



# Projektowanie obiektowe

## Artificial Intelligence - Design Patterns

### Autorzy:

- Szymon Migas
- Emil Siatka

## Rozpoznanie w Modelach AI

Po szybkim zestawieniu dostępnych na rynku modeli sztucznej inteligencji zdecydowaliśmy się na skorzystanie z modelu Claude 3.7 Sonnet dostarczanego przez firmę Anthropic .

Decyzja ta była motywowana przez dwie ważne dla nas kwestie:

- Wysoka pozycja w rankingach modeli AI oraz własne doświadczenia
- Dostępność tego modelu za darmo w dostępnym dla studentów narzędziu Copilot dostarczanym przez firmę Microsoft

## Propozycje przedstawione przez model

Zapytania do modelu napisane zostały przez nas w języku angielskim dla zapewnienia jak największej dokładności odpowiedzi generowanych przez ten model.

### Zapytanie 1

---

Take a look at the project structure. Please propose some design patterns I can use to refactor this code. Please make a short note which one you suggest, and don't provide any code yet

### Rezultat 1

---

Based on the project structure, here are some design patterns I'd recommend for refactoring:

1. **Strategy Pattern:** Ideal for implementing different pricing strategies using the existing `ObliczCenePoRabacie` interface. This would allow you to define various discount algorithms and switch between them dynamically.
2. **Decorator Pattern:** Could be used to add optional discounts or surcharges to products without modifying their original structure. Each decorator would wrap a `Towar` object and modify its price calculation.
3. **Observer Pattern:** Would be useful if you want changes in price or inventory to automatically update related components (like recalculating invoice totals when a product price changes).
4. **Factory Method Pattern:** Could help with creating different types of products or documents with specialized behavior while maintaining a common interface.
5. **Adapter Pattern:** Could integrate the external `LosowyRabat` with your internal discount calculation system in a more cohesive way.

I'd most strongly recommend the **Strategy Pattern** as it aligns perfectly with your existing interface and would provide the most immediate benefit for managing different pricing calculations in a clean, extensii

---

# Wybór i implementacja wybranych wzorców

## Wzorzec Strategii

Zdecydowaliśmy się na wybór tego wzorca, ponieważ model z którego korzystaliśmy zawzięcie uważał go za najlepszy z możliwych.

## Zapytanie 2.

---

Generate new files implementing strategy pattern.

## Rezultat 2.

---

I'll help you implement the Strategy Pattern based on your project structure. This pattern is perfect for your discount calculation system.

First, let's create concrete implementations for the `ObliczCenePoRabacie` interface:

```

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);
    }
}

```

```

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);
    }
}

```

Now, let's create an adapter for the external `LosowyRabat` :

```

package rabaty;

import losowy.LosowyRabat;

public class AdapterLosowegoRabatu implements ObliczCenePoRabacie {
    private final LosowyRabat losowyRabat;

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

    @Override
    public double obliczCenePoRabacie(double cena) {
        // Assuming LosowyRabat has a method that returns discount percentage
        double rabatProcent = losowyRabat.losujRabat();
        return cena * (1 - rabatProcent/100);
    }
}

```

Now, modify the `Towar` class to use the Strategy Pattern (assuming it exists):

```

package magazyn;

import rabaty.ObliczCenePoRabacie;

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

    // Constructor and other existing fields

    public void ustawStrategieRabatu(ObliczCenePoRabacie strategiaRabatu) {
        this.strategiaRabatu = strategiaRabatu;
    }

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

    // Other existing methods
}

```

These implementations allow you to dynamically set different discount strategies for products and easily extend the system with new types of discounts in the future.

## Komentarz do odpowiedzi 2.

Ponieważ model źle zimportował klasę `LosowyRabat` oraz zostawił komentarze w klasie `Towar` poprosiliśmy o poprawienie kodu wygenerowanego pliku.

## Zapytanie 3.

```

change occurrences of import statement:
import losowy.LosowyRabat;
into
import rabatlosowy.LosowyRabat;

```

## Rezultat 3.

Based on your request, I need to update the `import` statement in the generated code. I notice that in my previously generated code I used `import losowy.LosowyRabat;`, but the correct import statement should be `import rabatlosowy.LosowyRabat;`.

Here's the corrected `AdapterLosowegoRabatu` class with the proper import:

```
package rabat;

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) {
        // Assuming LosowyRabat has a method that returns discount percentage
        double rabatProcent = losowyRabat.losujRabat();
        return cena * (1 - rabatProcent/100);
    }
}
```

This correction ensures the adapter properly imports the `LosowyRabat` class from the correct package.

## Zapytanie 4.

---

Please fulfill existing fields, methods and constructors in place of your comments in Towar class

## Rezultat 4.

---

Looking at the project, I see you've implemented the Strategy Pattern but need to update the `Towar` class to integrate it. I'll modify the `Towar` class to incorporate the discount strategy functionality while preserving the existing fields and methods.

Here's the updated `Towar` class:

```

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);
    }
}

```

This implementation:

1. Adds the `strategiaRabatu` field to store the discount strategy
2. Provides a method to set the strategy
3. Adds `getCenaPoRabacie()` to calculate the price after applying the discount
4. Maintains all existing fields and methods

# Wzorzec Dekoratora

## Zapytanie 5.

---

Generate new files implementing decorator pattern. Dont leave comments like previously (Existing code).

## Rezultat 5.

---

Based on your request, I'll implement the Decorator pattern for your project to enhance the discount system. This pattern will allow you to stack multiple discounts or modifiers on products.

Here are the implementations:

```
package rabaty.decoratory;

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();
    }
}
```

```

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 + "%)";
    }
}

```

```

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)";
    }
}

```



```

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)";
    }
}

```

```

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 + "%";
    }
}

```

## Zapytanie 6.

---

Make some changes to the project to make use of created decorators

## Rezultat 6.

---

Looking at your project, I'll add some code to demonstrate the usage of the decorator pattern alongside the already implemented strategy pattern. Here's how you can modify the `Ui` class to showcase both patterns:

```

package main;

import java.util.Iterator;
import java.util.Calendar;
import magazyn.Towar;

import dokumenty.Faktura;
import dokumenty.Pozycja;

//ZEWNETRZNY RABAT
import rabatlosowy.LosowyRabat;

// Import strategy classes
import rabaty.RabatProcentowy;
import rabaty.RabatKwotowy;
import rabaty.AdapterLosowegoRabatu;

// Import decorator classes
import rabaty.dekoratory.RabatProcentowyDekorator;
import rabaty.dekoratory.RabatKwotowyDekorator;
import rabaty.dekoratory.DarmoweDostarczeniaDekorator;
import rabaty.dekoratory.PodatekDekorator;

public class Ui {

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

        // 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");

        // Przykładowa faktura z podstawowymi rabatami
        Faktura f = new Faktura(teraz.getTime(), "Fido");
    }
}

```

```

f.dodajPozycje(t1, 3);
f.dodajPozycje(t2, 5);

wypiszFakture(f);

// 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);
}

private static void wypiszFakture(Faktura faktura) {
    System.out.println("=====");
    System.out.println("FA z dnia: " + faktura.getDataSprzedazy().toString());
    System.out.println("Wystawiona dla: " + faktura.getKontrahent());
    System.out.println("Na kwotę: " + faktura.getSuma());
    Iterator<Pozycja> iteratorPozycji = faktura.getIteratorPozycji();
    while(iteratorPozycji.hasNext()) {
        Pozycja pozycja = iteratorPozycji.next();
        System.out.println("Towar: " + pozycja.getNazwa() + " Ilość: " + pozycja.getIlość() + " Wartość: " + pozycja.getWartosc());
    }
    System.out.println("=====");
}
}

```

This modified `Ui` class demonstrates:

1. **Strategy Pattern Usage:**

- Creating different discount strategies
- Applying them to products
- Changing strategies dynamically
- Using the adapter for the external random discount

2. **Decorator Pattern Usage:**

- Creating a base product
- Adding a percentage discount decorator
- Adding a tax decorator on top of that
- Adding a free delivery decorator on top of both
- Showing how different decorators can be combined independently

The output will clearly show how both patterns affect product pricing and presentation, with:

- The original products with strategy-based discounts
- The decorated products with layered modifications
- Both types of products in invoices

This demonstrates the flexibility and composability of both design patterns in your system.

## Opisy zastosowanych wzorców

Opisy te są zawarte w plikach

- `decorator-pattern.pdf`
- `strategy-pattern.pdf`