“ Wzorzec Dekorator ”

Martyna Folholc
Informatyka zaoczne III rok

Wstęp

Wzorzec projektowy **Dekorator (ang. Decorator)** należy do grupy wzorców **strukturalnych**. Jego głównym celem jest **dynamiczne** rozszerzanie funkcjonalności obiektów bez konieczności tworzenia licznych podklas poprzez dziedziczenie. Dekorator umożliwia „opakowywanie” obiektów w dodatkowe klasy dekorujące (tzw. dekoratory), które przechwytują wywołania metod i realizują dodatkowe operacje, a następnie przekazują wywołanie do obiektu dekorowanego. Dzięki temu uzyskujemy:

- **Elastyczność** – możemy dodawać i usuwać dodatkowe cechy w czasie działania programu.
- **Separację odpowiedzialności** – dodatkowa funkcjonalność jest umieszczona w dekoratorze, a nie w klasie bazowej.
- **Zachowanie interfejsu** – dekorowany obiekt i dekorator implementują ten sam interfejs, co umożliwia traktowanie ich tak samo w reszcie systemu.

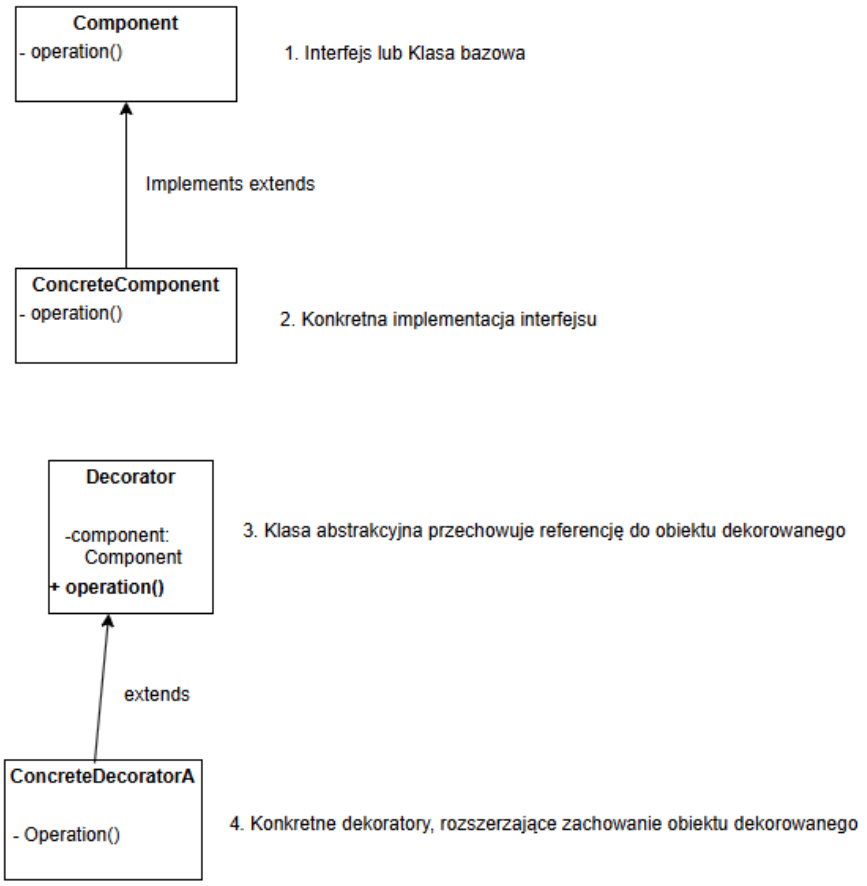
Ogólna idea i motywacja

W tradycyjnym podejściu do rozszerzania funkcjonalności obiektów moglibyśmy tworzyć nowe podklasy. Jednak wielokrotne dziedziczenie funkcjonalności powoduje eksplozję liczby klas i utrudnia zarządzanie kodem (tzw. „inheritance hell”). Dekorator rozwiązuje ten problem przez **kompozycję** zamiast dziedziczenia.

Wyobraźmy sobie sytuację, w której mamy klasę reprezentującą napój (*Beverage*) i różne dodatki (np. mleko, czekolada). Zamiast tworzyć osobne klasy: **KawaZMlekiem, KawaZCzekolada, KawaZMlekiemICzekolada** itp., możemy zdefiniować obiekt **Kawa** i “przyozdabiać” go dekoratorami (np.**MilkDecorator, ChocolateDecorator**). Każdy dekorator dodaje odpowiednie cechy lub modyfikuje istniejące.

Struktura wzorca(UML)

Diagram UML przedstawia strukturę wzorca Dekorator:



Szczegółowy opis diagramu UML

1. **Component**: Jest to interfejs lub klasa abstrakcyjna definiująca operację **operation()** , którą implementują wszystkie elementy hierarchii, w tym dekoratory.
2. **ConcreteComponent**: Klasa implementująca interfejs **Component**, która zawiera podstawową funkcjonalność i może być dekorowana.
3. **Decorator**: Klasa abstrakcyjna implementująca **Component** , przechowująca referencję do dekorowanego obiektu. Deleguje wywołania metody **operation()** do obiektu **Component**, który dekoruje.
4. **ConcreteDecorator**: Konkretne dekoratory, które dziedziczą po klasie **Decorator** i dodają dodatkową funkcjonalność do metody **operation()**.

Implementacja w języku Java - System napojów z dodatkami

1. Interfejs (Component)

```java
public interface Beverage {
    String getDescription();
    double cost();
}
```

getDescription() - zwraca opis napoju (np. “Kawa”, “Kawa z mlekiem” itp.)
cost() - zwraca cenę napoju

2. Konkretna implementacja (ConcreteComponent)

```java
public class Coffee implements Beverage {
    @Override
    public String getDescription() {
        return "Kawa";
    }

    @Override
    public double cost() {
        return 5.0; // bazowa cena kawy
    }
}
```

Ta klasa reprezentuje zwykłą kawę bez dodatków.

3. Klasa abstrakcyjna dekoratora (Decorator)

```java
public abstract class BeverageDecorator implements Beverage {
    // Kompozycja: przechowujemy referencję do obiektu "Beverage"
    protected Beverage beverage;

    public BeverageDecorator(Beverage beverage) {
        this.beverage = beverage;
    }

    // Opcjonalnie możemy zdefiniować podstawową implementację metod
    // lub zostawić je abstrakcyjne.
    @Override
    public String getDescription() {
        return beverage.getDescription();
    }

    @Override
    public double cost() {
        return beverage.cost();
    }
}
```

BeverageDecorator implementuje ten sam interfejs (**Beverage**), dzięki czemu dekorator i obiekt dekorowany mogą być traktowane tak samo.

4. Konkretne dekoratory (ConcreteDecorator)

Dekorator mleka (MilkDecorator)

```java
public class MilkDecorator extends BeverageDecorator {

    public MilkDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public String getDescription() {
        // Rozszerzamy opis bazowy
        return beverage.getDescription() + " + mleko";
    }

    @Override
    public double cost() {
        // Dodajemy koszt mleka do bazowego kosztu
        return beverage.cost() + 1.0;
    }
}
```

Dekorator czekolady (ChocolateDecorator)

```java
public class ChocolateDecorator extends BeverageDecorator {

    public ChocolateDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + " + czekolada";
    }

    @Override
    public double cost() {
        return beverage.cost() + 2.0;
    }
}
```

5. Użycie wzorca w kodzie

```java
public class DecoratorTest {
    public static void main(String[] args) {
        // Tworzymy zwykłą kawę
        Beverage coffee = new Coffee();
        System.out.println(coffee.getDescription() + " => " + coffee.cost() + " zł");

        // Dodajemy mleko
        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " => " + coffee.cost() + " zł");

        // Dodajemy kolejną warstwę dekoratora - czekoladę
        coffee = new ChocolateDecorator(coffee);
        System.out.println(coffee.getDescription() + " => " + coffee.cost() + " zł");
    }
}
```

wynik uruchomienia:

```
Kawa => 5.0 zł
Kawa + mleko => 6.0 zł
Kawa + mleko + czekolada => 8.0 zł
```

Jak widać, w sposób dynamiczny dodawane są nowe cechy (mleko, czekolada), a obiekt pozostaje w tym samym typie **Beverage**.

Zalety i wady wzorca

1. Zalety

Elastyczność – pozwala na dynamiczne dodawanie i usuwanie funkcjonalności obiektów.

Unikanie nadmiernej liczby podklas – zamiast tworzyć wiele odmian klas poprzez dziedziczenie, używamy kompozycji.

Otwarty na rozszerzenia, zamknięty na modyfikacje – zgodnie z zasadą Open/Closed, nie modyfikujemy kodu istniejącej klasy, tylko ją „opakowujemy”.

2. Wady

Wiele małych obiektów – intensywne użycie dekoratorów może prowadzić do dużej liczby obiektów w systemie, co bywa trudniejsze w debugowaniu.

Trudne śledzenie przepływu – wywołanie metody może przechodzić przez wiele dekoratorów, co utrudnia zrozumienie, gdzie i w jaki sposób właściwie zachowanie jest rozszerzane.

Zastosowania

Systemy graficzne: dekoratory mogą dodawać dodatkowe obramowania, cienie, efekty do kontrolki UI.

Strumienie wejścia/wyjścia (Java I/O): klasy dekorujące strumienie, np. **BufferedInputStream, DataInputStream, PushbackInputStream.**

Rozszerzanie funkcjonalności wtyczek: w dużych systemach modułowych, gdzie dynamicznie dodajemy nowe cechy bez modyfikowania kodu bazowego.

Przetwarzanie danych: filtracja i modyfikacja danych „w locie”, np. szyfrowanie, kompresja itp.