Decorator (dekorator)

Cel:

Dołącza dynamicznie nową funkcjonalność do obiektu – elastyczna alternatywa dziedziczenia.

Przykład:

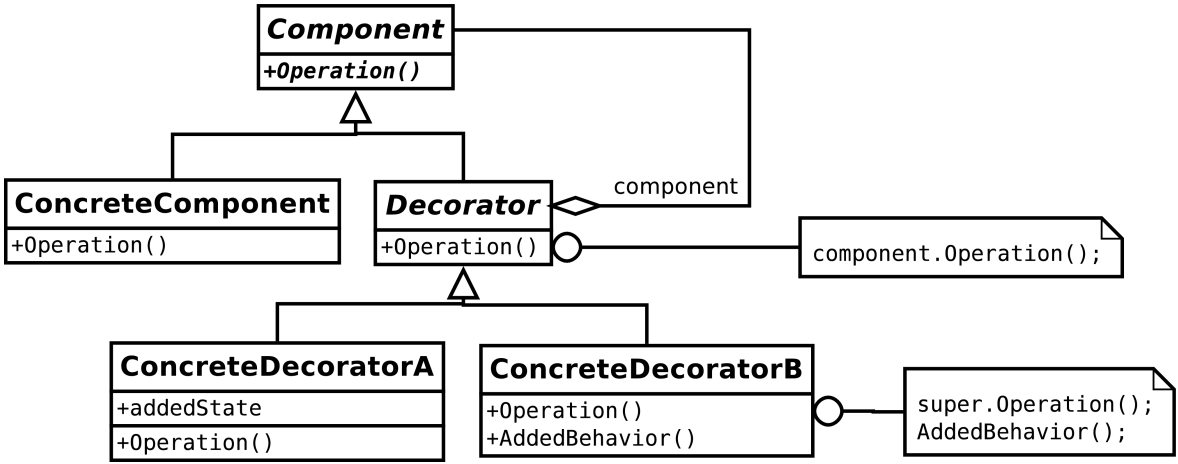
```
interface iPlik {
     void zapisz(String tekst);
     String odczytaj();
}
class Plik implements iPlik {
     public void zapisz(String tekst) {
          ...
     }
     public String odczytaj() {
          return ...
     }
     ...
}
```

```
class PlikSzyfrowany extends Plik {
     public void zapisz(String tekst) { ... }
     public String odczytaj() { ... }
}

class PlikSkompresowany extends Plik {
     public void zapisz(String tekst) { ... }
     public String odczytaj() { ... }
}

class PlikSzyfrowanySkompresowany extends Plik {
     public void zapisz(String tekst) { ... }
     public String odczytaj() { ... }
}
```

Zastosowanie:

- Aby dynamicznie dodać funkcjonalność pojedynczym obiektom, w sposób przeźroczysty, czyli bez wpływania na inne obiekty;
- aby móc później tę funkcjonalność usunąć;
- gdy dziedziczenie jest niepraktyczne, np. dostępne jest wiele niezależnych rozszerzeń, próba zapisania wszystkich ich kombinacji stworzyłaby ogromną liczbę klas (lub, z różnych powodów, nie możemy dziedziczyć z danej klasy).

Struktura:

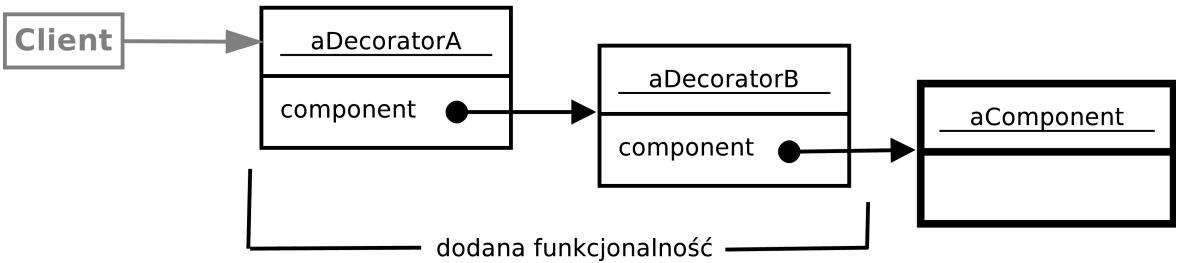


Składniki:

- ***Component*** – definiuje interfejs obiektów, które mają być dekorowane.
- ***ConcreteComponent*** - obiekt, który będzie dekorowany.
- ***Decorator*** – zawiera referencję do Componentu i ma interfejs zgodny z Componentem.
- ***ConcreteDecorator*** – dodaje funkcjonalność do swego komponentu.

Zależności:

Dekorator odsyła żądanie do swego Komponentu, swoje dodatkowe operacje wykonując przed lub po odesłaniu.



```
interface iPlik {
     void zapisz(String tekst);
     String odczytaj();
}

class Plik implements iPlik {
     public void zapisz(String tekst) { ... }
     public String odczytaj() { ... }
     ...
}

abstract class Dekorator implements iPlik {
     protected iPlik komp;
     public Dekorator(iPlik komponent) {
          this.komp = komponent;
     }
     public void zapisz(String tekst) { komp.zapisz(tekst); }
     public String odczytaj() { return komp.odczytaj(); }
}
```

```
class DekoratorSzyfrowanie extends Dekorator {
     public DekoratorSzyfrowanie(iPlik komponent) {
          super(komponent);
     }
     public void zapisz(String tekst) {
          super.zapisz(szyfruj(tekst));
     }
     public String odczytaj() {
          return odszyfruj(super.odczytaj());
     }
     public String szyfruj(String tekst) { ... }
     public String odszyfruj(String tekst) { ... }
}

class DekoratorKompresja extends Dekorator {
     public DekoratorKompresja(iPlik komponent) {
          super(komponent);
     }
     public void zapisz(String tekst) { ... }
     public String odczytaj() { ... }
     ...
}
```

```
// ... Client:
iPlik plik = new DekoratorKompresja(
                 new DekratorSzyfrowanie(
                     new Plik("./dane.txt")));
plik.zapisz("Ala ma kota");
plik.zamknij();
```

Konsekwencje:

1. Rozwiązanie bardziej elastyczne, niż dziedziczenie. Można dodawać i usuwać funkcjonalność dynamicznie (a także dowolnie łączyć funkcjonalności), bez tworzenia nowych klas.
2. Nie musimy projektować dużej, konfigurowalnej klasy, tylko małą, prostą, do której opcje zostaną dodane później. Definiowanie nowej funkcjonalności jest proste – klasy dekoratorów są niezależne od siebie i każdy odpowiada tylko za jedną funkcjonalność. Możliwe jest również dekorowanie klas potomnych.
3. Skutkuje systemem złożonym z dużej liczby drobnych, podobnych obiektów (różniących się nie stanem, a sposobem połączenia). Może być to trudne w utrzymaniu.

Implementacja:

1. Interfejs dekoratora musi pasować do interfejsu komponentów, które ma dekorować.

```
abstract class Dekorator implements iPlik {
     protected iPlik komp;
     public Dekorator(iPlik komponent) {
          this.komp = komponent;
     }
     ...
}
```

2. Bez abstrakcyjnej klasy dekoratora – np. gdy mamy dodać tylko jedną funkcjonalność. Tak przeważnie będzie, gdy dodajemy coś do już istniejącej hierarchii.

3. Klasa Komponent – powinna być lekka, gdyż dziedziczą z niej wszystkie dekoratory. Powinna zatem jedynie definiować interfejs, a nie przechowywać dane.

```
interface iPlik {
     void zapisz(String tekst);
     String odczytaj();
}
```

4. Dekorator jest niewidoczny dla Komponentu (nie wie, w jaki sposób jest udekorowany).
5. Dostęp do udekorowanego obiektu – przez interfejs Component, ale możliwe jest też rozszerzenie interfejsu, np. w IO Javy dekoratory dodają własne metody.

```
abstract class OutputStream {
     public void write(byte[] b) { ... }
     ...
}

class FileOutputStream extends OutputStream {
     public FileOutputStream(File file) { ... }
     ...
}

class ObjectOutputStream extends OutputStream {
     public void writeObject(Object obj) { ... }
     public void writeIntt(int val) { ... }
     ...
}
```

```
// Client:
ObjectOutputStream oos = new ObjectOutputStream(
                            new FileOutputStream("plik.tmp"));

oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());

oos.close();
```

Powiązania:

- Dekorator zmienia działanie obiektu, natomiast Adapter – interfejs.
- Proxy z kolei zmienia sposób dostępu do obiektu.
- Composite – Dekorator może być traktowany jako zubożony Kompozyt (tylko jeden komponent), ale podstawowym celem Dekoratora nie jest agregacja, a dodanie funkcjonalności.
- Dekorator to niejako „powłoka” obiektu, która zmienia jego zachowanie, alternatywą jest zmiana „jądra” – wzorce Bridge, czy Strategy – będą lepsze, gdy komponent jest być „ciężki”.
- Udekorowany obiekt może być posłużyć jako Prototyp.