

# Wzorce projektowe

## *Wzorce Kreatywne*

Konwersatorium 2

dr inż. Wojciech Skibiński



# WZORCE PROJEKTOWE (DESIGN PATTERNS)



## Kreacyjne (Creational)


- Budowniczy (Builder)
- Fabryka Abstrakcyjna (Abstract Factory)
- Metoda Wytwórcza (Factory Method)
- Prototyp (Prototype)
- Singleton (Singleton)

## Strukturalne (Structural)

- Adapter (Adapter)
- Dekorator (Decorator)
- Fasada (Façade)
- Kompozyt (Composite)
- Most (Bridge)
- Pełnomocnik (Proxy)
- Pylek (Flyweight)

## Czynnościowe (Behavioral)

- Interpreter (Interpreter)
- Iterator (Iterator)
- Łańcuch Zobowiązań (Chain of Responsibility)
- Mediator (Mediator)
- Metoda Szablonowa (Template Method)
- Obserwator (Observer)
- Odwiedzający (Visitor)
- Pamiętka (Memento)
- Poleceniec (Command)
- Stan (State)
- Strategia (Strategy)



# Wzorce projektowe są odpowiedzią na Open / Closed Principle\*

\***Open / Closed Principle** - jedna z zasad SOLID mówiąca, że elementy systemu takie, jak klasy, moduły, funkcje itd. powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje. Oznacza to, iż można zmienić zachowanie takiego elementu bez zmiany jego kodu. Jest to szczególnie ważne w środowisku produkcyjnym, gdzie zmiany kodu źródłowego mogą być niewskazane i powodować ryzyko wprowadzenia błędu. Program, który trzyma się tej zasady, nie wymaga zmian w kodzie, więc nie jest narażony na powyższe ryzyko.



## Wzorce Kreacyjne

Wzorce opisujące proces tworzenia nowych obiektów. Ich zadaniem jest tworzenie, inicjalizacja oraz konfiguracja obiektów, klas oraz innych typów danych. Należą do nich wzorce: Budowniczy, Fabryka, Prototyp, Singleton.

# Budowniczy (Builder)



Kreacyjny wzorzec projektowania, którego zadaniem jest stworzenie obiektu z innych mniejszych obiektów. Celem jest rozdzielenie sposobu tworzenia obiektów od ich reprezentacji. Budowanie obiektu oparte jest na jednym procesie konstrukcyjnym i podzielone jest na mniejsze etapy.

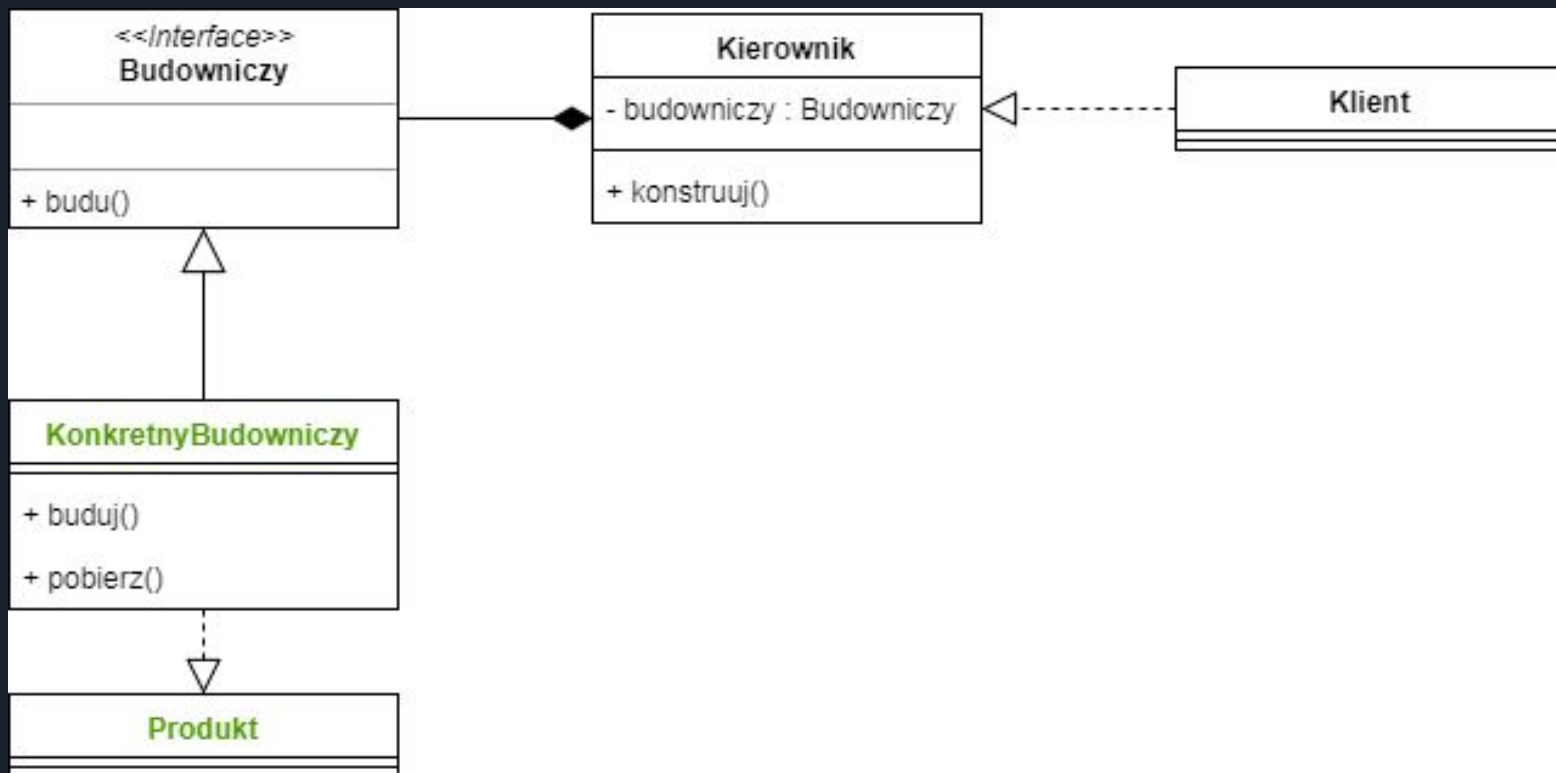
Zaletą wykorzystania tego wzorca jest możliwość łatwego sterowania, w jaki sposób przebiega proces tworzenia obiektów oraz większa skalowalność kodu. Ponadto izolujemy ten proces, który często może być skomplikowany.




# Motywacja

- Niektóre obiekty są proste i mogą być stworzone przy użyciu jednego wywołania konstruktora
- Inne obiekty wymagają ustawienia wielu pól i (być może) wymagają wielu konstruktorów
- Klasa, która ma 10 konstruktorów, a nawet konstruktor z 10 argumentami NIE JEST DOBRA
- Zamiast tego twórz obiekt w kawałkach
- Budowniczy dostarcza API do konstrukcji obiektu krok po kroku

# Budowniczy - UML





```
static void Main(string[] args)
{
    var hello = "hello";
    var sb = new StringBuilder();
    sb.Append("<p>");
    sb.Append(hello);
    sb.Append("</p>");
    WriteLine(sb);

    var words = new[] { "hello", "world" };
    sb.Clear();
    sb.Append("<ul>");
    foreach (var word in words)
    {
        sb.AppendFormat("<li>{0}</li>", word);
    }
    sb.Append("</ul>");
    WriteLine(sb);
}
```



```

public class HTMLElement
{
    public string Name, Text;
    public List<HTMLElement> Elements = new List<HTMLElement>();
    private const int indentSize = 2;

    Odwołania: 2
    public HTMLElement() { }

    1 odwołanie
    public HTMLElement(string name, string text)
    {
        Name = name ?? throw new ArgumentNullException(paramName: nameof(name));
        Text = text ?? throw new ArgumentNullException(paramName: nameof(text));
    }

    Odwołania: 2
    private string ToStringImpl(int indent)
    {
        var sb = new StringBuilder();
        var i = new string(' ', indentSize * indent);
        sb.AppendLine($"{i}<{Name}>");

        if (!string.IsNullOrEmpty(Text))
        {
            sb.Append(new string(' ', indentSize * (indent + 1)));
            sb.AppendLine(Text);
        }

        foreach (var e in Elements)
        {
            sb.Append(e.ToStringImpl(indent + 1));
        }
        sb.AppendLine($"{i}</{Name}>");
        return sb.ToString();
    }

    Odwołania: 3
    public override string ToString()
    {
        return ToStringImpl(0);
    }
}

```

```

public class HtmlBuilder
{
    private readonly string rootName;
    HTMLElement root = new HTMLElement();

    1 odwołanie
    public HtmlBuilder(string rootName)
    {
        this.rootName = rootName;
        root.Name = rootName;
    }

    Odwołania: 2
    public void AddChild(string childName, string childText)
    {
        var e = new HTMLElement(childName, childText);
        root.Elements.Add(e);
    }

    Odwołania: 3
    public override string ToString()
    {
        return root.ToString();
    }

    Odwołania: 0
    public void Clear()
    {
        root = new HTMLElement {Name = rootName};
    }
}

```

# Fluent interfaces

```
public void AddChild(string childName, string childText)
{
    var e = new HtmlElement(childName, childText);
    root.Elements.Add(e);
}
```



```
static void Main(string[] args)
{
    var builder = new HtmlBuilder("ul");
    builder.AddChild("li", "hello");
    builder.AddChild("li", "world");
    Console.WriteLine(builder.ToString());
}
```

```
public HtmlBuilder AddChild(string childName, string childText)
{
    var e = new HtmlElement(childName, childText);
    root.Elements.Add(e);
    return this;
}
```

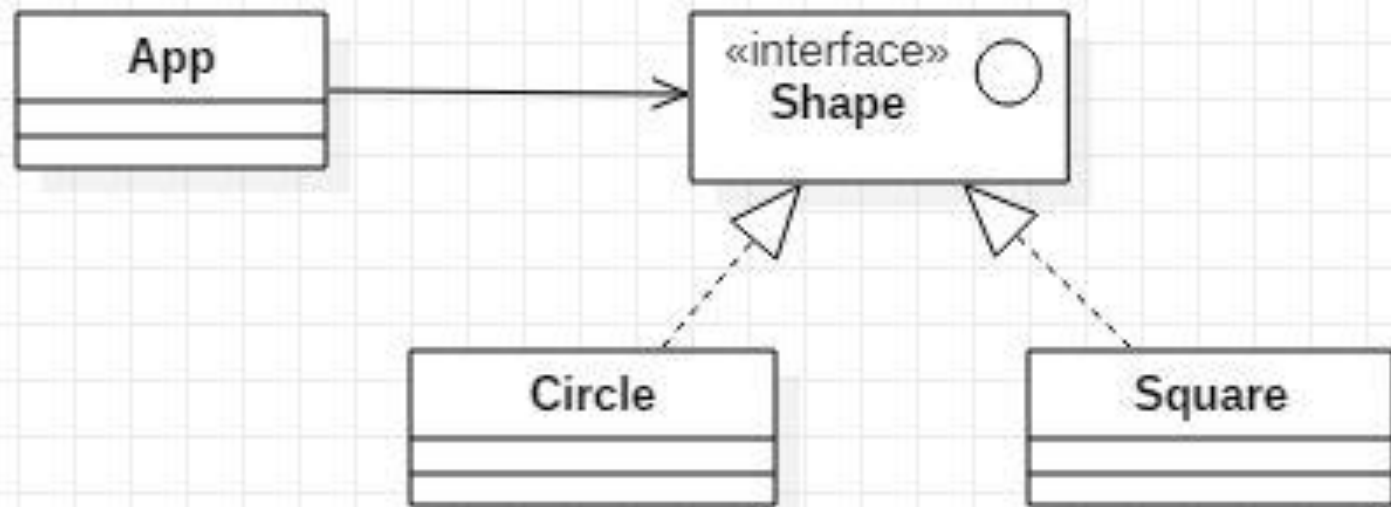


```
static void Main(string[] args)
{
    var builder = new HtmlBuilder("ul");
    builder.AddChild("li", "hello").AddChild("li", "world");
    Console.WriteLine(builder.ToString());
}
```

# Fabryki (Factories)

- Prosta fabryka
- Metoda fabryczna
- Fabryka abstrakcyjna







# Prosta fabryka (Simple factory)

Metoda zwracająca obiekt.

Często stosuje się prostą fabrykę, gdy chce się ukryć konstruktor, lub gdy tworzony obiekt ma wiele pól, które są inicjalizowane domyślnymi wartościami.

Uważany przez niektórych za antywzorzec!



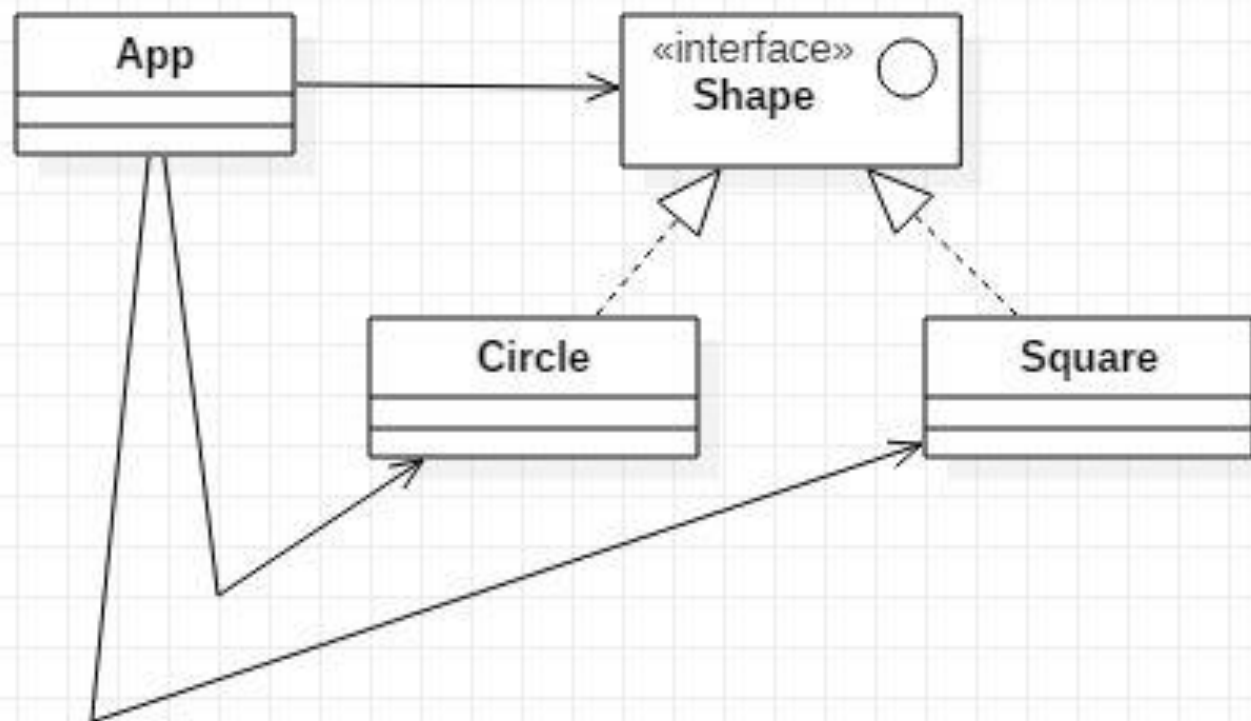
```
Odwołania: 3
public interface Shape
{
    Odwołania: 2
    void DrawShape();
}

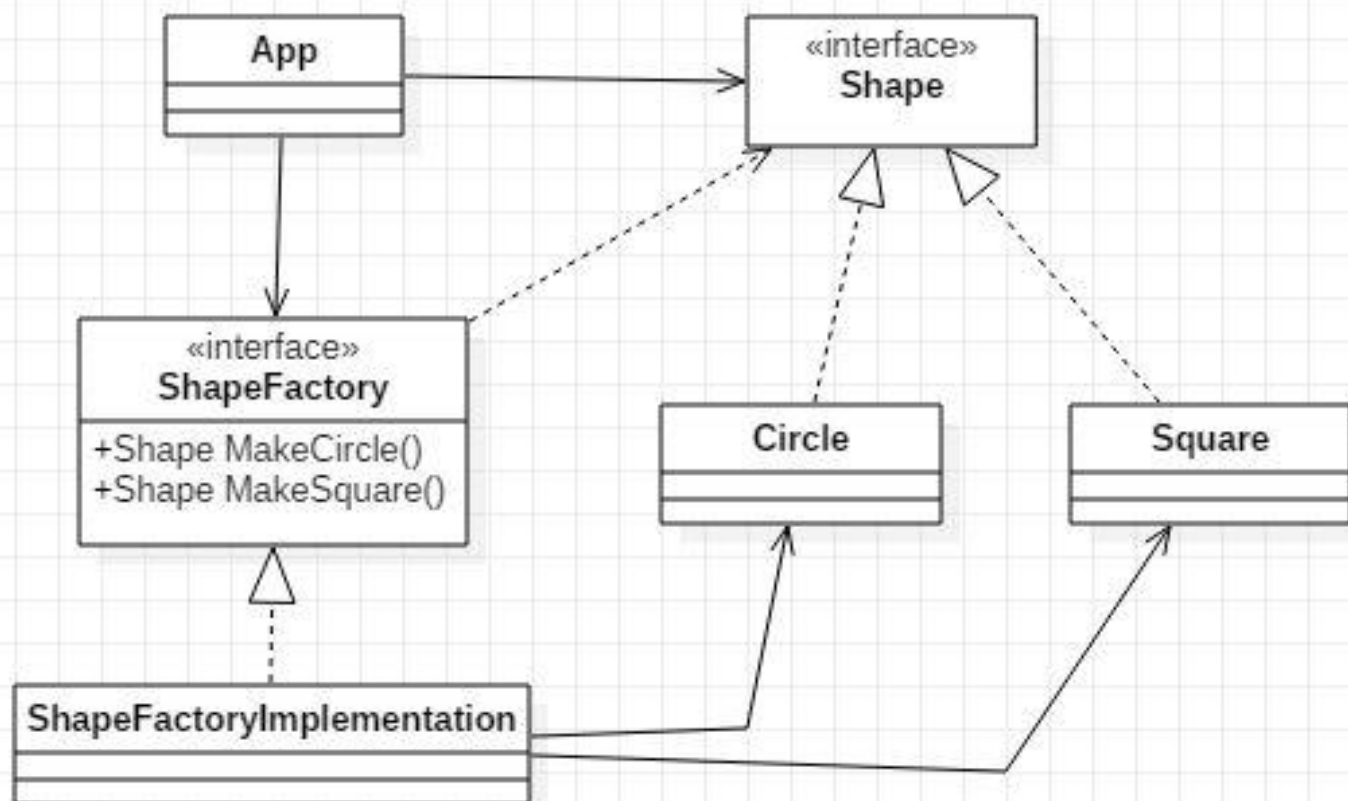
1 odwołania
public class Square : Shape
{
    Odwołania: 2
    public void DrawShape()
    {
        Console.WriteLine("Narysowano kwadrat");
    }
}

Odwołania: 0
public class Program
{
    Odwołania: 0
    static void Main()
    {
        Shape sq = FactoryMethod();

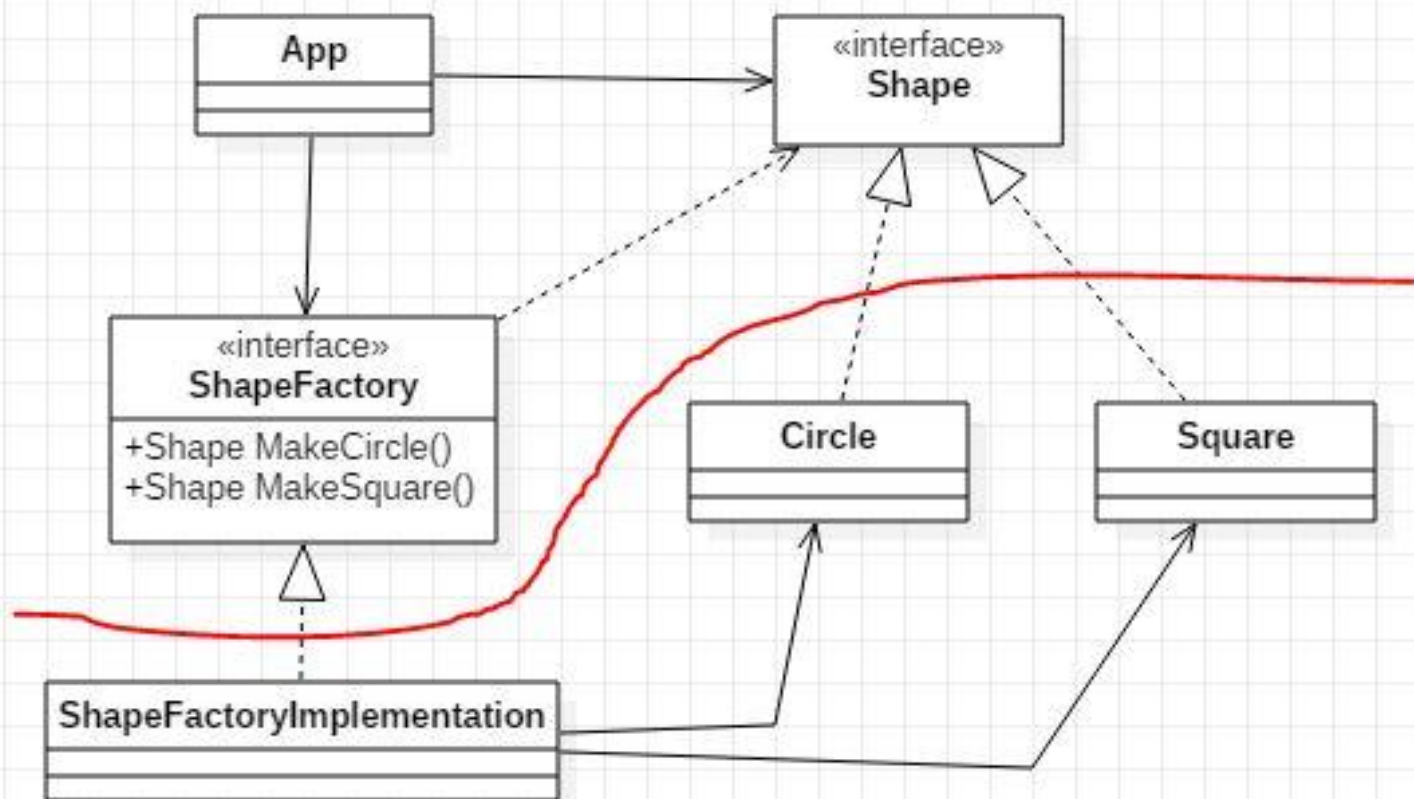
        sq.DrawShape();
    }

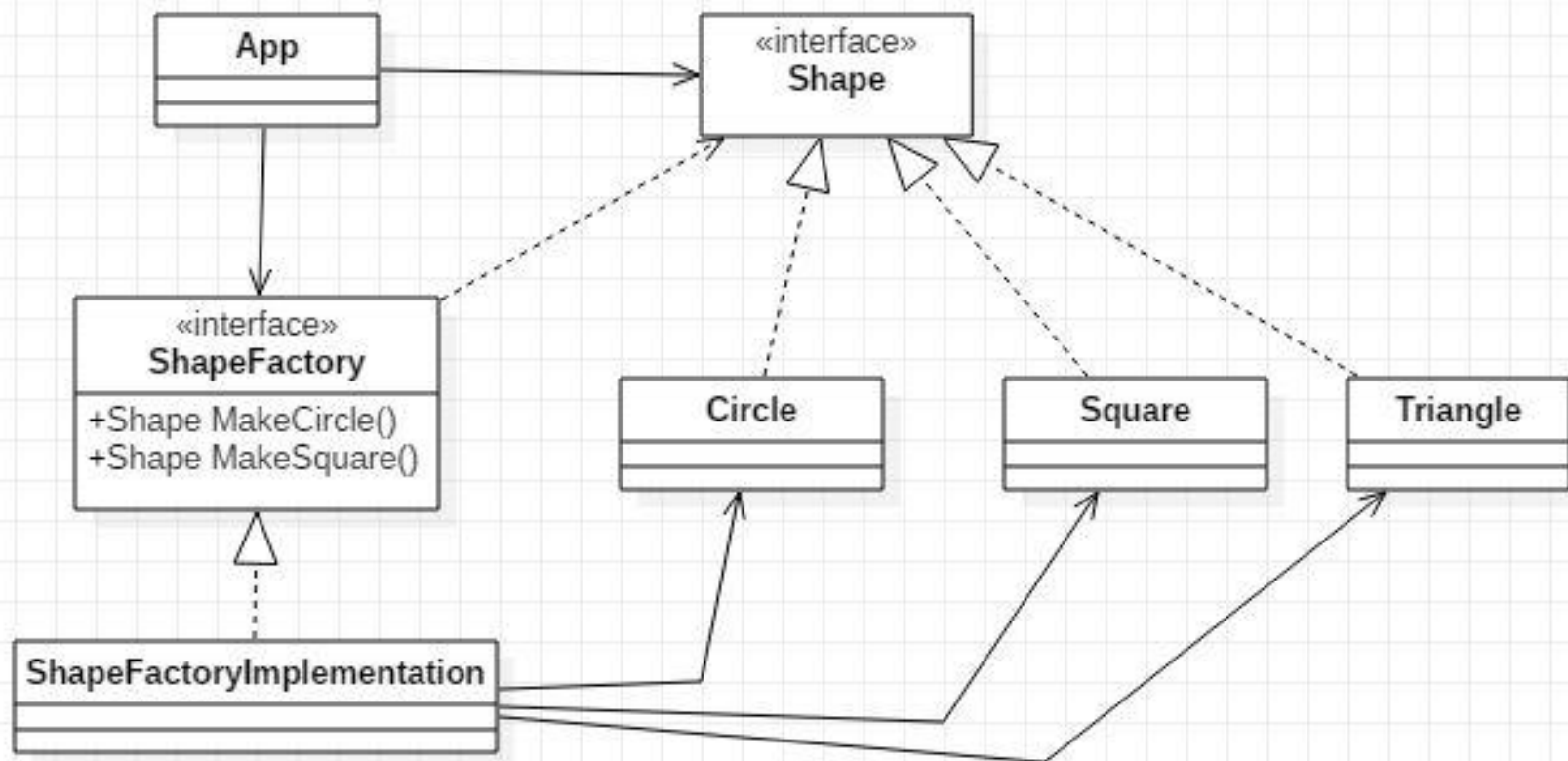
    1 odwołania
    private static Shape FactoryMethod()
    {
        return new Square();
    }
}
```

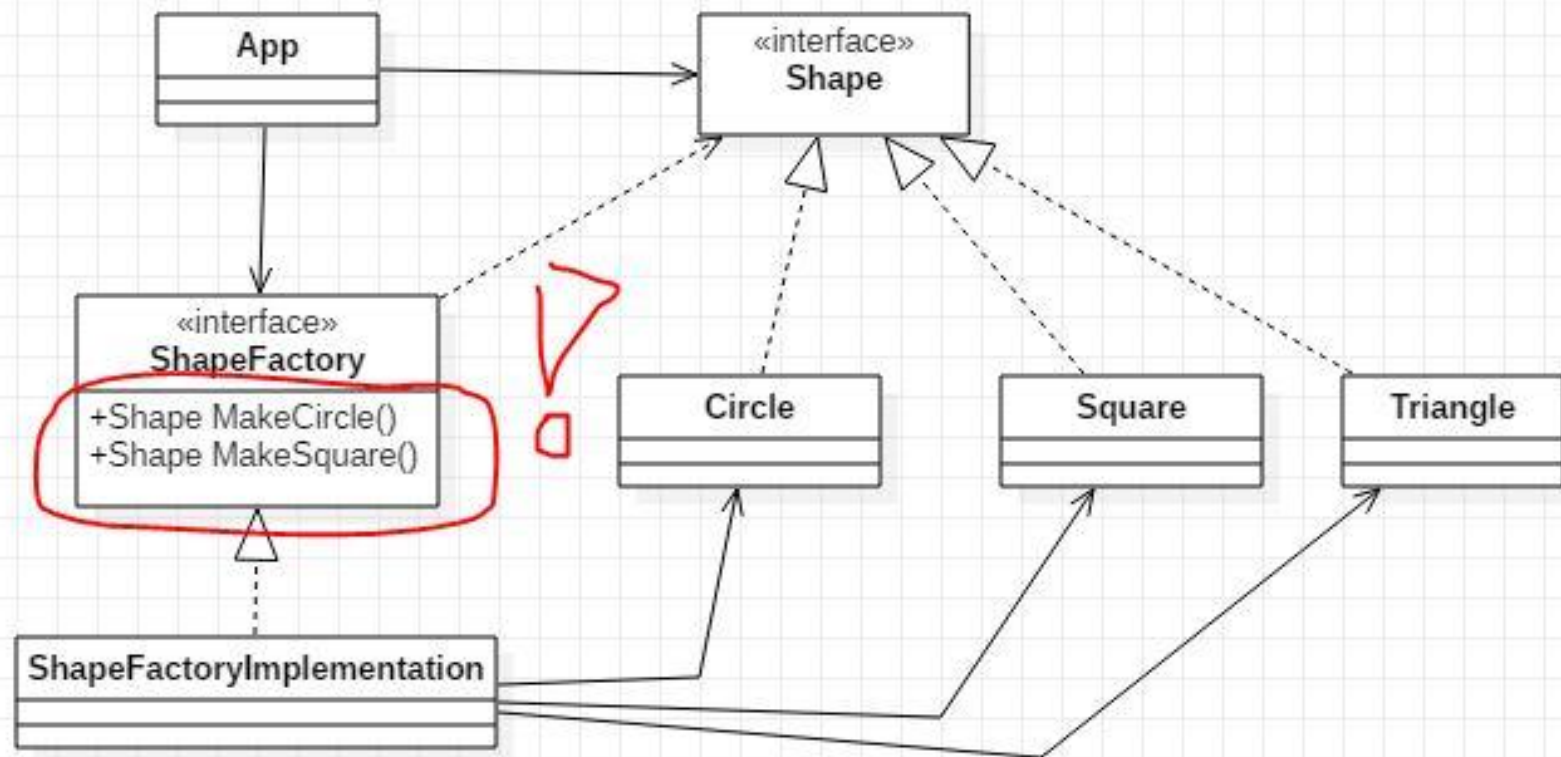








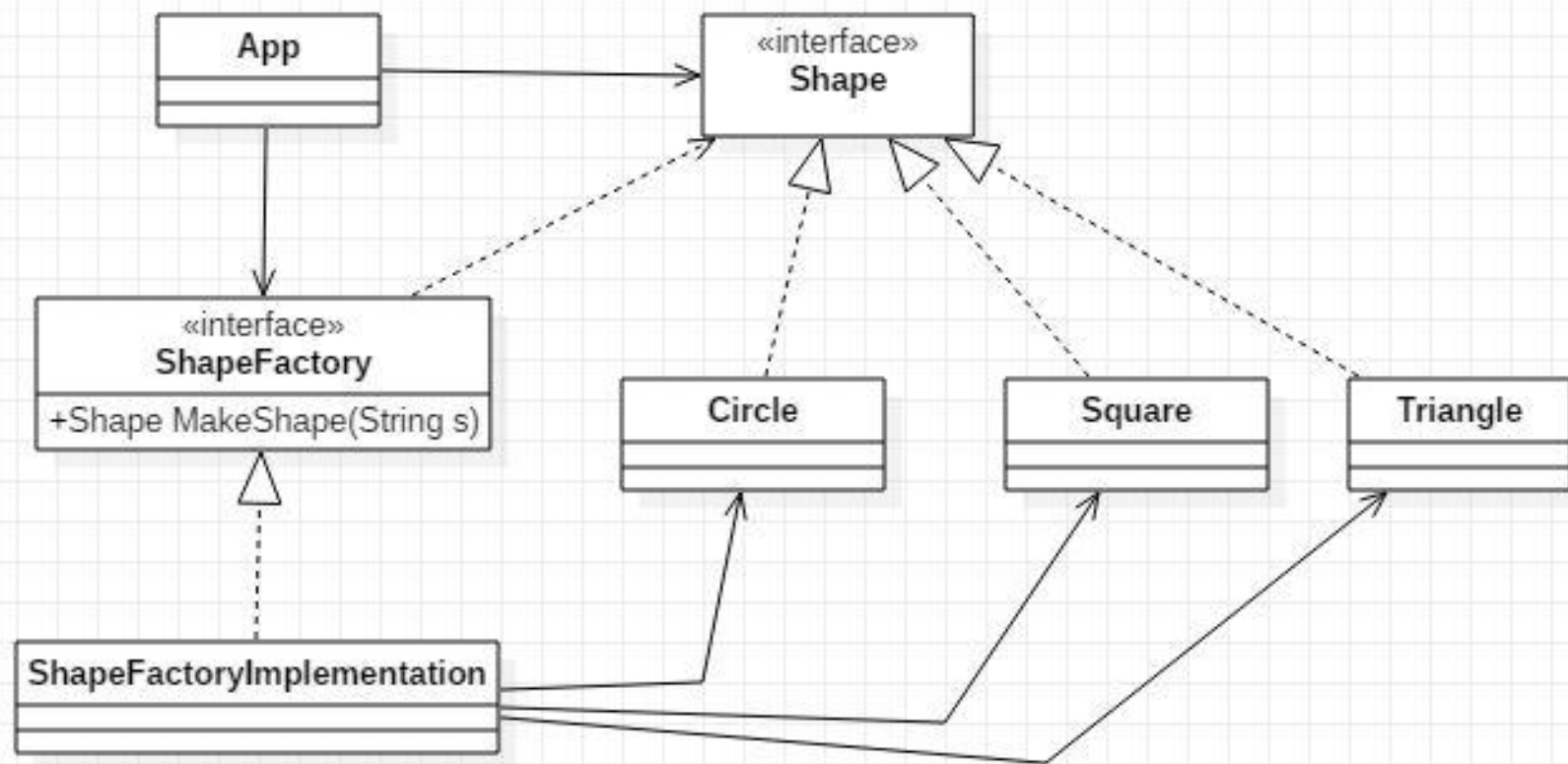




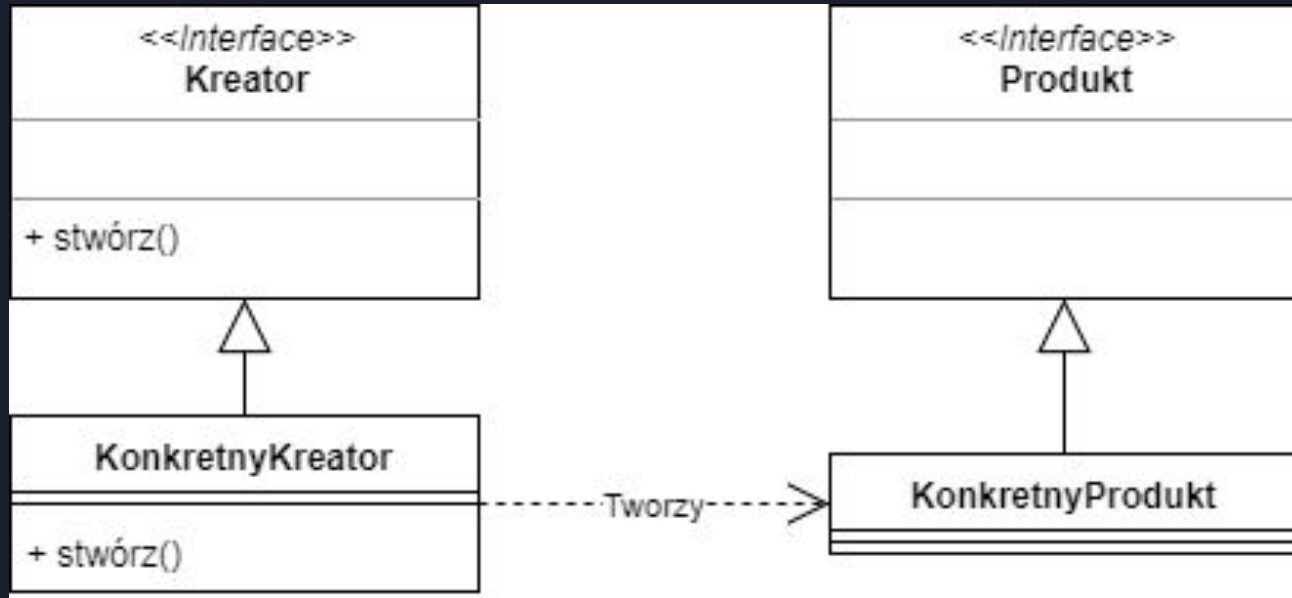


# Metoda Wytwórcza (Factory Method)

kreacyjny wzorzec projektowy, którego celem jest dostarczenie interfejsu do tworzenia obiektów nieokreślonych jako powiązanych typów.  
Tworzeniem egzemplarzy zajmują się podklasy.



# Metoda Wytwórcza - UML



Odwolania: 3

```
public interface Shape
{
    Odwolania: 2
    void DrawShape();
}
```

1 odwołanie

```
public class Square : Shape
```

```
{
    Odwolania: 2
    public void DrawShape()
    {
        Console.WriteLine("Narysowano kwadrat");
    }
}
```

Odwolania: 2

```
public interface Factory
```

```
{
    Odwolania: 2
    Shape CreateShape();
}
```

1 odwołanie

```
public class SquareFactory : Factory
```

```
{
    Odwolania: 2
    public Shape CreateShape()
    {
        return new Square();
    }
}
```

Odwolania: 0

```
public class Program
```

```
{
    Odwolania: 0
    static void Main()
    {
        Factory ShapeFactory = new SquareFactory();

        ShapeFactory.CreateShape().DrawShape();
    }
}
```

```

Odwołania: 4
public interface Shape
{
    Odwołania: 3
    void DrawShape();
}

```

```

1 odwołanie
public class Square : Shape
{
    Odwołania: 3
    public void DrawShape()
    {
        Console.WriteLine("Narysowano kwadrat");
    }
}

```

```

1 odwołanie
public class Circle : Shape
{
    Odwołania: 3
    public void DrawShape()
    {
        Console.WriteLine("Narysowano koło");
    }
}

```

```

Odwołania: 0
public class Program
{
    Odwołania: 0
    static void Main()
    {
        ShapeFactory Factory = new ShapeFactoryImplementation();

        Factory.MakeShape("kwadrat").DrawShape();
    }
}

```

```

Odwołania: 2
public interface ShapeFactory
{
    Odwołania: 2
    Shape MakeShape(string s);
}

1 odwołanie
public class ShapeFactoryImplementation : ShapeFactory
{
    Odwołania: 2
    public Shape MakeShape(string s)
    {
        if(s == "kwadrat")
        {
            return new Square();
        }
        if (s == "koło")
        {
            return new Circle();
        }
        else
        {
            throw new Exception(
                $"ShapeFactoryImplementation nie mogła utworzyć obiektu klasy{s}");
        }
    }
}

```

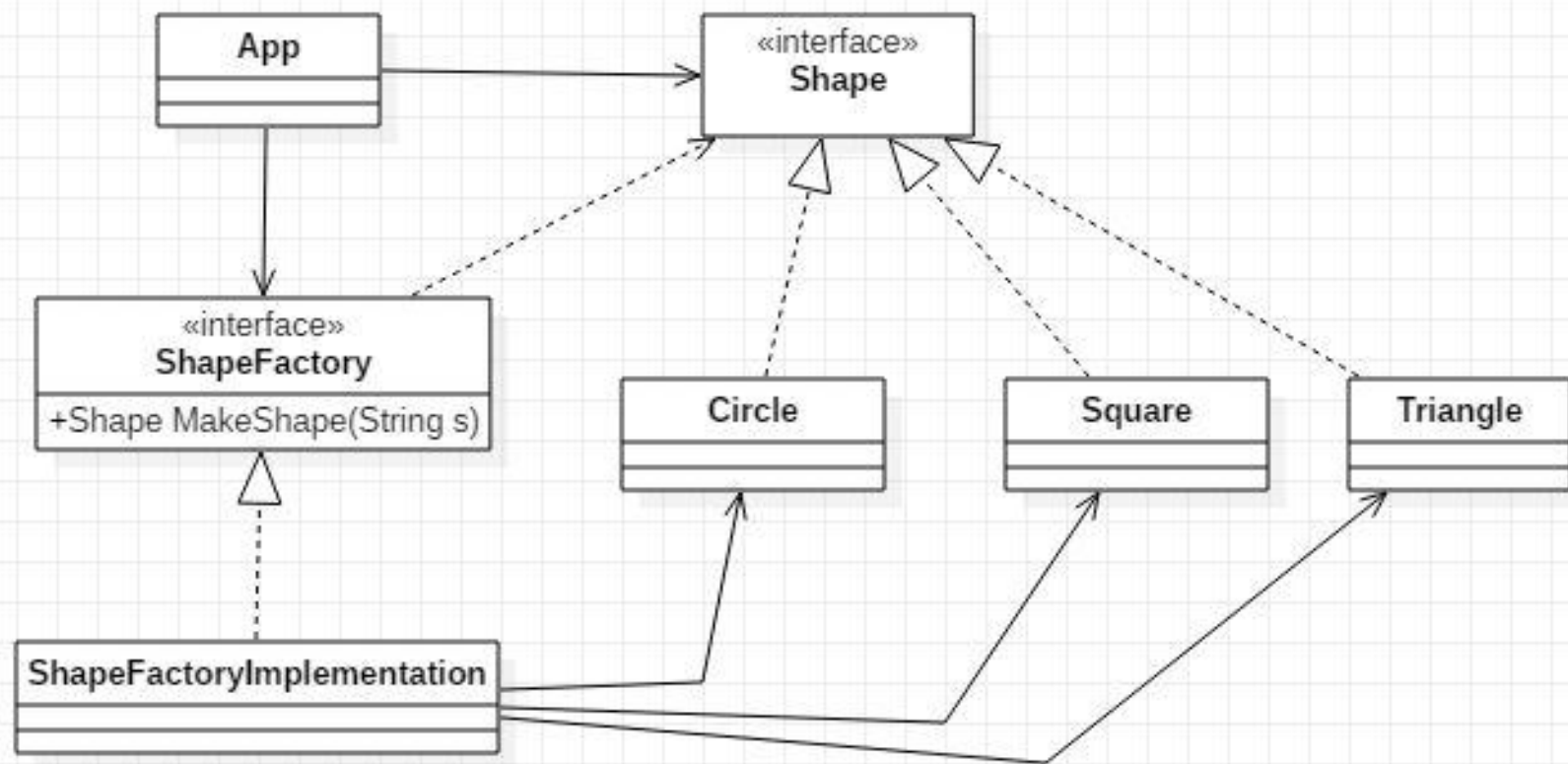


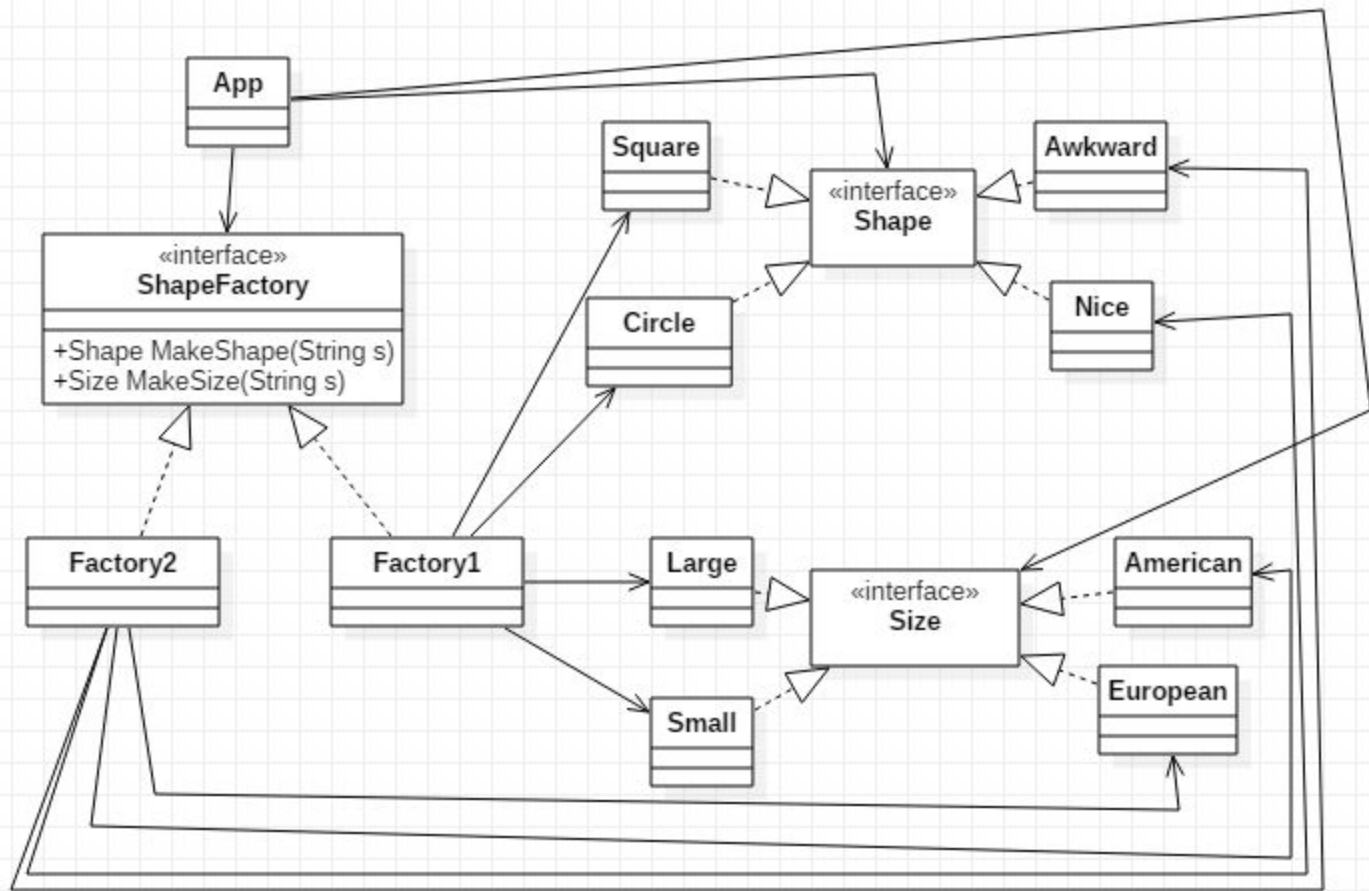


# Fabryka Abstrakcyjna (Abstract Factory)

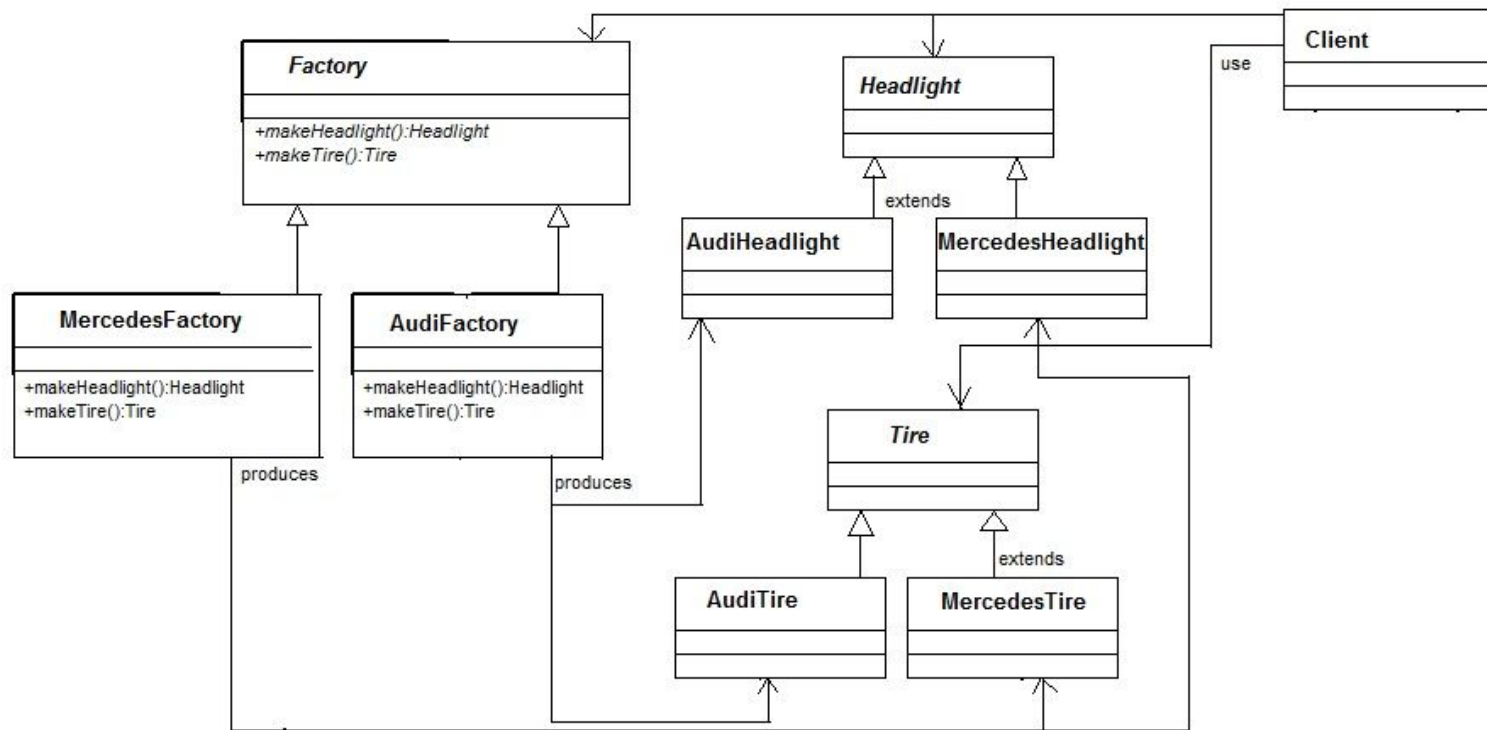
Wzorzec konstrukcyjny, służący do tworzenia nowych obiektów, związanych z jednym, wspólnym interfejsem. Główną zaletą stosowania takiej metody tworzenia nowych obiektów jest uniezależnienie się od konkretnej implementacji oraz od procesu tworzenia instancji. Obiekty tworzone są w przewidywalny sposób, wg. ustalonego interfejsu, przez co proces ten jest scentralizowany oraz hermetyczny.

Independent Deployability  
Dependency Inversion Principle

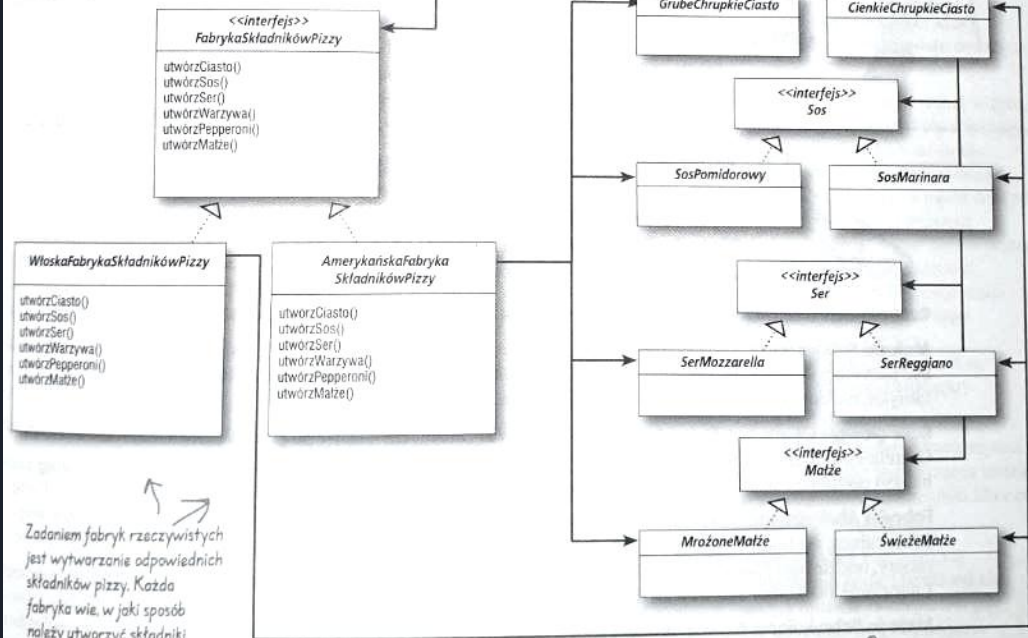




# Fabryka Abstrakcyjna - UML



Klasa abstrakcyjna *FabrykaSkładnikówPizzy* jest interfejsem, który określa, w jaki sposób powinna być tworzona rodzina spokrewnionych produktów — wszystkie składniki, które są niezbędne do zrobienia pizzy.



Zadaniem fabryk rzeczywistych jest wytworzenie odpowiednich składników pizzy. Każda fabryka wie, w jaki sposób należy utworzyć składniki odpowiednie dla danego regionu.

Każda fabryka wytwarza różne implementacje tej samej rodziny produktów.



# Podsumowanie - Fabryki

Prosta fabryka: Jest najprostszym sposobem na oddzielenie klienta od implementacji. Przeważnie wywoływana przez klienta przez metodę i zwraca jeden z wielu obiektów tego samego interfejsu.

Metoda fabryczna (wytwórcza, fabrykująca): Wykorzystuje mechanizm dziedziczenia. Implementacja polega na stworzeniu metody abstrakcyjnej która zostanie zaimplementowana w klasie po niej dziedziczącej.

Fabryka abstrakcyjna: Wykorzystuje kompozycję. Zwraca całą rodzinę powiązanych ze sobą obiektów. Przeważnie wykorzystuje metodę fabryczną do tworzenia obiektów.

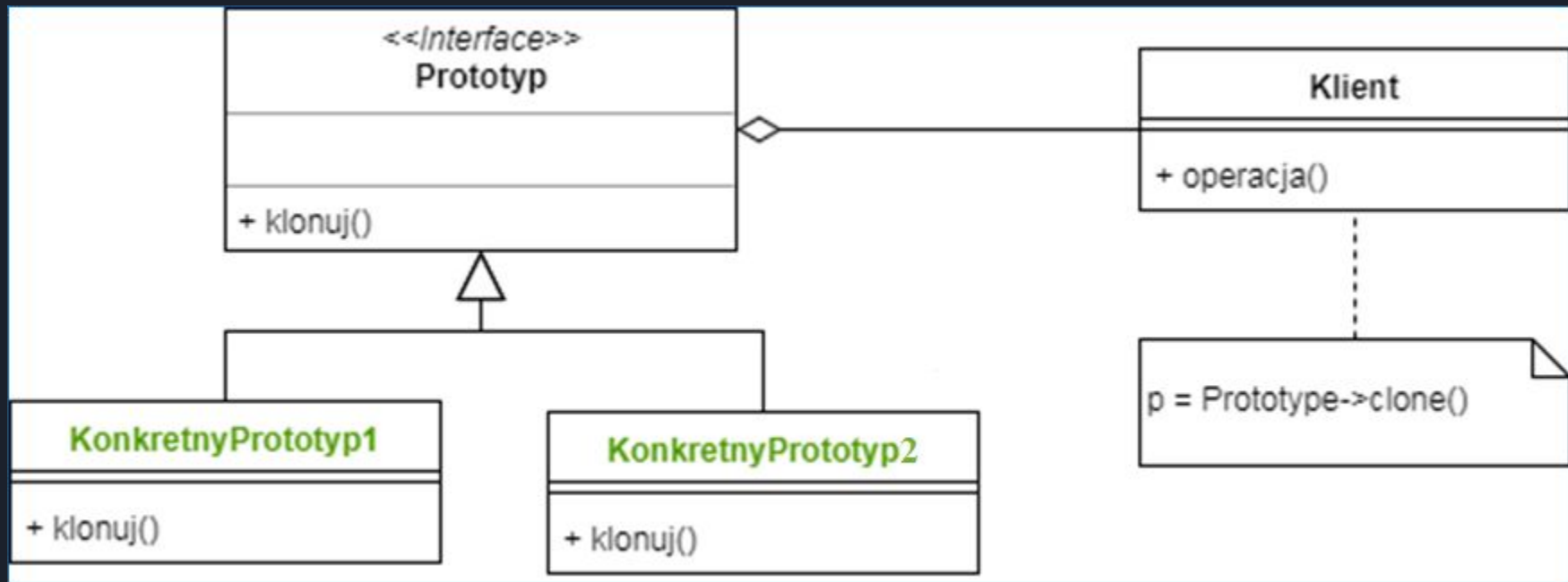


# Prototyp (Prototype)

Kreacyjny wzorzec projektowy, umożliwiający tworzenie obiektów danej klasy wykorzystując do tego już istniejący wzorcowy obiekt, nazywany prototypem. W tym celu wykorzystywane są mechanizmy klonowania lub kopiowania obiektów.

Wzorzec ten wykorzystywany jest, gdy chcemy utworzyć wiele obiektów tego samego typu lub o podobnych właściwościach oraz gdy chcemy uniezależnić system od sposobu w jaki tworzone są w nim jego produkty.

# Prototyp - UML







# Co się wypisze?

```
public class Person : ICloneable
{
    public readonly string[] Names;

    Odwołania: 2
    public Person(string[] names)
    {
        Names = names;
    }
    Odwołania: 0
    public override string ToString()
    {
        return $"{nameof(Names)}: {string.Join(", ", Names)}";
    }
    1 odwołanie
    public object Clone()
    {
        return new Person(Names);
    }
}
— odwołania
public class Program
{
    Odwołania: 0
    static void Main()
    {
        var john = new Person(new[] { "John", "Smith" });

        var jane = (Person)john.Clone();
        jane.Names[0] = "Jane";

        Console.WriteLine(john);
        Console.WriteLine(jane);
    }
}
```



# IClonable



Interfejs ICloneable w większości przypadków dokonuje “Shallow Copy”, a nie “Deep Copy”, na czym nam zależy.

Shallow Copy (płytkie kopiowanie) - Kopiowana jest struktura obiektu, a zmienne wskazują na ten sam obszar pamięci

Deep Copy (głębokie kopiowanie) - Kopiowanie struktury obiektu oraz wartości zmiennych



# Konstruktor kopiujący

Konstruktor kopiujący to konstruktor, który przyjmuje referencję do obiektu swojego typu i tworzy identyczny obiekt.



# Konstruktor kopiujący

```
public class Person
{
    public readonly string[] Names;

    1 odwołanie
    public Person(string[] names)
    {
        Names = names;
    }

    1 odwołanie
    public Person(Person other)
    {
        Names = new string[other.Names.Length];
        for (int i = 0; i < other.Names.Length; i++)
        {
            Names[i] = other.Names[i];
        }
    }

    Odwołania: 0
    public override string ToString()
    {
        return $"{nameof(Names)}: {string.Join(", ", Names)}";
    }
}

— odwołania
public class Program
{
    Odwołania: 0
    static void Main()
    {
        var john = new Person(new[] { "John", "Smith" });

        var jane = new Person(john);
        jane.Names[0] = "Jane";

        Console.WriteLine(john);
        Console.WriteLine(jane);
    }
}
```

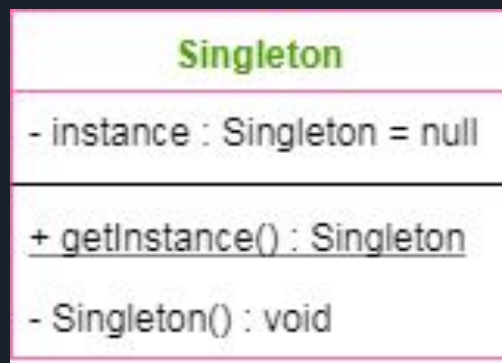
# Singleton

Kreacyjny wzorzec projektowania,  
ograniczający tworzenie obiektów danej  
klasy do tylko jednej instancji oraz  
zapewniający globalny dostęp do tego  
obiektu.





# Singleton UML





Nie używaj tego wzorca!

```
public sealed class Singleton
{
    private static Singleton instance = null;

    1 odwołanie
    private Singleton()
    {
    }

    1 odwołanie
    public static Singleton Instance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }

    1 odwołanie
    public void PrintSth()
    {
        Console.WriteLine("Metoda wywołana przy użyciu Singletona");
    }
}

— odwołania
public class Program
{
    — odwołania
    static void Main(string[] args)
    {
        Singleton s = Singleton.Instance();
        s.PrintSth();
    }
}
```



# Thread safe

```
public sealed class Singleton
{
    private static Singleton instance = null;
    private static readonly object padlock = new object();

    1 odwołanie
    Singleton()
    {
    }

    1 odwołanie
    public static Singleton Instance()
    {
        lock (padlock)
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }

    1 odwołanie
    public void PrintSth()
    {
        Console.WriteLine("Metoda wywołana przy użyciu Singletona");
    }
}

— odwołania
public class Program
{
    — odwołania
    static void Main(string[] args)
    {
        Singleton s = Singleton.Instance();
        s.PrintSth();
    }
}
```





Q&A

