Decorator pattern

Many structural design patterns are needed to avoid unnecessary work and rewriting of code many times over.

Usually, these patterns are made in such a way that you can avoid interfering with classes that were previously defined.

Decorator design pattern

A **decorator** is a structural design pattern that allows using special wrappers for objects, giving them new functionality while avoiding changes to their structure.

This approach simplifies the coding process by delegating the implementation of new functions to other classes, instead of rewriting new object classes.

A **wrapper** (or decorator) is an object that works with your initial object in a way that alters its behavior.

Wrapper shares the same interface and functions with that object. But, in case the user needs to call the initial object, the wrapper will pass the user's request through itself. In this process, the result can be adapted.

This approach is useful when you can't just modify your class and it will be hard to extend it with some child objects.

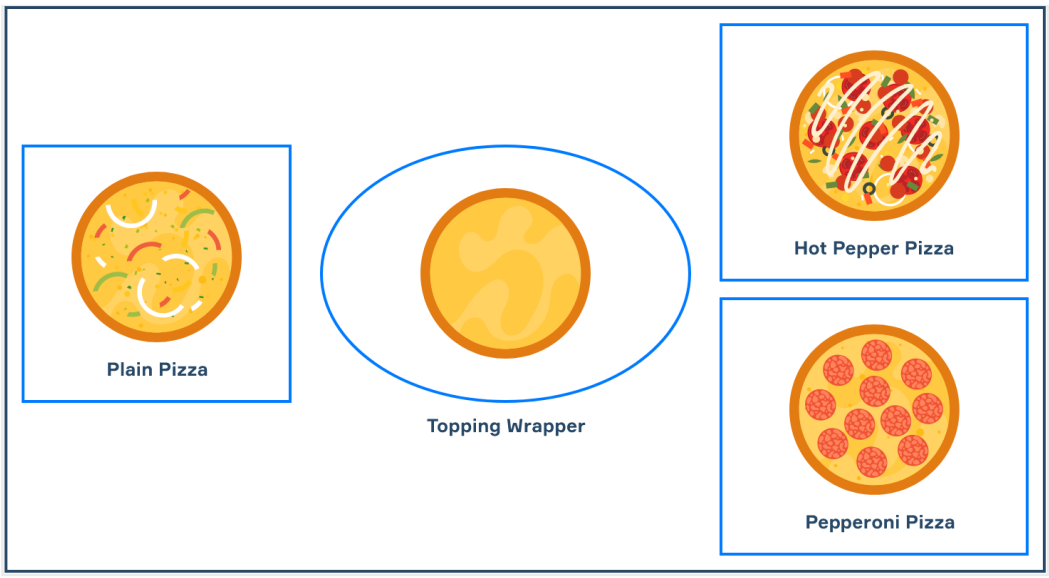
Using a decorator, you can add new behaviors and even use multiple level wrappers, without touching any part of the existing class. But this can be a little tricky to implement, considering that you may end up being unable to remove some of the wrappers without ruining all of the code.

Decorator implementation

Let's use the pizza-making process as an example. Cooks make pizza. They have abundant options for toppings and types of pizza.

But in the end, a pizza is just some dough with something on top of it.

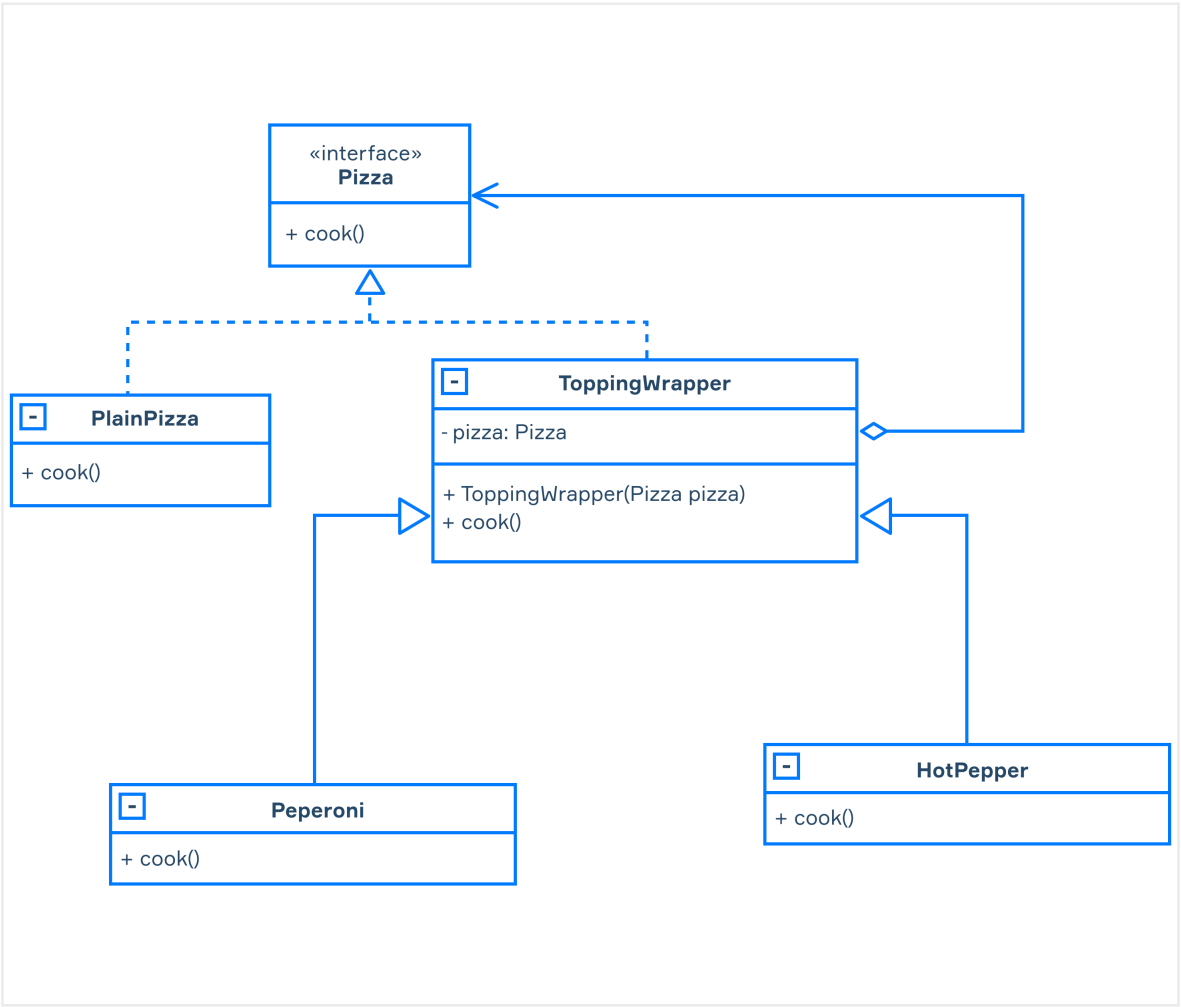
So, if we try to depict the pizza-making process as code, we can create a new class for each pizza type. And if we want to create combinations of pizza toppings, we'll end up with a large number of classes which will complicate your work.



Using this pattern, we need to introduce these elements:

- an interface that represents methods for basic pizza making;
- a class that creates our pizza;
- one base wrapper which will store our pizza object to combine it with our topping wrappers;
- for this particular example, we'll have two wrappers for pizza toppings.

If we look at the resulting structure as a class diagram, it will look similar to this:



Decorator in pseudocode

Now let's try to depict this pattern in pseudocode. First of all, we need to describe our Pizza interface:

```
interface Pizza is
  method cook()
```

Here, we described all common methods for both our pizza maker and pizza wrapper. Next, we'll describe them as PlainPizza class and ToppingWrapper:

```
class PlainPizza implements Pizza is
  method cook() is
    return "Pizza"


class ToppingWrapper implements Pizza is
  field pizza: Pizza

  constructor ToppingWrapper(Pizza pizza)
    this.pizza = pizza
```

```
  method cook() is
    pizza.cook()
```

Both these classes will implement the Pizza interface. Our PlainPizza will return an object that represents a simple pizza that will be referenced within the ToppingWrapper class and then combined with our other wrappers.
Next, we'll define our Pepperoni and HotPepper wrappers:

```
class Pepperoni extends ToppingWrapper is
  method cook() is
    return pizza.cook() + " Pepperoni"

class HotPepper extends ToppingWrapper is
  method cook() is
    return pizza.cook() + " HotPepper"
```

As you can see, these wrappers will add some additional parts to our existing objects.

Example of using decorator pattern

When we have all of our classes ready, we can call our topping wrappers in a client code, in order to add some new parts to our initial object:

```
//Client code:

Pizza pepperoni = new Pepperoni(new PlainPizza()) //decorate plain pizza
print("Description: " + pepperoni.cook())

//Output:
//Description: Pizza Pepperoni
```

If we call our PlainPizza constructor inside of the Pepperoni wrapper, we will have a modified object with additional parts. We can also put one wrapper inside of another. The result will look like this:

```
//Client code:

Pizza pepperoni = new Pepperoni(new PlainPizza()) //decorate plain pizza
Pizza hotPepperoni = new HotPepper(pepperoni) //decorate already decorated pizza
print("Description: " + hotPepperoni.cook())

//Output:
//Description: Pizza Pepperoni HotPepper
```