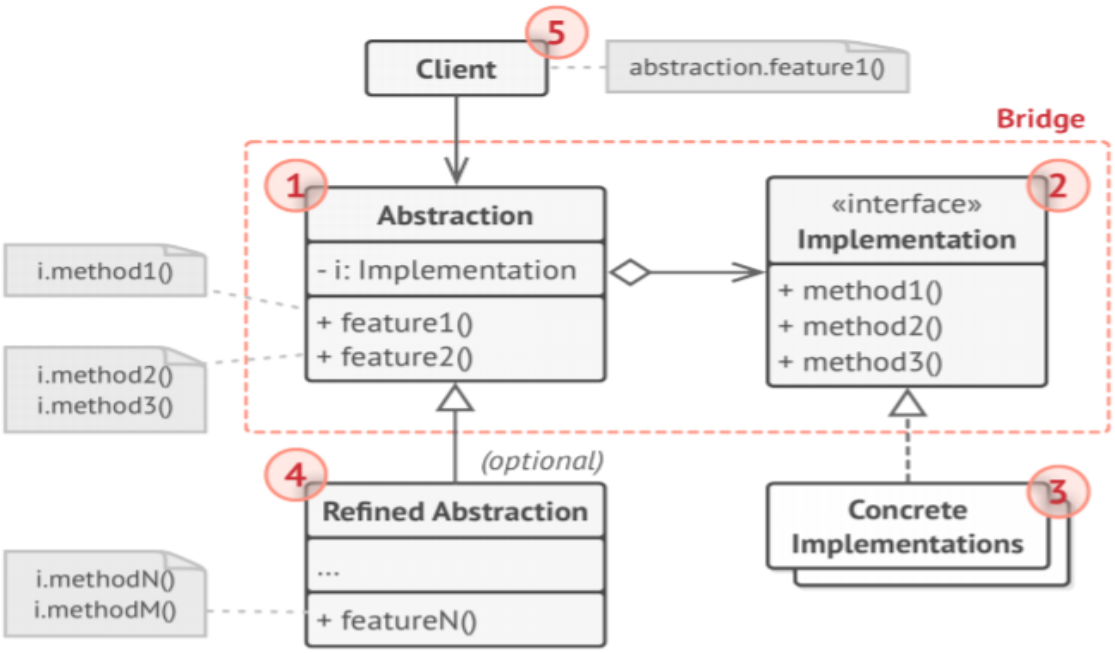
Bridge Pattern Implementation Lab Task 1

Introduction & Concept

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

- Lets you split a large class or a set of closely related classes into two separate hierarchies - abstraction and implementation.
- Decouple an abstraction from its implementation so that the two can vary independently.
- This patten is used to separate an abstraction from its implementation so that both can be modified independently.
- It involves an interface which acts as a bridge between the abstraction class and implementer classes.
- The following UML diagram illustrates the structure of bridge pattern.

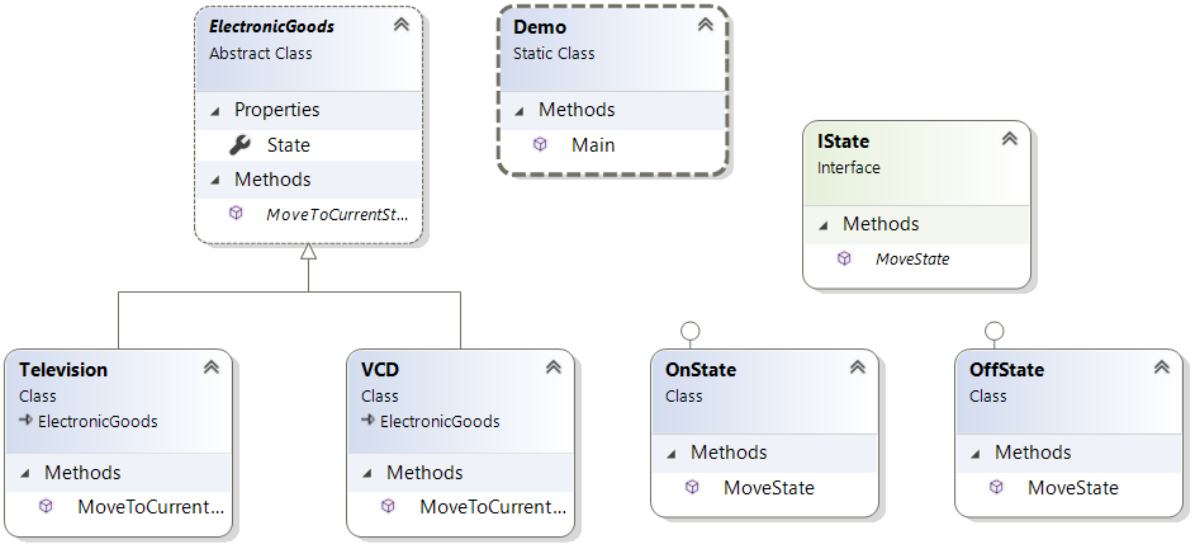


Aamir Shabbir Pare

The Problem

An electronic device manufactures company decided to design a remote control for controlling different devices. They want its initial version to have only two functions ON and OFF and operate for VCD and TV.

Class Diagram



Bridge Design Pattern Implementation Class Diagram

Implementation

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

```
/// <summary>
/// Implementor
/// </summary>
public interface IState
{
    void MoveState();
}
```

The above created interface declares a single method *MoveState ()*. The functionality of implementor to be implemented by the concreted implementors.

In the next step we are going to provide the concrete implementation for the above created interface.

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

```
/// <summary>
/// Concrete Implementor 1
/// </summary>
public class OnState : IState
{
    public void MoveState()
    {
        Console.Write("On State");
    }
}
```

This concreate implementor is responsible to turn the device state on. It implements the *IState* interface to provide the concreted implementation.

Next, we will create another concrete implementor the same *OnState* implementor is created.

5. Create **OffState.cs** file in this folder and write the following code in it:

```
/// <summary>
/// Concrete Implementor 2
/// </summary>
public class OnState : IState
{
    public void MoveState()
    {
        Console.Write("Off State");
    }
}
```

This concreate implementor is responsible to turn the device state off. It implements the *IState* interface to provide the concreted implementation.

We have successfully finished the development for the implementor hierarchy according to the bridge design pattern solution.

Next, we are going to start developing the abstraction hierarchy of classes according to the bridge design pattern guidelines.

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

```
/// <summary>
/// Abstration
/// </summary>
public abstract class ElectronicGoods
{
    //Reference to the Implementor
    public IState State { get; set; } = new OffState();
    public abstract void MoveToCurrentState();
}
```

In above abstract class we prefer composition over inheritance and composed the implementor and set its default state to OffState because it the default behavior of almost every device. We also defined *MoveToCurrentState () asbstract* method that will be redefined according to the device implementation.

Next, we are going to create concrete implementation for redefining abstraction for the Television device.

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

```
/// <summary>
/// Redefined Abstration for Television
/// </summary>
public class Television : ElectronicGoods
{
    public override void MoveToCurrentState()
    {
        Console.Write($"Television is fucntioning at : ");
        State.MoveState();
        Console.WriteLine();
    }
}
```

We inherit from the abstract base class *ElectronicGoods* and redefined the abstraction in it. We overridden the *MoveToCurrentState()* method and execute the *MoveState()* behavior of concrete implementor through IState interface.

Next, we are going to create another concrete implementation for redefining the abstraction for VCD device.

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

```
/// <summary>
/// Redefined Abstration for VCD
/// </summary>
public class VCD : ElectronicGoods
{
    public override void MoveToCurrentState()
    {
        Console.Write("VCD is fucntioning at : ");
        State.MoveState();
        Console.WriteLine();
    }
}
```

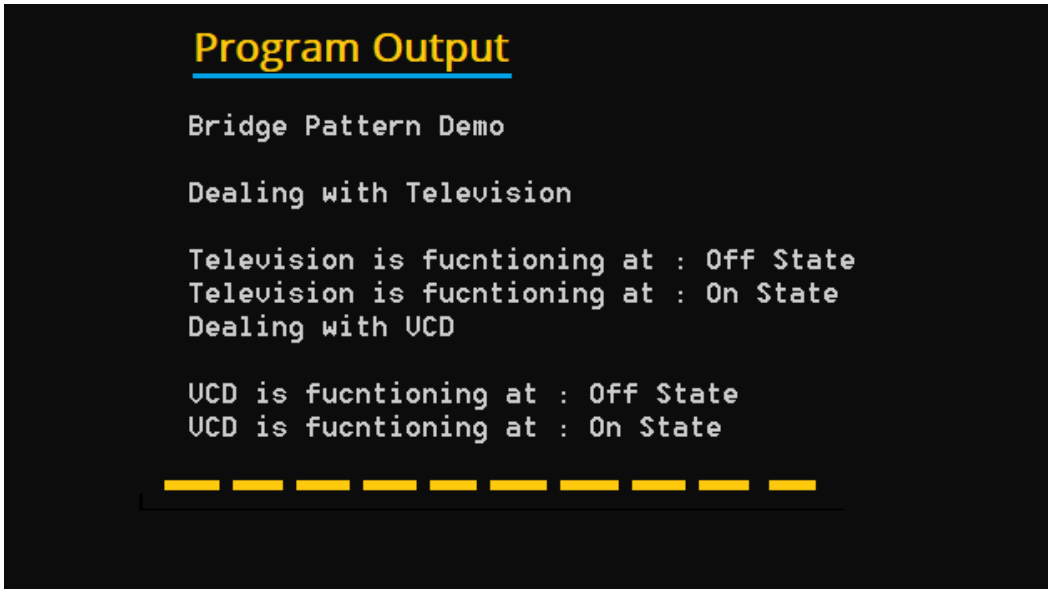
The VCD redefines the abstract base class abstraction the same way it's redefined in the Television class above.

Next, we will create a client class that will show how to use above implementation of the bridge design pattern in the client code.

9. Create ***Demo.cs*** file in this folder and write the following code in it:

```
/// <summary>
/// Client for Bridge Demo
/// </summary>
public static class Demo
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Bridge Pattern Demo\n");
        Console.WriteLine("Dealing with Television\n");

        //Default State if off
        ElectronicGoods device = null;



        device = new Television();
        device.MoveToCurrentState();

        //Change State to On
        device.State = new OnState();
        device.MoveToCurrentState();


        Console.WriteLine("Dealing with VCD\n");
        //Default State if off
        device = new VCD();
        device.MoveToCurrentState();

        //Change State to On
        device.State = new OnState();
        device.MoveToCurrentState();

        Console.ReadKey();
    }

}
```

Program Output



Summary

The main purpose of the Bridge Pattern is to split a large class or a set of closely related classes into two separate hierarchies - abstraction and implementation.

In this lab task we have learned what is a bridge pattern and how to implement it using C#. We have also learned how this pattern prefers object composition over inheritance. We composed the implementor interface to store the concrete implementor reference in abstract base class that defines the abstraction. We redefined the abstractions by providing implementations through concreate classes i.e. Television and VCD.

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