# 32. Decorator

Decorator design pattern allows developers to modify the functionality of existing objects without modifying the objects themselves. Just like the Adapter pattern, Decorator provides a wrapper for an existing object. However, unlike Adapter, it implements exactly the same interface as the original object. This allows you to apply additional processing steps to the original API without modifying its original behavior.

Decorator can be summarized as follows:

- There is a class, which we will refer to as the **Original Object**.
- The **Original Object** implements **Service Interface**.
- There is a **Decorator** class that also implements **Service Interface**.
- Because **Decorator** class acts as a wrapper for **Service Interface**, any implementation of this **Service Interface** can be used, including another **Decorator**.
- **Decorator** can add additional functionality to any public members of its internal **Service Interface** implementation, such as pre-processing or post-processing steps.

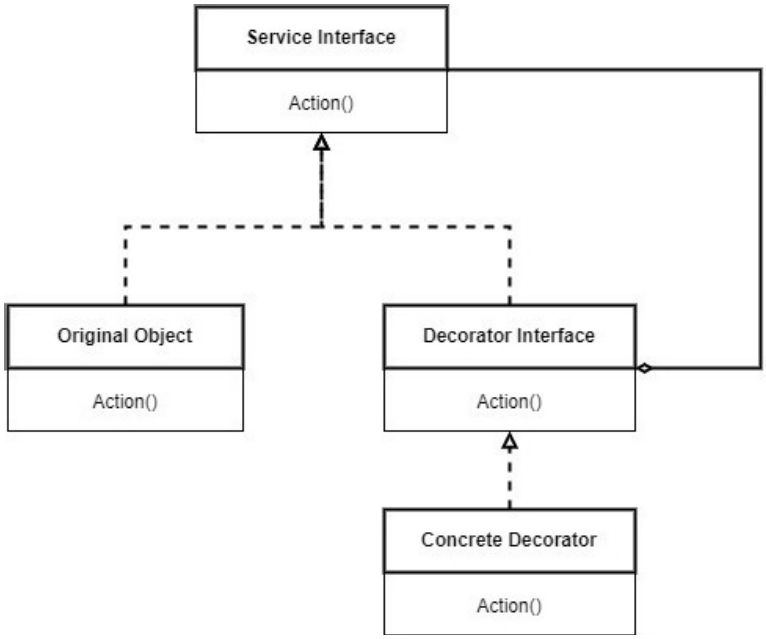


**Figure 32.1 - Decorator UML diagram**

We will now go through an example implementation of Decorator. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Structural_Patterns/Decorator

## Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

# Decorator implementation example

One real-life analogy of Decorator would be a power drill that can have a hammer extension attached to it. The hammer extension doesn't change its base functionality of drilling holes, but it gives the tool the ability to drill through concrete. And this is what we will use in the code as an abstraction.

We will create a .NET console application. Once created, we will add the `IPowerDrill` interface to it, which will be the **Service Interface** used by both our gun object (the **Original Object**) and all of its **Decorators**. The interface will look like this:

```
internal interface IPowerDrill
{
    bool Drill();
    void Recharge();
}
```

We can drill and we can recharge the drill. If the drill fails, (e.g. if it's out of power), the `Drill` method returns `false`. Otherwise, it returns `true`.

And this is what the implementation of our **Original Object** looks like:

```
namespace Decorator_Demo;

internal class PowerDrill : IPowerDrill
{
    private int powerUnits = 0;

    public void Recharge()
    {
        powerUnits = 6;
```

```
10      }
11
12      public bool Drill()
13      {
14          if (powerUnits > 0)
15          {
16              Console.WriteLine("Drill used.");
17              powerUnits--;
18              return true;
19          }
20
21          Console.WriteLine("Out of power.");
22          return false;
23      }
24  }
```

So, we have an integer that represents how much power we have left. Every time we recharge, the integer gets reset to 6. Every time we drill, the power units get decremented by one. If there is any power left at the beginning of the drilling attempt, `true` is returned. Otherwise, if the power unit counter is zero, `false` is returned. In either situation, the outcome is printed in the console.

Now we will start adding some **Decorator** objects. To demonstrate how multiple decorators can be applied, we will create two of them. To help us with this, we will create an abstract class that both of them will implement. This class will look as follows:

```
namespace Decorator_Demo;

internal abstract class PowerDrillDecorator :
    IPowerDrill
{
    private IPowerDrill? powerDrill;

    public void SetPowerDrill(
        IPowerDrill powerDrill)
    {
        this.powerDrill = powerDrill;
    }

    public virtual void Recharge()
    {
        if (powerDrill is not null)
        {
            powerDrill.Recharge();
        }
    }

    public virtual bool Drill()
    {
        if (powerDrill is not null)
        {
            return powerDrill.Drill();
        }

        return false;
    }
}
```

As you can see, the base **Decorator** class implements the same **Service Interface** as our **Original Object**. Now, we will add some implementations to it. First, we will add the `HammerExtension` class, which will look as follows:

```
namespace Decorator_Demo;

internal class HammerExtension :
    PowerDrillDecorator
{
    public override bool Drill()
    {
        if (base.Drill())
        {
            Console.WriteLine(
                "Drill used in hammer mode.");
            return true;
        }

        return false;
    }
}
```

So, we have left the default implementation of the `Recharge` method. But our `Drill` method had some additional functionality added to it. If the shot has been successful, we notify the console that a hammer extension has been used.

Now, we will add another **Decorator**. We will call it `ExtendedBattery`. And this is what it will look like:

```
namespace Decorator_Demo;

internal class ExtendedBattery :
    PowerDrillDecorator
{
    private int extraPower = 0;

    public override void Recharge()
    {
        base.Recharge();
```

```
11          extraPower = 6;
12      }
13
14      public override bool Drill()
15      {
16          if (!base.Drill())
17          {
18              if (extraPower > 0)
19              {
20                  Console.WriteLine(
21                      "Extended battery used.");
22                  extraPower--;
23                  return true;
24              }
25
26              return false;
27          }
28
29          return true;
30      }
31  }
```

This **Decorator** provides some additional functionality. Its `Recharge` method allows us to apply additional power capacity. Its `Drill` method ensures that if the **Original Object** runs out of energy, we can still continue operating while using the additional power reserve.

Now, let's see how these two decorators work together. To do so, we will replace the content of the `Program.cs` file with the following:

```
using Decorator_Demo;

var powerDrill = new PowerDrill();

powerDrill.Recharge();
powerDrill.Drill();
powerDrill.Drill();

Console.WriteLine("Adding a hammer extension.");
var silencer = new HammerExtension();
silencer.SetPowerDrill(powerDrill);
silencer.Drill();
silencer.Drill();

Console.WriteLine("Adding an extended battery.");
var extendedMag = new ExtendedBattery();
extendedMag.SetPowerDrill(silencer);
extendedMag.Recharge();
extendedMag.Drill();
extendedMag.Drill();
extendedMag.Drill();
extendedMag.Drill();
extendedMag.Drill();
extendedMag.Drill();
extendedMag.Drill();
extendedMag.Drill();

Console.ReadKey();
```

In here, we are creating an instance of the `PowerDrill` class, recharging it, and using the drill. Then we apply an instance of `HammerExtension` to it and use this extension while drilling. Afterward, we apply an instance of `ExtendedBattery` to it, recharge it, and use the drill multiple times again. You can see the output on the following screenshots:

```
Shot fired.
Shot fired.
Applying a silencer.
Shot fired.
Shot has been silenced.
Shot fired.
Shot has been silenced.
Applying an extended magazine.
Shot fired.
Shot has been silenced.
Shot fired.
Shot has been silenced.
Shot fired.
Shot has been silenced.
Shot fired.
Shot has been silenced.
Shot fired.
Shot has been silenced.
Shot fired.
Shot has been silenced.
Out of ammo.
Shot fired from the extended magazine.
Out of ammo.
Shot fired from the extended magazine.
```

**Figure 32.2 - The results of applying multiple decorators**

this concludes our Decorator example. Let's now summarize the main benefits of using this design pattern.

## Benefits of using Decorator

The benefits of using Decorator are as follows:

- This design pattern allows you to modify the behavior of objects without having access to the internal functionality of those objects.
- It strongly enforces the open-closed principle, so nothing that uses the original objects will have to be modified.
- It enforces the single responsibility principle, as each **Decorator** implementation is responsible for a specific functionality.

- It's easy to write automated tests against **Decorator** implementations because they are very specific and focused.

But, despite all of its benefits, Decorator has some caveats. And this is what we will have a look at next.

## Caveats of using Composite

It's a good thing that Decorator allows you to apply multiple layers of modified functionality to your **Original Object**. But when you have too many of them in the stack, it may become hard to manage the functionality. Removal and replacement of **Decorator** instances becomes especially problematic. So you need to watch out for that.

In the above example, I have intentionally demonstrated another problem you may encounter while using Decorator. We have added a silencer to our gun, which added silencing functionality to our shots. So far so good. Then we added an extended magazine on top of the silencer. And this combination worked too. That was until we ran out of ammo from the original gun. We could still make shots from the additional ammo reserve, but those were not silenced, making our silencer ineffective in this scenario.

So that exposes a problem. **Decorator** implementations need to be designed in such a way that they don't interfere with each other's functionality.

Finally, there's not always a possibility to use the Decorator pattern when you want to modify the behavior of the objects you can't modify from the inside. Or, at least, you can't modify it in a specific way. And this is where Adapter will be more appropriate.