I6SWD Decorator Pattern

Tri Nguyen au564065 & Arseniy Malyutin au508734

November 2020

Contents

1 The Decorator Pattern

1.1 Description of the pattern

The decorator pattern is a method of attaching functionality and/or responsibility to an object dynamically without the need of having to form objects from some abstract base super class.

The idea is that if you have some sort of objects that share a common abstract class, but implement different variations of it. One would not need to modify the base class each time new object variations arise, instead one could decorate those objects with another class that shares the base class.

An example from the book headfirst describes how one runs a beverage store that sells different varieties of beverages, but with different types(coffee, tea, shakes, etc) and condiments(caramel, cream, milk etc). One could use the base class beverage and that creates a concrete object (tea) and decorates it with, say milk. The idea is, as you can see in figure 1 is that you have a component from the base class that gets wrapped in however many decorators you wish to add to it that add these different varieties to the object depending on what you wish it to do.

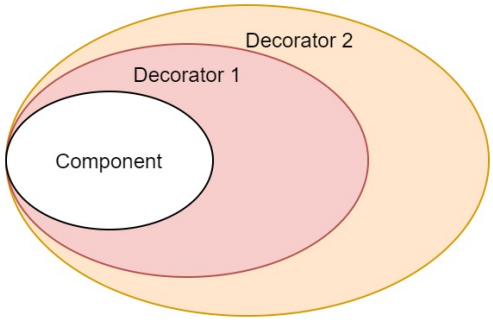

Figure 1: Concept of Decorator Pattern wrapping decorators around a component

The beverage example however does not account for an important aspect of the decorator class. The idea is that each decorator can add new functionality to the component. Say for example instead that you run a pizza place where the customer is allowed to choose toppings for the pizza, but as an added bonus can also choose what shape he wishes the pizza to be and what

kind of colour dough the baker should use. The added decorators for the pizza in this case would be a shape and color that add a new functionality to your standard pizza(assuming the customer wants to choose these things of course). This pattern is very identical to the chain of responsibility pattern, however in the chain of responsibility only one class handles the request, while in the decorator pattern all the classes handle the request.

1.2 Purpose of the Pattern

One issue that pattern can solve is if you imagine having a situation where you have multiple types of subclasses that subscribe to one superclass as in the example with beverages talked about previously as can be seen from figure 2. In the figure you have each subclass calculate its costs manually inside of it depending on what condiments it uses. One can imagine a situation where this can become a maintenance nightmare as more subclasses keep getting added and changes in their functionality(in this case cost of condiments) can take ages to correct since one would have to manually go into each subclass and edit it.

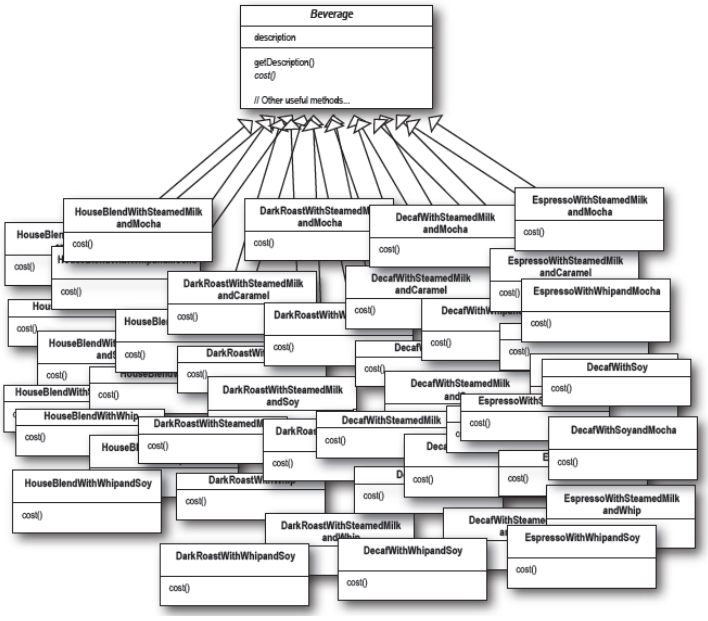

Figure 2: An unmanageable amount of subclasses. Page 81 from the book Head First - Design Patterns.

Another example of an issue is, if one generates a superclass that has several methods denoting functionality and then each subclass writes a boolean true or false into each class to determine what functionality it desires(i.e. functionality in this case being condiments). This can be seen in figure 3.

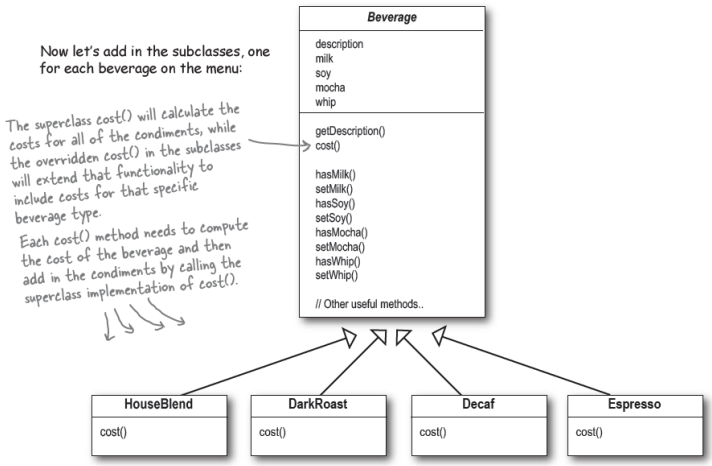

Figure 3: The coffee implmentation using has[condiment] methods. Page 83 from the book Head First - Design Patterns.

These issues show concerns with wanting to add/remove functionality dynamically, create more abstract code structure less reliant on subclassing and increase maintenance that the 2 methods described do not provide.

This is where decorator patterns come into play. It provides a method where one can dynamically allocate functionality to classes, allow for more abstract coding to lessen the burden for individual classes and increase extensibility and maintainability by providing a method where one can add a decorator object to fulfil functionality needs if a class desires it. The maintainability can be seen as each decorator uses the single responsibility principle, this makes it easier to modify the decorator if responsibility changes occur.

1.3 Pattern type

Decorator Pattern is a behavioural design pattern that can change/add behaviour/functionality to a component object while not having to copy and

paste much code, or even rely on inheriting from a base class and implementing a lot of concrete instances of the base class. Instead of having to change the behaviour and functionality during compile-time, one can do it dynamically during run-time with Decorator Pattern. Desiring changed behaviour to an object happens by implementing an abstract Decorator from the base class and changes behaviour in the concrete implementations of the Decorator.

1.4 Pattern Structure

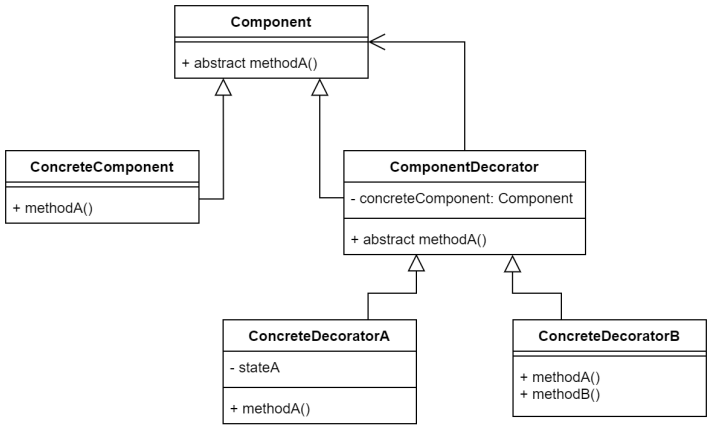

Figure 4: Generic UML-diagram of Decorator Pattern.

Decorator pattern bases its pattern in the use of an abstract base class, which can be seen on figure 4 with the Component class. Concrete component implementations will act as a base component that can be decorated upon. The decorators can then partake in adding more behaviour and functionality to the base components without having to change the code written in the base component.
Using inheritance, the component decorator also acts as a component, and to be able to execute its methods when wrapped around a concrete component, it therefore also has a component. So the decorator has and is a component.
Our implementation of Decorator Pattern revolves around an Italian trattoria that serves three different dishes: pizza, calzone, and sandwich. Each dish is plain by itself and therefore needs condiments and topping that can be

added onto the dish, each condiment will add to the total cost of the meal. If the trattoria would want to make the dishes and its possible combinations in compile-time, they would have to make a specific implementation of each dish and its array of condiments and toppings. As explained earlier on figure 2, that would result in a lot of classes to manage. With the implementation shown on figure 5, Decorator Pattern is used to take away the hassle of having to rewrite so much redundant code and just be able to edit code one place instead of other places, as well as being able to make different combinations at run-time.

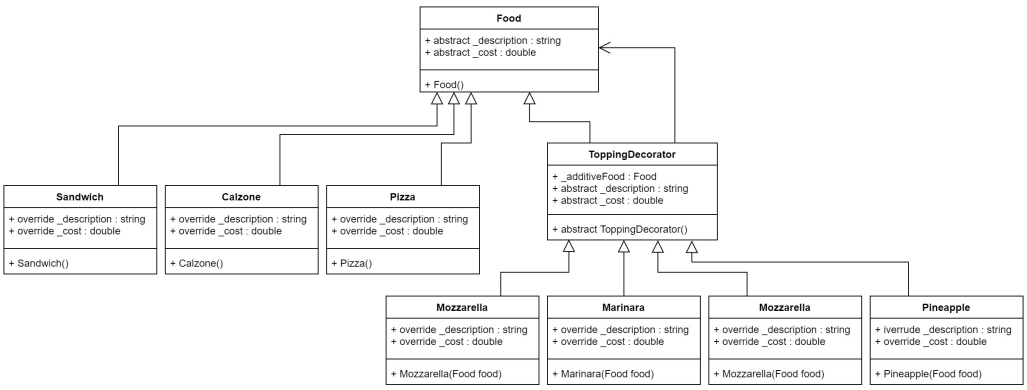

Figure 5: Trattoria example UML-diagram with Decorator Pattern.

Instead of having an exponential amount of different classes of dishes, the dishes can be made by wrapping dishes with toppings and adding as many wrappers as are wanted and needed.

1.5 Dynamics

With Decorator Pattern, one can dynamically add new functionality and behaviour to a component by decorating it as one sees fit. Instead of having to edit the code of the component, a specific decorator can be wrapped around the component and decorate it with new behaviour. Instead of having statically made components with each their extreme specific use, despite it still abiding by the Single Responsibility Principle, the structure of the program and its components are parted so that base components are decorated to fit a specific use rather than hard coding components to that specific use. Being able to change the behaviour and functionality of a base component

at run-time makes it much more flexible and shows its dynamic structure it can work upon.

Instead of having a specific class written out like

```
class PizzaMarinaraMozzarella {
    public PizzaMarinaraMozzarella(){
        _cost = 70.00;
        _description = "Pizza, Marinara, Mozzarella"
    }

    //cost of base pizza + marinara + mozzarella
    public double _cost{get;set;}
    public string _description{get;set}
};
```

Listing 1: Subclass of pizza

We can wrap a Pizza component with the Marinara decorator and Mozzarella decorator for more flexibility and dynamic run-time combinations like so

```
Pizza pizza1 = new Pizza();
Marinara marinara1 = new Marinara(pizza1);
Mozzarella mozzarella1 = new Mozzarella(marinara1);
```

Listing 2: Component pizza in marinara and mozzarella decorators

With the outermost wrapper, mozzarella1, we can call the description or cost, and the decorator will traverse into the center to its base component, call its description or cost, and add each decorator upon decorator.
If we instead wanted a

```
PizzaMarinaraGorgonzolaSausage
```

or

```
PizzaMarinaraCheddarSausage
```

or

```
PizzaBechamelMozzarellaBacon
```

We can just wrap a Pizza component in whatever combination of decorators we want.

The given example

1.6 Consequences of Decorators

Though it is clear that the benefits of having objects that can extend functionality dynamically make it easier to maintain and modify. There are some drawbacks concerning using the decorator pattern. Due to the nature of decorators each performing their required functionality it can become tedious to maintain if there are far too many decorators, although it can be easy to find responsibilities of each decorator if the number of decorators grow it increases the cost of maintenance. When a client uses code that requires decorators it can be difficult for the client to understand what kind of decorators the object requires to get the desired result. Since each decorator adds functionality, a client has to read through each decorator to get to the desired functionality of the object. This issue leads to another problem is that if the object requires a certain set of decorators to be done in order, in order to get the desired result. The pattern seems inherently flexible as one can always add decorators, but in reality this makes it rigid due to the choice of decorators to use and in which order.