
Decorator Design Pattern
Na'ama Bar-Ilan, August 2020

1.
Introduction

Structural

This design pattern is about the structure of objects and code

An Alternative to Inheritance?

Add Functionality...

The Decorator design pattern seeks to add new functionality to an existing object without changing that object's definition.

... Outside the Class

It adds new responsibilities to an individual instance of an object, without adding those responsibilities to the class of objects.

The Participants

Component

Defines the interface for objects which will have responsibilities or abilities added to them dynamically.

Decorator

Maintains a reference to a Component and defines an interface that conforms to the Component interface.

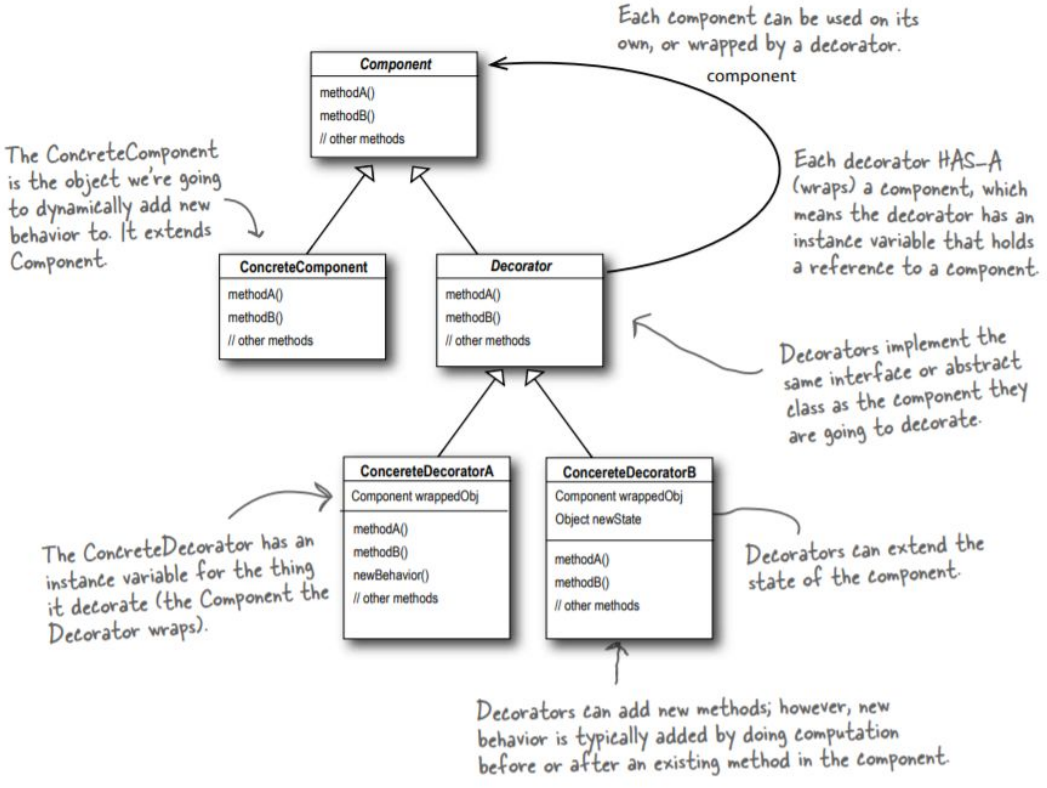
ConcreteComponent

These are the objects to which said responsibilities are added.

ConcreteDecorator

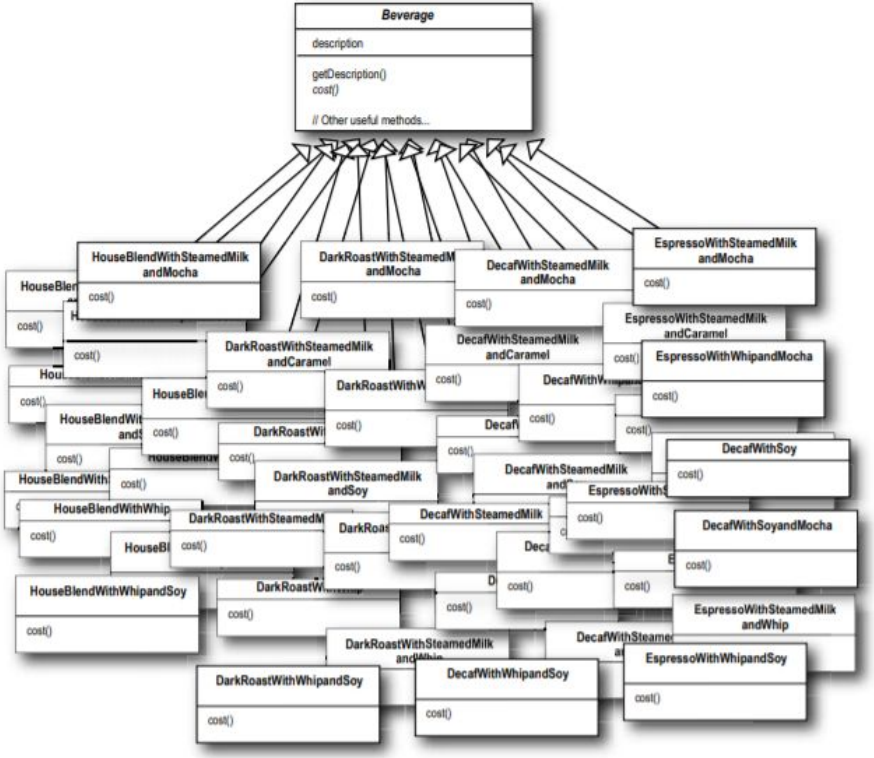
These are the classes which actually add responsibilities to the ConcreteComponent classes.

UML Class Diagram



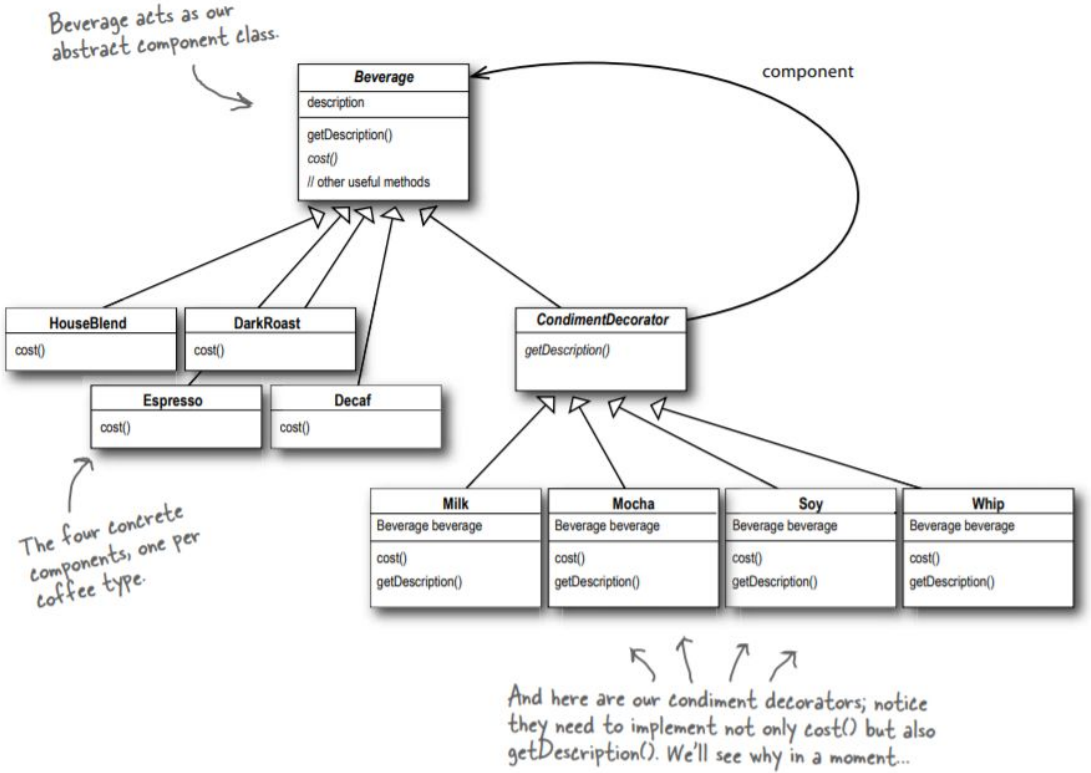
UML Source: Head First Design Patterns: A Brain-Friendly Guide

UML : Cafe Example - Inheritance



UML Source: Head First Design Patterns: A Brain-Friendly Guide

UML : Cafe Example - Decorator



UML Source: Head First Design Patterns: A Brain-Friendly Guide

2.
Pros & Cons

Advantages

- The pattern has the power to add flexibility to designs.
- Decorating allows us to give objects new responsibilities without making any code changes to the underlying classes.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.
- Helps designs stay true to the Open-Closed SOLID Principle: open for extension, but closed for modification.

Disadvantages

- Can sometimes add a lot of small classes to a design and this occasionally results in a design that's harder for others to understand
- Some code is dependent on specific types. Introducing decorators without thinking through everything can create typing issues.
- Introducing decorators can increase the complexity of the code needed to instantiate the component, because it needs to be wrapped in decorators.

3.
Real World Example

A Farm to Table Restaurant

Why Farm-to-Table?

- The restaurant can only make dishes from ingredients that are available from the farm
- Ingredients come in and out based on season and demand.
- When an ingredient is not available, or runs out, the restaurant needs to be able to mark related dishes as 'Sold Out' on the fly.
- For example: we need to be able to track the number of zucchini pasta dishes we have in the kitchen, and immediately mark it as unavailable when we run out.

5.
Code Snippet

1.

```
/// <summary>
/// The abstract Component class
/// </summary>
abstract class RestaurantDish
{
    public abstract void Display();
}
```

2.

```
/// <summary>
/// A ConcreteComponent class
/// </summary>
class FreshSalad : RestaurantDish
{
    private string _greens;
    private string _cheese; //I am going to use this pun everywhere I can
    private string _dressing;

    public FreshSalad(string greens, string cheese, string dressing)
    {
        _greens = greens;
        _cheese = cheese;
        _dressing = dressing;
    }

    public override void Display()
    {
        Console.WriteLine("\nFresh Salad:");
        Console.WriteLine(" Greens: {0}", _greens);
        Console.WriteLine(" Cheese: {0}", _cheese);
        Console.WriteLine(" Dressing: {0}", _dressing);
    }
}

/// <summary>
/// A ConcreteComponent class
/// </summary>
class Pasta : RestaurantDish
{
    private string _pastaType;
    private string _sauce;

    public Pasta(string pastaType, string sauce)
    {
        _pastaType = pastaType;
        _sauce = sauce;
    }

    public override void Display()
    {
        Console.WriteLine("\nClassic Pasta:");
        Console.WriteLine(" Pasta: {0}", _pastaType);
        Console.WriteLine(" Sauce: {0}", _sauce);
    }
}
```

3.

```
/// <summary>
/// The abstract Decorator class.
/// </summary>
abstract class Decorator : RestaurantDish
{
    protected RestaurantDish _dish;

    public Decorator(RestaurantDish dish)
    {
        _dish = dish;
    }

    public override void Display()
    {
        _dish.Display();
    }
}
```

Code source: https://www.exceptionnotfound.net/decorator-pattern-in-csharp/

4.

```
/// <summary>
/// A ConcreteDecorator. This class will impart "responsibilities" onto the dishes
/// (e.g. whether or not those dishes have enough ingredients left to order them)
/// </summary>
class Available : Decorator
{
    public int NumAvailable { get; set; } //How many can we make?
    protected List<string> customers = new List<string>();
    public Available(RestaurantDish dish, int numAvailable) : base(dish)
    {
            NumAvailable = numAvailable;
    }

    public void OrderItem(string name)
    {
        if (NumAvailable > 0)
        {
            customers.Add(name);
            NumAvailable--;
        }
        else
        {
            Console.WriteLine("\nNot enough ingredients for " + name + "'s
order!");
        }
    }

    public override void Display()
    {
        base.Display();

        foreach(var customer in customers)
        {
            Console.WriteLine("Ordered by " + customer);
        }
    }
}
```

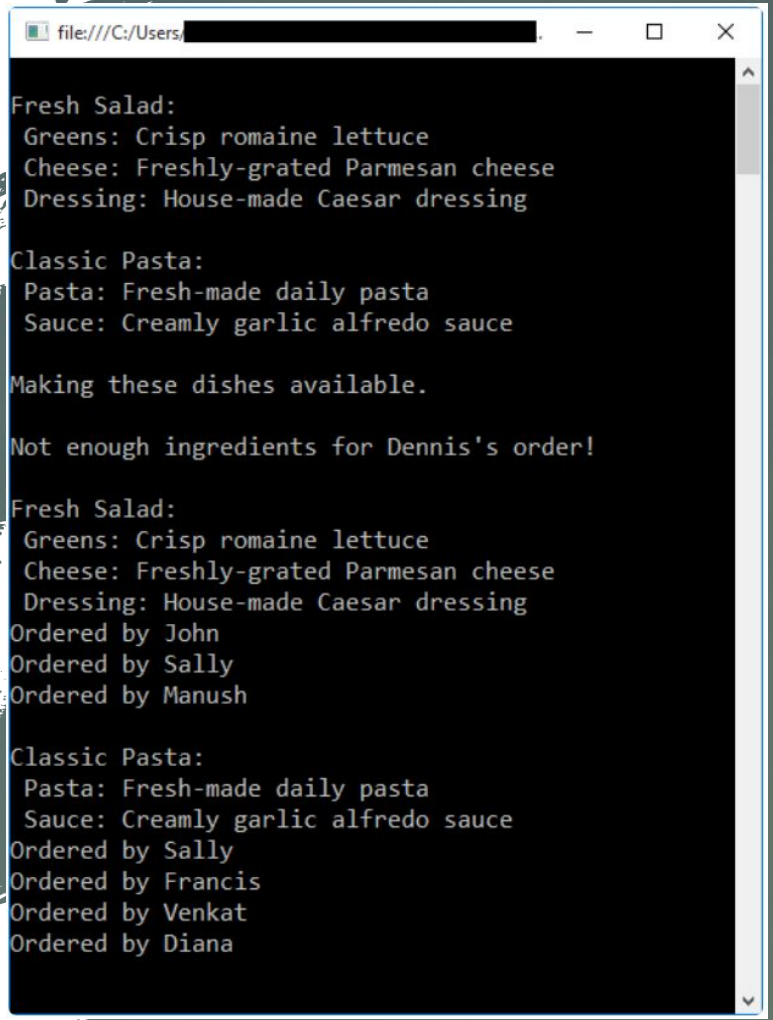
5.

```
static void Main(string[] args)
{
    //Step 1: Define some dishes, and how many of each we can make
    FreshSalad caesarSalad = new FreshSalad("Crisp romaine lettuce", "Freshly-
grated Parmesan cheese", "House-made Caesar dressing");
    caesarSalad.Display();

    Pasta fettuccineAlfredo = new Pasta("Fresh-made daily pasta", "Creamly garlic
alfredo sauce");
    fettuccineAlfredo.Display();

    Console.WriteLine("\nMaking these dishes available.");

    //Step 2: Decorate the dishes; now if we attempt to order them once we're out
of ingredients, we can notify the customer
    Available caesarAvailable = new Available(caesarSalad, 3);
    Available alfredoAvailable = new Available(fettuccineAlfredo, 4);

    //Step 3: Order a bunch of dishes
    caesarAvailable.OrderItem("John");
    caesarAvailable.OrderItem("Sally");
    caesarAvailable.OrderItem("Manush");

    alfredoAvailable.OrderItem("Sally");
    alfredoAvailable.OrderItem("Francis");
    alfredoAvailable.OrderItem("Venkat");
    alfredoAvailable.OrderItem("Diana");
    alfredoAvailable.OrderItem("Dennis"); //There won't be enough for this order.

    caesarAvailable.Display();
    alfredoAvailable.Display();

    Console.ReadKey();
}
```

Code source: https://www.exceptionnotfound.net/decorator-pattern-in-csharp/

The App Output

Dennis can't order the fettuccine alfredo because we've run out of ingredients.



Code source: https://www.exceptionnotfound.net/decorator-pattern-in-csharp/

6. Highlighted Summary

Decorator Summary

- The Decorator pattern seeks to dynamically add functionality to instances of objects at runtime, without needing to change the definition of the instance's class
- This design flexibility is especially useful in scenarios where different instances of the same object might behave differently (e.g., dishes in a restaurant or items in a library)
- While it is useful in certain circumstances, it is not used very often outside of demos.
- Can add complexity and hurt the readability of the code.

OOP Principles

Abstraction

Abstract classes are not supposed to be instantiated. They are only supposed to be used as a template that can be derived further down for more clarity.

Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another.

Encapsulation

Encapsulation is the act of hiding methods and attributes that should not be exposed to unauthorized or unneeded classes or methods. Examples of encapsulation are Public, private, and protected.

Polymorphism

Within polymorphism, we want the ability to change the behavior of a specific class. The ability to override an abstract or virtual method is polymorphism.

OOP Principles

Abstraction

The *Component* and the *Decorator* are abstract classes that do not get instantiated. These are templates that get derived further down in the *ConcreteComponent* and *ConcreteDecorator* classes, e.g: FreshSalad : RestaurantDish, Available: Decorator

Inheritance

ConcreteComponent and *ConcreteDecorator* take on the attributes and methods of their abstract base class, the *Component* and *Decorator*, e.g:

FreshSalad inherits the Display() method from RestaurantDish

Polymorphism

We change the behaviors within the child classes, *ConcreteComponent* and *ConcreteDecorator*, e.g. in FreshSalad we override the abstract Display() method to include ConsoleWriteLine("\nFresh Salad:")

Encapsulation

The *ConcreteComponent* attributes are Private so they are only available within the body of that class.

The *Decorator* attribute is protected, so it is only accessible within that class and by the derived *ConcreteDecorator* class instances.

Sources & Links

- https://www.exceptionnotfound.net/decorator-pattern-in-csharp/
- https://www.dofactory.com/net/decorator-design-pattern
- Head First Design Patterns: A Brain-Friendly Guide, by Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates
- Presentation template by SlidesCarnival


Thanks!
Any questions?