AI - Kwiecień 2025

Autorzy: Radosław Szepielak, Grzegorz Rybiński

Silnik AI: Claude 3.7 Sonnet

1. Przeanalizuj zmiany, które wprowadziłeś, między kodem źródłowym (oryginalnym) a kodem zmodyfikowanym z wdrożonymi wzorcami Strategia i Obserwator. Najpierw wymień wprowadzone zmiany, następnie je rozwiń.

Główne zmiany wprowadzone w kodzie:

1. **Wzorzec Strategia:**
   - Zastosowano interfejs `ObliczCenePoRabacie` (który już istniał w oryginalnym kodzie)
   - Dodano dwie konkretne strategie rabatu: `RabatProcentowy` i `RabatKwotowy`
   - Zmodyfikowano klasę `Pozycja` dodając pole `strategiaRabatu` oraz metodę `zastosujRabat()`
   - Zmieniono metodę `przeliczWartosc()` aby uwzględniała zastosowany rabat
2. **Wzorzec Obserwator:**
   - Stworzono nowy interfejs `ObserwatorPozycji`
   - Dodano w klasie `Pozycja` listę obserwatorów oraz metody do zarządzania nimi
   - Zmodyfikowano wszystkie metody zmieniające stan `Pozycji` aby powiadamiały obserwatorów
   - Faktura stała się obserwatorem pozycji, reagującym na zmiany automatycznym przeliczeniem sumy
3. **Inne zmiany funkcjonalne:**
   - Dodano metodę `dodajPozycjeZRabatem()` w klasie `Faktura`
   - Dodano metodę `getPozycja()` w klasie `Faktura` dla łatwiejszego dostępu do konkretnych pozycji
   - Rozbudowano klasę `Ui` o demonstrację działania wzorców
   - Dodano metodę `wypiszPozycje()` do klasy `Faktura`

Korzyści z wprowadzonych zmian:

1. **Korzyści ze wzorca Strategia:**
   - Elastyczne zarządzanie różnymi typami rabatów
   - Możliwość łatwego dodawania nowych strategii rabatowych bez modyfikacji istniejącego kodu
   - Dynamiczna zmiana sposobu obliczania rabatu w czasie działania programu
2. **Korzyści ze wzorca Obserwator:**
   - Automatyczna aktualizacja sumy faktury po każdej zmianie pozycji
   - Luźniejsze powiązania między klasami (Pozycja nie musi znać Faktury)
   - Eliminacja błędów wynikających z zapomnienia o ręcznym przeliczeniu sumy

Analiza wdrożonych wzorców projektowych

1. Wzorzec Strategia (Strategy Pattern)

Cel wzorca:

Wzorzec Strategia definiuje rodzinę algorytmów, hermetyzuje je i umożliwia ich wymienne stosowanie. Pozwala na zmianę algorytmu niezależnie od klientów, którzy go używają.

Zmiany w kodzie:

1.1. Interfejs ObliczCenePoRabacie:

Interfejs już istniał w oryginalnym kodzie, ale został rozszerzony przez implementacje.

```
java

// Oryginalny interfejs
package rabaty;

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

1.2. Nowe klasy implementujące strategie rabatu:

```
java

// Nowa klasa: RabatProcentowy.java
package rabaty;

public class RabatProcentowy implements ObliczCenePoRabacie {
    private double procentRabatu;

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

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

// Nowa klasa: RabatKwotowy.java
package rabaty;

public class RabatKwotowy implements ObliczCenePoRabacie {
    private double kwotaRabatu;

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

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

1.3. Modyfikacja klasy Pozycja do obsługi strategii rabatu:

Oryginalny kod:

```java
private void przeliczWartosc() {
    this.wartosc = this.ilosc * this.cena;
}
```

Zmodyfikowany kod:

```java
// Dodanie pola strategii rabatu
private ObliczCenePoRabacie strategiaRabatu;

// Dodanie metody do zastosowania rabatu
public void zastosujRabat(ObliczCenePoRabacie strategiaRabatu) {
    this.strategiaRabatu = strategiaRabatu;
    this.przeliczWartosc();
    this.powiadomObserwatorow(); // powiązanie z wzorcem Obserwator
}

// Zmodyfikowana metoda przeliczania wartości
private void przeliczWartosc() {
    if (strategiaRabatu != null) {
        double cenaPoRabacie = strategiaRabatu.obliczCenePoRabacie(this.cena);
        this.wartosc = this.ilosc * cenaPoRabacie;
    } else {
        this.wartosc = this.ilosc * this.cena;
    }
}
```

2. Wzorzec Obserwator (Observer Pattern)

Cel wzorca:

Wzorzec Obserwator definiuje zależność jeden-do-wielu między obiektami w taki sposób, że gdy jeden obiekt zmienia stan, wszystkie zależne od niego obiekty są powiadamiane i aktualizowane automatycznie.

Zmiany w kodzie:

2.1. Nowy interfejs ObserwatorPozycji:

```java
// Nowy interfejs: ObserwatorPozycji.java
package dokumenty;

public interface ObserwatorPozycji {
    void aktualizujPozycje();
}
```

2.2. Modyfikacja klasy Pozycja do obsługi obserwatorów:

Dodane pola i metody:

```java
// Lista obserwatorów
private ArrayList<ObserwatorPozycji> obserwatorzy = new ArrayList<>();

// Metody zarządzania obserwatorami
public void dodajObserwatora(ObserwatorPozycji obserwator) {
    obserwatorzy.add(obserwator);
}

public void usunObserwatora(ObserwatorPozycji obserwator) {
    obserwatorzy.remove(obserwator);
}

protected void powiadomObserwatorow() {
    for (ObserwatorPozycji obserwator : obserwatorzy) {
        obserwator.aktualizujPozycje();
    }
}
```

Modyfikacja istniejących metod:

W oryginalnej wersji po zmianie pozycji trzeba było ręcznie przeliczać sumę faktury. W nowej wersji po każdej modyfikacji Pozycji automatycznie powiadamiani są obserwatorzy:

```java
// Przykład modyfikacji metody setIlosc
// Oryginalny kod:
public void setIlosc(double ilosc) {
    this.ilosc = ilosc;
    this.przeliczWartosc();
}

// Zmodyfikowany kod:
public void setIlosc(double ilosc) {
    this.ilosc = ilosc;
    this.przeliczWartosc();
    this.powiadomObserwatorow(); // Powiadomienie obserwatorów o zmianie
}
```

2.3. Modyfikacja klasy Faktura do pełnienia roli obserwatora:

Zmiany w metodzie dodajPozycje:

```java
// Oryginalny kod:
public void dodajPozycje(Towar towar, double ilosc) {
    pozycje.add(new Pozycja(towar, ilosc));
    this.przeliczSume();
}

// Zmodyfikowany kod:
public void dodajPozycje(Towar towar, double ilosc) {
    Pozycja pozycja = new Pozycja(towar, ilosc);

    // Implementacja wzorca Obserwator - dodanie faktury jako obserwatora pozycji
    pozycja.dodajObserwatora(new ObserwatorPozycji() {
        @Override
        public void aktualizujPozycje() {
            przeliczSume();
        }
    });

    pozycje.add(pozycja);
    this.przeliczSume();
}
```

Nowa metoda dodajPozycjeZRabatem:

```java
// Nowa metoda pozwalająca dodać pozycję z rabatem
public void dodajPozycjeZRabatem(Towar towar, double ilosc, ObliczCenePoRabacie rabat) {
    Pozycja pozycja = new Pozycja(towar, ilosc);

    // Implementacja wzorca Obserwator
    pozycja.dodajObserwatora(new ObserwatorPozycji() {
        @Override
        public void aktualizujPozycje() {
            przeliczSume();
        }
    });

    // Zastosowanie strategii rabatu
    pozycja.zastosujRabat(rabat);

    pozycje.add(pozycja);
    this.przeliczSume();
}
```

Nowa metoda getPozycja:

```java
// Metoda pomocnicza do pobierania konkretnej pozycji
public Pozycja getPozycja(int index) {
    if (index >= 0 && index < pozycje.size()) {
        return pozycje.get(index);
    }
    return null;
}
```

3. Modyfikacja klasy Ui (Demonstracja działania wzorców)

Zmiany w metodzie main:

Oryginalny kod:

```java
public static void main(String[] args) {
    Calendar teraz = Calendar.getInstance();

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

    // I przykładowa faktura
    Faktura f = new Faktura(teraz.getTime(), "Fido");
    f.dodajPozycje(t1, 3);
    f.dodajPozycje(t2, 5);

    wypiszFakture(f);

    // TEST ZEWN. rabatu
    LosowyRabat lr = new LosowyRabat();
    System.out.println(lr.losujRabat());
}
```

Zmodyfikowany kod:

```java
public static void main(String[] args) {
    Calendar teraz = Calendar.getInstance();

    // Tworzymy towary
    Towar t1 = new Towar(10, "buty");
    Towar t2 = new Towar(2, "skarpety");
    Towar t3 = new Towar(50, "kurtka");

    // Tworzymy strategie rabatów (wzorzec Strategia)
    ObliczCenePoRabacie rabatProcentowy = new RabatProcentowy(10); // 10% rabatu
    ObliczCenePoRabacie rabatKwotowy = new RabatKwotowy(5); // 5 zł rabatu

    // Tworzymy fakturę
    Faktura faktura = new Faktura(teraz.getTime(), "Fido");

    // Dodajemy pozycje - normalne i z rabatami
    faktura.dodajPozycje(t1, 3);
    faktura.dodajPozycjeZRabatem(t2, 5, rabatProcentowy);
    faktura.dodajPozycjeZRabatem(t3, 1, rabatKwotowy);

    System.out.println("Początkowa faktura:");
    wypiszFakture(faktura);

    // Demonstracja działania wzorca Obserwator - zmiana ilości automatycznie zaktualizuj
    System.out.println("\nPo zmianie ilości w pierwszej pozycji:");
    Pozycja pozycja = faktura.getPozycja(0);
    pozycja.setIlosc(5); // To automatycznie zaktualizuje sumę faktury

    wypiszFakture(faktura);

    // Demonstracja zmiany ceny
    System.out.println("\nPo zmianie ceny w drugiej pozycji:");
    Pozycja pozycja2 = faktura.getPozycja(1);
    pozycja2.setCena(3); // To automatycznie zaktualizuje sumę faktury

    wypiszFakture(faktura);

    // Demonstracja dodania rabatu do istniejącej pozycji
    System.out.println("\nPo dodaniu rabatu do trzeciej pozycji:");
    Pozycja pozycja3 = faktura.getPozycja(2);
    pozycja3.zastosujRabat(new RabatProcentowy(20)); // 20% rabatu

    wypiszFakture(faktura);
}
```

4. Korzyści z wprowadzonych zmian

4.1. Korzyści z wprowadzenia wzorca Strategia:

- **Elastyczność rabatów**: Możliwość łatwego dodawania nowych rodzajów rabatów bez modyfikacji istniejącego kodu.
- **Wymienność algorytmów**: Możliwość dynamicznej zmiany sposobu obliczania rabatu w czasie działania programu.
- **Enkapsulacja algorytmów**: Każdy typ rabatu jest zamknięty w osobnej klasie zgodnie z zasadą pojedynczej odpowiedzialności.

4.2. Korzyści z wprowadzenia wzorca Obserwator:

- **Automatyczne aktualizacje**: Po zmianie wartości pozycji, suma na fakturze jest automatycznie aktualizowana.
- **Luźne powiązania**: Klasa `Pozycja` nie musi wiedzieć nic o klasie `Faktura`, tylko powiadamia obserwatorów o zmianach.
- **Redukcja błędów**: Eliminacja problemów wynikających z zapomnienia ręcznego przeliczenia sumy po zmianach.

4.3. Ogólne korzyści:

- **Lepsza struktura kodu**: Kod jest bardziej modularny i zgodny z zasadami SOLID.
- **Łatwiejsza rozszerzalność**: Dodawanie nowych funkcjonalności jest prostsze dzięki luźnym powiązaniom między klasami.
- **Czytelność**: Jaśniejszy podział odpowiedzialności między klasami.