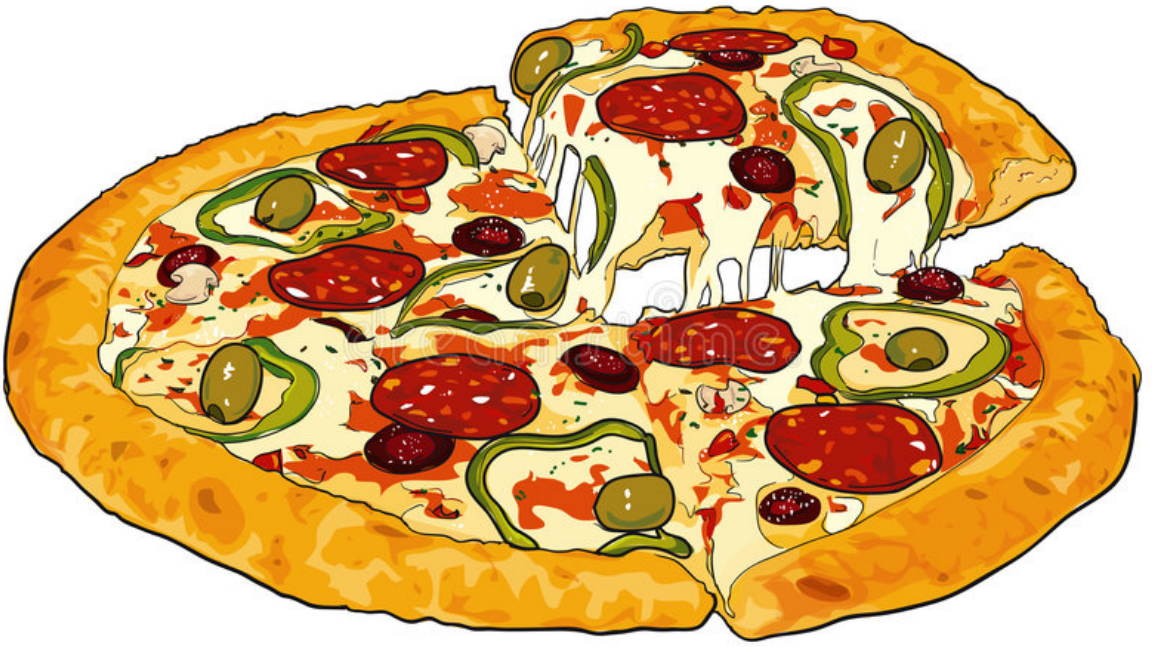
Decorator Design Pattern

Notes by Bhavuk Jain



Definition: The **Decorator Pattern** allows us to attach additional responsibilities to an object *dynamically*. Decorators provide a flexible **alternative to subclassing** for extending functionality.
Decorator Pattern is useful in the cases where we have some common base functionality and we want to add some extra features on top of it.

Example: ***Pizza!*** Each Pizza has a base with let's say, *base bread*, *veggies*, *mayonnaise*, *cheese*, *sauce*, etc. On top of it, if we want we can ask for **additional toppings** such as *mushrooms, extra cheese, jalapenos, black olives,* and what not! So here, the **base pizza** will be our **base object** and the **toppings** will be the **decorators** which are adding additional features on top of our base object.

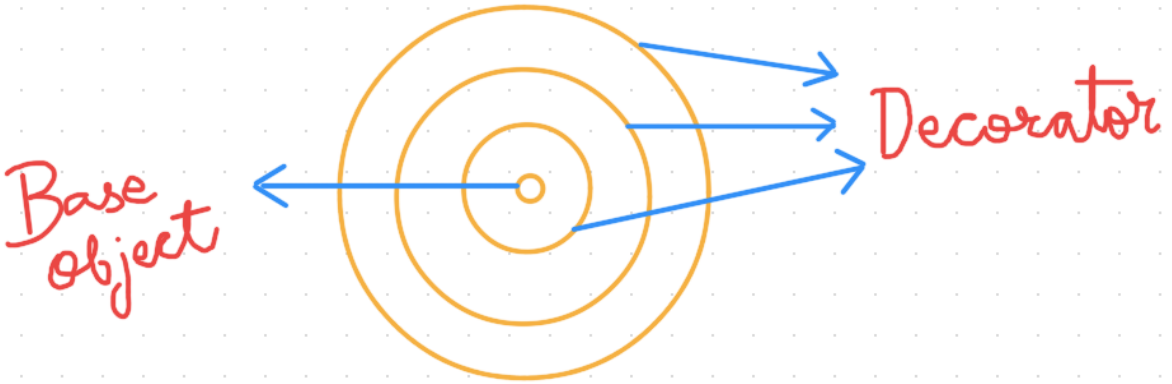
CORE CONCEPT

We have an object that has some features. Now, we want to have *additional features* on top of it, so what we can do is to **wrap it inside another object** . Now, this new object has it's own features as well as the base object's features.

Again, we can **wrap** this newly created object with another **object** to add more features. Like this, we can add **n** number of wrapper objects. In this case, the **wrapper objects** are called as **decorators** **which can again be *decorated* by some other *wrapper* object**. Here, the wrapper object is itself also acting as a **Base Pizza** and it also has a Base Pizza.

It's very important to understand the **is-a** and **has-a** relationship the **wrapper** object has with **Base Pizza**.

You can imagine it to be something like this:

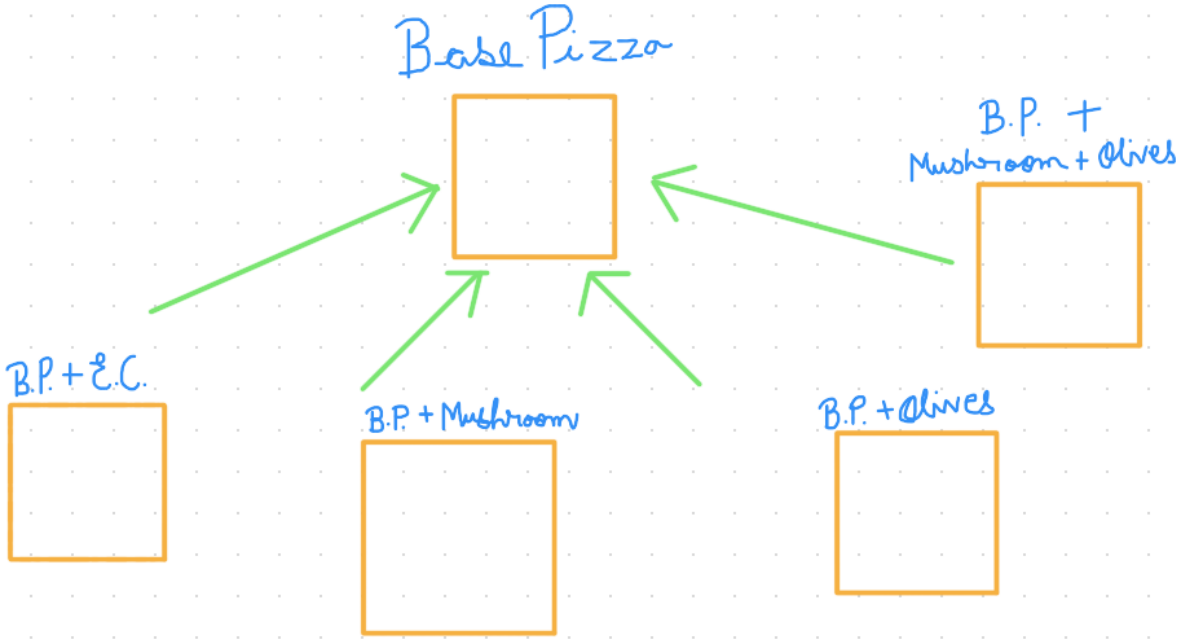


WHAT & WHY?

We use the Decorator Pattern to **avoid** something known as **Class Explosion** .

Class explosion occurs when we tend to create **n** number of **sub classes** for different combinations of the features that we would want.

Like in our **Pizza example** (refer the diagram given below), we would end up creating so many sub classes for different combinations of Pizza Base with other toppings. This would become *unmanageable* as we add more toppings and different base pizzas.



B.P. -> Base Pizza
E.C. -> Extra Cheese

Solution With Code Snippets

(Go through the code comments for better perspective)

1.) We create an abstract class **BasePizza** with an abstract method **cost()**

BasePizza.java × ToppingDecorator.java

```
1  package decoraterpattern;
2
3  public abstract class BasePizza {
4      public abstract int cost();
5  }
6
```

2.) We create a class called **VegDelight** which is a **BasePizza**, and because it is extending an abstract class, so it has to provide implementation to the

abstract methods

```java
package decoraterpattern;

public class VegDelight extends BasePizza{

    // Provide implementation to abstract cost() in BasePizza
    public int cost(){
        // Assume price of VegDelight to be Rs. 100
        return 100;
    }

}

```

3.) We create another class called **Farmhouse** which is a **BasePizza**, and because it is extending an abstract class, so it has to provide implementation to the abstract methods

```java
package decoraterpattern;

public class Farmhouse extends BasePizza{

    // Provide implementation to abstract cost() in BasePizza
    public int cost(){
        // Assume price of Farmhouse to be Rs. 200
        return 200;
    }

}

```

4.) Now, we'll create a **ToppingDecorator** Class (which is kind of an **helper class** so that we can distinguish between classes that act as **BasePizza** and the classes that act as a **ToppingDecorator**. ToppingDecorator is extending BasePizza to denote that it itself is also an BasePizza and can be decorated by other ToppingDecorator(s)

```java
package decoraterpattern;

public abstract class ToppingDecorator extends BasePizza {
/*
ToppingDecorator Class is a BasePizza, hence,
if any other class extends ToppingDecorator, it will need to
provide the implementation of abstract cost() method as well
 */

}

```

5.) We create a class **ExtraCheese** which is a **ToppingDecorator**. It has a **BasePizza** object. The cost is calculated by adding up the cost of the BasePizza and adding topping's own cost.

```java
package decoraterpattern;

// ExtraCheese is a ToppingDecorator which again is a BasePizza
public class ExtraCheese extends ToppingDecorator{

  /* Reference variable for BasePizza,
     to signify that ExtraCheese has a BasePizza
   */
  BasePizza basePizza;

  // Assigning basePizza object through constructor injection
  public ExtraCheese(BasePizza basePizza){
    this.basePizza = basePizza;
  }

  // Providing implementation to abstract cost()
  public int cost(){

    // Assume price of ExtraCheese topping to be Rs. 20
    return this.basePizza.cost()+20;
  }

}

```

6.) We create a class **BlackOlive** which is a **ToppingDecorator**. It has a **BasePizza** object. The cost is calculated by adding up the cost of the

BasePizza and adding topping's own cost.

```java
package decoraterpattern;

// BlackOlive is a ToppingDecorator which again is a BasePizza
public class BlackOlive extends ToppingDecorator{

    /* Reference variable for BasePizza,
       to signify that BlackOlive has a BasePizza
    */
    BasePizza basePizza;

    // Assigning basePizza object through constructor injection
    public BlackOlive(BasePizza basePizza){
        this.basePizza = basePizza;
    }

    // Providing implementation to abstract cost()
    public int cost(){

        // Assume price of Mushroom topping to be Rs. 50
        return this.basePizza.cost()+50;
    }
}

```

7.) We create a class **Mushroom** which is a **ToppingDecorator**. It has a **BasePizza** object. The cost is calculated by adding up the cost of the BasePizza and adding topping's own cost.

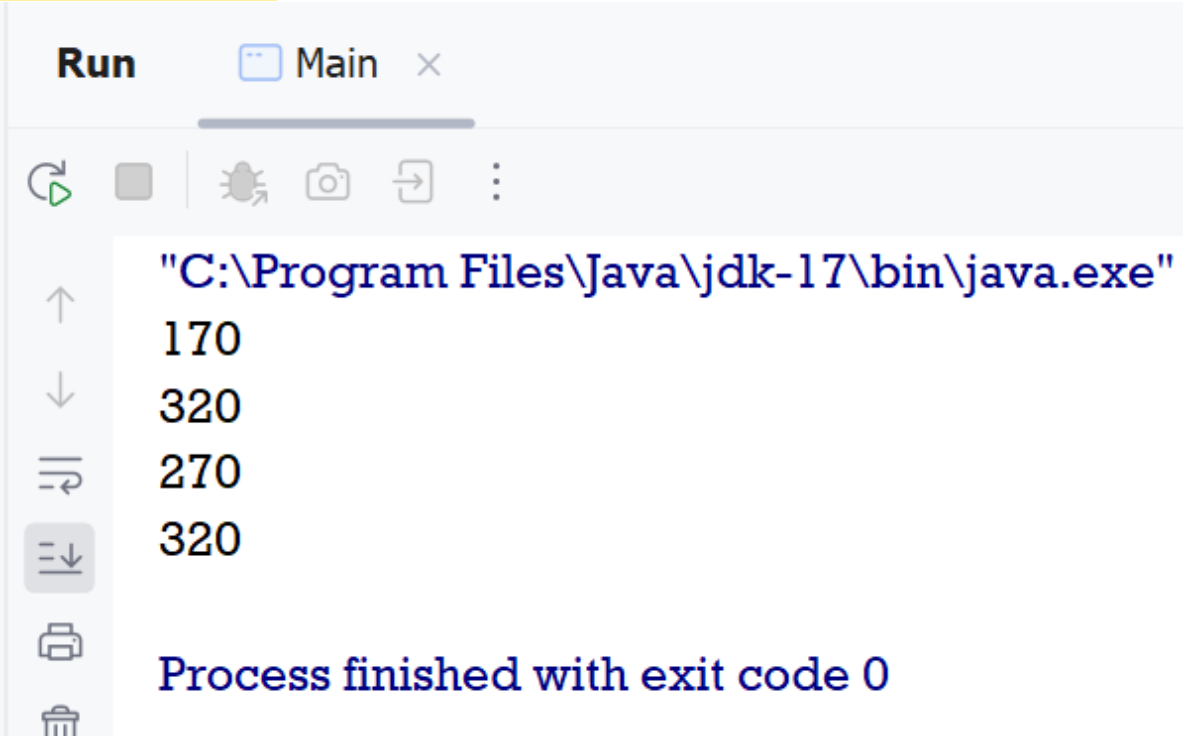
```java
package decoraterpattern;

// Mushroom is a ToppingDecorator which again is a BasePizza
public class Mushroom extends ToppingDecorator{

   /* Reference variable for BasePizza,
      to signify that Mushroom has a BasePizza
   */
   BasePizza basePizza;

   // Assigning basePizza object through constructor injection
   public Mushroom(BasePizza basePizza){
      this.basePizza = basePizza;
   }

   // Providing implementation to abstract cost()
   public int cost(){

      // Assume price of Mushroom topping to be Rs. 100
      return this.basePizza.cost()+100;
   }
}

```

Decorator Pattern In Action

Creating different combinations of various Base Pizza(s) and toppings (By passing objects of **BasePizza** to **ToppingDecorators**):-

```java
package decoraterpattern;

public class Main {
    public static void main(String[] args) {

        // Create a new Pizza having Base as VegDelight & Toppings -> ExtraCheese + BlackOlive
        BasePizza myCustomPizza1 = new BlackOlive(new ExtraCheese(new VegDelight()));
        System.out.println(myCustomPizza1.cost()); // Print the total cost

        // Create a new Pizza having Base as Farmhouse & Toppings -> ExtraCheese + Mushroom
        BasePizza myCustomPizza2 = new Mushroom(new ExtraCheese(new Farmhouse()));
        System.out.println(myCustomPizza2.cost()); // Print the total cost

        // Create a new Pizza having Base as VegDelight & Toppings -> ExtraCheese + BlackOlive + Mushroom
        BasePizza myCustomPizza3 = new Mushroom(new BlackOlive(new ExtraCheese(new VegDelight())));
        System.out.println(myCustomPizza3.cost()); // Print the total cost

        // Create a new Pizza having Base as Farmhouse & Toppings -> ExtraCheese + BlackOlive + BlackOlive
        BasePizza myCustomPizza4 = new BlackOlive(new BlackOlive(new ExtraCheese(new Farmhouse())));
        System.out.println(myCustomPizza4.cost()); // Print the total cost

    }
}

```

Output



Hope you found it helpful! Thanks for reading!

By
Bhavuk Jain