6.22 Mẫu thiết kế DECORATOR

6.22.1 Mở đầu

Để hiểu được mẫu thiết kế *Decorator* (người trang trí), giả sử giúp công ty Pizza thực hiện tinh toán thêm cho bề mặt. Một người sử dụng có thể yêu cầu bổ sung một lớp bề mặt thêm trên pizza và công việc của chúng ta là bổ sung bề mặt và tăng giá của nó khi sử dụng hệ thống.

Đây là một cái gì đó giống như bổ sung thêm trách nhiệm cho pizza của chúng ta trong thời gian thực và mẫu thiết kế *Decorator* là phù hợp với kiểu yêu cầu này. Nhưng trước hết chúng ta sẽ tìm hiểu thêm về mẫu hành vi này.

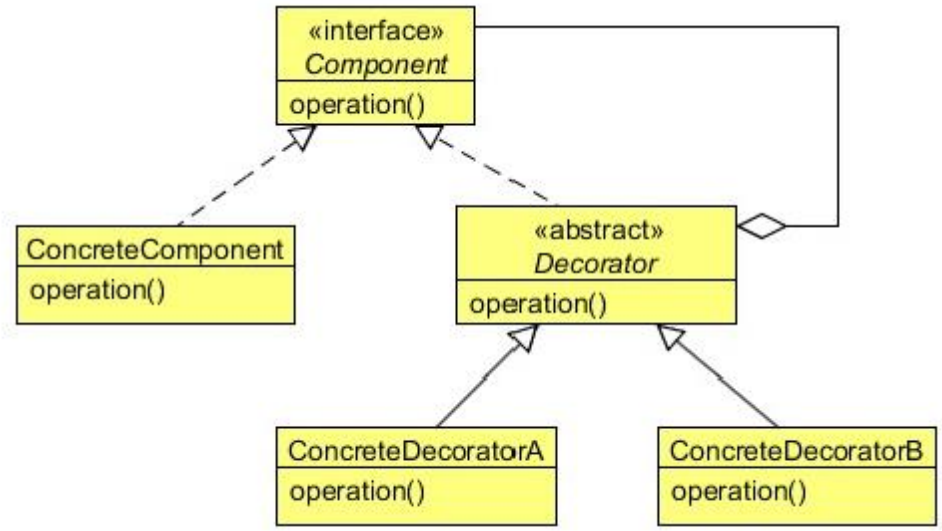
6.22.2 Mẫu thiết kế Decorator là gì

Mục đích của mẫu thiết kế *Decorator* là đính trách nhiệm thêm vào một đối tượng một cách động. *Decorator* cung cấp một lựa chọn linh hoạt cho lớp con để mở rộng chức năng.

Mẫu *Decorator* được sử dụng để mở rộng chức năng của đối tượng một cách động mà không thay đổi lớp gốc hoặc sử dụng giao diện. Điều này được thực hiện bằng cách tạo một đối tượng bao bọc được tham chiếu như *Decorator* bao quanh đối tượng thực tế. .

Một đối tượng *Decorator* được thiết kế để có cùng một giao diện như đối tượng bên dưới. Nó cho phép đối tượng client tương tác với đối tượng *Decorator* theo cách chính xác như nhau như có thể với chính đối tượng bên dưới. Đối tượng *Decorator* chứa một tham chiếu đến đối tượng thực tế. Đối tượng *Decorator* nhận một yêu cầu từ client. Đến lượt nó chuyển tiếp các lời gọi này đến đối tượng bên dưới. Đối tượng *Decorator* bổ sung một số chức năng trước hoặc sau khi chuyển tiếp một số yêu cầu cho đối tượng bên dưới. Điều này đảm bảo rằng chức năng bổ sung có thể được thêm vào bên ngoài cho đối tượng cho trước trong thời gian chạy mà không thay đổi cấu trúc của nó.

Decorator ngăn cản sự gia tăng của các lớp con dẫn đến độ phức tạp và hỗn độn ít hơn. Dễ dàng bổ sung bất cứ tổ hợp khả năng nào. Một khả năng như vậy có thể bổ sung hai lần. Nó trở nên có thể có các đối tượng *decorator* khác nhau đồng thời cho cùng một đối tượng. Một client có thể chọn các khả năng nào mà muốn bằng cách gửi thông điệp đến *decorator* phù hợp.



Component

- Định nghĩa giao diện cho các đối tượng mà có trách nhiệm bổ sung vào chúng một cách động.

ConcreteComponent

- Định nghĩa một đối tượng mà gắn vào nó các trách nhiệm bổ sung có thể đính kèm

Decorator

- Duy trì một tham chiếu đến một đối tượng *Component* và định nghĩa giao diện hợp với giao diện của *Component.*

ConcreteDecorator

- Bổ sung trách nhiệm cho *Component.*

6.22.3 Cài đặt mẫu thiết kế Decorator

Để đơn giản chúng ta tạo giao diện Pizza mà chỉ có hai phương thức.

```
package com.javacodegeeks.patterns.decoratorpattern;

public interface Pizza {

	public String getDesc();
	public double getPrice();
}
```

Phương thức *getDesc* được sử dụng để nhận được mô tả pizza, trong khi phương thức *getPrice* được dùng để nhận giá.

Dưới đây là hai lớp Pizza cụ thể.

```
package com.javacodegeeks.patterns.decoratorpattern;

public class SimplyVegPizza implements Pizza{

	@Override
	public String getDesc() {
		return "SimplyVegPizza (230)";
	}

	@Override
	public double getPrice() {
		return 230;
	}


}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class SimplyNonVegPizza implements Pizza{

	@Override
	public String getDesc() {
		return "SimplyNonVegPizza (350)";
	}

	@Override
	public double getPrice() {
		return 350;
	}

}
```

Decorator đóng gói đối tượng mà chức năng cần thiết cần được tăng cường cho nó, như vậy cần cài đặt cùng giao diện. Dưới đây là *abstract decorator* mà sẽ được mở rộng bởi mọi *decorator* cụ thể.

```
package com.javacodegeeks.patterns.decoratorpattern;

public abstract class PizzaDecorator implements Pizza {

	@Override
	public String getDesc() {
		return "Toppings";
	}

}
```

Sau đây là các lớp *Decorator* cụ thể:

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Broccoli extends PizzaDecorator{

	private final Pizza pizza;

	public Broccoli(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Broccoli (9.25)";
	}
```

```
	@Override
	public double getPrice() {
		return pizza.getPrice()+9.25;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Cheese extends PizzaDecorator{

	private final Pizza pizza;

	public Cheese(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Cheese (20.72)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+20.72;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Chicken extends PizzaDecorator{

	private final Pizza pizza;

	public Chicken(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Chicken (12.75)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+12.75;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class FetaCheese extends PizzaDecorator{

	private final Pizza pizza;

	public FetaCheese(Pizza pizza){

		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Feta Cheese (25.88)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+25.88;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class GreenOlives extends PizzaDecorator{

	private final Pizza pizza;

	public GreenOlives(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Green Olives (5.47)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+5.47;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Ham extends PizzaDecorator{

	private final Pizza pizza;

	public Ham(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Ham (18.12)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+18.12;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Meat extends PizzaDecorator{

	private final Pizza pizza;

	public Meat(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Meat (14.25)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+14.25;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class RedOnions extends PizzaDecorator{

	private final Pizza pizza;

	public RedOnions(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Red Onions (3.75)";
	}


	@Override
	public double getPrice() {
		return pizza.getPrice()+3.75;
	}

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class RomaTomatoes extends PizzaDecorator{

	private final Pizza pizza;

	public RomaTomatoes(Pizza pizza){
		this.pizza = pizza;
	}

	@Override
	public String getDesc() {
		return pizza.getDesc()+", Roma Tomatoes (5.20)";
	}
```

```
        @Override
        public double getPrice() {
                return pizza.getPrice()+5.20;
        }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Spinach extends PizzaDecorator{

        private final Pizza pizza;

        public Spinach(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Spinach (7.92)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+7.92;
        }

}
```

Chúng ta cần trang trí đối tượng pizza của chúng ta với các lớp bề mặt trên. Các lớp trên chứa tham chiếu đến đối tượng pizza mà cần được trang trí.. Đối tượng decorator bổ sung chức năng của nó cho decorator sau khi gọi hàm của decorator.

```
package com.javacodegeeks.patterns.decoratorpattern;

import java.text.DecimalFormat;

public class TestDecoratorPattern {

        public static void main(String[] args) {

                DecimalFormat dformat = new DecimalFormat("#.##");
                Pizza pizza = new SimplyVegPizza();

                pizza = new RomaTomatoes(pizza);
                pizza = new GreenOlives(pizza);
                pizza = new Spinach(pizza);

                System.out.println("Desc: "+pizza.getDesc());
                System.out.println("Price: "+dformat.format(pizza.getPrice()));

                pizza = new SimplyNonVegPizza();

                pizza = new Meat(pizza);
                pizza = new Cheese(pizza);
                pizza = new Cheese(pizza);
                pizza = new Ham(pizza);

                System.out.println("Desc: "+pizza.getDesc());
                System.out.println("Price: "+dformat.format(pizza.getPrice()));
        }

}
```

Code trên cho kết quả đầu ra như sau:

```
Desc: SimplyVegPizza (230), Roma Tomatoes (5.20), Green Olives (5.47), Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72), Cheese (20.72), Ham (18.12)
Price: 423.81
```

Trong lớp trên, trước hết chúng ta tạo đối tượng *SimplyVegPizza* và sau đó trang trí nó với *RomaTomatoes, GreenOlives* và *Spinach*. Mô tả desc trong đầu ra chỉ rõ các lớp bề mặt này được bổ sung cho *SimplyVegPizza* và giá của nó là tổng tất cả.

Chúng ta làm như vậy cho *SimplyNonVegPizza* và bổ sung lớp khác lên nó. Lưu ý rằng bạn có thể trang trí cùng một vật nhiều hơn một lần cho một đối tượng. Trong ví dụ trên, chúng ta đã bổ sung *cheese* hai lần, nó cũng cộng hai lần vào giá, mà có thể thấy ở đầu ra.

Mẫu thiết kế *Decorator* trông có vẻ tốt khi bạn cần bổ sung thêm chức năng cho đối tượng với việc thay đổi nó trong thời gian chạy. Nhưng nó cho kết quả trong nhiều đối tượng nhỏ. Một thiết kế mà sử dụng *Decorator* thường cho kết quả trong hệ thống tích hợp từ nhiều đối tượng nhỏ mà trông giống như vậy. Các đối tượng khác nhau chỉ ở cách chúng được kết nối với nhau, không ở trong lớp của chúng hoặc ở giá trị của các biến của chúng. Mặc dù các hệ thống này là dễ tùy biến bởi những người hiểu chúng, nhưng chúng có thể vẫn là khó để học và bắt lỗi.

6.22.4 Khi nào sử dụng mẫu thiết kế Decorator

Sử dụng mẫu *Decorator* trong các trường hợp sau:

- Bổ sung trách nhiệm cho các đối tượng riêng lẻ một cách động và trong suốt, mà là không làm ảnh hưởng đến các đối tượng khác.
- Đối với các trách nhiệm mà có thể gỡ bỏ.
- Khi mở rộng bởi các lớp con là không thể. Đôi khi số lớn các mở rộng độc lập là có thể và có thể tạo ra sự bùng nổ các lớp con để hỗ trợ mỗi tổ hợp. Hoặc định nghĩa một lớp có thể được che giấu hoặc ngược lại là không sẵn sàng cho tách các lớp con.