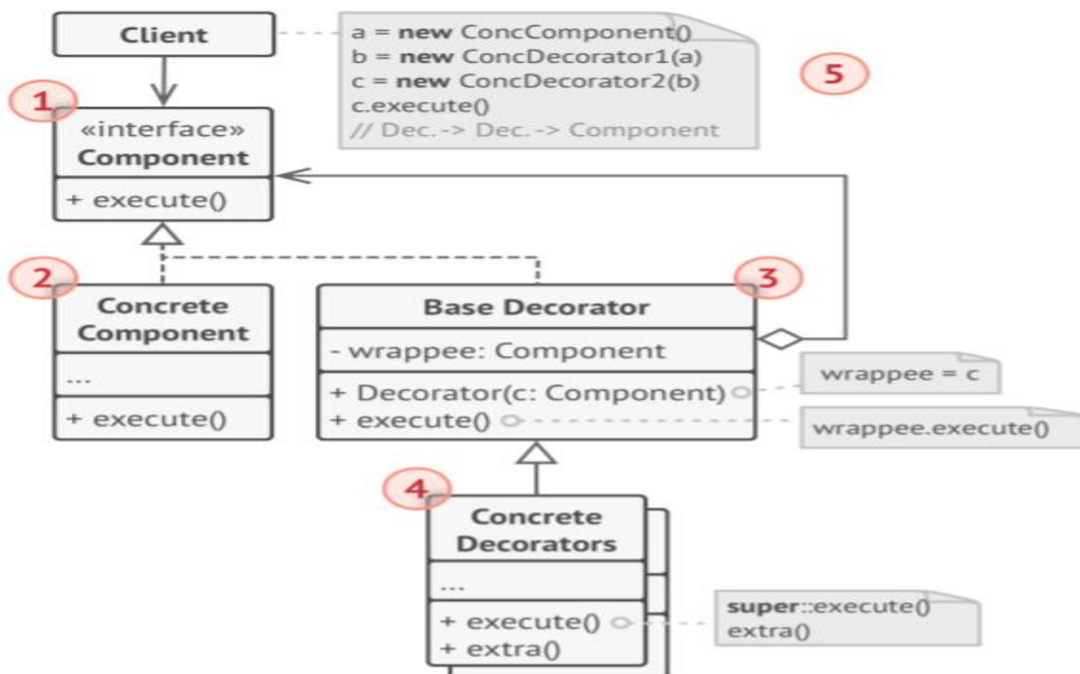
Decorator Pattern Implementation Lab Task 1

Introduction & Concept

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

- Decorator is a structural design pattern that allows us to extend the behavior of objects by placing into a special wrapper class. It allows to attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.
- The Decorator is also known as Wrapper. Both names for the same design pattern can be used interchangeably.
- The composite pattern is a partitioning design pattern, it describes that the group of objects are treated same way as a single instance of the same type of object.
- The concept is useful when you want to add some special functionality to a specific object instead of the whole class. This pattern prefers object composition over inheritance.
- The following UML diagram illustrates the structure of decorator pattern.

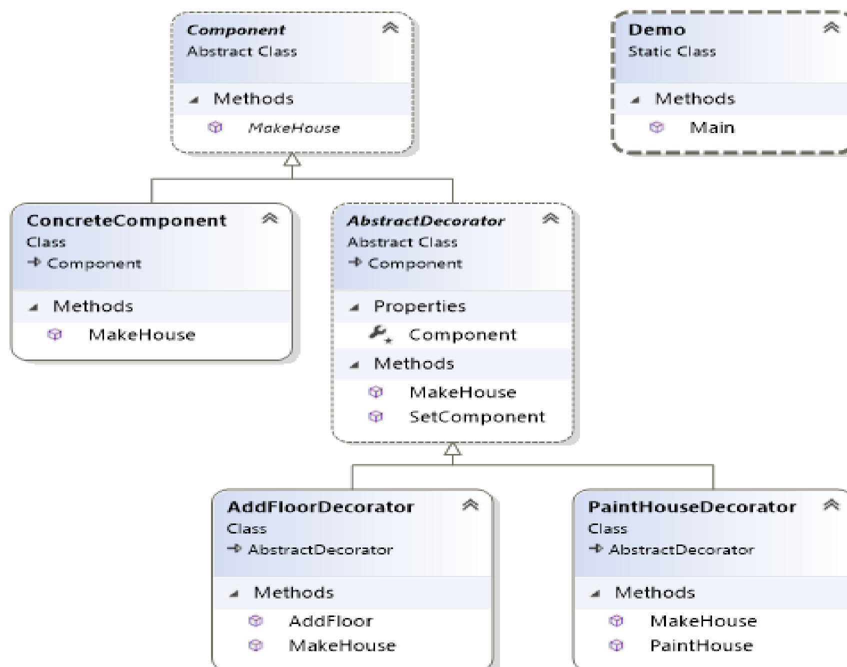


Aamir Shabbir Pare

The Problem

Suppose you own a single-story house, and you decide to build a second floor on top of it. Obviously, you may not want to change the architecture of the ground floor. But you may want to change the design of the architecture for the newly added floor without affecting the existing architecture.

Class Diagram



Basic Implementation of Decorator Design Pattern

Implementation

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

```
/// <summary>
/// Decorator Base functionality
/// </summary>
public abstract class Component
{
    public abstract void MakeHouse();
}
```

The above created abstract class declares a single method *MakeHouse ()*. The base functionality that will be provided by the Concrete Component. The additional functionality will be attached by the decorators.

In the next step we are going to provide the concrete implementation for the above created abstract class Component.

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

```
/// <summary>
/// Implementation for Component
/// </summary>
public class ConcreteComponent : Component
{
    public override void MakeHouse()
    {
        Console.WriteLine("The original house is complete and is
                            closed for modification.");
    }
}
```

Aamir Shabbir Pare

This is the base functionality in decorator that works as a first floor in our example and on top of it we will build another floor using decorators.

Next, we are going to create an abstraction for addition responsibilities to be attached and make use of existing component. Here we will prefer object composition over inheritance.

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

```
/// <summary>
/// Abstraction for decorators
/// </summary>
public abstract class AbstractDecorator : Component
{
    protected Component Component { get; set; }

    public void SetComponent(Component c)
    {
        Component = c;
    }

    public override void MakeHouse()
    {
        if(Component != null)
        {
            Component.MakeHouse(); //Delegating responsibility
        }
    }
}
```

In above abstract decorator we defined the *SetComponent()* method. This will allow to inject the objects that implement the Component abstract class. We also overridden the *MakeHouse()* and execute the Component’s existing behavior.

Next, we are going to create concrete decorators.

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

```
/// <summary>
/// Add Floor Concrete decorator
/// </summary>
public class AddFloorDecorator : AbstractDecorator
```

```
{
    public override void MakeHouse()
    {
        base.MakeHouse();
        Console.WriteLine("***Add floor decorator start
                                working***");
         AddFloor();
         Console.WriteLine("***Add floor decorator finished
                                construction***\n");
    }

    public void AddFloor()
    {
        Console.WriteLine("Making an additional floor on to of the
                                original");
    }
}
```

We inherit form the *AbstractDecorator* and defined the *AddFloor()* method. This is the additional responsibility that we are attaching in this concrete decorator. We also overridden the base decorator *MakeHouse().* In this method we executed the existing base behavior *base.MakeHouse()* and after that we called the *AddFloor()* method.

Next, we are going to create another decorator the same way the *AddFloorDecorator* is created.

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

```
 /// <summary>
/// Paint House Concrete decorator
/// </summary>
public class PaintHouseDecorator : AbstractDecorator
{
    public override void MakeHouse()
    {
         base.MakeHouse();
         Console.WriteLine("***Paiting house decorator start work
                                Ing***");
         PaintHouse();
         Console.WriteLine("***Paiting house decorator finished
                                Painting***\n");

    }
```

```
        public void PaintHouse()
        {
            Console.WriteLine("Now i am painting the house");
        }
    }
```

The above *PaintHouseDecorator* is same as *AddFloorDecorator*. It inherits from the *BaseDecorator* and overrides its *MakeHouse()* and defines additional *PaintHouse()* method. Notice the execution in the overridden *MakeHouse()*, once the house is built the *PaintHouse()* is executed.

Next, we will create a client class that will show how to use the decorators in client code.

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

```
public static class DecoratorDemo
{
    public static void Main(string[] args)
    {
        Console.WriteLine("DECORATOR PATTERN DEMO");
        Console.WriteLine("========================\n");

        //Add a new floor on top of other using decorator
        ConcreteComponent cc = new ConcreteComponent();
        AddFloorDecorator afd = new AddFloorDecorator();
        afd.SetComponent(cc);

        //Paint the house using another decorator
        PaintHouseDecorator phd = new PaintHouseDecorator();
        phd.SetComponent(afd);
        phd.MakeHouse();

        Console.ReadKey();
    }
}
```

Program Output

```
Program Output

DECORATOR PATTERN DEMO
=======================

The original house is completed and is closed for modification.

***Add floor decorator start working***
I am making an additional floor on top of the original
***Add floor decorator finished construction***

***Paiting house decorator start working***
Now i am painting the house
***Paiting house decorator finished painting***
```

Summary

The main purpose of the Decorator Pattern is to promote the OCP (Open-Closed Principle) concept. That means, we can add new functionality without altering the existing functionalities.

In this lab task, a little longer than usual, we have learned what is a decorator pattern and how to implement it using C#. We have also learned how this pattern prefers object composition over inheritance. We wrapped the object into concrete decorators to attach new functionality without altering the existing one made accessible through the object composition in the abstract base decorator.

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