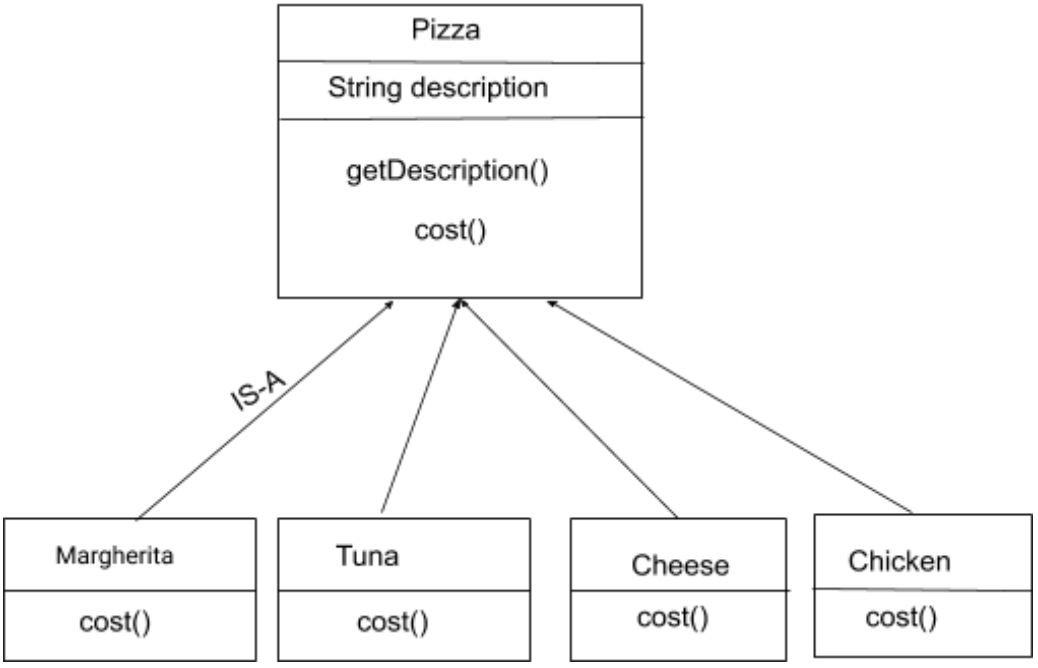
Decorator Design Pattern

We're now going to be looking at a pattern called the **decorator pattern**. And to get a better understanding of that pattern, we're going to start with a small example.

Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Margherita, Tuna, Cheese and Chicken. Customers can add a number of toppings to the pizza like Tomato, Olive, Mushroom and Black pepper. And each topping has a small cost in addition to the cost of the pizza.

So let's sketch out a class design for the pizza restaurant.



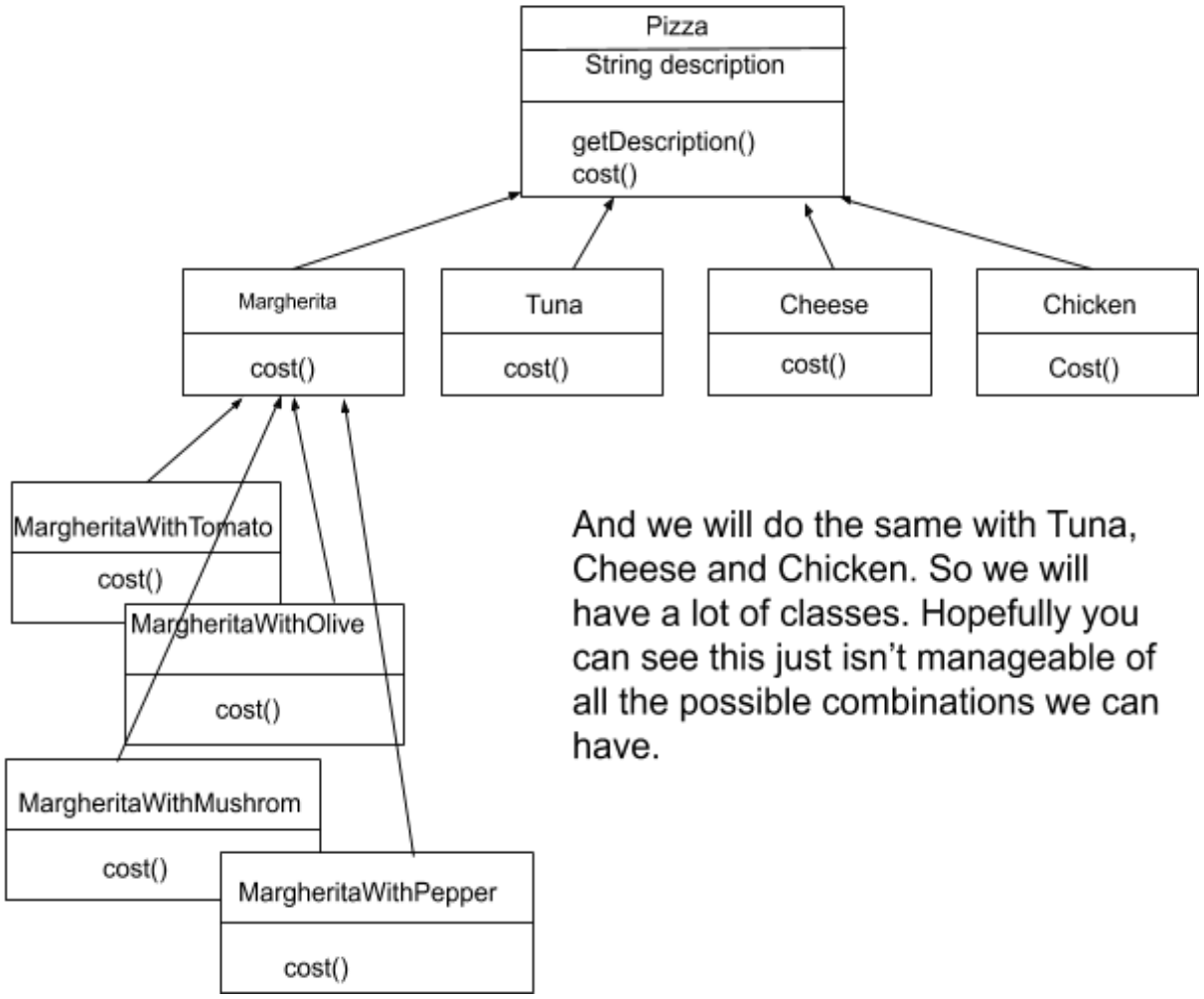
We'll give the Pizza superclass a description field and a getter method as well as a cost method. Now let's add some subclasses. We have Margherita, Tuna, Cheese and chicken In each subclass, we can override the cost method to calculate the cost of a particular pizza.

That looks like a nice, simple design that we can extend for other pizza in the future, but we're forgetting that there are many variants of these pizza, like a cheese with olive.

Let us look at various options.

Option 1

Create a new subclass for every topping with a pizza. The class diagram would look like:



This looks very complex. There are way too many classes and it is a maintenance nightmare. Also if we want to add a new topping or pizza we have to add so many classes. This is obviously a very bad design.

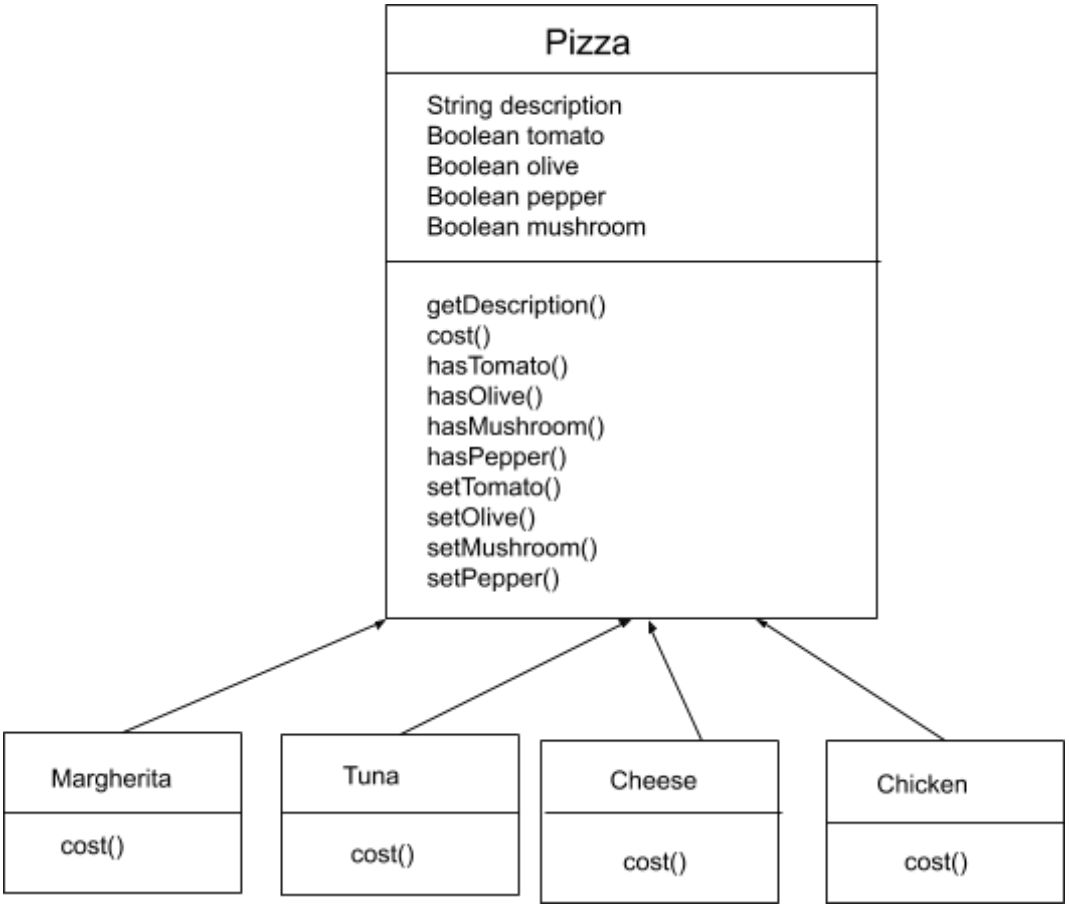
So let’s take a different approach.

Option 2:

This time, let's try using object properties to keep track of the pizza and the toppings. What we can do is use some fields in the superclass to track which toppings are in the pizza.

And for each toppings we have a Boolean field and a method to track if that topping is being used in the pizza.

The class diagram would look like:



Now we could write code like this for say Margherita with tomato and mushroom.

```
Margherita marPizza = new Margherita();
marPizza.setTomato();
marPizza.setMushroom();
float cost = marPizza.cost();
```

The cost depends on the cost method in each pizza subclass.

```
if (hasTomato()){
      Cost += .30;
}
if (hasOlive()){
      Cost += .90;
}

if (hasMushroom()){
      Cost += 1.30;
}
if (hasPepper()){
      Cost += .08;
}
```

we always have to test every possible topping, but this might work, so let's analyze this approach.

It looks much simpler, but we have to consider

- Price changes in toppings will lead to alteration in the existing code.
- New toppings will force us to add new methods, open up the code in the superclass and change it.
- For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
- What if a customer wants double Mushroom?

Design Principle #5

Classes should be open for extension, but closed for modification.

Now what does that mean?

Well think about our current design for the pizza restaurant. We know in the future, we'll have to support new toppings types. But that means we'll have to modify existing code.

And that's exactly what we don't want. We want to leave our design open for toppings
Types, but closed in the sense we don't want to touch existing code.

Let's check this principle in a little more detail. First, we have to call this the **open-closed principle**.

And, again, the principle says that code should be open for extension but closed for modification.

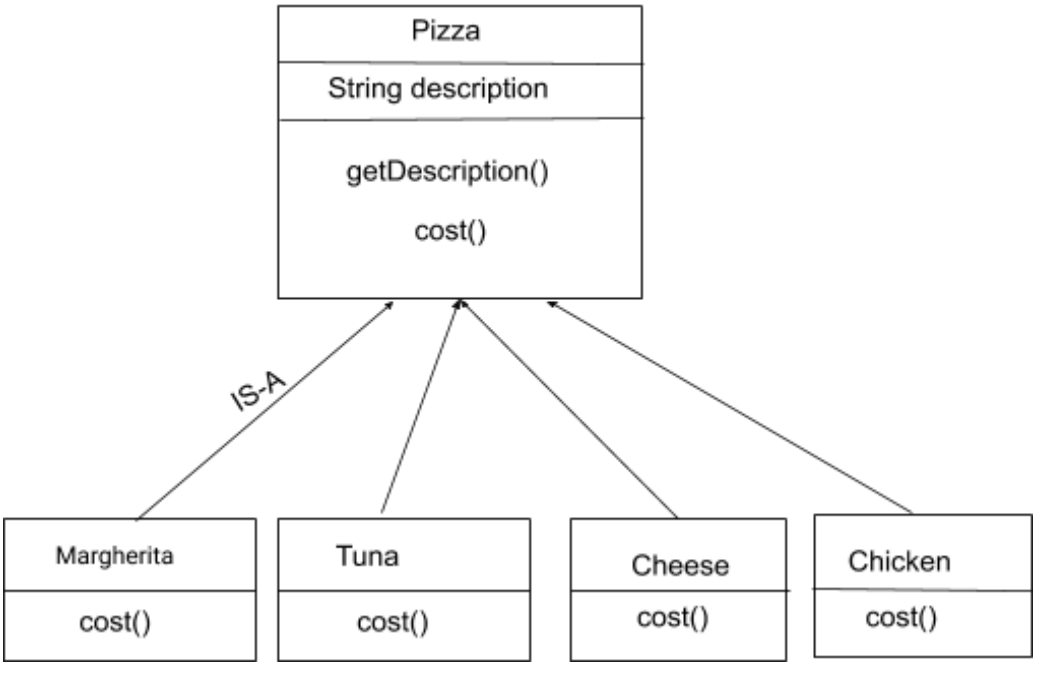
If we can do that, we have **flexibility and maintainability**, because we can **add new behavior** but without the **risk of introducing new bugs** into the code that we've already written.

So our goal is to have designs that we can **extend** at any time, but to do it **without touching existing code**.

This is one of the most important design principles and if you can implement this principle into your designs, then your designs are going to be much more flexible and maintainable in the long run.

Before we jump into the formal definition of the decorator pattern, let's get a conceptual feel for how a decorator works, in particular, how we can compose objects together to achieve it.

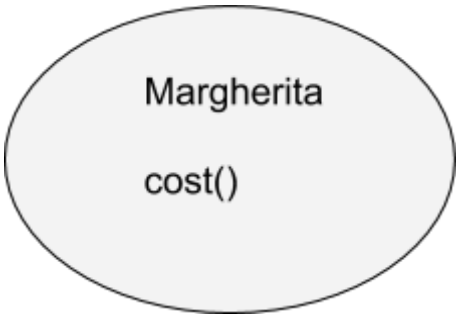
Our first design on the pizza hierarchy wasn't too bad.



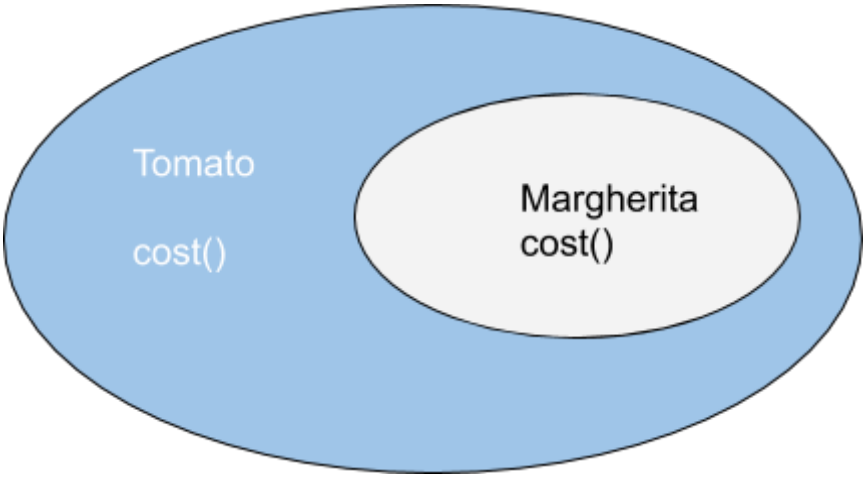
A Margherita is a pizza, and a Tuna is a pizza, and so on.
And so let's start there, Where we got into trouble was really with the toppings.
Say a customer wants a Margherita with Tomato and Olive.

Rather than using a specific class for the entire pizza, like we did before, let's create a Margherita object and and “decorate” it with toppings:

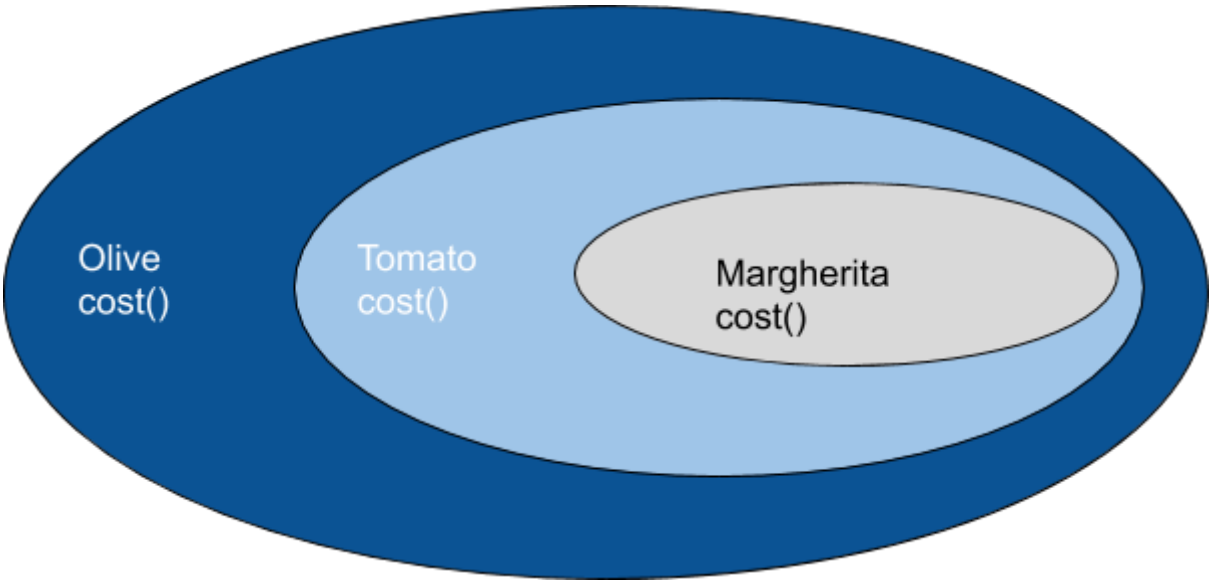
1- Take a Margherita object.



2- “Decorate” or compose it with a Tomato object

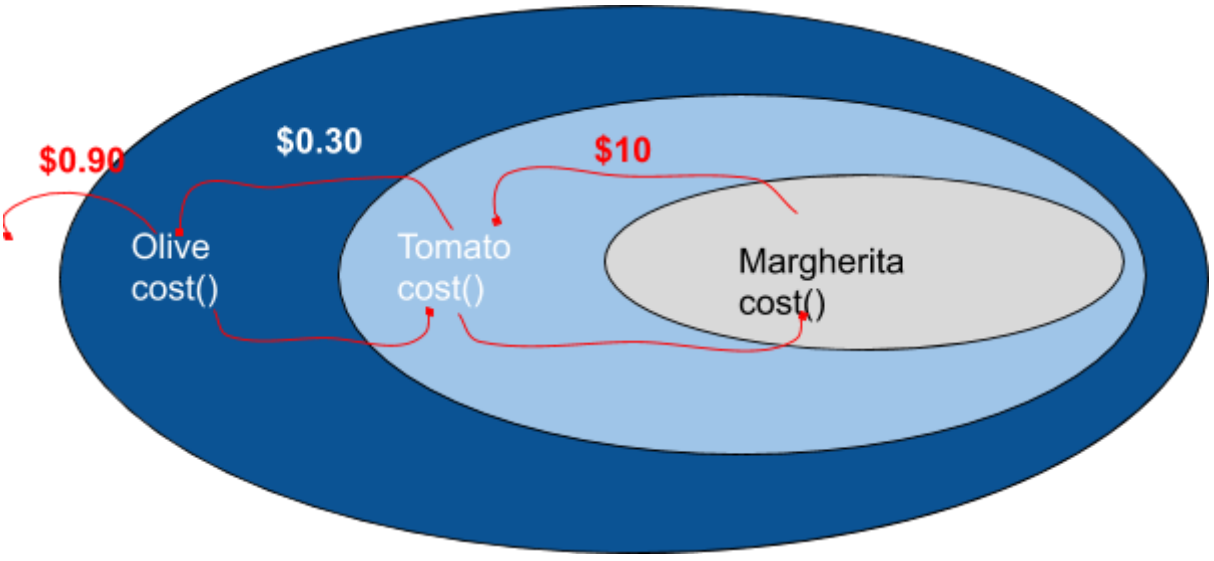


3- “Decorate” or compose it with a Olive object.



We read the class names from the inside out, happens to match the pizza order.

Margherita with Tomato and Olive. But what about cost?
Well, as you can see, each of these objects looks alike. They all have cost methods, and they're all responsible for their own part of the cost.



We call cost on the outermost object, which then delegates the cost to the next object, which then delegates the cost to the next object, and so on.

Each time we delegate, we get a value back, and then we add that cost to our own, and finally return that cost, which is the accumulation of all cost, back to the caller.

Of course you've noticed how flexible this design is, We can create just about any pizza we want.

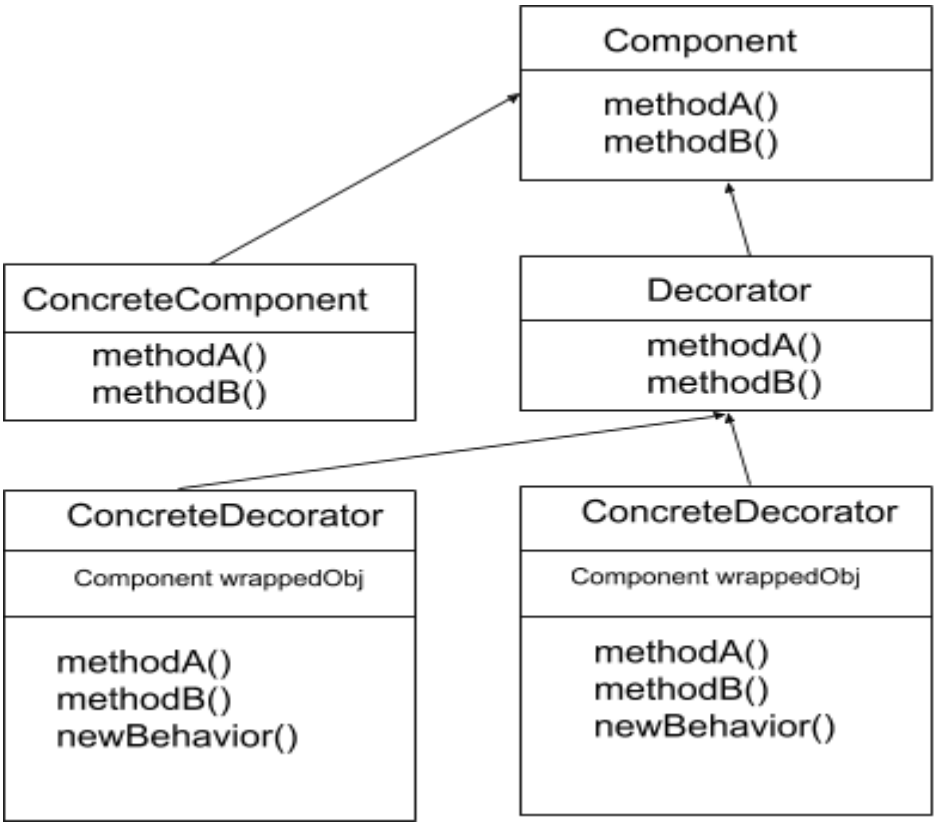
Now it's time to look at how this is actually achieved in classes and code.
We are going to use a decorator pattern to design our pizza restaurant, So let's begin by looking at the definition of the pattern.

The **decorator pattern** attaches additional responsibilities to an object dynamically, Decorators provide a flexible alternative to subclassing for extending functionality.

You're probably starting to get the sense that these pattern definitions don't always help you to see how to implement the pattern.

As with all design pattern definitions, we really need to look at a class diagram to get a better sense for how the pattern works.

There are two important parts to the decorator pattern, there are the **Components**, which are the **pizza** in our example, and the **Decorators**, which are the **toppings** in our example.



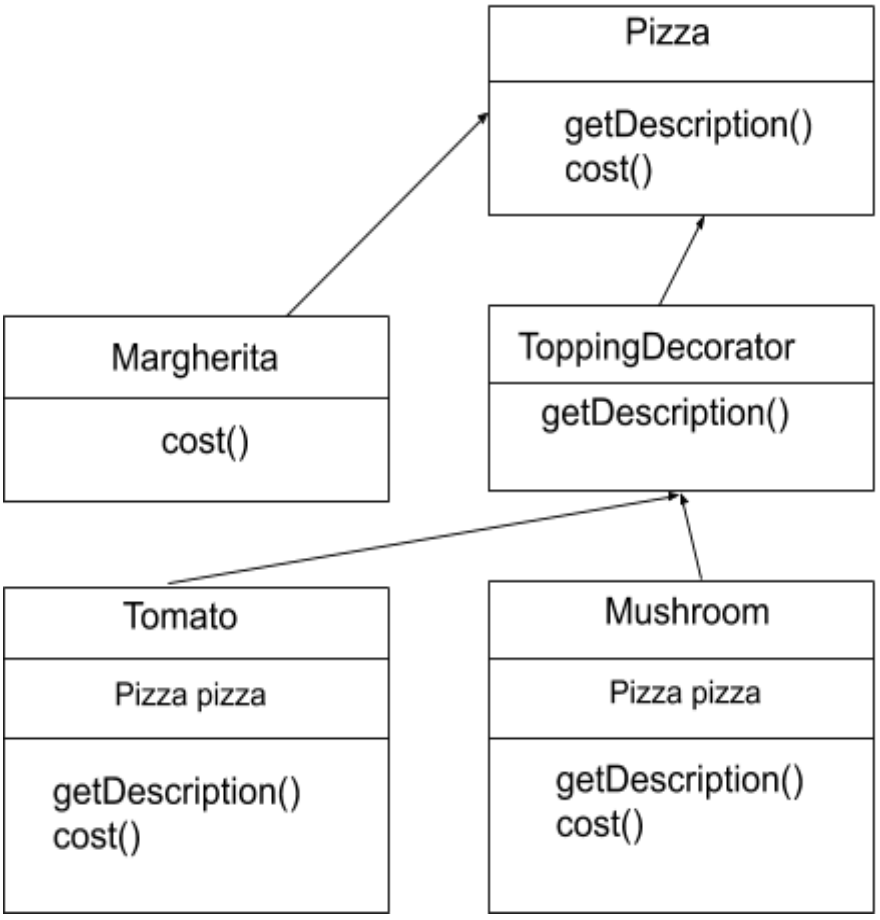
Starting at the top, you can see we have a component class, this is an interface or abstract class that's implemented by the concrete components.

So in our example, the component class will be the pizza, and the concrete components will be the different types of pizza like Margherita and etc.

We also have a decorator class, which is often an abstract class. And then we have the concrete decorators that implement it. And these are the tomato, olive, mushroom and pepper toppings in our example.

The reason it's so important for both the concrete components and the decorators to implement the component superclass, is because we want to make sure that we can treat each class in the same way.

That is, we want to be able to wrap any decorator around any of the components.



For instance, in the pizza restaurant example, we want to be able to wrap any of the toppings around any of the pizzas, and then call the cost and get description methods on any of these wrapped objects, Or even an unwrapped object and get the correct result.

Please check the code to make it clear and I will try to upload a video to make it more clearer soon.

#StaySafe ❤
#StudyWell 📖