Dokumentacja wdrożonych wzorców projektowych

Poniżej opisano dwa wzorce projektowe zastosowane w kodzie:

1. **Strategy** – wydzielenie logiki obliczania rabatu
2. **Builder** – czytelne i bezpieczne tworzenie obiektów `Faktura`

Dokument zawiera:

- Krótki opis wzorca
- Fragmenty kodu przed i po zmianie
- Korzyści wynikające ze zmiany
- Krótka nota dotycząca użycia AI do wsparcia refaktoryzacji

---

1. Wzorzec Strategy

Opis wzorca

Wzorzec **Strategy** pozwala wydzielić rodzinę algorytmów (tutaj: wyliczanie rabatu) i sprawić, że można je wzajemnie wymieniać w czasie działania programu bez modyfikacji kodu klienta.

Kod przed zmianą

```
// Ui.java
LosowyRabat lr = new LosowyRabat();
System.out.println(lr.losujRabat());

// brak ujednoliconego interfejsu ani możliwości podmiany innej strategii
```

Kod po zmianie

```
// Interfejs strategii
package rabaty;
public interface Rabat {
    double oblicz(double suma);
}

// Implementacja domyślna
package rabaty;
public class BrakRabat implements Rabat {
    @Override
    public double oblicz(double suma) {
        return suma;
    }
}

// Losowy rabat 0–10%
package rabaty;
import java.util.Random;
```

```java
public class LosowyRabat implements Rabat {
    private Random rnd = new Random();
    @Override
    public double oblicz(double suma) {
        double procent = rnd.nextDouble() * 0.10;
        return suma * (1 - procent);
    }
}

// Faktura korzysta ze strategii
package dokumenty;
import rabaty.Rabat;
public class Faktura {
    private Rabat rabatStrategy;
    // konstruktor przyjmuje strategię
    Faktura(..., Rabat rabatStrategy) {
        this.rabatStrategy = rabatStrategy;
        ...
    }
    public double getSumaPoRabacie() {
        return rabatStrategy.oblicz(suma);
    }
}
```

Korzyści

- **Łatwa rozbudowa**: nowy sposób liczenia rabatu (np. progowy, sezonowy) wystarczy dodać jako kolejną klasę implementującą `Rabat`.
- **Brak zmiany kodu `Faktura`** przy dodawaniu kolejnych strategii.
- **Testowalność**: można wstrzyknąć w testach stub lub mock strategii.

---

2. Wzorzec Builder

Opis wzorca

Wzorzec **Builder** ułatwia tworzenie złożonych obiektów krok po kroku, pozwala na czytelny kod i zapobiega tworzeniu obiektów w niepełnym stanie.

Kod przed zmianą

```java
// Ui.java
Faktura f = new Faktura(teraz.getTime(), "Fido", new LosowyRabat());
```

Kod po zmianie

```java
// Faktor budujący
package dokumenty;
import rabaty.Rabat;
import rabaty.BrakRabat;
import java.util.Date;
public class FakturaBuilder {
    private Date dataSprzedazy;
```

```
    private String kontrahent;
    private Rabat rabat = new BrakRabat();

    public FakturaBuilder forKontrahent(String kontrahent) {
        this.kontrahent = kontrahent;
        return this;
    }

    public FakturaBuilder atDate(Date data) {
        this.dataSprzedazy = data;
        return this;
    }

    public FakturaBuilder withRabat(Rabat rabat) {
        this.rabat = rabat;
        return this;
    }

    public Faktura build() {
        return new Faktura(dataSprzedazy, kontrahent, rabat);
    }
}

// Użycie
Faktura f = new FakturaBuilder()
    .atDate(teraz.getTime())
    .forKontrahent("Fido")
    .withRabat(new LosowyRabat())
    .build();
```

Korzyści

- **Czytelność**: kolejne wywołania metod opisują krok po kroku składanie obiektu.
- **Bezpieczeństwo**: wymuszenie niezbędnych parametrów przed wywołaniem `build()`.
- **Rozszerzalność**: łatwo dodać kolejne opcje (np. termin płatności, komentarze) bez zmiany sygnatury konstruktora.

Dokumentacja użycia AI

Refaktoryzacja kodu i proponowane zmiany zostały przygotowane przy wsparciu narzędzia ChatGPT (model o4-mini). AI pomogło w:

- Zidentyfikowaniu odpowiednich wzorców projektowych,
- Opracowaniu przykładów kodu przed i po zmianie,
- Wyjaśnieniu korzyści w formie czytelnej dokumentacji.

Dzięki AI praca nad lepszą architekturą kodu została przyspieszona i ustrukturyzowana zgodnie z dobrymi praktykami projektowymi.