

# Lista 3 | POO

---

## Zadanie 1

---

1. (2p) (GRASP) Dwie wybrane zasady projektowe GRASP zilustrować przykładowym kompilującym się kodem. Umieć uzasadnić, że zaproponowany kod rzeczywiście ilustruje wybrane reguły.

**Creator:** Klasa `Order` odpowiada za tworzenie obiektów `OrderItem`, bo *posiada* dane do ich utworzenia.

**Information Expert:** Klasa `Order` posiada wszystkie informacje o swoich elementach i dlatego oblicza łączną wartość zamówienia.

```
// Klasa reprezentująca pojedynczy element zamówienia
public class OrderItem {
    private String name;
    private int quantity;
    private double unitPrice;

    public OrderItem(String name, int quantity, double unitPrice) {
        this.name = name;
        this.quantity = quantity;
        this.unitPrice = unitPrice;
    }

    public String getName() { return name; }
    public int getQuantity() { return quantity; }
    public double getUnitPrice() { return unitPrice; }
}

// Klasa reprezentująca zamówienie
import java.util.ArrayList;
import java.util.List;

public class Order {
    private List<OrderItem> items;

    public Order() {
        items = new ArrayList<>();
    }

    // Zasada Creator - Order tworzy obiekt OrderItem,
    // ponieważ "zawiera" dane do jego utworzenia.
    public void addItem(String name, int quantity, double unitPrice) {
        items.add(new OrderItem(name, quantity, unitPrice));
    }

    // Zasada Information Expert - Order zna swoje elementy
    // i potrafi obliczyć łączną wartość.
    public double calculateTotal() {
        double total = 0;
        for (OrderItem item : items) {
            total += item.getUnitPrice() * item.getQuantity();
        }
        return total;
    }
}
}
```

```
public class Task1Demo {
    public static void main(String[] args) {
        Order order = new Order();
        order.addItem("Produkt A", 2, 10.0);
        order.addItem("Produkt B", 1, 20.0);
        double total = order.calculateTotal();
        System.out.println("Łączna wartość zamówienia: " + total);
    }
}
```

STDIN

Input for the program ( Optional)

Output:

Łączna wartość zamówienia: 40.0

## Zadanie 2

---

2. (1p) (Single Responsibility Principle) Dokonać analizy projektu obiektowego pod kątem zgodności z zasadą SRP. Zaproponować zmiany. Zaimplementować działający kod dla przykładu przed i po zmianach.

```
public class ReportPrinter {  
    public string GetData();  
    public void FormatDocument();  
    public void PrintReport();  
}
```

Ile klas docelowo powstanie z takiej jednej klasy? Dlaczego akurat tyle? Czy refaktoryzacja klasy naruszającej SRP oznacza automatycznie, że **każda** metoda powinna trafić do osobnej klasy?

Początkowo jedna klasa `ReportPrinter` łączy w sobie pobieranie danych, formatowanie oraz drukowanie raportu. Podzielimy to na trzy oddzielne klasy.

### Przed zmianami

```
// Klasa łącząca pobieranie danych, formatowanie i drukowanie raportu  
public class ReportPrinter {  
    public String getData() {  
        return "Dane raportu";  
    }  
  
    public void formatDocument() {  
        System.out.println("Dokument sformatowany");  
    }  
  
    public void printReport() {  
        System.out.println("Raport wydrukowany");  
    }  
}
```

### Po zmianach

```
public class ReportDataProvider {  
    public String getData() {  
        return "Dane raportu";  
    }  
}
```

```
public class ReportFormatter {  
    public String formatReport(String data) {  
        return "Sformatowany raport: " + data;  
    }  
}
```

```
public class ReportPrinter {  
    public void printReport(String formattedReport) {  
        System.out.println(formattedReport);  
    }  
}
```

<pre>1 public class Task2Demo { 2     public static void main(String[] args) { 3         ReportDataProvider dataProvider = new ReportDataProvider(); 4         ReportFormatter formatter = new ReportFormatter(); 5         ReportPrinter printer = new ReportPrinter(); 6 7         String data = dataProvider.getData(); 8         String formatted = formatter.formatReport(data); 9         printer.printReport(formatted); 10    } }</pre>	STDIN
	Input for the program ( Optional )
	Output: Sformatowany raport: Dane raportu

## Odpowiedzi na pytania:

- Powstają trzy klasy.
- Refaktoryzacja klasy naruszającej SRP nie oznacza, że każda metoda musi trafić do osobnej klasy. Kluczowe jest grupowanie tak, że grupy stanowią jedną spójną odpowiedzialność.

## Zadanie 3

---

3. (2p) (**Open-Closed Principle**) Dokonać analizy projektu obiektowego pod kątem zgodności klasy `CashRegister` z zasadą OCP. Klasa ta posiada dwie odpowiedzialności - obliczania zobowiązań podatkowych oraz przygotowania wydruku.

Wstępna analiza wykazuje, że klasa ta nie spełnia postulatu OCP (dlaczego?).

Zaproponować takie zmiany, które uczynią ją niezmienną a równocześnie rozszerzalną jeśli chodzi o możliwość implementowania różnych taryf podatkowych oraz drukowania paragonów z uwzględnieniem różnego porządkowania towarów (alfabetycznie, według kategorii itp.)

Zaimplementować działający kod dla przykładu przed i po zmianach demonstrując kilka różnych rozszerzeń.

```
public class TaxCalculator {
    public Decimal CalculateTax( Decimal Price ) { return Price * 0.22 }
}

public class Item {
    public Decimal Price { get { ... } }
    public string Name { get { ... } }
}

public class CashRegister {
    public TaxCalculator taxCalc = new TaxCalculator();

    public Decimal CalculatePrice( Item[] Items ) {
        Decimal _price = 0;
        foreach ( Item item in Items ) {
            _price += item.Price + taxCalc.CalculateTax( item.Price );
        }
        return _price;
    }

    public string PrintBill( Item[] Items ) {
        foreach ( var item in Items )
            Console.WriteLine( "towar {0} : cena {1} + podatek {2}",
                                item.Name, item.Price, taxCalc.CalculateTax( item.Price ) );
    }
}
```

Początkowa implementacja klasy `CashRegister` realizuje dwie odpowiedzialności (obliczanie podatku i drukowanie paragonu) bez możliwości łatwej rozbudowy.

Refaktoryzacja będzie polegać na wprowadzeniu interfejsów, które umożliwią łatwe rozszerzenie funkcjonalności.

## Przed refaktoryzacją

```
public class TaxCalculator {
    public decimal CalculateTax(decimal price) {
        return price * 0.22m;
    }
}

public class Item {
    public decimal Price { get; set; }
    public string Name { get; set; }
}

public class CashRegister {
    public TaxCalculator taxCalc = new TaxCalculator();

    public decimal CalculatePrice(Item[] items) {
        decimal total = 0;
        foreach (var item in items) {
            total += item.Price + taxCalc.CalculateTax(item.Price);
        }
        return total;
    }

    public string PrintBill(Item[] items) {
        foreach (var item in items)
            Console.WriteLine("towar {0} : cena {1} + podatek {2}",
                               item.Name, item.Price, taxCalc.CalculateTax(item.Price));
        return "Paragon wydrukowany";
    }
}
```

## Po refaktoryzacji

```
// Interfejs dla kalkulatora podatku
public interface ITaxCalculator {
    decimal CalculateTax(decimal price);
}

// Domyślna implementacja
public class DefaultTaxCalculator : ITaxCalculator {
    public decimal CalculateTax(decimal price) => price * 0.22m;
}

// Przykładowa alternatywna implementacja taryfy specjalnej
public class SpecialTaxCalculator : ITaxCalculator {
    public decimal CalculateTax(decimal price) => price * 0.15m;
}

// Interfejs dla drukarki paragonu
public interface IBillPrinter {
    string PrintBill(Item[] items, ITaxCalculator taxCalculator);
}

// Implementacja drukarki sortującej alfabetycznie
public class BillPrinterAlphabetical : IBillPrinter {
    public string PrintBill(Item[] items, ITaxCalculator taxCalculator) {
        var sortedItems = items.OrderBy(i => i.Name).ToArray();
        StringBuilder sb = new StringBuilder();
        foreach (var item in sortedItems) {
            sb.AppendLine($"towar {item.Name} : cena {item.Price} + podatek {taxCalculator.CalculateTax(item.Price)}");
        }
        return sb.ToString();
    }
}

// Implementacja drukarki sortującej według ceny rosnąco
public class BillPrinterByPrice : IBillPrinter {
    public string PrintBill(Item[] items, ITaxCalculator taxCalculator) {
        var sortedItems = items.OrderBy(i => i.Price).ToArray();
        StringBuilder sb = new StringBuilder();
        foreach (var item in sortedItems) {
            sb.AppendLine($"towar {item.Name} : cena {item.Price} + podatek {taxCalculator.CalculateTax(item.Price)}");
        }
        return sb.ToString();
    }
}
```



```

public class CashRegister {
    private readonly ITaxCalculator _taxCalculator;
    private readonly IBillPrinter _billPrinter;

    // Wstrzykiwanie zależności
    public CashRegister(ITaxCalculator taxCalculator, IBillPrinter billPrinter) {
        _taxCalculator = taxCalculator;
        _billPrinter = billPrinter;
    }

    public decimal CalculatePrice(Item[] items) {
        decimal total = 0;
        foreach (var item in items) {
            total += item.Price + _taxCalculator.CalculateTax(item.Price);
        }
        return total;
    }

    public string PrintBill(Item[] items) {
        return _billPrinter.PrintBill(items, _taxCalculator);
    }
}

```

```

1 using System;
2
3 public class Task3Demo {
4     public static void Main() {
5
6         Item[] items = new Item[] {
7             new Item { Name = "Produkt A", Price = 10m },
8             new Item { Name = "Produkt B", Price = 20m },
9             new Item { Name = "Produkt C", Price = 15m }
10        };
11
12        // Używamy domyślnego kalkulatora podatku i drukarki sortującej alfabetycznie
13        ITaxCalculator taxCalculator = new DefaultTaxCalculator();
14        IBillPrinter billPrinter = new BillPrinterAlphabetical();
15        CashRegister register = new CashRegister(taxCalculator, billPrinter);
16
17        decimal totalPrice = register.CalculatePrice(items);
18        Console.WriteLine("Łączna cena: " + totalPrice);
19
20        string bill = register.PrintBill(items);
21        Console.WriteLine("Paragon:\n" + bill);
22    }
23 }

```

STDIN

Input for the program ( Optional )

Output:

Łączna cena: 54.90

Paragon:

towar Produkt A : cena 10 + podatek 2.20

towar Produkt B : cena 20 + podatek 4.40

towar Produkt C : cena 15 + podatek 3.30



## Zadanie 4

---

4. (1p) (Liskov Substitution Principle) Zaprojektowano klasy `Rectangle` i `Square` i w "naturalny" sposób relację dziedziczenia między nimi (*każdy kwadrat jest prostokątem*).

```
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
}

public class Square : Rectangle
{
    public override int Width
    {
        get { return base.Width; }
        set { base.Width = base.Height = value;}
    }

    public override int Height
    {
        get { return base.Height; }
        set { base.Width = base.Height = value; }
    }
}
```

Co można powiedzieć o spełnianiu przez taką hierarchię zasady LSP w kontekście poniższego kodu klienckiego?

```
public class AreaCalculator
{
    public int CalculateArea( Rectangle rect )
    {
        return rect.Width * rect.Height;
    }
}

int w = 4, h = 5;

Rectangle rect = new Square() { Width = w, Height = h };

AreaCalculator calc = AreaCalculator();

Console.WriteLine( "prostokąt o wymiarach {0} na {1} ma pole {2}",
    w, h, calc.CalculateArea( rect ) );
```

Jak należałoby zmodyfikować przedstawioną hierarchię klas, żeby zachować zgodność z LSP w kontekście takich wymagań? Jak potraktować klasy `Rectangle` i `Square`? Odpowiedź zilustrować działającym kodem.

Oryginalna hierarchia (gdzie `Square` dziedziczy po `Rectangle`) może prowadzić do nieoczekiwanego zachowania.

Lepszym rozwiązaniem jest stworzenie wspólnego interfejsu `Shape`, który definiuje metodę obliczania pola, a klasy `Rectangle` i `Square` implementują ją zgodnie ze swoimi właściwościami.

```
// Interfejs reprezentujący kształt
public interface IShape {
    int GetArea();
}

public class Rectangle : IShape {
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea() {
        return Width * Height;
    }
}

public class Square : IShape {
    public int Side { get; set; }

    public int GetArea() {
        return Side * Side;
    }
}

public class AreaCalculator {
    public int CalculateArea(IShape shape) {
        return shape.GetArea();
    }
}
```

```
2 using System;
3
4 public class Task4Demo {
5     public static void Main() {
6         IShape rectangle = new Rectangle(4, 5);
7         IShape square = new Square(5);
8
9         AreaCalculator calc = new AreaCalculator();
10        Console.WriteLine("Pole prostokąta: " + calc.CalculateArea(rectangle));
11        Console.WriteLine("Pole kwadratu: " + calc.CalculateArea(square));
12    }
13 }
14
```

Input for the program

Output:

Pole prostokąta: 20  
Pole kwadratu: 25

## Zadanie 5

5. (1p) (**Interface Segregation Principle**) Znaleźć w bibliotece standardowej dowolnego języka programowania przykład interfejsu, klasy abstrakcyjnej lub klasy przewidzianej do dziedziczenia, które łamią zasadę ISP tzn. istnieją zastosowania, w których korzysta się tylko z części tego interfejsu.

W razie trudności ze znalezieniem, proszę przyjrzeć się interfejsom z biblioteki **Microsoft.AspNetCore.Identity**.

Interfejsy zdokumentowano tu:

<https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity?view=aspnetcore-9.0>

Zaproponować refaktoryzację interfejsu wolną od zaobserwowanej przypadłości.

Przykładem z biblioteki .NET może być interfejs **IUserStore<TUser>**.

Interfejs ten definiuje szereg metod, takich jak:

CreateAsync(TUser, CancellationToken)	Tworzy element określony <code>user</code> w magazynie użytkownika.
DeleteAsync(TUser, CancellationToken)	Usuwa określony <code>user</code> element ze sklepu użytkownika.
FindByIdAsync(String, CancellationToken)	Wyszukuje i zwraca użytkownika, jeśli istnieje, który ma określony <code>userId</code> element .
FindByNameAsync(String, CancellationToken)	Znajduje i zwraca użytkownika, jeśli istnieje, który ma określoną znormalizowaną nazwę użytkownika.
GetNormalizedUserNameAsync(TUser, CancellationToken)	Pobiera znormalizowaną nazwę użytkownika dla określonego <code>user</code> elementu .
GetUserIdAsync(TUser, CancellationToken)	Pobiera identyfikator użytkownika dla określonego <code>user</code> elementu .
GetUserNameAsync(TUser, CancellationToken)	Pobiera nazwę użytkownika dla określonego <code>user</code> elementu .
SetNormalizedUserNameAsync(TUser, String, CancellationToken)	Ustawia podaną znormalizowaną nazwę dla określonego <code>user</code> elementu .
SetUserNameAsync(TUser, String, CancellationToken)	Ustawia daną <code>userName</code> wartość dla określonego <code>user</code> elementu .
UpdateAsync(TUser, CancellationToken)	Aktualizację określony <code>user</code> w magazynie użytkowników.


Nie każda implementacja UserStore musi korzystać ze wszystkich metod – niektóre systemy mogą np. potrzebować tylko wyszukiwania, a nie modyfikacji.

Narusza to ISP, bo zmuszamy klasę implementującą interfejs do implementowania nieużywanych metod.

## Proponowana refaktoryzacja

Podzielić interfejs na mniejsze, bardziej wyspecjalizowane interfejsy, np.:

```
public interface IUserCreator<TUser> {  
    Task CreateAsync(TUser user, CancellationToken cancellationToken);  
}  
  
public interface IUserUpdater<TUser> {  
    Task UpdateAsync(TUser user, CancellationToken cancellationToken);  
}  
  
public interface IUserDeleter<TUser> {  
    Task DeleteAsync(TUser user, CancellationToken cancellationToken);  
}  
  
public interface IUserFinder<TUser> {  
    Task<TUser> FindByIdAsync(string userId, CancellationToken cancellationToken);  
    Task<TUser> FindByNameAsync(string normalizedUserName, CancellationToken cancellationToken);  
}
```



Implementacje mogą wybrać tylko te interfejsy, które są im potrzebne, dzięki czemu klient nie jest zmuszony do zależności od metod, których nie używa.

## Zadanie 6

---

### 6. (1p) (SRP vs ISP) Wytłumaczyć różnicę między SRP a ISP.

#### SRP (Single Responsibility Principle)

- Dotyczy projektowania klas
- Każda klasa powinna mieć tylko jedną odpowiedzialność, czyli tylko jeden powód do zmiany.
- **Przykład:** Klasa odpowiedzialna za pobieranie danych nie powinna zajmować się ich formatowaniem ani drukowaniem.

#### ISP (Interface Segregation Principle)

- Dotyczy projektowania interfejsów
- Klient nie powinien być zmuszany do bycia zależnym od metod, których nie używa.
- **Przykład:** Zamiast implementować duży interfejs zawierający wiele metod, lepiej podzielić go na mniejsze interfejsy, z których klient wybiera tylko te, które są mu

potrzebne.

## Zadanie 7

---

7. (1p) (**Dependency Inversion Principle**) Przykład z zadania o SRP zrefaktoryzować - wprowadzić klasę `ReportComposer` która obsługuje *wstrzykiwanie* zależności do obiektów usługowych.

W tym przykładzie wprowadzamy klasę `ReportComposer`, która łączy usługi odpowiedzialne za pobieranie danych, formatowanie oraz drukowanie raportu. Zależności są wstrzykiwane przez interfejsy.

### Definicja interfejsów dla usług

```
public interface IReportDataProvider {
    String getData();
}

public interface IReportFormatter {
    String formatReport(String data);
}

public interface IReportPrinter {
    void printReport(String formattedReport);
}
```

### Implementacje

```
public class ReportDataProviderImpl implements IReportDataProvider {
    @Override
    public String getData() {
        return "Dane raportu z bazy";
    }
}

public class ReportFormatterImpl implements IReportFormatter {
    @Override
    public String formatReport(String data) {
        return "*** Sformatowany raport ***\n" + data;
    }
}
```

```

public class ReportPrinterImpl implements IReportPrinter {
    @Override
    public void printReport(String formattedReport) {
        System.out.println(formattedReport);
    }
}

```

## Klasa ReportComposer

```

public class ReportComposer {
    private IReportDataProvider dataProvider;
    private IReportFormatter formatter;
    private IReportPrinter printer;

    public ReportComposer(IReportDataProvider dataProvider, IReportFormatter formatter,
        this.dataProvider = dataProvider;
        this.formatter = formatter;
        this.printer = printer;
    }

    public void composeAndPrintReport() {
        String data = dataProvider.getData();
        String formatted = formatter.formatReport(data);
        printer.printReport(formatted);
    }
}

```

<pre> 1 public class Main { 2     public static void main(String[] args) { 3         IReportDataProvider dataProvider = new ReportDataProviderImpl(); 4         IReportFormatter formatter = new ReportFormatterImpl(); 5         IReportPrinter printer = new ReportPrinterImpl(); 6 7         ReportComposer composer = new ReportComposer(dataProvider, formatter, printer); 8         composer.composeAndPrintReport(); 9     } 10 } 11 </pre>		<p>STDIN</p> <p>Input for the program (Optional)</p>
		<p>Output:</p> <p>*** Sformatowany raport ***</p> <p>Dane raportu z bazy</p>