

Wzorce projektowe

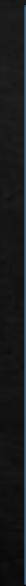
Jerzy Grynczewski

Plan bloku

Plan bloku

- ❖ Wstęp
- ❖ Paradygmat obiektowe
 - Podstawy
 - Relacje
 - Interfejsy
- ❖ Zasady projektowania
 - OOA, OOD, OOP
 - UML
 - YAGNI, KISS, DRY
 - GRASP
 - SOLID
 - STUPID i CUPID
- ❖ Wzorce projektowe
 - GoF
- ❖ Metryki kodu
- ❖ "Zapachy" kodu
- ❖ PEP8 i dobre praktyki

Wstęp



Co to jest wzorzec projektowy

Wzorzec projektowy to modelowe rozwiązanie powszechnie występującego problemu projektowego. Opisuje problem i ogólne podejście do jego rozwiązania.

Co to jest wzorzec projektowy

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to the problem".

Christopher Alexander (architect and design theorist not a software developer)

A Pattern Language, Oxford University Press, New York, 1977.

Przykłady wzorców projektowych



Budynki

Przykłady wzorców projektowych



Budynki



Kody elektroniczne
i hydrauliczne

Przykłady wzorców projektowych



Budynki



Kody elektroniczne
i hydrauliczne



Samochody

Co dają nam wzorce projektowe

Wzorce projektowe dają nam pewność, że wyniki naszej pracy są spójne, niezawodne i zrozumiałe.

Przykłady wzorców projektowych



Budynki



Kody elektroniczne i
hydrauliczne



Samochody



Interfejsy telefonów
komórkowych

Paradygmat obiektowy

Paradygmat obiektowy

- Klasy
- Obiekty
- Atrybuty (obiektowe, klasowe)
- Metody (obiektowe, klasowe, statyczne)

Podstawowe cechy paradymatu obiektowego

- Enkapsulacja (aka hermentyzacja, kapsułkowanie)
- Abstrakcja
- Dziedziczenie
- Polimorfizm

Relacje (aka powiązania)

- Interfejs (IMPLEMENTED-A)
- Dziedziczenie (IS-A)
- Kompozycja (HAS-A)
- Zależność (USE-A)

Interfejsy

Interfejsy w Pythonie

I w SOLID

Interfejsy w Pythonie

I w SOLID

Wsparcie w Java,
C#, Visual Basic
dla definicji
interfejsu

Interfejsy w Pythonie

I w SOLID

Wsparcie w Java,
C#, Visual Basic
dla definicji
interfejsu

Wsparcie w C++ za
pomocą klas
abstrakcyjnych

Interfejsy w Pythonie

I w SOLID

Wsparcie w Java,
C#, Visual Basic
dla definicji
interfejsu

Wsparcie w C++ za
pomocą klas
abstrakcyjnych

Domyślnie brak
wsparcia w
Pythonie

Interfejsy w Pythonie

I w SOLID

Domyślnie brak
wsparcia w
Pythonie

Wsparcie w Java,
C#, Visual Basic
dla definicji
interfejsu

Wsparcie w C++ za
pomocą klas
abstrakcyjnych

Wprowadzone
przez PEP 3119
za pomocą klas
abstrakcyjnych

Interfejsy w Pythonie

I w SOLID

Domyślnie brak wsparcia w Pythonie

Wsparcie w Java, C#, Visual Basic dla definicji interfejsu

Wprowadzone przez PEP 3119 za pomocą klas abstrakcyjnych

Wsparcie w C++ za pomocą klas abstrakcyjnych

Pierwszy raz pojawiły się w Pythonie 2.6 i 3

Definiowanie abstrakcyjnych klas bazowych

```
import abc

class MyABC(abc.ABC):
    """Abstract Base Class Definition"""

    @abc.abstractmethod
    def do_something(self, value):
        """Required method"""

    @property
    @abc.abstractmethod
    def some_property(self):
        """Required property"""
```

Definiowanie abstrakcyjnych klas bazowych

Moduł abc

```
import abc
```

```
class MyABC(abc.ABC):  
    """Abstract Base Class Definition"""
```

```
@abc.abstractmethod  
def do_something(self, value):  
    """Required method"""
```

```
@property  
@abc.abstractmethod  
def some_property(self):  
    """Required property"""
```

Definiowanie abstrakcyjnych klas bazowych

Moduł abc

```
import abc
```

Klasa abstrakcyjna

```
class MyABC(abc.ABC):  
    """Abstract Base Class Definition"""
```

```
@abc.abstractmethod  
def do_something(self, value):  
    """Required method"""
```

```
@property  
@abc.abstractmethod  
def some_property(self):  
    """Required property"""
```

Definiowanie abstrakcyjnych klas bazowych

Moduł abc

```
import abc
```

Klasa abstrakcyjna

```
class MyABC(abc.ABC):  
    """Abstract Base Class Definition"""
```

Metoda abstrakcyjna

```
@abc.abstractmethod  
def do_something(self, value):  
    """Required method"""
```

```
@property  
@abc.abstractmethod  
def some_property(self):  
    """Required property"""
```

Definiowanie abstrakcyjnych klas bazowych

Moduł abc

```
import abc
```

Klasa abstrakcyjna

```
class MyABC(abc.ABC):  
    """Abstract Base Class Definition"""
```

Metoda abstrakcyjna

```
@abc.abstractmethod  
def do_something(self, value):  
    """Required method"""
```

Property abstrakcyjne

```
@property  
@abc.abstractmethod  
def some_property(self):  
    """Required property"""
```

Implementacja konkretnej klasy

Dziedziczy z ABC

```
class MyClass(MyABC):
    """Implementation of MyABC"""

    def __init__(self, value=None):
        self._my_prop = value

    def do_something(self, value):
        """Implementation of abstract method"""
        self._myprop *= 2+value

    @property
    def some_property(self):
        """Implementation of abstract property"""
        return self._myprop
```

Implementacja konkretnej klasy

Dziedziczy z ABC

```
class MyClass(MyABC):  
    """Implementation of MyABC"""
```

Standardowy konstruktor

```
def __init__(self, value=None):  
    self._my_prop = value
```

```
def do_something(self, value):  
    """Implementation of abstract method"""  
    self._myprop *= 2+value
```

```
@property  
def some_property(self):  
    """Implementation of abstract property"""  
    return self._myprop
```

Implementacja konkretnej klasy

Dziediczy z ABC

```
class MyClass(MyABC):  
    """Implementation of MyABC"""
```

Standardowy konstruktor

```
def __init__(self, value=None):  
    self._my_prop = value
```

Implementacja
abstrakcyjnej metody

```
def do_something(self, value):  
    """Implementation of abstract method"""  
    self._myprop *= 2+value
```

```
@property  
def some_property(self):  
    """Implementation of abstract property"""  
    return self._myprop
```

Implementacja konkretnej klasy

Dziedziczy z ABC

```
class MyClass(MyABC):  
    """Implementation of MyABC"""
```

Standardowy konstruktor

```
def __init__(self, value=None):  
    self._my_prop = value
```

Implementacja abstrakcyjnej metody

```
def do_something(self, value):  
    """Implementation of abstract method"""  
    self._myprop *= 2+value
```

Implementacja abstrakcyjnego property

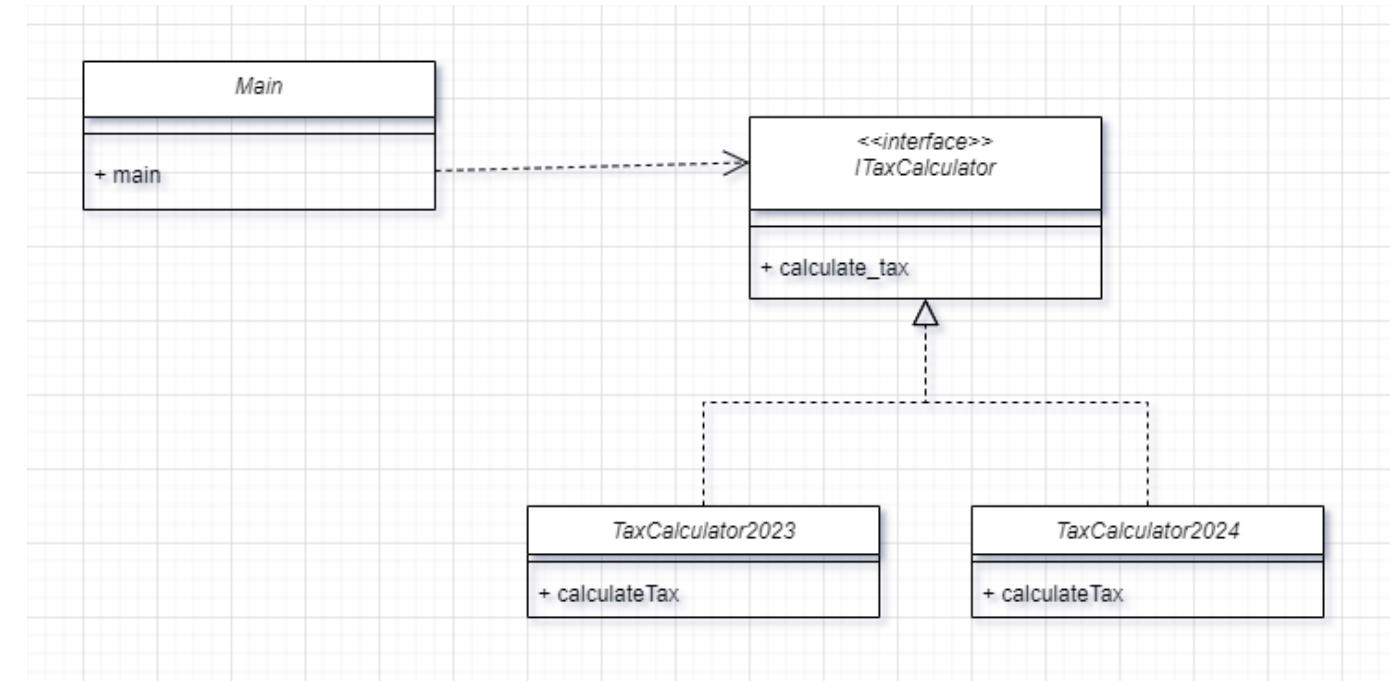
```
@property  
def some_property(self):  
    """Implementation of abstract property"""  
    return self._myprop
```

A co jeżeli **nie**
zaimplementujemy
abstrakcyjnej metody ?

A co jeżeli nie
zaimplementujemy
abstrakcyjnej metody ?

TypeError: Can't instantiate abstract class BadClass with
abstract methods do_something, some_property

Interfejs



Zasady projektowania

CRAIG LARMAN (1997)

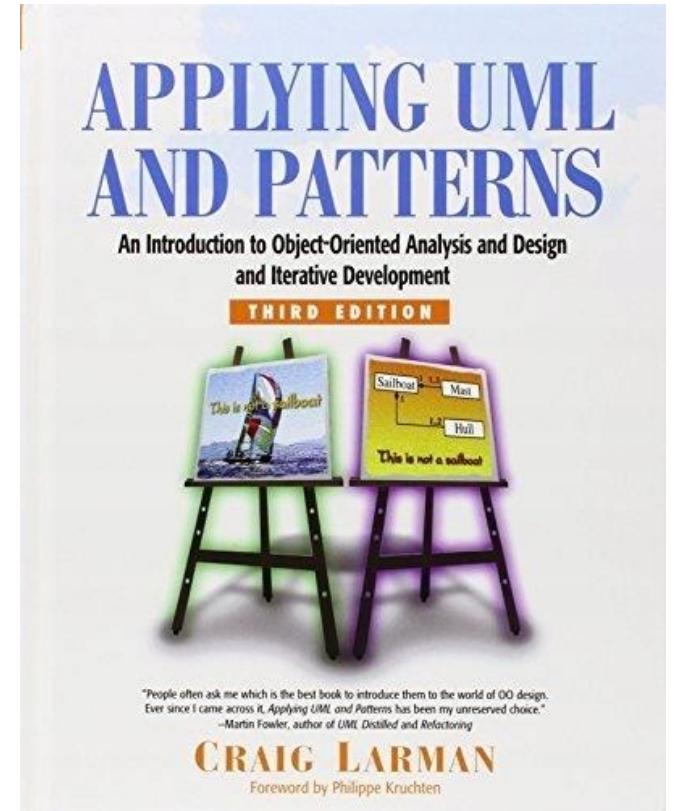
UML – Unified Modeling Language

OOA – Object Oriented Analysis

OOD – Object Oriented Design

OOP – Object Oriented Programming

GRASP - General Responsibility Assignment Software Patterns



Obiektowe modelowanie dziedziny

OOA, OOD, OOP

OOA

Object Oriented Analysis

OOA

Analiza zorientowana obiektowo

Badanie i klasyfikacj obiektów pojęciowych

OOA
zrób co należy



OOA

zrób co należy

- ❖ Opis słowny dziedziny

OOA

zrób co należy

- ❖ Opis słowny dziedziny
- ❖ Identyfikacja **klas pojęciowych** (lista kategorii klas pojęciowych lub analiza fraz rzeczownikowych)

Klasy pojęciowe

GraWMonopol

Gracz

Pionek

Plansza

Pole

KompletPól

Kostka

OOA

zrób co należy

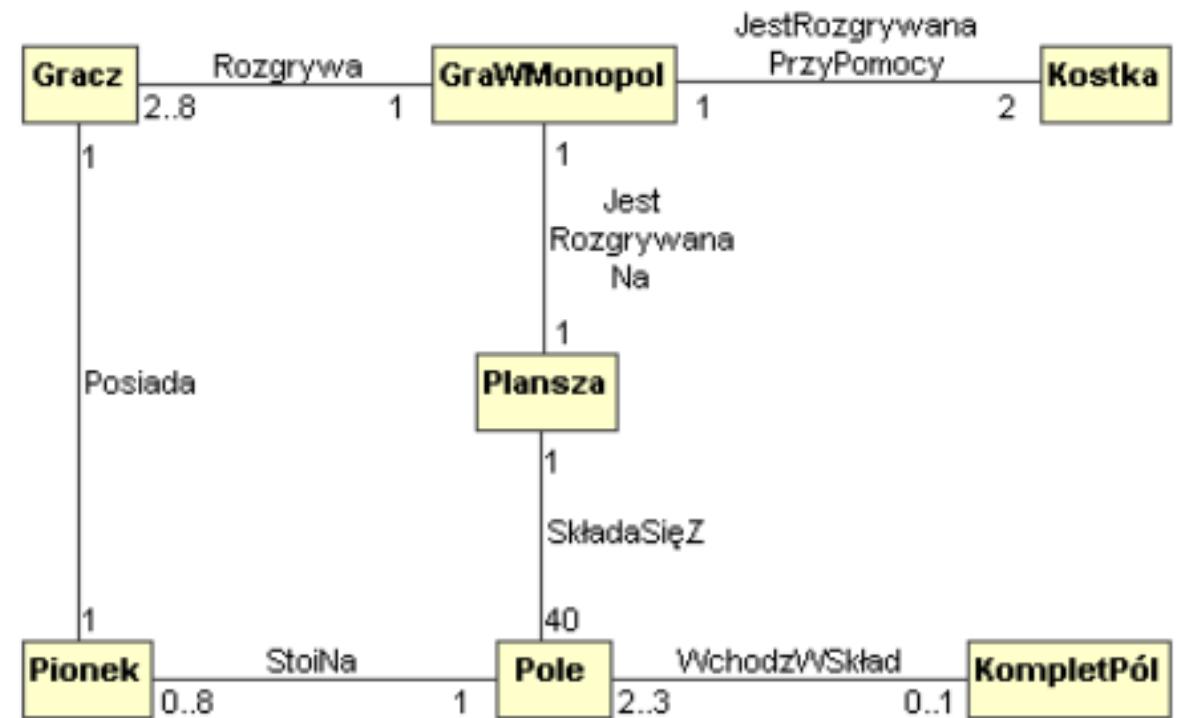
- ❖ Opis słowny dziedziny
- ❖ Identyfikacja **klas pojęciowych** (lista kategorii klas pojęciowych lub analiza fraz rzeczownikowych)
- ❖ Identyfikacja **powiązań** (lista kategorii powiązań, analiza fraz czasownikowych)

OOA

zrób co należy

- ❖ Opis słowny dziedziny
- ❖ Identyfikacja **klas pojęciowych** (lista kategorii klas pojęciowych lub analiza fraz rzeczownikowych)
- ❖ Identyfikacja **powiązań** (lista kategorii powiązań, analiza fraz czasownikowych)
- ❖ Określenie **liczebności** powiązań

Powiązania

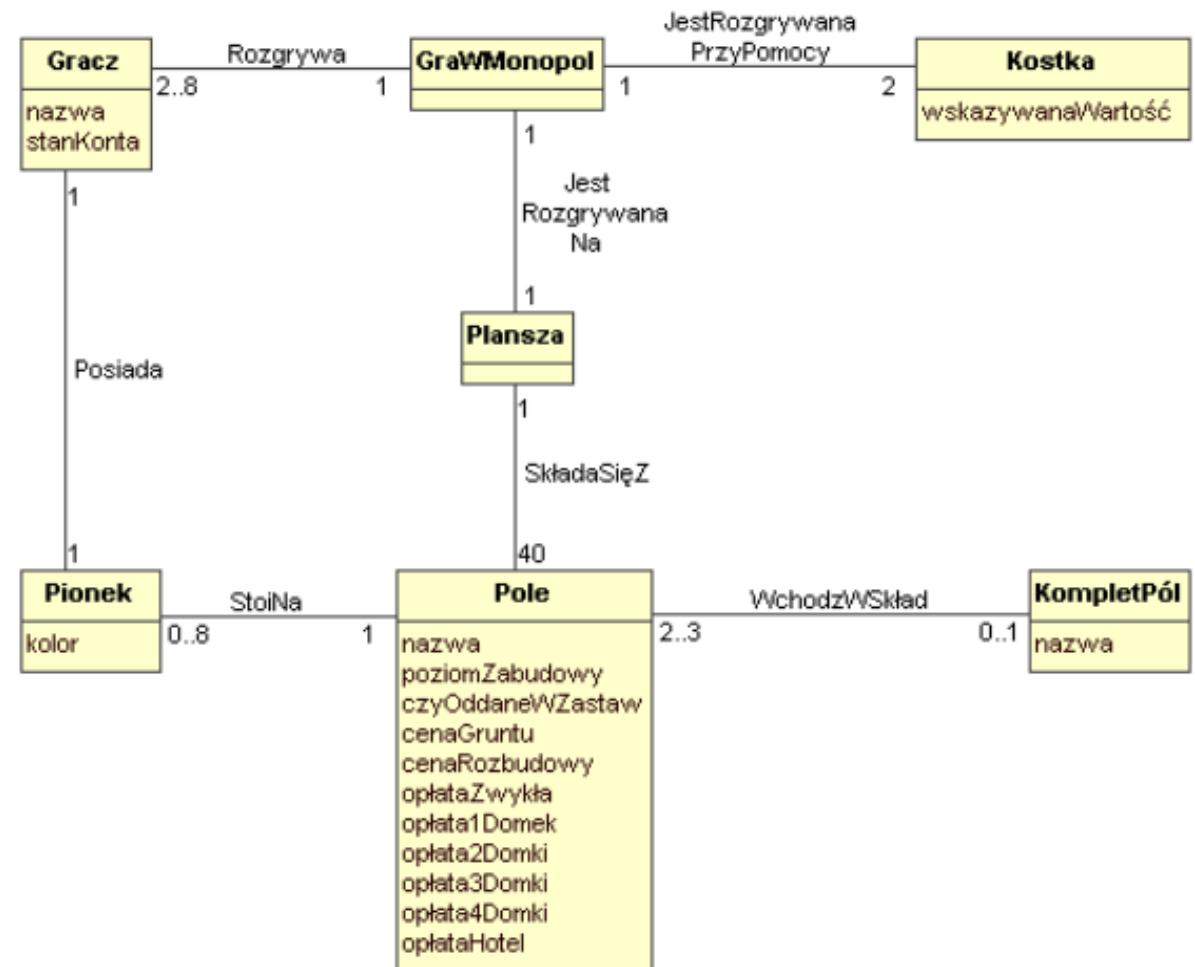


OOA

zrób co należy

- ❖ Opis słowny dziedziny
- ❖ Identyfikacja **klas pojęciowych** (lista kategorii klas pojęciowych lub analiza fraz rzeczownikowych)
- ❖ Identyfikacja **powiązań** (lista kategorii powiązań, analiza fraz czasownikowych)
- ❖ Uwzględnienie atrybutów klas pojęciowych

Atrybuty



Obiektowy model dziedziny

Obiektowy model dziedziny identyfikuje **klasy pojęciowe**, ich **atrybuty i powiązania** pomiędzy nimi. Stanowi bazę do tworzenia projektu obiektowego.

OOD

Object Oriented Design



OOD

Projektowanie zorientowane obiektowo

Opracowywanie **projektu obiektowego** na podstawie obiektowego modelu dziedziny. Polega na przypisaniu obiektom odpowiedzialności (za pamiętanie danych i za wykonywanie operacji na tych danych). To tutaj wykorzystywana jest wiedza o **wzorcach projektowych**.

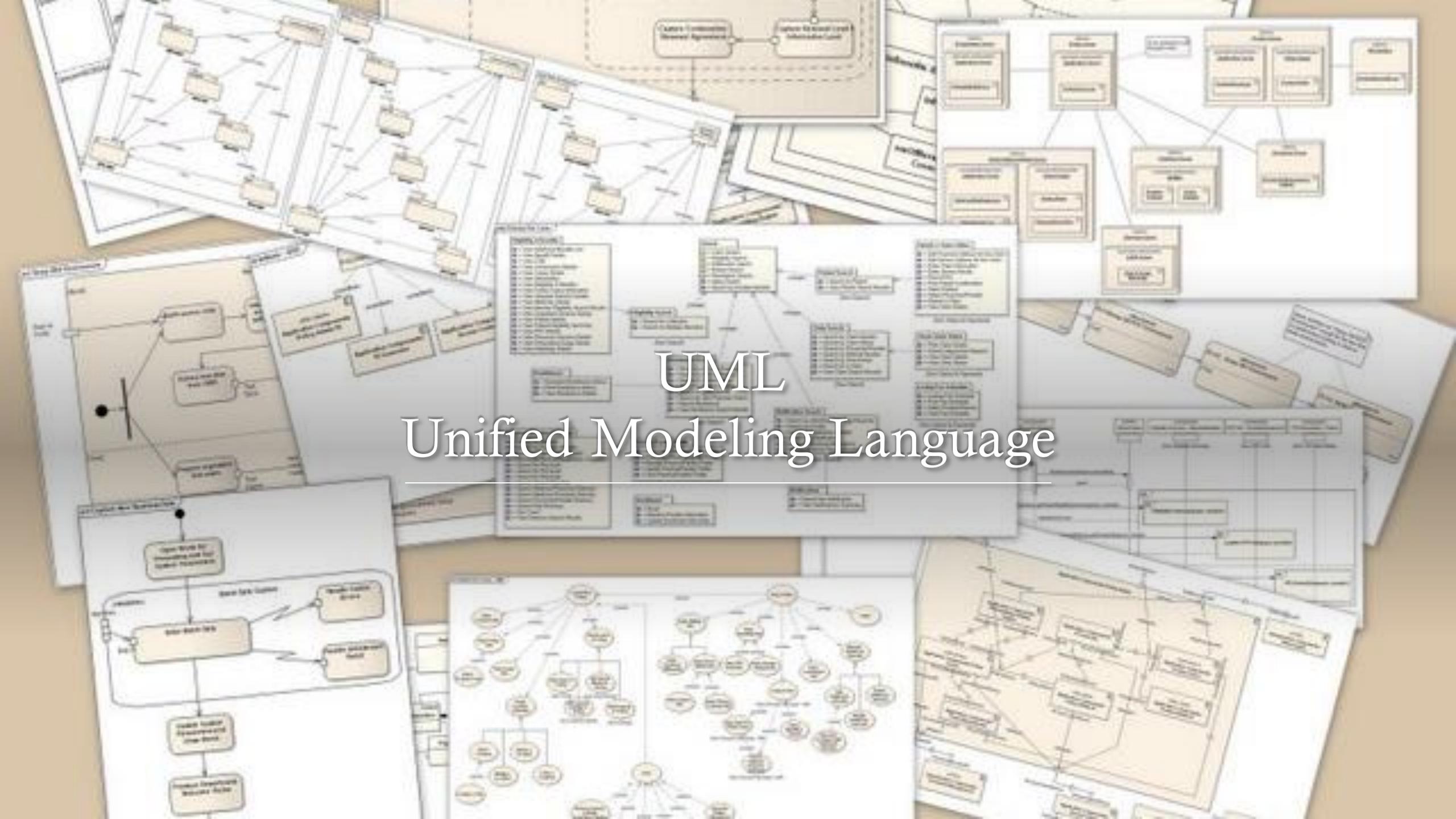
OOD

zrób jak należy

- ❖ Opracowanie **klas projektowych** (zastosowanie wzorców projektowych)

OOD
zrób jak należy

-
- ❖ Nałożenie więzów



The background of the image consists of a large, overlapping stack of various UML diagrams and system architecture sketches. These include class diagrams with objects and associations, sequence diagrams showing interactions between objects, state transition diagrams illustrating system behavior over time, and network architecture diagrams showing the relationships between different components or systems.

UML

Unified Modeling Language

UML

Zunifikowany język modelowania

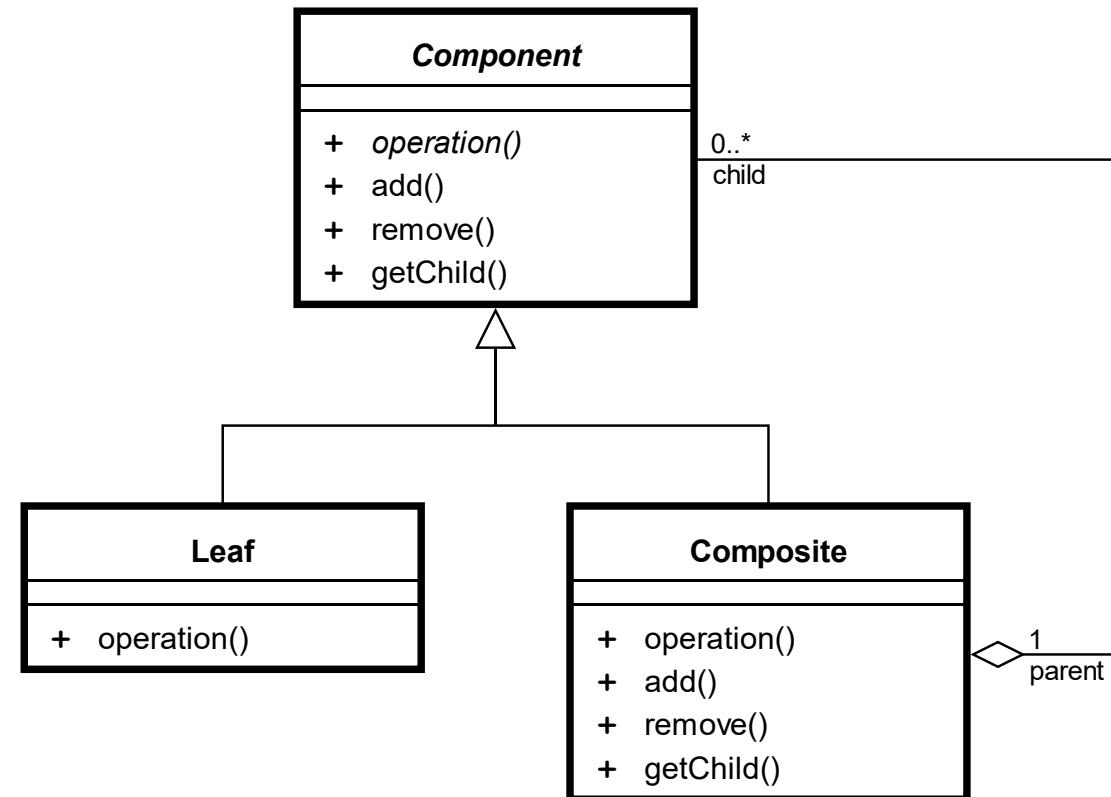
Graficzny język modelowania.

Twórcami języka są: Grady Booch, Ivar Jacobson I James Rumbaugh. Pierwszy standard – UML 1.0 powstał w 1997. Dalszym rozwojem języka zajęła się grupa OMG.

Obecny standard - UML 2.5.1 wyróżnia 17 rodzajów diagramów (14 diagramów głównych i 3 abstrakcyjne).

Diagram klas

Class diagram



Prezentacja klas i zależności pomiędzy nimi.

Diagram obiektów UML

Object Diagram

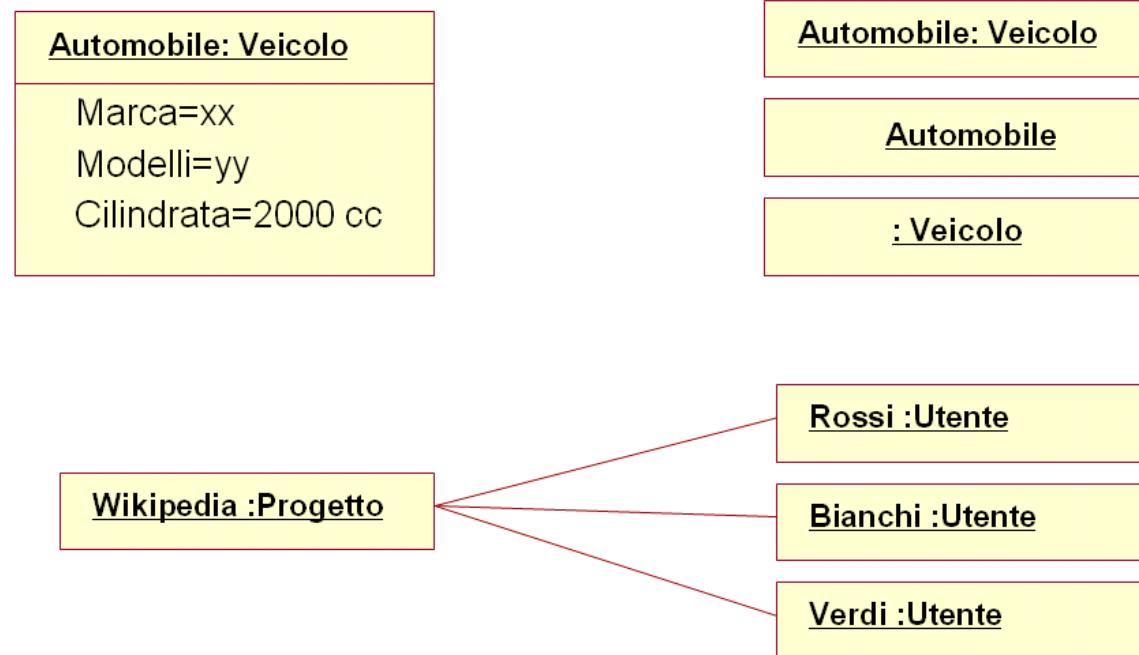


Diagram klas

Klasa

```
class Shape:  
    def __init__(self, position_x, position_y):  
        self.position_x = position_x  
        self.position_y = position_y  
  
    def render(self):  
        ...
```

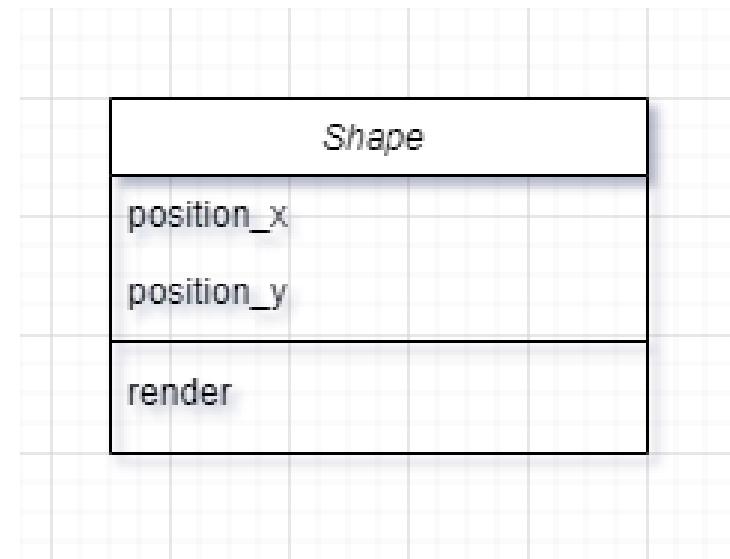


Diagram klas

Szczegóły klasy

```
class Person:
    def __init__(self, name: str, phone_number: str, age: int):
        self.name = name
        self._phone_number = phone_number
        self.__age = age

    def calculate_future_age(self, years: int) -> int:
        future_age = self.__age + years
        return future_age

    def _save_phone_number_to_file(self) -> None:
        ...

    def __update_age(self) -> None:
        ...
```

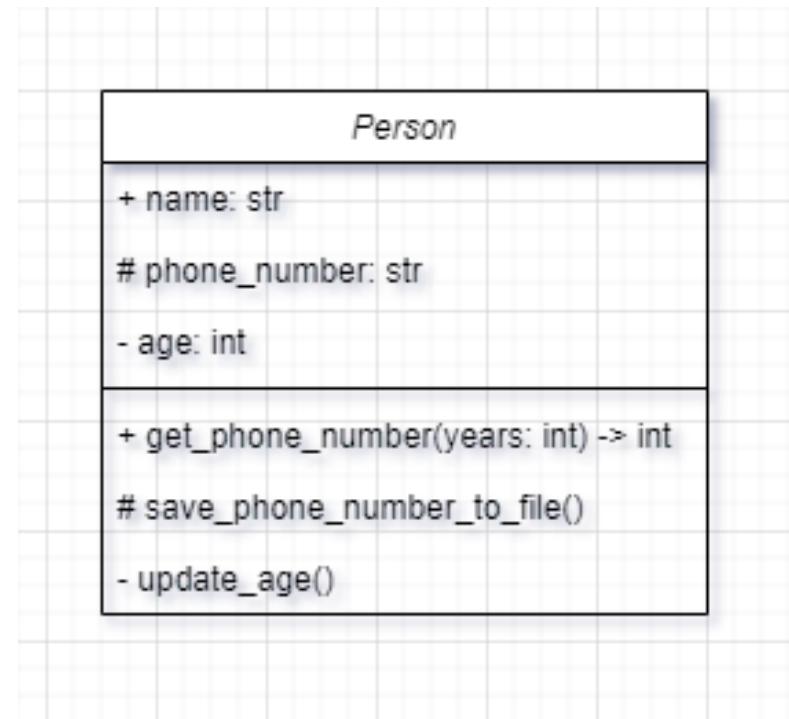
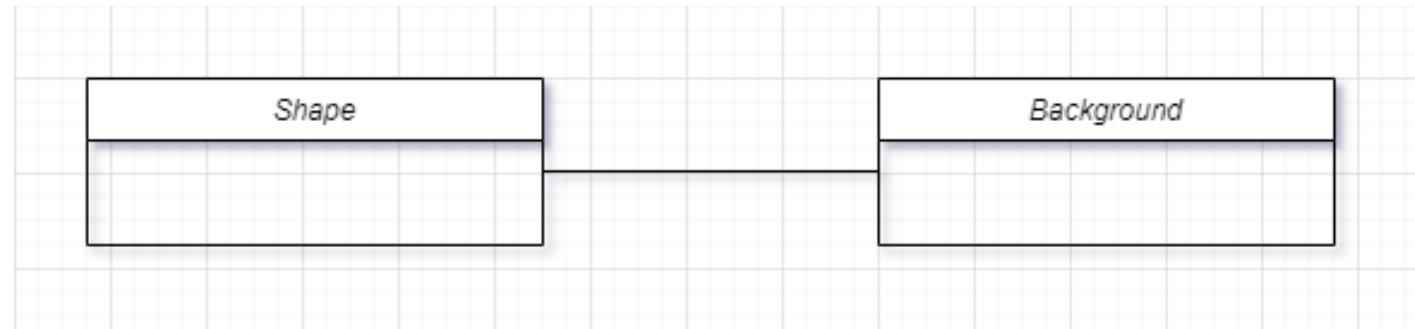


Diagram klas - relacje

Asocjacja *ang. association*



UML

Relacje

Uwaga!

Terminologia używana w modelowaniu obiektowym do opisu relacji pomiędzy klasami różni się miejscami od terminologii wprowadzonej w UML.

Terminologia wprowadzona w kolejnych slajdach jest charakterystyczna dla modelowania obiektowego.

Diagram klas - relacje

Dziedziczenie *ang. inheritance* (IS-A)

```
class Shape:  
    ...  
  
class Rectangle(Shape):  
    ...
```

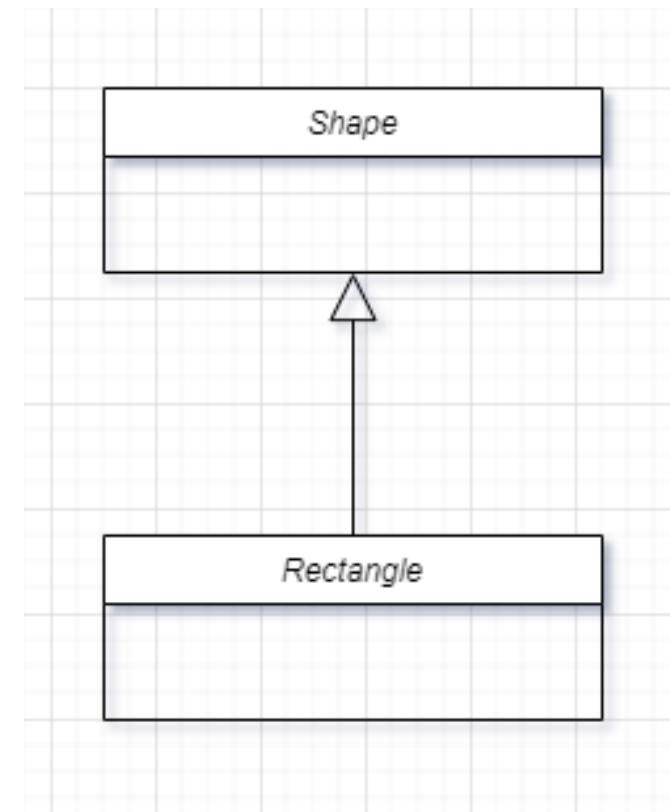


Diagram klas - relacje

Kompozycja *ang. composition* (OWNS-A)

```
class Size:  
    ...  
  
class Shape:  
    def __init__(self):  
        self.size = Size()
```

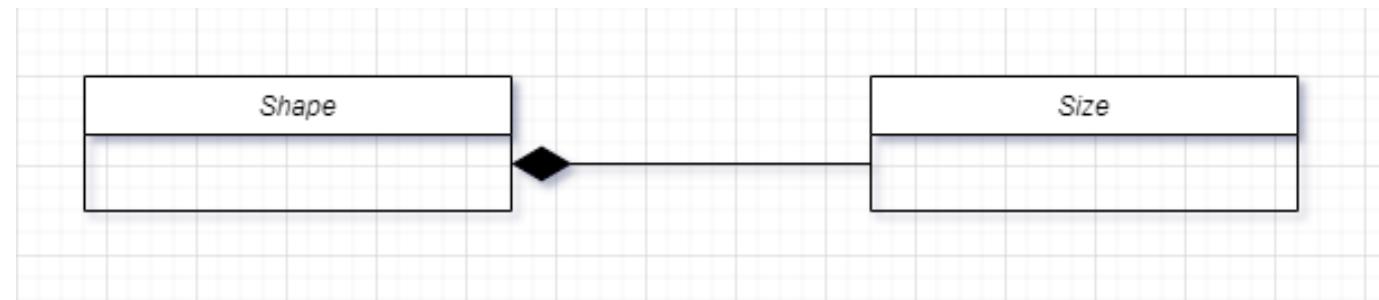


Diagram klas - relacje

Zależność ang. *dependency* (USE-A)

```
class Document:  
    ...  
  
class Shape:  
    def render(self, doc: Document):  
        ...  
  
document = Document()  
Shape().render(document)
```



UML

Relacje

Czasami w modelowaniu obiektowym wprowadza się jeszcze dwa typy relacji.

Diagram klas - relacje

Interfejs ang. *interface* (IMPLEMENT-A)

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def render(self):
        ...

class Rectangle(Shape):
    def render(self):
        print("Rectangle")
```

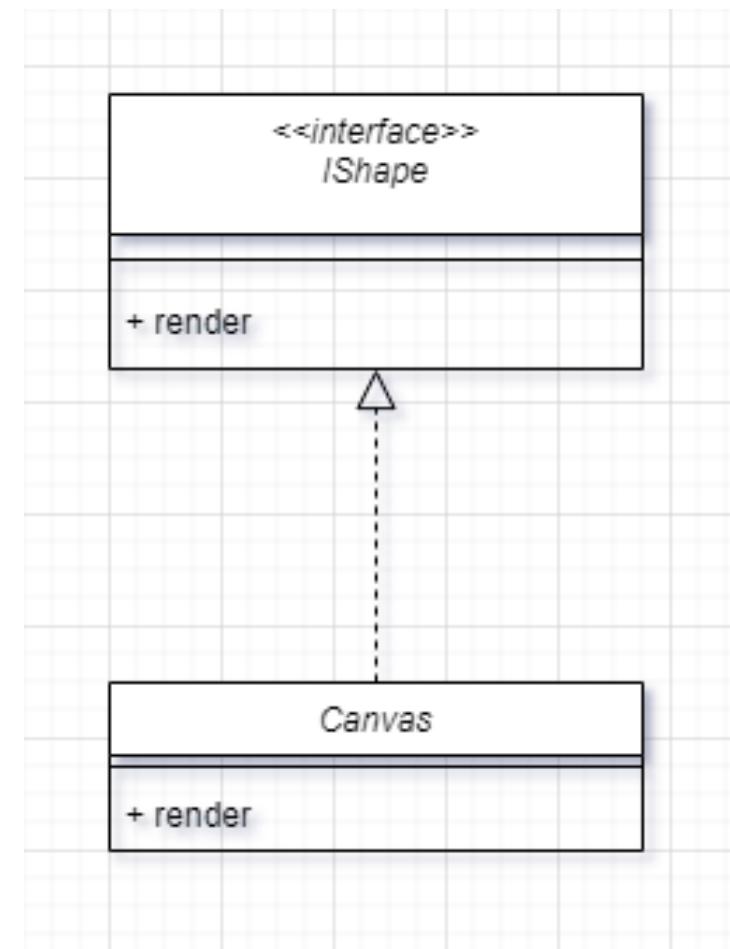
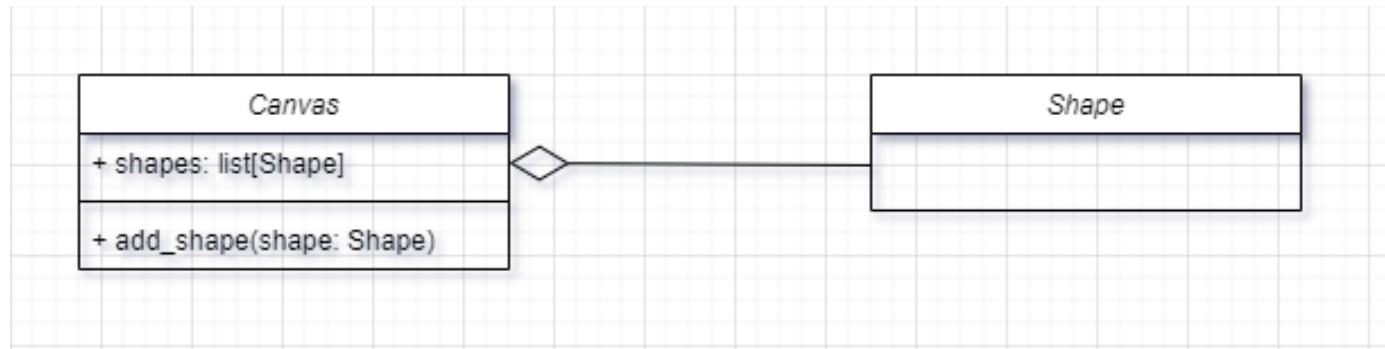


Diagram klas - relacje

Agregacja *ang. aggregation* (HAS-A)

```
class Shape:  
    ...  
  
class Canvas:  
    def __init__(self):  
        self.shapes: list[Shape] = []  
  
    def add_shape(self, shape: Shape):  
        self.shapes.append(shape)  
  
  
s = Shape()  
canvas = Canvas()  
canvas.add_shape(s)
```



OOP

Object Oriented Programming

OOP

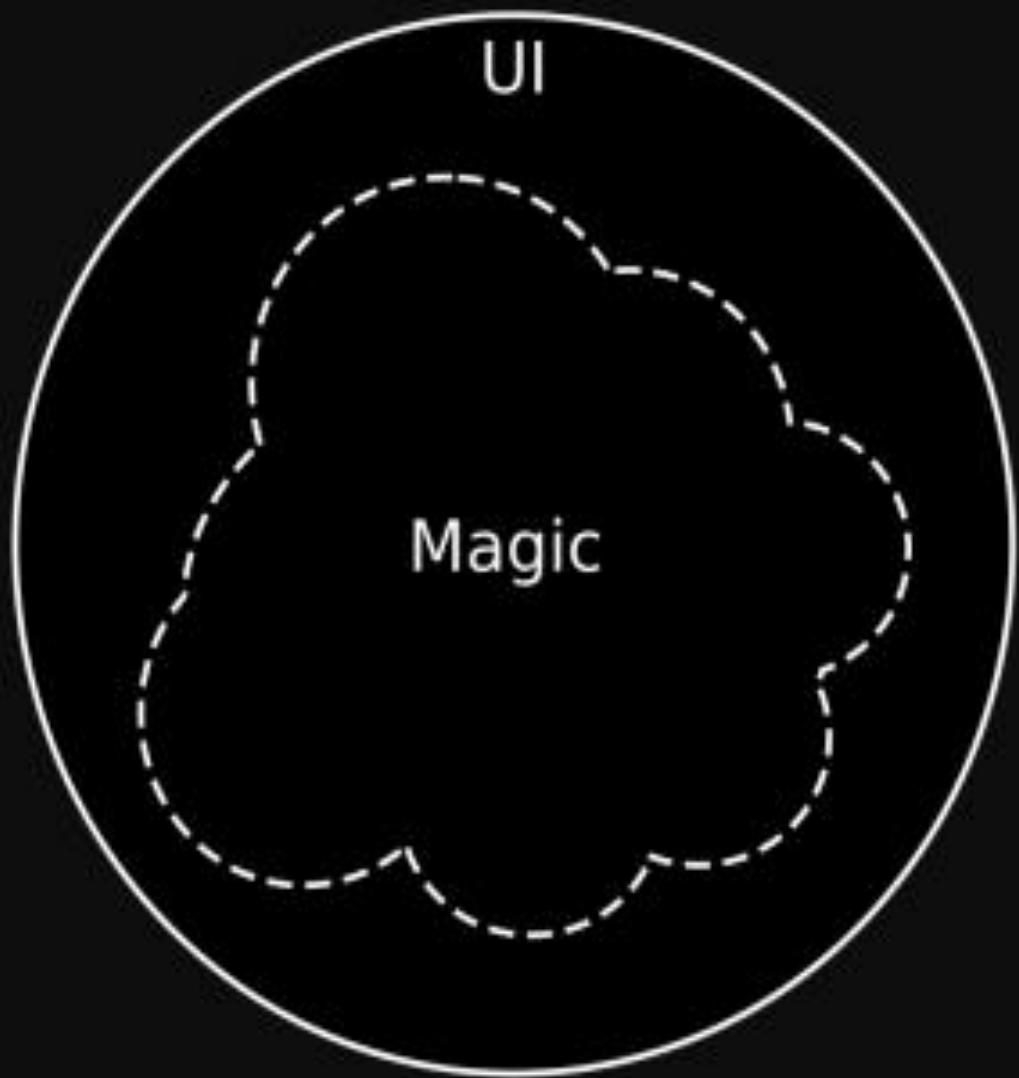
Programowanie zorientowane obiektowo

Implementacja klas projektowych w wybranym języku

Podstawowe zasady projektowania



What other see



What you see





Projektowanie - podstawy

- SoC (Seperation of Concern)
Separacja zagadnień
- KISS (Keep It Simple, Stupid)
- YAGNI (You Aren't Gonna Need It)
 - MoSCoW (Must have, Should have, Could have, Won't have)

Podstawowe zasady programowania



YAGNI

Do the NEEDED



KISS

Do it SIMPLE

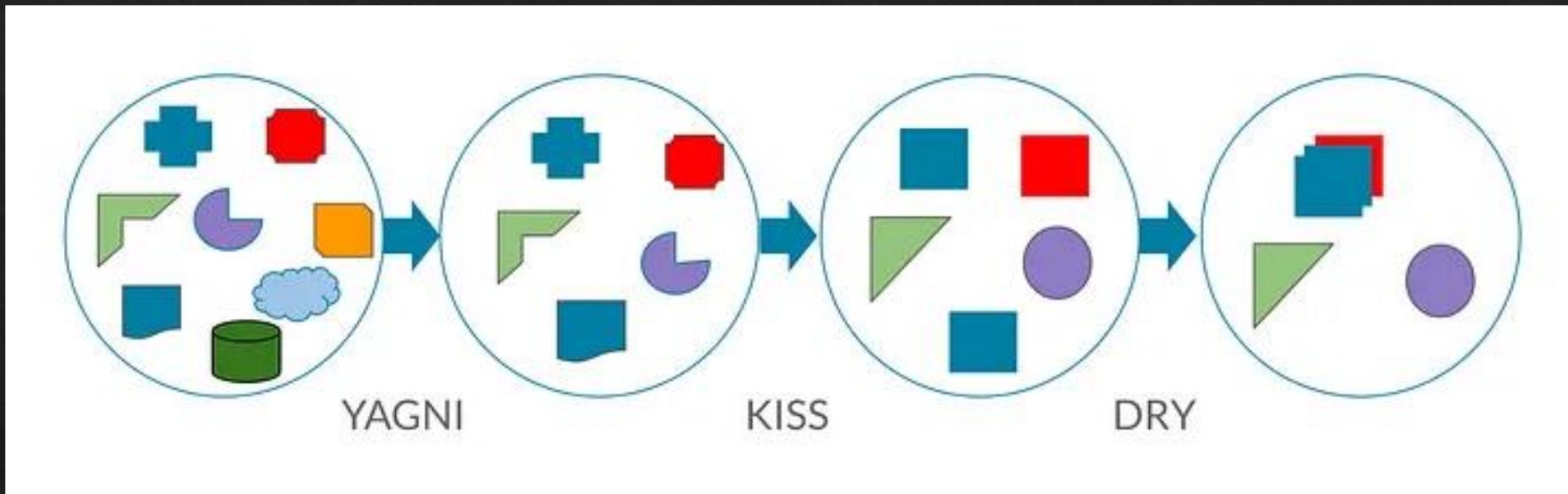


DRY

Do it ONCE



Suche pocałunki Jagny



Suche pocałunki Jagny

Programowanie - podstawy

- SoC (Separation of Concern)
Separacja zagadnień
- DRY (Don't Repeat Yourself)
 - RoT (Rule of Three)
- KISS (Keep It Simple, Stupid)
- YAGNI (You Aren't Gonna Need It)
 - DTSTTCPW (Do The Simplest Thing That Could Possibly Work)

GRASP

General Responsibility Assignment Software Patterns

Zasady GRASP

- **High cohesion**
- **Low couplings**
- Information expert
- Pure fabrication
- Creator
- Controller
- Indirection
- Polymorphism
- Protected variations

Cohesion

Spójność

Spójność to miara dotycząca wewnętrznej struktury **komponentu oprogramowania**, czymkolwiek ten komponent oprogramowania jest (pakietem, modułem, klasą, funkcją, ...).

Komponent o wysokiej spójności wykonuje jedno, precyzyjnie określone zadanie. **Komponent o niskiej spójności** to **komponent**, który realizuje wiele, niepowiązanych ze sobą zadań.

```
def handle_stuff(d: Data, quantity: int, screenX: int,
                  screenY: int, status: int, c: Color):

    update_corporate_database(d, q, status)
    for i in range(0, quantity):
        profit[i] = revenue[i] - expense[i] * status
    new_color = c
    status = SUCCESS
    display_profits(screenX, screenY, status, d, new_color)
```

Funkcja o niskiej spójności

Cohesion Spójność

Niska spójność świadczy o nieprawidłowym zaprojektowaniu **komponentu** i jego wysokiej złożoności. Taki komponent należy rozbić na kilka mniejszych komponentów. Po jednym **komponencie** na każde zadanie (odpowiedzialność).

W przypadku klas, spójność **klasy** oceniamy sprawdzając, czy metody **klasy** pracują na jednym i tym samym zbiorze atrybutów.

Cohesion

Spójność

Wyróżniamy 7 typów spójności:

- functional cohesion
- sequential cohesion
- communicational cohesion
- procedural cohesion
- temporal cohesion
- logical cohesion
- coincidental cohesion

Coupling Sprzężenia

Sprzężenie - miara stopnia zależności modułów od siebie nawzajem.

Coupling Sprzężenia

Wyróżniamy 5 typów sprzężeń:

- data coupling
- control coupling
- external coupling
- common coupling
- content coupling



```
def check_email_security(email):
    if email.header.bearer.invalid():
        raise Exception("Email header bearer is invalid")
    elif email.header.received != email.header.received_spf:
        raise Exception("Received mismatch")
    else:
        print("Email header is secure")
```

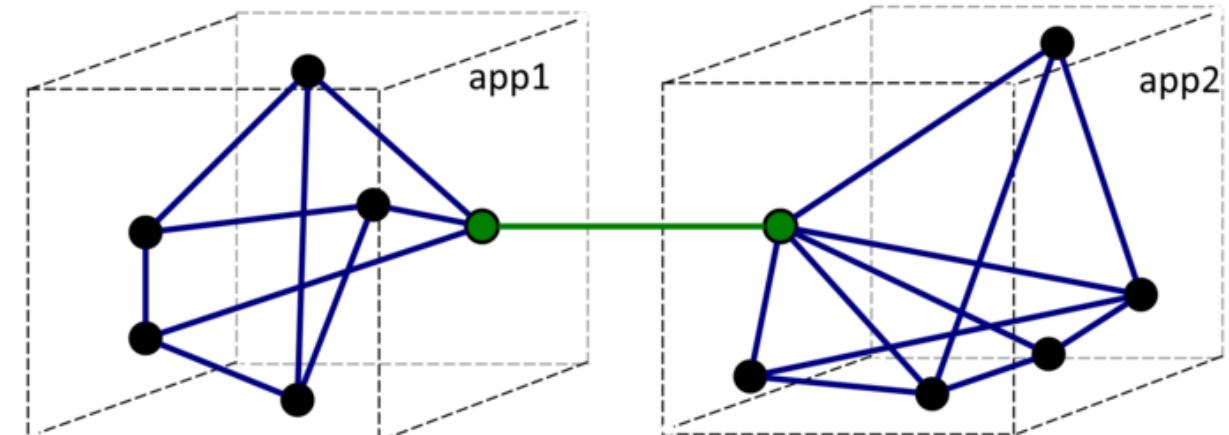
Funkcja o silnych sprzężeniach

*W kodzie staramy się utrzymać wysoką spójność i luźne
sprzężenia*

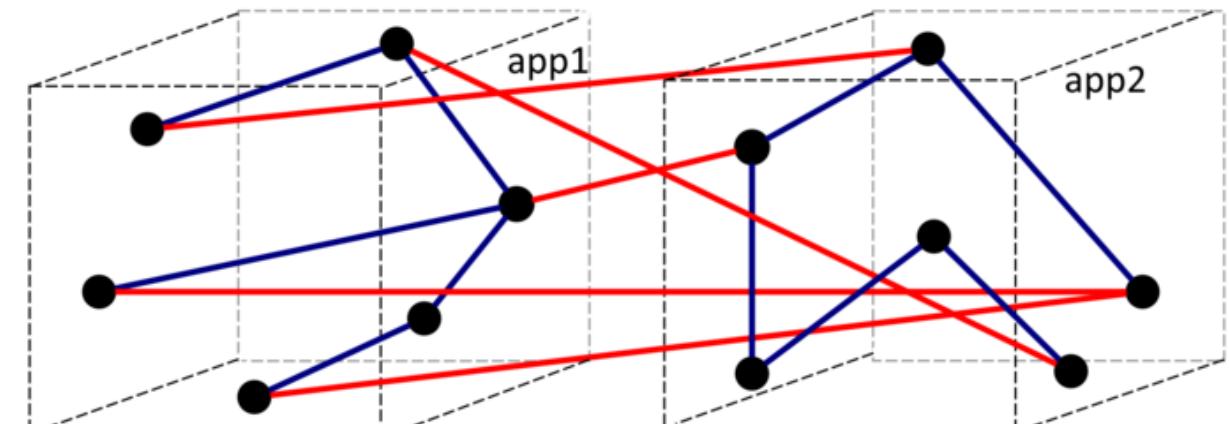
W kodzie staramy się utrzymać wysoką spójność i luźne sprzężenia

Klasy o wysokiej spójności jest znacznie łatwiej utrzymywać. Poza tym klasa, której jesteśmy w stanie nadać jasno określona odpowiedzialność jest łatwiejsza do zrozumienia.

Wysoka spójność
(high cohesion)



a) Good



b) Bad

Luźne sprzężenia
(loose coupling)

