Metodyki zwinne wytwarzania oprogramowania

Wykład 7

Marcin Młotkowski

23 listopada 2016

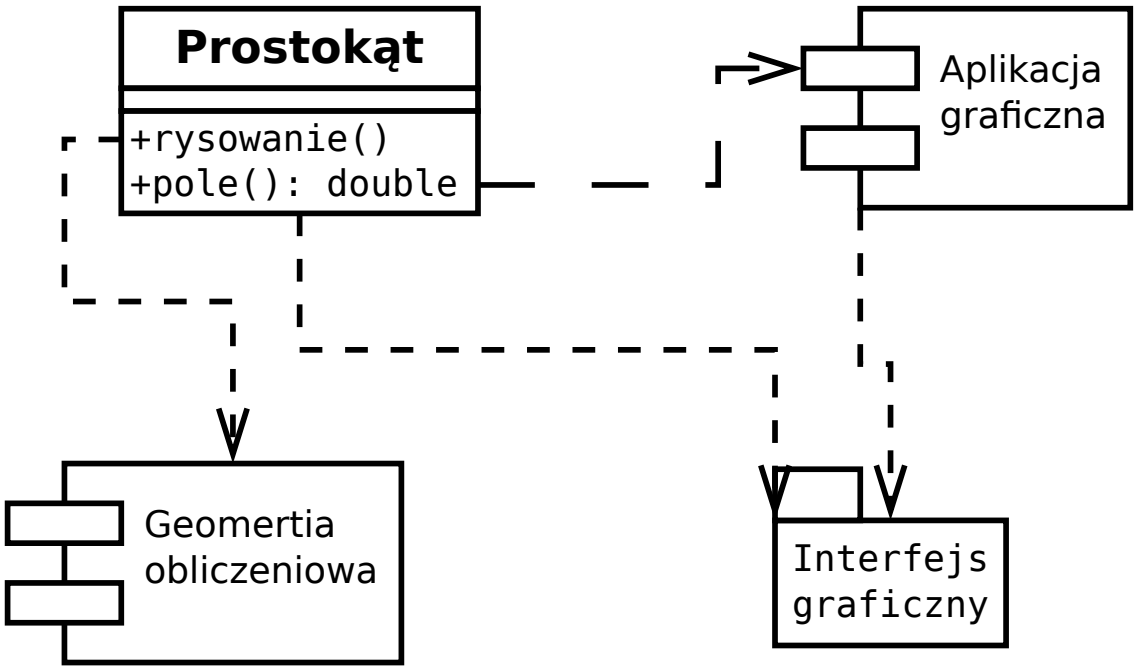
Plan wykładu

Single–Responsibility Principle (SRP)

Definicja

Za chwilę...

Przykład


Prostokąt
+rysowanie()
+pole(): double
Aplikacja graficzna
Geomertia obliczeniowa
Interfejs graficzny

Co robi klasa **Prostokąt**

Geometria

Modeluje matematyczny obiekt *prostokąt*, dzięki czemu może obliczyć jego pole.

Grafika

Wizualizuje prostokąt za pomocą interfejsu graficznego.

Zmiana projektu

Zmiana interfejsu graficznego z 2D na 3D.

Zmiana projektu

Zmiana interfejsu graficznego z 2D na 3D.

Zmiana wymaga

Ponowna implementacja metody **rysowanie()**

Zmiana projektu

Zmiana interfejsu graficznego z 2D na 3D.

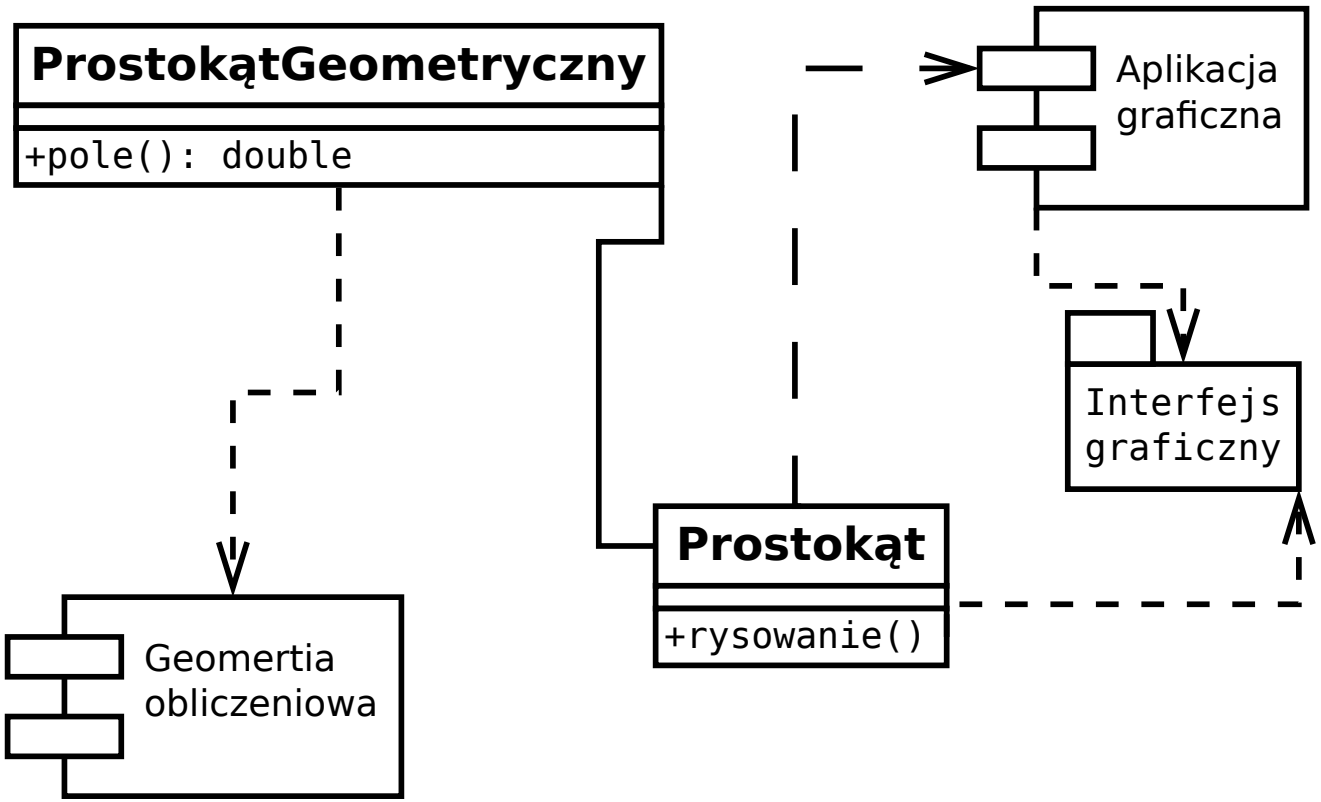
Zmiana wymaga

Ponowna implementacja metody **rysowanie()**

Konieczne

Skompilowanie, przetestowanie i wdrożenie *Aplikacji geometrycznej*, czy przypadkiem coś się nie zmieniło.

Zmiana projektu



Obserwacje

Zmiana projektu wymusiła zmiany tylko części klasy, tj. części "graficznej".

Może być druga przyczyna zmiany projektu: zmiana części "geometrycznej".

Zasada pojedynczej odpowiedzialności

Definicja

Żadna klasa nie może być modyfikowana z więcej niż jednego powodu.

Inny przykład

Klasa Pracownik

+ ObliczPensję(): decimal
+ Zapisz()
+ EdycjaGUI()

Plan wykładu

Tydzień temu na wykładzie:

Sztywność

Drobna zmiana w specyfikacji powoduje kaskadowe zmiany

Cel

Chcemy aby:

rozbudowa systemu wymagała tylko dodawania kodu, a nie jego modyfikacji.

Zasada otwarte–zamknięte (open–close principle, OCP)

Definicja

Składniki oprogramowania (klasy, moduły, procedury etc.) muszą być otwarte na rozbudowę, ale zamknięte na dla modyfikacji.

Objaśnienia

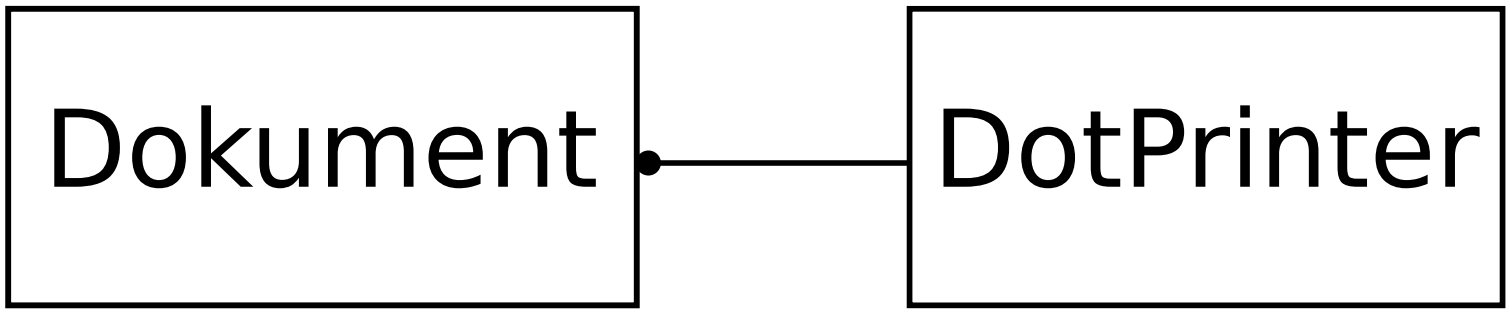
Otwarte na rozszerzenia

łatwo rozbudować funkcjonalność modułu

Zamknięte na modyfikacje

Zmiany nie mogą skutkować modyfikacją kodu źródłowego, ani nawet zmianami na poziomie binariów.

Przykład niezgodności z zasadą OCP: architektura klient-serwer



Kod źródłowy

```
public class DotPrinter {
   ...
}

public class Dokument {

  public void drukuj(DotPrinter prn)
  {
    ...
  }
}
```

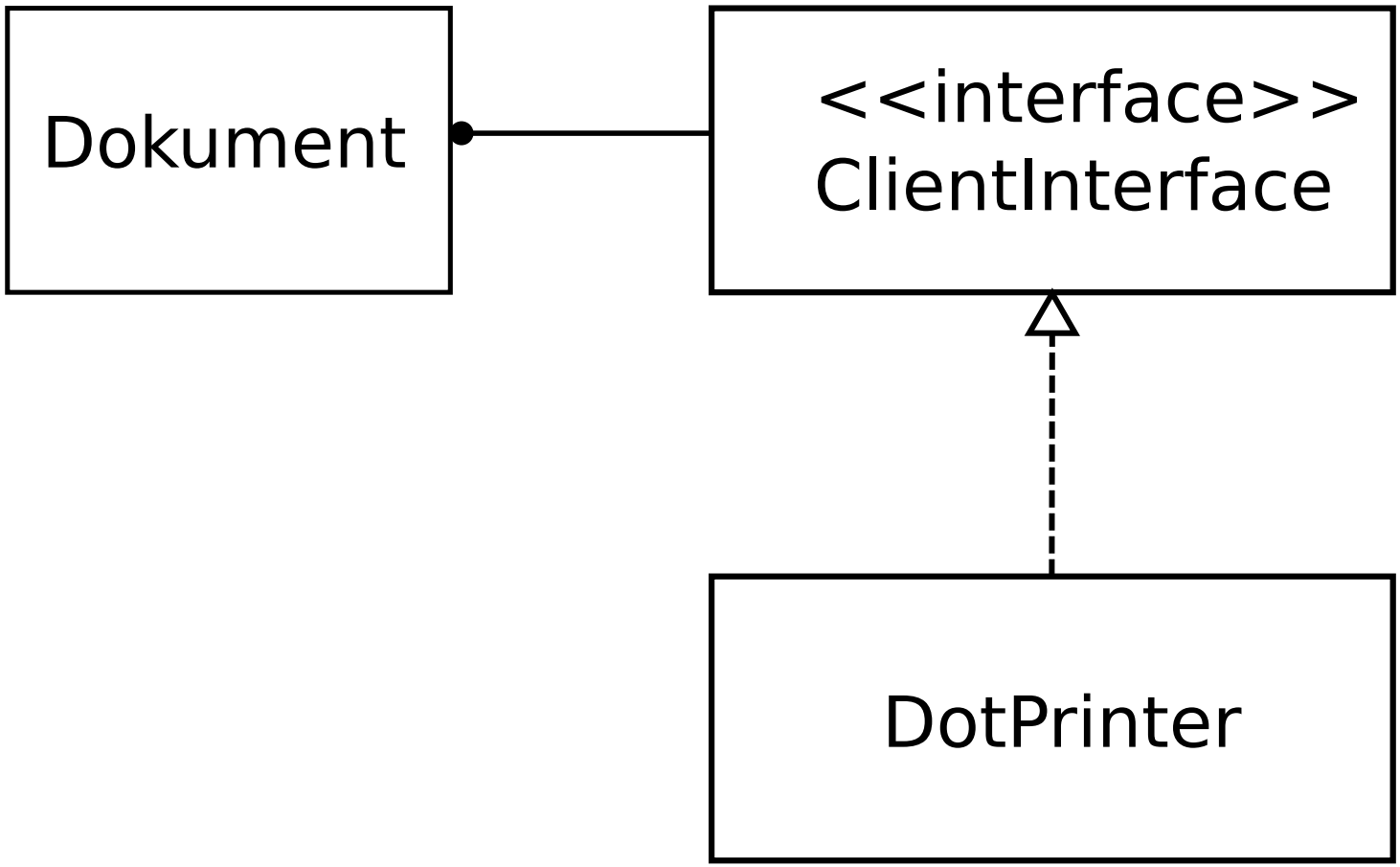
Kod źródłowy

```
public class DotPrinter {
   ...
}

public class Dokument {

  public void drukuj(DotPrinter prn)
  {
     ...
  }
}
```

Wymiana sprzętu

```
public class LaserPrinter {
   ...
```

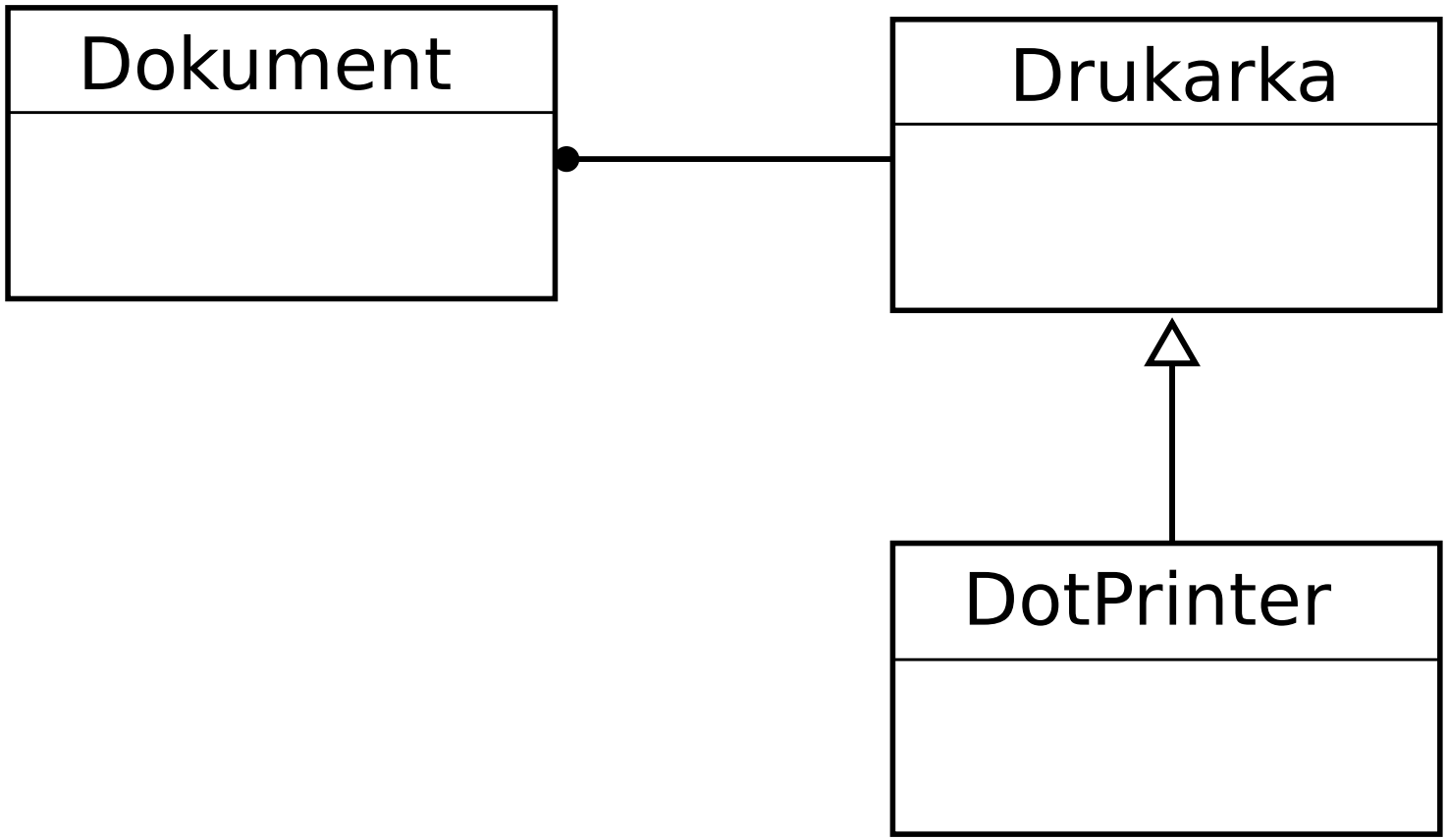
Rozwiązanie nr 1: wzorzec projektowy Strategia



Implementacja

```
public interface Printer {
   ...
}

public class DotPrinter : Printer {
   ...
}

public class Dokument {

  public void drukuj(Printer prn)
  {
     ...
  }
}
```

Rozwiązanie nr 2: wzorzec projektowy Template Method



Inny przykład złego projektu: figury geometryczne

```
enum ShapeType { circle, square };
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
  int i;
  for(i = 0; i < n; i++)
  {
    struct Shape *s = list[i];
    switch (s->itsType)
    {
      case square:
        DrawSquare( (struct Square*)s);
        break;
      case circle:
        DrawCircle( (struct Circle*)s);
        break;
    }
  }
}
```

Ocena takiego systemu

Dodanie nowej figury, np. trójkąta:

- wyszukanie wszystkich warunków sprawdzających typ figury;
- konieczność ponownej kompilacji modułu i wszystkich modułów zależnych;
- konieczność ponownej instalacji bibliotek.

Wady

Sztywność

Dodanie nowego elementu do wyliczenia wymusza serię działań: kompilacja, testowanie, instalacja.

Wady

Sztywność

Dodanie nowego elementu do wyliczenia wymusza serię działań: kompilacja, testowanie, instalacja.

Wrażliwość

Rozszerzenie implementacji wymaga starannego przejrzenia kodu w poszukiwaniu `switch--case` i `if--else` badających typ figury.

Dobre rozwiązanie

```
public interface Shape
{
  void Draw();
}

public class Square : Shape
{
  public void Draw() { ... }
}

public class Circle : Shape
{
  public void Draw() { ... }
}
```

Zgodność z zasadą

Otwartość

Łatwość zmian w zakresie dodawania nowych figur.

Zgodność z zasadą

Otwartość

Łatwość zmian w zakresie dodawania nowych figur.

Zamkniętość

Nie ma konieczności modyfikacji istniejącego kodu przy rozbudowie.

A jakie są wady

A jakie są wady

Trzeba przewidzieć kierunek zmian projektu.

Kiedy ten projekt zawiedzie

```
Shape shapes[];

foreach (Shape sh in shapes)
  sh.Draw();
```

Kiedy ten projekt zawiedzie

```
Shape shapes[];

foreach (Shape sh in shapes)
  sh.Draw();
```

Dodatkowe założenie

Chcemy, aby najpierw były kwadraty, a potem okręgi.

Kiedy ten projekt zawiedzie

```
Shape shapes[];

foreach (Shape sh in shapes)
  sh.Draw();
```

Dodatkowe założenie

Chcemy, aby najpierw były kwadraty, a potem okręgi.

Smutny wniosek

Nie da się zaprojektować tak system, aby był otwarty na dowolne zmiany.

Próba poprawienia

```
public interface Shape : IComparable
{
  void Draw();
}

...
shapes.Sort();
foreach (Shape sh in shapes)
  sh.Draw();
```

Przypomnienie

```
public interface IComparable
{
  int CompareTo(Object obj)
}
```

Próba implemetacji

```
public class circle : Shape
{
  public int CompareTo(object o)
  {
    if (o is Square)
      return -1;
    else
      return 0;
  }
}
```

Ocena implementacji

Czy ta implementacja spełnia warunek ocp?

Przewidywanie kierunku zmian

- szybkie wydania;

Przewidywanie kierunku zmian

- szybkie wydania;
- szybkia prezentacja;

Przewidywanie kierunku zmian

- szybkie wydania;
- szybkia prezentacja;
- szybkie uzyskanie informacji zwrotnej od klienta.

Plan wykładu

Oryginalne sformułowanie

Oczekujemy czegoś na kształt właściwości podstawienia: jeżeli dla każdego obiektu o_1 typu S istnieje taki obiekt o_2 typu T, że zachowanie wszystkich programów P zdefiniowanych na bazie typu T nie zmieni się, jeśli obiekt o_1 zastąpimy obiektem o_2, typ S jest podtypem typu T.

Barbara Liskov, 1988

Pierwszy przykład: figury geometryczne

```
public enum ShapeType { square, circle };

public class Shape
{
  private ShapeType type;
  public Shape(ShapetType t) { this.type = t; }

  public static void DrawShape(Shape s) {
    if (s.type == ShapeType.square)
      (s as Square).Draw();
    else if (s.type == ShapeType.circle)
      (s as Circle).Draw();
  }
}
```

Implementacja klas

```
public class Circle : Shape
{
  ...
  public Circle() : base(ShapeType.circle) {}
  public void Draw() { ... }
}

public class Square : Shape
{
  public Square() : base(ShapeType.square) {}
  public void Draw() { ... }

}
```

Użycie

Chcemy uniknąć polimorficznych wywołan bo kosztowne...

```
Shape[] obrazek = new Shape[100];
...
foreach(Shape sh in Shape)
    Shape.DrawShape(sh);
```

Użycie

Chcemy uniknąć polimorficznych wywołan bo kosztowne...

```
Shape[] obrazek = new Shape[100];
...
foreach(Shape sh in Shape)
    Shape.DrawShape(sh);
```

Ocena implementacji

Narusza zasadę otwarte–zamknięte

Drugi błąd

```
Shape s = new Circle(ShapeType.circle);
s.Draw();
```

Drugi błąd

```
Shape s = new Circle(ShapeType.circle);
s.Draw();
```

```
Shape s = new Shape(ShapeType.circle);
s.Draw();
```

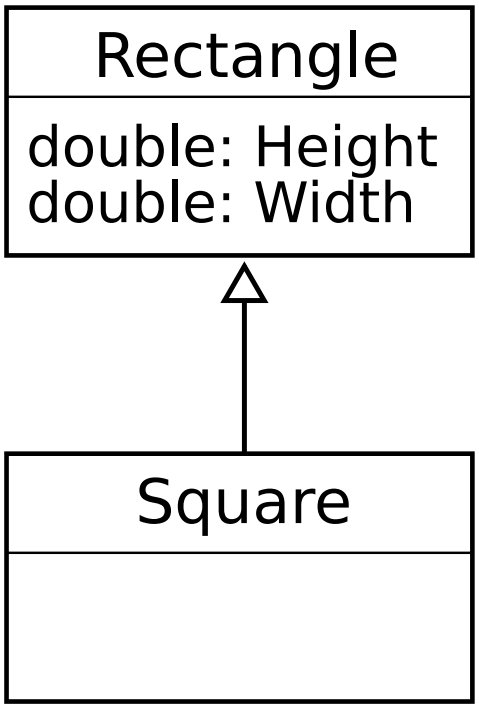
Ciekawszy przykład

```
public class Rectangle
{
  double width;
  double height;

  public double Width {
    get { return width; }
    set { width = value; }
  }

  public double Height {
    get { return height; }
    set { height = value; }
  }
```

Ciąg dalszy

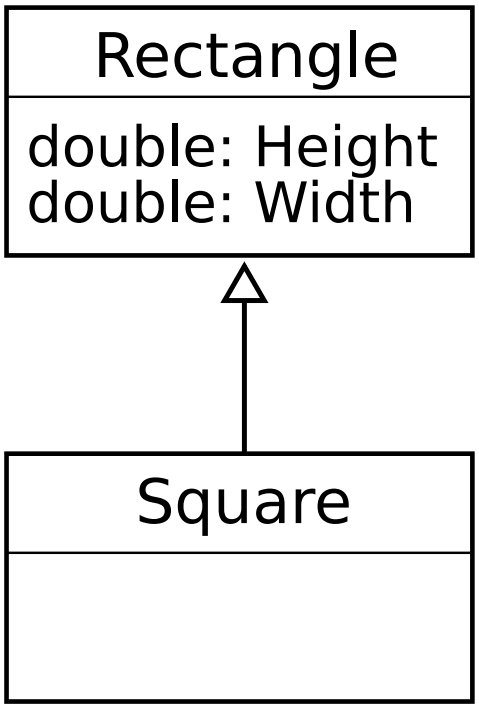
Rozszerzamy implementację o **Kwadraty**.

Jak rozwinąć projekt



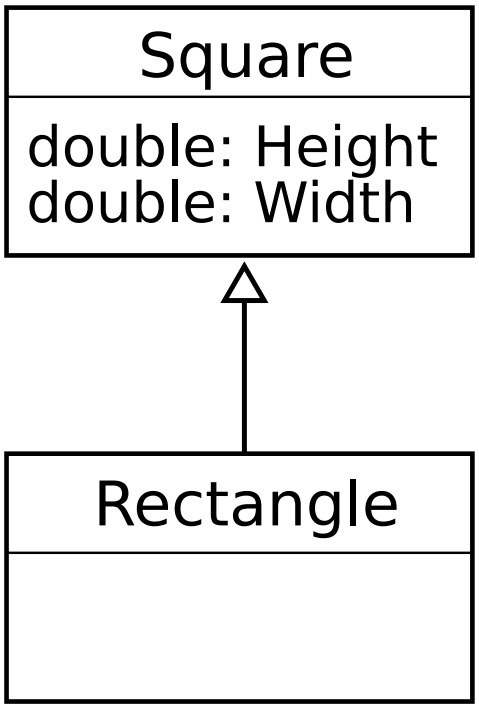
- kwadrat jest prostokątem;

Jak rozwinąć projekt



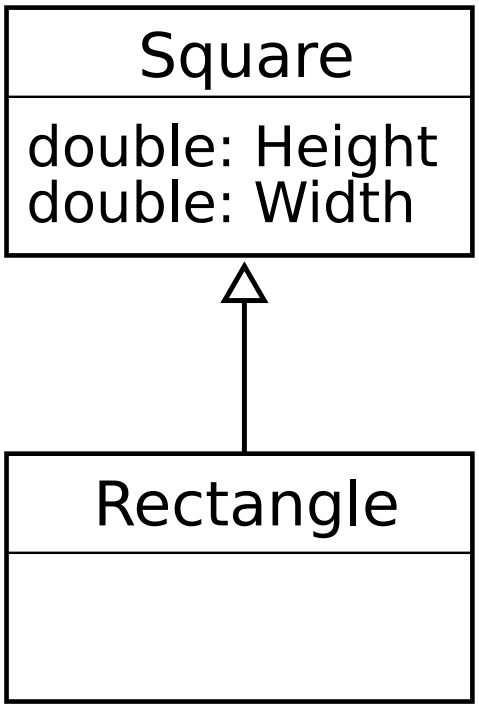
- kwadrat jest prostokątem;
- ale w klasie `Square` nie potrzebujemy tylu atrybutów!

A może tak



- W kwadracie wysokość i szerokość są równe, w prostokącie musimy usunąć ten warunek;

A może tak



- W kwadracie wysokość i szerokość są równe, w prostokącie musimy usunąć ten warunek;
- musimy zmienić implementację klasy Square (czego programiści zwinni unikają).

Poprawiamy projekt

```
public class Square : Rectangle
{
  public new double Width
  {
    set {
      base.Width = value;
      base.height = value;
    }
  }

  public new double Height
  {
    set {
      base.width = value;
      base.height = value;
```

Czy to działa?

```
void obszar(Rectangle r)
{
  r.Width = 4.0;
  r.Height = 5.0;
  if (r.Area() != 20.0)
    throw new Exception("Nieprawidłowa metoda Area!");
}
```

Czy to działa?

```
void obszar(Rectangle r)
{
  r.Width = 4.0;
  r.Height = 5.0;
  if (r.Area() != 20.0)
    throw new Exception("Nieprawidłowa metoda Area!");
}


obszar(new Square());
```

Morał

Nie można używać obiektów klasy **Square** zamiast **Rectangle**.

Morał

Nie można używać obiektów klasy **Square** zamiast **Rectangle**.

Szukanie winnego

- klasa `Rectangle`?
- klasa `Square`?
- funkcja `obszar`?

Analiza przypadków

Funkcja **obszar**

Wygląda na to, że programista użył prawidłowo interfejsu klasy **Rectangle**.

Analiza przypadków

Funkcja **obszar**

Wygląda na to, że programista użył prawidłowo interfejsu klasy **Rectangle**.

Klasa **Rectangle**

?

Analiza przypadków

Funkcja **obszar**

Wygląda na to, że programista użył prawidłowo interfejsu klasy **Rectangle**.

Klasa **Rectangle**

?

Klasa **Square**

Programista zmodyfikował działanie metod.

Może klasa **Square** nie musi być podklasą **Rectangle**?

Zasada podstawiania Liskov

Definicja

Musi istnieć możliwość zastępowania typów bazowych ich podtypami.