Wzorzec Strategia (Strategy Pattern)

Wygenerowany przy pomocy AI

> Please describe strategy pattern in strategy-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 Strategia należy do grupy wzorców behawioralnych i umożliwia zdefiniowanie rodziny algorytmów, enkapsulację każdego z nich oraz uczynienie ich wymiennymi. Strategia pozwala na zmianę algorytmu niezależnie od klientów, którzy z niego korzystają.

Główne cechy wzorca Strategia:

- Definiuje rodzinę algorytmów
- Enkapsuluje każdy algorytm w osobnej klasie
- Umożliwia wymianę algorytmów w czasie działania programu
- Oddziela implementację algorytmu od kodu, który go używa
- Eliminuje instrukcje warunkowe związane z wyborem algorytmu

Implementacja wzorca Strategia w projekcie

W projekcie wzorzec Strategia został zastosowany do implementacji różnych typów rabatów dla towarów. Wprowadzono następujące klasy:

1. Interfejs strategii

```
package rabaty;

public interface ObliczCenePoRabacie {
    double obliczCenePoRabacie(double cena);
}
```

2. Konkretne strategie

RabatProcentowy

```
package rabaty;

public class RabatProcentowy implements ObliczCenePoRabacie {
    private final double procentRabatu;

    public RabatProcentowy(double procentRabatu) {
        this.procentRabatu = procentRabatu;
    }

    @Override
    public double obliczCenePoRabacie(double cena) {
        return cena * (1 - procentRabatu/100);
    }
}
```

RabatKwotowy

```
package rabaty;

public class RabatKwotowy implements ObliczCenePoRabacie {
    private final double kwotaRabatu;

    public RabatKwotowy(double kwotaRabatu) {
        this.kwotaRabatu = kwotaRabatu;
    }

    @Override
    public double obliczCenePoRabacie(double cena) {
        return Math.max(0, cena - kwotaRabatu);
    }
}
```

3. Adapter dla zewnętrznej biblioteki rabatów

```java
package rabaty;

import rabatlosowy.LosowyRabat;

public class AdapterLosowegoRabatu implements ObliczCenePoRabacie {
    private final LosowyRabat losowyRabat;

    public AdapterLosowegoRabatu(LosowyRabat losowyRabat) {
        this.losowyRabat = losowyRabat;
    }

    @Override
    public double obliczCenePoRabacie(double cena) {
        // Zakładam, że LosowyRabat zwraca procent rabatu
        double rabatProcent = losowyRabat.losujRabat();
        return cena * (1 - rabatProcent/100);
    }
}
```

4. Klasa kontekstu (Towar)

```
package magazyn;

import rabaty.ObliczCenePoRabacie;

public class Towar {
 private double cena;
 private String nazwa;
 private ObliczCenePoRabacie strategiaRabatu;

 public Towar(double cena, String nazwa)
 {
  this.cena=cena;
  this.nazwa=nazwa;
 }

 //operacje na cenie
 public void setCena(double cena)
 {
  this.cena=cena;
 }

 public double getCena()
 {
  return cena;
 }
 //operacje na nazwie towaru
 public String getNazwa()
 {
  return nazwa;
 }
 public void setNazwa(String nazwa)
 {
  this.nazwa=nazwa;
 }

 // operacje związane ze strategią rabatu
 public void ustawStrategieRabatu(ObliczCenePoRabacie strategiaRabatu) {
  this.strategiaRabatu = strategiaRabatu;
 }

 public double getCenaPoRabacie() {
  if (strategiaRabatu == null) {
   return cena;
  }
  return strategiaRabatu.obliczCenePoRabacie(cena);
 }
}
```

5. Zastosowanie w kodzie UI

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

```
// Część 1: Demonstracja wzorca Strategii
System.out.println("=== DEMONSTRACJA WZORCA STRATEGII ===");

// Tworzymy towary
Towar t1 = new Towar(10, "buty");
Towar t2 = new Towar(2, "skarpety");

// Tworzymy strategie rabatowe
RabatProcentowy rabat20procent = new RabatProcentowy(20);
RabatKwotowy rabat1zl = new RabatKwotowy(1);

// Stosujemy strategie do towarów
t1.ustawStrategieRabatu(rabat20procent);
t2.ustawStrategieRabatu(rabat1zl);

System.out.println("Buty - cena bez rabatu: " + t1.getCena() + " zł");
System.out.println("Buty - cena po rabacie 20%: " + t1.getCenaPoRabacie() + " zł");

System.out.println("Skarpety - cena bez rabatu: " + t2.getCena() + " zł");
System.out.println("Skarpety - cena po rabacie 1 zł: " + t2.getCenaPoRabacie() + " zł");

// Adapter do losowego rabatu
LosowyRabat lr = new LosowyRabat();
AdapterLosowegoRabatu adapterLosowy = new AdapterLosowegoRabatu(lr);

// Zmiana strategii rabatowej
t1.ustawStrategieRabatu(adapterLosowy);
System.out.println("Buty - cena po losowym rabacie: " + t1.getCenaPoRabacie() + " zł");
```

Korzyści z wdrożenia wzorca Strategia w projekcie

1. **Elastyczność systemu rabatów** - Możemy łatwo dodawać nowe rodzaje rabatów bez modyfikowania klasy `Towar`. Wystarczy stworzyć nową klasę implementującą interfejs `ObliczCenePoRabacie`.
2. **Wymienność algorytmów** - Możemy dynamicznie zmieniać sposób obliczania rabatu dla danego towaru w trakcie działania programu poprzez metodę `ustawStrategieRabatu()`.
3. **Integracja z zewnętrznymi bibliotekami** - Dzięki zastosowaniu wzorca Adapter w połączeniu ze Strategią, łatwo zintegrowaliśmy zewnętrzną klasę `LosowyRabat` z naszym systemem.
4. **Eliminacja rozbudowanych warunków** - Zamiast rozbudowanych instrukcji warunkowych w metodzie obliczającej cenę po rabacie, każdy algorytm jest zamknięty w osobnej klasie.
5. **Większa testowalność** - Każda strategia rabatu może być testowana niezależnie, co ułatwia pisanie testów jednostkowych.
6. **Łatwe rozszerzanie** - System jest otwarty na rozszerzenia (nowe strategie rabatów) bez konieczności modyfikacji istniejącego kodu.
7. **Czytelność kodu** - Kod jest bardziej czytelny, ponieważ każda strategia ma jasno określoną odpowiedzialność.

Podsumowując, wzorzec Strategia znacząco poprawił elastyczność i możliwość rozszerzania systemu rabatów w projekcie, jednocześnie utrzymując kod czysty i zgodny z zasadą pojedynczej odpowiedzialności (SRP) oraz zasadą otwarte-zamknięte (OCP).