Wzorzec projektowy - dekorator

Kamil Oratowski, Piotr Szewerniak, Paula Chajduła, Marcin Caputa

Politechnika Krakowska
Wydział Inżynierii
Materiałowej i Fizyki

Wydział Inżynierii Materiałowej i Fizyki
Politechnika Krakowska

June 26, 2020

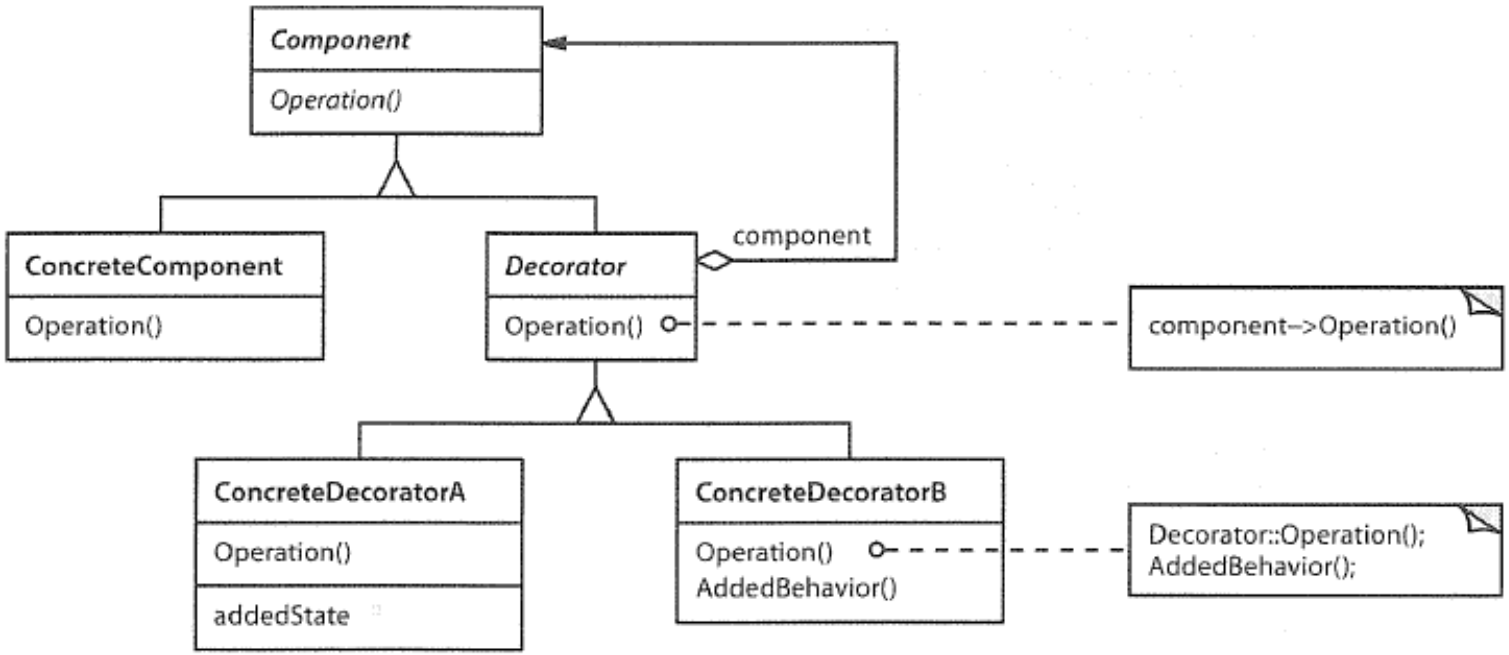
Przeznaczenie i Uzasadnienie

Dynamicznie dołacza dodatkowe obowiazki do obietku. Wzorzec ten udostepnia elestyczny sposób tworzenia podklas o wzbogaconych funkcjach. Dekoratory sa alternatywa dla dziedziczenia. Dziedziczenie rozszerza zachowanie klasy w trakcie kompilacji, w przeciwieństwie do dekoratorów, które rozszerzaja klasy w czasie działania programu.Takie rozwiazanie jest bardziej estetyczne i pozwala na umieszczenie komponentu w innym obiekcie. Dekorator przekazuje żadania do komponentum a przed ich wysłaniem lub potem może wykonywać dodatkowe działania. Umożliwia to rekurencyjne zagnieżdżanie dekoratorówm, a tym samym dodanie dowolnej liczby nowych zadań.

Kiedy stosować?

- W przypadku jezyka Java wzorzec projektowy dekorator jest dość czesto używany w bibliotece standardowej. Za przykład moga tu posłużyć strumienie wykorzystywane przy operacjach na plikach. InputStream jest klasa abstrakcyjna, która posiada wiele dekoratorów, na przykład FileInputStream czy BufferedInputStream.
- Innym przykładem, również z jezyka Java, moga być dekoratory kolekcji. Dekoratory te na przykład pozwalaja na utworzenie kolekcji, która jest synchronizowana czy niemodyfikowalna. Collections zawiera szereg metod zaczynajacych sie od synchronized albo unmodifiable, które tworza instancje dekoratorów.
- W jezyku Python istnieje składnia, która pozwala na łatwe użycie dekoratorów. Można powiedzieć, że ten wzorzec projektowy został wbudowany w jezyk. Notacja @dekorator pozwala dekorować zarówno klasy jak i funkcje. Przykładami dekoratorów dostepnych w bibliotece standardowej moga być @property, @contextlib.contextmanager czy @functools.wraps.

Struktura - diagram UML w przypadku ogólnym



Zalety

- Jedna z czesto polecanych praktyk w programowaniu obiektowym jest preferowanie kompozycji przed dziedziczeniem. Wzorzec projektowy dekorator jest flagowym przykładem użycia tej reguły. Takie podejście pozwala na dynamiczne rozszerzanie funkcjonalności obiektu bez potrzeby kompilacji kodu.
- Niewatpliwa zaleta dekoratora jest możliwość dowolnego łaczenia istniejacych dekoratorów. Każdy z nich bedzie opakowywał kolejny obiekt nie majac świadomości, że jest kolejnym dekoratorem w kolejce. Jest to istotne w przypadku gdy istnieje kilka dodatkowych funkcjonalności, które powinna zawierać rozszerzana klasa.

Wady

- Interfejs dekoratora musi być dokładnie taki sam jak klasy dekorowanej. W niektórych jezykach programowania (na przykład w Javie) może prowadzić to do klas, które maja sporo metod, których implementacja polega na przekazaniu wywołania do dekorowanego obiektu (jeśli dekorator implementuje interfejs). Te wade można rozwiazać stosujac dziedziczenie.
- Dekorator czesto jest płaska klasa". Rozszerza on dekorowana klase o jedna, podstawowa funkcjonalność. Prowadzić to może do sytuacji, w której system zawiera wiele niewielkich klas. W sytuacji gdy zazwyczaj używa sie stałego zbioru dekoratorów użycie standardowego dziedziczenia może ograniczyć te liczbe.

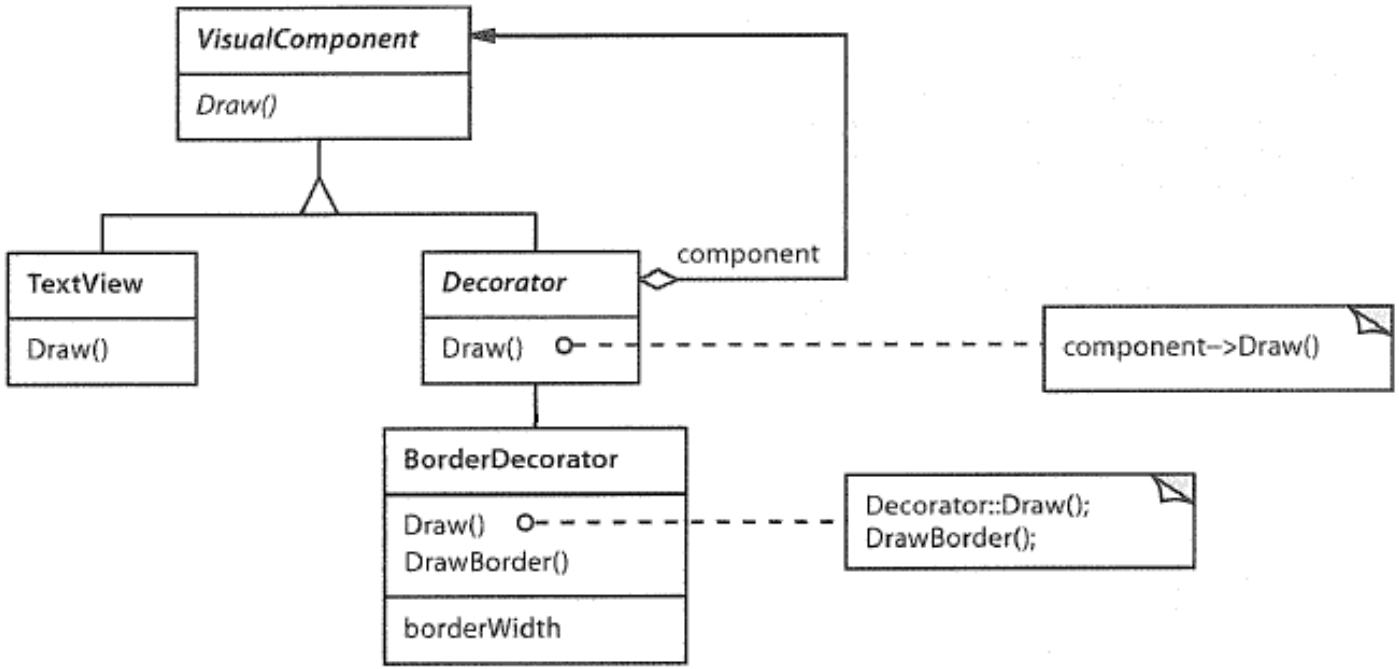
Wykorzystanie w praktyce nr 1

```
1   class VisualComponent{   ///utowrzenie klasy VisualComponent
2       public:
3           VisualComponent(); ///utworzenie konstruktora
4           virtual void Draw(); ///utowrzenie operatora Draw
5           virtual void Resize(); ///utworzenie operatora Resize
6   };
7   class Decorator : public VisualComponent{ ///Zdefiniowanie podklasy Decorator
8       public:
9           Decorator(VisualComponent*); /// utworzenie konstruktora
10
11          virtual void Draw(); ///utworzenie operatora Draw
12          virtual void Resize(); ///utworzenie operatora Resize
13      public:
14          VisualComponent* _component; //zdefiniowanie domyslnej implementacji do zmiennej
15  };
16  void Decorator::Draw(){  ///zdefiniowanie operatora Draw
17      _component->Draw();
18  }
19
20  void Decorator::Resize(){ ///zdefiniowanie operatora Resize
21      _component->Resize();
22  }
```

Wykorzystanie w praktyce nr 1

```
25  class BorderDecorator: public Decorator{    /// dodanie ramki do komponentu zawierającego obiekt tej klasy
26  public:
27      BorderDecorator(VisualComponent*, int borderWidth);
28
29      virtual void Draw();
30  private:
31      void DrawBorder(int);
32  private:
33      int_width;
34  };
35
36  void BorderDecorator::Draw(){ /// podklasa klasy Decorator
37      Decorator::Draw();  /// wyświetlenie obramowania
38      DrawBorder(_width);     /// narysowanie ramki
39  }
40
41  void Windows:SetContents(VisualComponent* contents){
42      Window* window = new Window; /// utworzenie okna, w którym umieścimy obiekt TextView
43      TextView* textView = new TextView; /// utworzenie obiektu TextView
44      window->SetContents(textView); /// umieszczenie obiektu TextView w oknie
45      window->SetContents(new BorderDecorator(new ScrollDecorator(textView),1))
46  }
```

Diagram UML w przypadku naszego kodu



Wykorzystanie w praktyce nr 2

```java
public class Pizza{ ///utworzenie klasy która reprezentuje podstawową pizze
    private static final BigDecimal BASE_PRICE = new BigDecimal(12);

    public BigDecimal getPrice(){
        return BASE_PRICE;
    }
    public String toString(){
        return "Pizza";
    }
}
public class PizzaWithMozzarella extends Pizza{  ///utworzenie dekoratora który okresla pizze z mozzarella
    private static final BigDecimal MOZZARELLA_PRICE= new BigDecimal(5);
    private final Pizza basePizza;

    public PizzaWithMozzarella(Pizza basePizza){   ///utworzenie konstruktora klasy
    this.basePizza=basePizza;
}
public BigDecimal getPrice(){
    return basePizza.getPrice().add(MOZZARELLA_PRICE);  ///dodanie do ceny bazowej pizzy ceny sera
}
}
```

Wykorzystanie w praktyce nr 2

```
22  public class Restaurant{
23      public static void main(String[] args){
24          Pizza margherita - new Pizza(); ///utworzenie bazowe pizzy
25          Pizza withMozzarella = new PizaWithMozzarella(margherita);  ///utworzenie Pizzy z Mozzarella
26          Pizza withMozzarellaAndHam = new PizzaWithHam(withMozzarella); ///utworzenie Pizzy z Mozzarella i Szynka
27          Pizza withMozzarellaHamAndBasil=new PizzaWithBasil(withMozzarellaAndHam); ///utworzenie Pizzy z Mozzarella,Szynka i Bazylia
28
29          DecimalFormat df = new DecimalFormat("#,00 zł"); /// zdefiniowanie formatu
30          for(Pizza pizza : List.of(margherita, withMozzarella, withMozzarellaAndHam, withMozzarellaHamAndBasil)){
31              System.out.println(String.format("%s costs %s.", pizza, df.format(pizza.getPrice())));
32          }
33          }
34      }
```

Bibliografia

- "Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku" - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- https://bykowski.pl/dekorator-wzorzec-projektowy/
- https://www.samouczekprogramisty.pl/wzorzec-projektowy-dekorator
- https://pl.wikipedia.org/wiki/Dekorator_(wzorzec_projektowy)