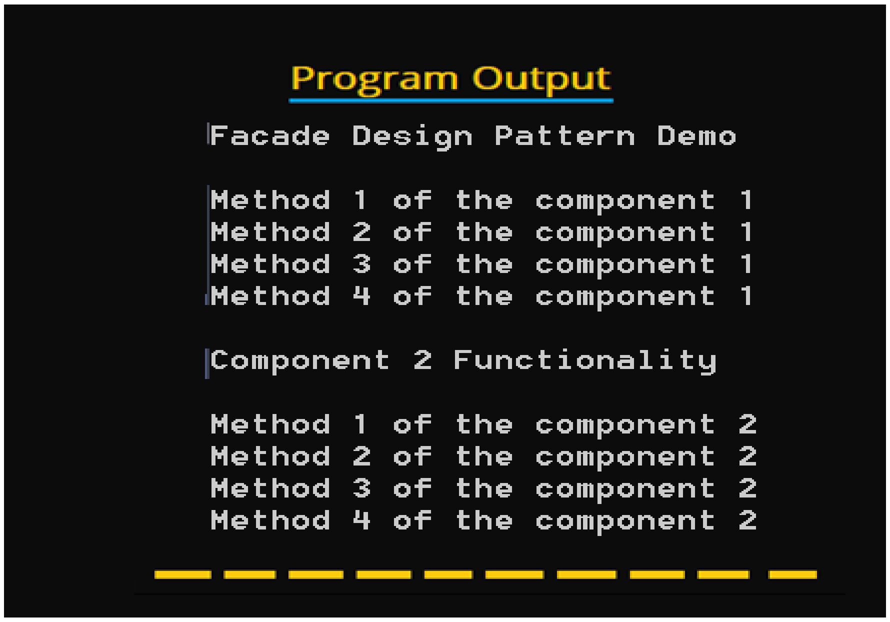
```
/// <summary>
/// Facade implementation usage
/// </summary>
public static class FacadeClientDemo
{
    public static void Main(string[] args)
    {
        IFacade facade = new Facade();
        facade.Execute();

        Console.ReadKey();
    }
}
```

Notice the client code, the facade implementation has reduced the complexity. The client can’t see any details of the subsystem because the facade implementation hides unnecessary functionality that is not required by the client.

Aamir Shabbir Pare

Program Output



```
Program Output

Facade Design Pattern Demo

Method 1 of the component 1
Method 2 of the component 1
Method 3 of the component 1
Method 4 of the component 1

Component 2 Functionality

Method 1 of the component 2
Method 2 of the component 2
Method 3 of the component 2
Method 4 of the component 2
```

You can also compare the program output before and after implementation of the facade. There is no difference of in the output, the result is same but check out the slim and smart code of ***FacadeClientDemo*** client class.

Summary

The main purpose of the Facade Pattern is to provide a simplified interface to the complex subsystem, library, framework, or a component.

In this lab task we have learned what is a facade design pattern and how to implement it using C#. We have also learned how this pattern prefers object composition over inheritance. We composed the components into the concrete facade class to delegate the responsibility. Then we demonstrated the usage of the facade pattern in the client code. The client code became very simple that is one of the advantages of facade design pattern.

Goodbye, wish you all the best and see you in next lab task!