Wzorzec Dekorator (Decorator Pattern)

Wygenerowany przy pomocy AI

> Please describe decorator pattern in decorator-pattern.md file. Then show changes you've made to files in the project and describe why is is helpful.
> Generate this file in polish.

Charakterystyka wzorca

Wzorzec Dekorator należy do grupy wzorców strukturalnych i pozwala na dynamiczne dodawanie nowych funkcjonalności do istniejących obiektów bez modyfikacji ich struktury. Zamiast korzystania z dziedziczenia (które jest statyczne), Dekorator wykorzystuje kompozycję do rozszerzania funkcjonalności w czasie wykonania programu.

Główne cechy wzorca Dekorator:

- Pozwala na dodawanie funkcjonalności do obiektów bez modyfikacji ich kodu
- Umożliwia łączenie wielu dekoratorów dla uzyskania złożonych zachowań
- Jest alternatywą dla dziedziczenia, gdy potrzebna jest elastyczność
- Respektuje zasadę otwarte-zamknięte (open/closed principle)

Implementacja wzorca Dekorator w projekcie

W projekcie wzorzec Dekorator został zastosowany do rozszerzania funkcjonalności klasy `Towar`. Wprowadzono następujące klasy:

1. Abstrakcyjny Dekorator

```
package rabaty.dekoratory;

import magazyn.Towar;

public abstract class TowarDekorator extends Towar {
    protected Towar dekorowany;

    public TowarDekorator(Towar dekorowany) {
        super(dekorowany.getCena(), dekorowany.getNazwa());
        this.dekorowany = dekorowany;
    }

    @Override
    public double getCena() {
        return dekorowany.getCena();
    }

    @Override
    public double getCenaPoRabacie() {
        return dekorowany.getCenaPoRabacie();
    }

    @Override
    public String getNazwa() {
        return dekorowany.getNazwa();
    }
}
```

2. Konkretne Dekoratory

RabatProcentowyDekorator

```
package rabaty.dekoratory;

import magazyn.Towar;

public class RabatProcentowyDekorator extends TowarDekorator {
    private final double procentRabatu;

    public RabatProcentowyDekorator(Towar dekorowany, double procentRabatu) {
        super(dekorowany);
        this.procentRabatu = procentRabatu;
    }

    @Override
    public double getCenaPoRabacie() {
        double cenaPoRabacie = dekorowany.getCenaPoRabacie();
        return cenaPoRabacie * (1 - procentRabatu/100);
    }

    @Override
    public String getNazwa() {
        return dekorowany.getNazwa() + " (rabat " + procentRabatu + "%)";
    }
}
```

RabatKwotowyDekorator

```
package rabaty.dekoratory;

import magazyn.Towar;

public class RabatKwotowyDekorator extends TowarDekorator {
    private final double kwotaRabatu;

    public RabatKwotowyDekorator(Towar dekorowany, double kwotaRabatu) {
        super(dekorowany);
        this.kwotaRabatu = kwotaRabatu;
    }

    @Override
    public double getCenaPoRabacie() {
        double cenaPoRabacie = dekorowany.getCenaPoRabacie();
        return Math.max(0, cenaPoRabacie - kwotaRabatu);
    }

    @Override
    public String getNazwa() {
        return dekorowany.getNazwa() + " (rabat " + kwotaRabatu + " zł)";
    }
}
```

DarmoweDostarczeniaDekorator

```
package rabaty.dekoratory;

import magazyn.Towar;

public class DarmoweDostarczeniaDekorator extends TowarDekorator {
    private final double wartoscGratisu;

    public DarmoweDostarczeniaDekorator(Towar dekorowany, double wartoscGratisu) {
        super(dekorowany);
        this.wartoscGratisu = wartoscGratisu;
    }

    @Override
    public String getNazwa() {
        return dekorowany.getNazwa() + " + darmowa dostawa (wartość " + wartoscGratisu + " zł)";
    }
}
```

PodatekDekorator

```
package rabaty.dekoratory;

import magazyn.Towar;

public class PodatekDekorator extends TowarDekorator {
    private final double stawkaPodatku;

    public PodatekDekorator(Towar dekorowany, double stawkaPodatku) {
        super(dekorowany);
        this.stawkaPodatku = stawkaPodatku;
    }

    @Override
    public double getCenaPoRabacie() {
        double cenaPoRabacie = dekorowany.getCenaPoRabacie();
        return cenaPoRabacie * (1 + stawkaPodatku/100);
    }

    @Override
    public String getNazwa() {
        return dekorowany.getNazwa() + " + podatek " + stawkaPodatku + "%";
    }
}
```

3. Zastosowanie w kodzie UI

W klasie `Ui` dodano kod demonstracyjny pokazujący wykorzystanie wzorca Dekoratora:

```
// Część 2: Demonstracja wzorca Dekoratora
System.out.println("\n=== DEMONSTRACJA WZORCA DEKORATORA ===");

// Podstawowy towar
Towar laptopPodstawowy = new Towar(3000, "Laptop");
System.out.println("Podstawowy produkt: " + laptopPodstawowy.getNazwa());
System.out.println("Cena podstawowa: " + laptopPodstawowy.getCena() + " zł");

// Dekorator 1: Rabat procentowy 10%
Towar laptopRabat10 = new RabatProcentowyDekorator(laptopPodstawowy, 10);
System.out.println("\nPo dodaniu rabatu procentowego: " + laptopRabat10.getNazwa());
System.out.println("Cena po rabacie: " + laptopRabat10.getCenaPoRabacie() + " zł");

// Dekorator 2: Dodanie podatku 23%
Towar laptopRabat10Podatek = new PodatekDekorator(laptopRabat10, 23);
System.out.println("\nPo dodaniu podatku: " + laptopRabat10Podatek.getNazwa());
System.out.println("Cena końcowa: " + laptopRabat10Podatek.getCenaPoRabacie() + " zł");

// Dekorator 3: Darmowa dostawa
Towar laptopRabat10PodatekDostawa = new DarmoweDostarczeniaDekorator(laptopRabat10Podatek, 20);
System.out.println("\nZ darmową dostawą: " + laptopRabat10PodatekDostawa.getNazwa());
System.out.println("Cena końcowa: " + laptopRabat10PodatekDostawa.getCenaPoRabacie() + " zł");

// Przykład niezależnego łączenia dekoratorów
Towar telefon = new Towar(1500, "Telefon");
telefon = new RabatKwotowyDekorator(telefon, 200);
telefon = new PodatekDekorator(telefon, 23);

System.out.println("\nInny przykład: " + telefon.getNazwa());
System.out.println("Cena końcowa: " + telefon.getCenaPoRabacie() + " zł");

// Dodanie do faktury elementu z dekoratorem
Faktura f2 = new Faktura(teraz.getTime(), "Dekoracja Sp. z o.o.");
f2.dodajPozycje(laptopRabat10PodatekDostawa, 1);
f2.dodajPozycje(telefon, 2);

wypiszFakture(f2);
```

Korzyści z wdrożenia wzorca Dekorator w projekcie

1. **Elastyczność modyfikowania towarów** - Możemy dynamicznie dodawać różne modyfikatory do towarów (rabaty, podatki, usługi dodatkowe) bez konieczności tworzenia osobnych klas dla każdej kombinacji modyfikatorów.
2. **Kombinowanie modyfikatorów** - Możemy łączyć różne dekoratory w dowolnej kolejności, tworząc zaawansowane konfiguracje produktów (np. produkt z rabatem procentowym, potem z podatkiem, a na końcu z darmową dostawą).
3. **Zgodność z zasadami SOLID** - Respektujemy zasadę otwarte-zamknięte, umożliwiając rozszerzanie funkcjonalności bez modyfikacji istniejącego kodu.
4. **Separacja odpowiedzialności** - Każdy dekorator odpowiada za dokładnie jedną modyfikację funkcjonalności, co ułatwia testowanie i konserwację kodu.
5. **Lepsza alternatywa dla dziedziczenia** - Zamiast tworzyć dużą hierarchię klas dziedziczących (np. TowarZRabatemProcentowym, TowarZRabatemProcentowymIPodatkiem itp.), wykorzystujemy kompozycję obiektów.
6. **Czytelny kod klienta** - Tworzenie złożonych obiektów jest przejrzyste i czytelne, a dodawanie nowych dekoratorów nie

wymaga zmian w istniejącym kodzie klienta.

Podsumowując, wzorzec Dekorator znacząco poprawił elastyczność systemu sprzedaży, umożliwiając łatwe łączenie różnych modyfikatorów cen i funkcji produktów bez tworzenia skomplikowanej hierarchii klas.