The decorator pattern allows behavior to be added to individual objects, without affecting the behavior of other objects in the same class.

```java
//Component Interface
interface PizzaOption{
    void assemble();
}
```

This is the PizzaOption interface which declares the assemble method, serving as the component interface for pizza toppings.

```java
class basicPizza implements PizzaOption{
    @Override
    public void assemble(){
        System.out.println("Assembling a simple Pizza");
    }
}
```

This basicPizza class implants the PizzaOption interface.
This class basically represents the basic pizza.
This defines the base behavior that can be enhanced or modified.

```java
abstract class PizzaDecorator implements PizzaOption{
    protected PizzaOption decoratedPizza;

    public PizzaDecorator(PizzaOption nonvegpizza){
        this.decoratedPizza = nonvegpizza;
    }

    @Override

    public void assemble(){
        decoratedPizza.assemble();
    }
}
```

This class PizzaDecorator implements the PizzaOption interface.
This class acts as a base for concrete decorators. This introduces a common structure for decorators which contains a reference to a PizzaOption enabling the dynamic composition of behaviors. The assemble() method here makes sure that the decorated pizza retains its base behavior while incorporating additional features.

```java
//Concrete decorator 1

class Pepperoni extends PizzaDecorator{
    public Pepperoni(PizzaOption nonvegpizza){
        super(nonvegpizza);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.println("Adding pepperoni to the Pizza");
    }
}
```

This is the class Pepperoni which extends the class PizzaDecorator. I mean by that, it will accomplish it by calling the assemble() method. This will add the specific features of pepperoni to be added to the pizza. The super in this case is needed because it will call the constructor of the superclass PizzaDecorator. It is passing the nonveg pizza argument to the constructor of PizzaDecorator.

```java
//Concrete decorator 2
class baconCrumble extends PizzaDecorator{
    public baconCrumble(PizzaOption nonvegpizza){
        super(nonvegpizza);

    }

    @Override
    public void assemble(){
    super.assemble();
    System.out.println("Adding bacon crumble to the Pizza");

    }

}
```

This is the class baconCrumble which extends the PizzaDecorator class.
This also uses the assemble() method which will add the feature of bacon crumble, which then will be added to the pizza. The actions which will be performed are the same as in the Pepperoni class.

There are many benefits of using the decorator pattern because first it extends the class without modifying the source code. It supports open and closed principle by enabling the addition of new functionality(decorators) without changing the existing classes(basicPizza)
It also provides a flexible alternative to subclassing for extended behavior, this means that I can create complex combinations of behaviors dynamically. It also enhances code maintainability by separating the concerns, which means that I have two decorators in my code example. Each decorator will focus on a specific aspect of the behavior. This separation of concerns makes it easier to understand and modify the code. In case if i were not to use the decorator pattern then I would face a lot of issues like Class Explosion, which means that each combination of pizza would require me to use separate subclasses. Depends on the number of options.
By not adding the Decorator pattern I would also violate the Open/Close principle. This means that adding new pizza options would require me to modify existing classes. This could lead to unintended consequences and potential bugs in the system.