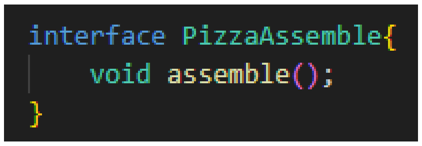
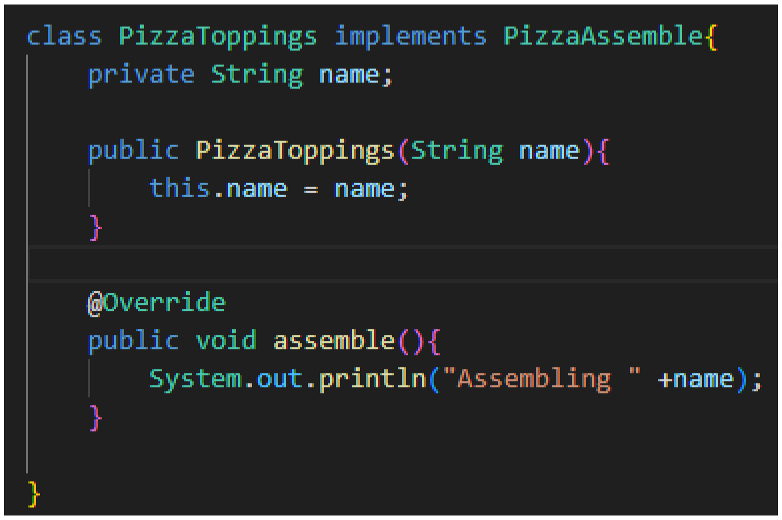
The composite design pattern is a structural design pattern which lets me compose objects into tree structures to represent part-whole hierarchies. This allows clients to treat individual objects and compositions of objects uniformly.

THE COMPOSITE DESIGN PATTERN EXPLAINED WITH MY CODE EXAMPLE



```java
interface PizzaAssemble{
    void assemble();
}
```

This is the component interface that declares the common interface assemble().



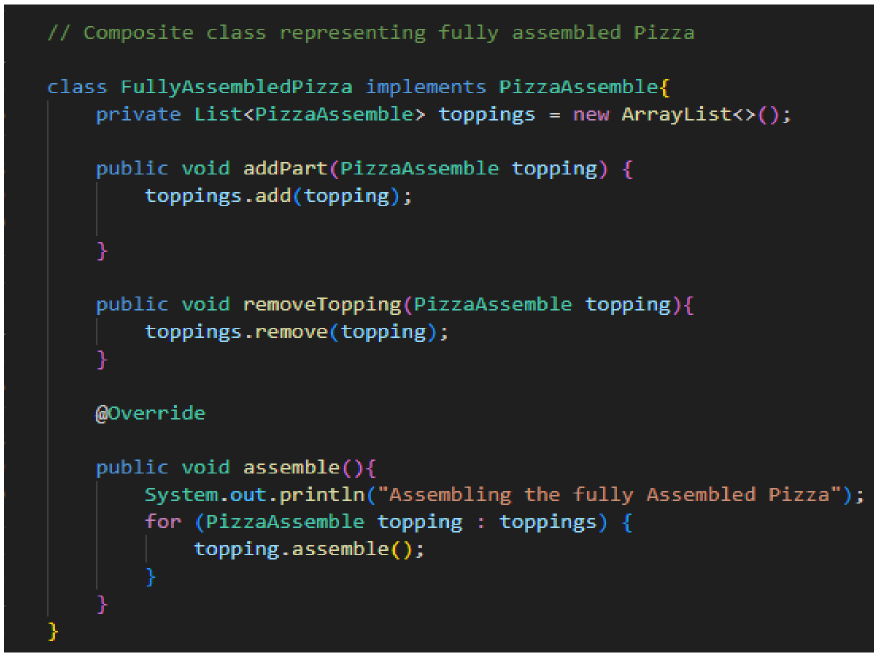
```java
class PizzaToppings implements PizzaAssemble{
    private String name;

    public PizzaToppings(String name){
        this.name = name;
    }

    @Override
    public void assemble(){
        System.out.println("Assembling " +name);
    }

}
```

This is the PizzaToppings class which implements the interface PizzaAssemble. This class has a name attribute which will represent the type of topping.



```java
// Composite class representing fully assembled Pizza

class FullyAssembledPizza implements PizzaAssemble{
    private List<PizzaAssemble> toppings = new ArrayList<>();

    public void addPart(PizzaAssemble topping) {
        toppings.add(topping);

    }

    public void removeTopping(PizzaAssemble topping){
        toppings.remove(topping);
    }

    @Override

    public void assemble(){
        System.out.println("Assembling the fully Assembled Pizza");
        for (PizzaAssemble topping : toppings) {
            topping.assemble();
        }
    }
}
```

This is the FullyAssembledPizza class which implements the PizzaAssemble interface.
This is a composite class which can contain a list of ‘PizzaAssemble’ objects, they can either be individual toppings or other fully assembled pizzas. It uses the PizzaAssemble interface.
This class has two methods addPart() and removeTopping(). The assemble method is implemented to iterate over all the toppings or fully assembled pizza it contains and calls the assembly method.

There are a lot of benefits of the composite pattern for my code example and for software in general.
It provides a Uniform interface which means that the interface PizzaAssemble can be used for both individual toppings and fully assembled pizza.
Recursive Composition is also possible, which means that it allows me the creation of more complex Pizzas(Objects) with different types of toppings and nested structures.
There is also an ease of Adding and Removing Components which means in my case that the FullyAssembledPizza class provides methods like addPart() and removeToppings(), which makes it easier for me to modify the pizzas structure without impacting the whole code.
If I were not to use the Composite design pattern for this code then I would face a lot of issues like lack of interface. This means that the interface would not be uniform, which would lead to that I would need to create a more complex and error prone client code, as the clients may have to handle all the individual toppings and the fully assembled pizzas differently. The other issue which I could face is also that there would be limited Recursive Composition which means that creating pizzas with nested structures could be challenging. I would lose the ability to create a pizza with diverse and nested toppings. The client code cloud also turn out to be more complex which means that, managing different types of pizza components separately could lead to more complex client code, because the clients may have to deal with various component types

differently, making the code less readable and more difficult to maintain as the complexity of the pizza structure will increase.

The Composite Pattern greatly benefits the provided pizza assembly code by providing a consistent interface, allowing for recursive composition of complex structures, and making component addition and removal easier. Without the pattern, issues such as interface uniformity, limited recursive composition, and increased complexity in client code management arise. The Composite Pattern, in essence, improves code flexibility, maintainability, and ease of modification.