Xideral

Java Academy

Week 3-Day 3

Decorator pattern & JUnit Testing

Presented by:

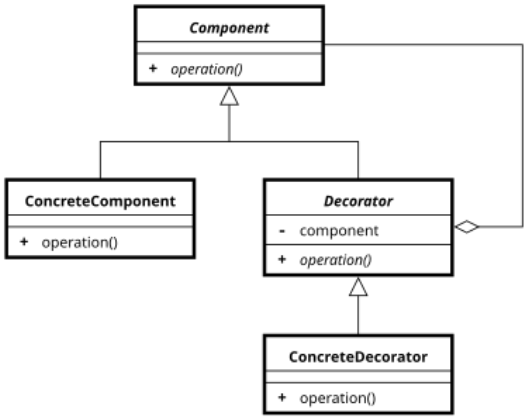
Edgar Itzak Sánchez Rogers

Monterrey Nuevo León México at 29 august 2024

Introduction:

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Decorator pattern allows behavior to be added to an individual object, dynamically, without affecting the behavior of other instances of the same class. by allowing the functionality of a class to be extended without being modified. With decorator pattern it's possible to add and remove responsibilities from an object dynamically at run-time.



Decorator pattern demonstration:

Menu Seller Interface:

```
1 package com.edgaritzak.decoratorpattern;
2
3 public interface MenuSellerInterface {
4
5     String makeOrder(String type, int quantity);
6 }
```

Abstract Seller Decorator:

```java
package com.edgaritzak.decoratorpattern;

public abstract class SellerDecorator implements MenuSellerInterface {

    protected MenuSellerInterface menuSeller;

    public SellerDecorator(MenuSellerInterface menuSeller) {
        super();
        this.menuSeller = menuSeller;
    }

    @Override
    public String makeOrder(String type, int quantity) {
        menuSeller.makeOrder(type, quantity);
        return "type"+type+" quantity"+quantity;
    }
}
```

Concrete Menu Seller:

```java
package com.edgaritzak.decoratorpattern;

public class MenuSeller implements MenuSellerInterface {

    String type;
    int quantity;

    @Override
    public String makeOrder(String type, int quantity) {
        System.out.println(quantity+" "+type+" pizza");
        return "type"+type+" quantity"+quantity;
    }

}
```

Concrete Seller Decorator:

```java
package com.edgaritzak.decoratorpattern;

public class ComboMealDecorator extends SellerDecorator{

    private String drink;
    private String dip;
    private String side;


    public ComboMealDecorator(MenuSellerInterface menuSeller) {
        super(menuSeller);
    }

    @Override
    public String makeOrder(String type, int quantity) {
        System.out.println("You have ordered a combo, it includes:");
        System.out.print("- ");super.makeOrder(type, quantity);
        System.out.println("Complements:");
        System.out.println("- Side: "+side);
        System.out.println("- Dip: "+dip);
        System.out.println("- Drink: "+drink);
        return "type:"+type+" quantity:"+quantity+" side:"+side+" dip:"+dip+" drink:"+drink;
    }



    public String getDip() {
        return dip;
    }

    public void setDip(String dip) {
        this.dip = dip;
    }

    public String getSide() {
        return side;
    }

    public void setSide(String side) {
        this.side = side;
    }

    public String getDrink() {
        return drink;
    }
    public void setDrink(String drink) {
        this.drink = drink;
    }
}

```

Main class:

```java
package com.edgaritzak.decoratorpattern;

public class Main {

    public static void main(String[] args) {
        Main app = new Main();
        app.run(args);
    }

     public void run(String[] args) {
        //Selling pizza
        MenuSeller menuSeller = new MenuSeller();
        menuSeller.makeOrder("Pepperoni", 2);

        //Sell pizza combo with decorator
        ComboMealDecorator ComboMeal  = new ComboMealDecorator(menuSeller);
        ComboMeal.setSide("Breadsticks");
        ComboMeal.setDrink("Coca-Cola 2L bottles, 67.6 Oz");
        ComboMeal.setDip("Ranch Dressing");
        ComboMeal.makeOrder("Supreme", 2);
        }
}
```

Output:

```
2 Pepperoni pizza
You have ordered a combo, it includes:
- 2 Supreme pizza
Complements:
- Side: Breadsticks
- Dip: Ranch Dressing
- Drink: Coca-Cola 2L bottles, 67.6 Oz
```

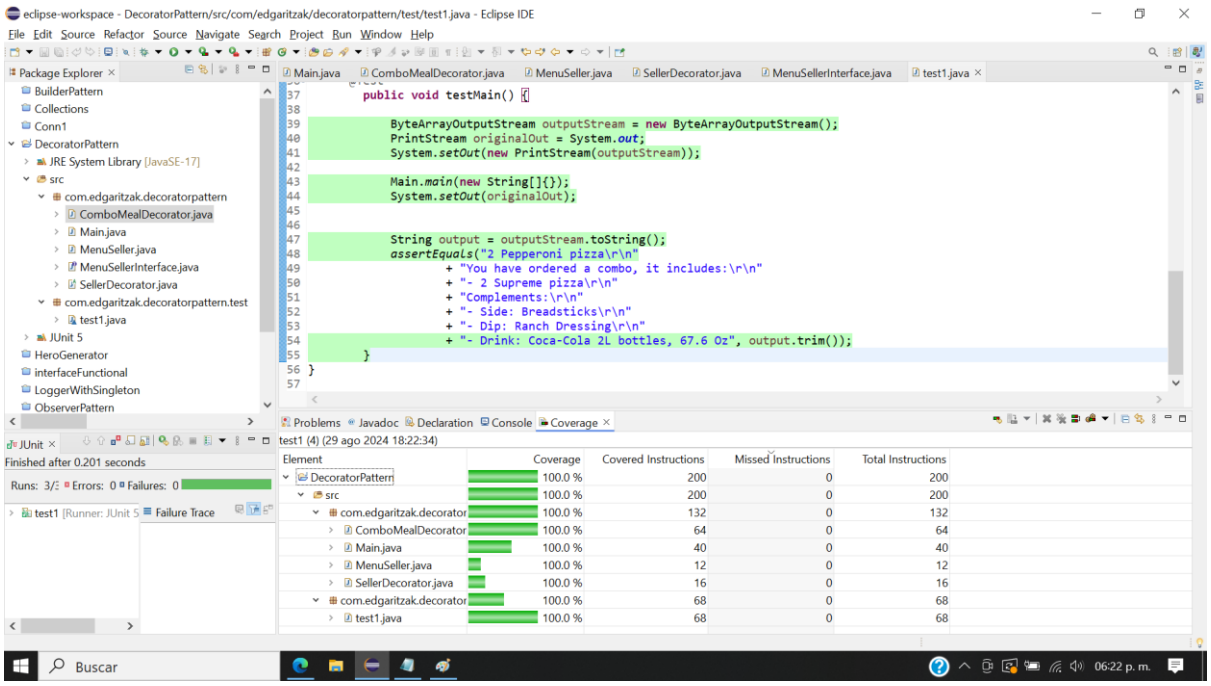
Code explanation:

In this example 2 objects are created using the MenuSellerInterface, a MenuSeller and a MenuSellerDecorator. MenuSellerDecorator adds additional functionality to MenuSeller, it adds the ability to add a drink, a side and a dip. To add functions to a MenuSeller just send it as a parameter to create a new MenuSellerDecorator.

JUnit Testing:

JUnit is a unit testing framework for the Java programming language. It is primarily used to write and run automated tests, which is crucial in software development to ensure that code works correctly and to prevent regression.

Its main advantage is to run tests automatically every time a change is made to the code, which helps to catch bugs quickly during development.



Conclusion:

The Decorator pattern allows to structure business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way, since they all follow a common interface. Some of the advantages of Decorator pattern are:

- Allows to extend an object's behavior without making a new subclass.
- Allows to add or remove responsibilities from an object at runtime.
- Allows to combine several behaviors by wrapping an object into multiple decorators.
- Single Responsibility Principle. It's possible implements many possible variants of behavior into several smaller classes.

References:

[1] Decorator pattern - Wikipedia

[2] Decorator (refactoring.guru)

[3] JUnit - Wikipedia, la enciclopedia libre