

# Python: Wzorce projektowe

## Część 2

Jerzy Grynczewski

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)
- ❖ Interfejsy w Pythonie

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)
- ❖ Interfejsy w Pythonie
- ❖ Wzorzec - Strategia

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)
- ❖ Interfejsy w Pythonie
- ❖ Wzorzec – Strategia
- ❖ Wzorzec – Iterator

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)
- ❖ Interfejsy w Pythonie
- ❖ Wzorzec – Strategia
- ❖ Wzorzec – Iterator

Dodatkowo mówiliśmy o:

- ❖ Protokołach

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasyfikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)
- ❖ Interfejsy w Pythonie
- ❖ Wzorzec – Strategia
- ❖ Wzorzec – Iterator

Dodatkowo mówiliśmy o:

- ❖ Protokołach
- ❖ Enkapsulacji i property

# Co zrobiliśmy wczoraj

- ❖ Co to są wzorce projektowe ?
- ❖ Czemu ich potrzebujemy ?
- ❖ Klasifikacja wzorców projektowych.
- ❖ Zasady programowania obiektowego (SOLID)
- ❖ Interfejsy w Pythonie
- ❖ Wzorzec – Strategia
- ❖ Wzorzec – Iterator

Dodatkowo mówiliśmy o:

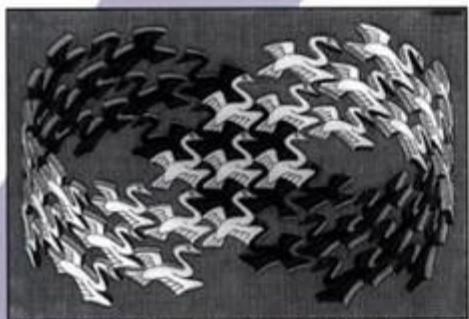
- ❖ Protokołach
- ❖ Enkapsulacji i property
- ❖ Diagramie UML

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

# Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Pierwszy raz opublikowana w 1995

Gamma, Helm, Johnson, Vlissides – Gang of Four

Pierwsza kompletna publikacja dotycząca wzorców projektowych.

Do tej pory pozostaje niepodważalnym autorytetem.

# Podstawowa klasyfikacja wzorców projektowych

## Kreacyjne

Tworzenie obiektów

## Strukturalne

Kompozycja obiektów

## Behawioralne (Czynnościowe)

Interakcja pomiędzy obiektami i  
odpowiedzialność

# Alternatywna klasyfikacja wzorców projektowych

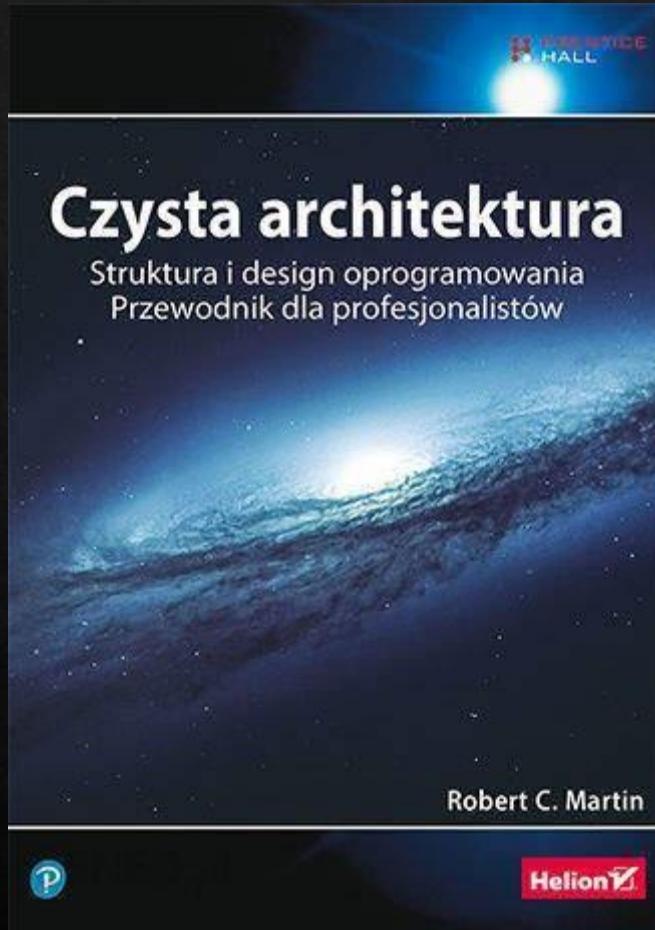
## Klasowe

Opisujące statyczne związki  
pomiędzy klasami

## Obiektowe

Opisujące dynamiczne związki  
pomiędzy obiektami

# SOLID Principles of Object Oriented Design



Robert C. Martin (wujek Bob)

# S – Single Responsibility Principle



Każdy moduł powinien mieć jedną i tylko jedną przyczynę zmiany.

Klasa powinna mieć tylko jedną odpowiedzialność, czyli albo gotowanie, albo zmywanie, ale nigdy to i to.

# O - Open Close Principle



Element oprogramowania powinien być otwarty na rozbudowę, a zamknięty na modyfikacje.

Klasa powinna być otwarta na rozbudowę (przeważnie poprzez zastosowanie dziedziczenia), ale zamknięta na modyfikacje.

# L – Liskov Substitution Principle



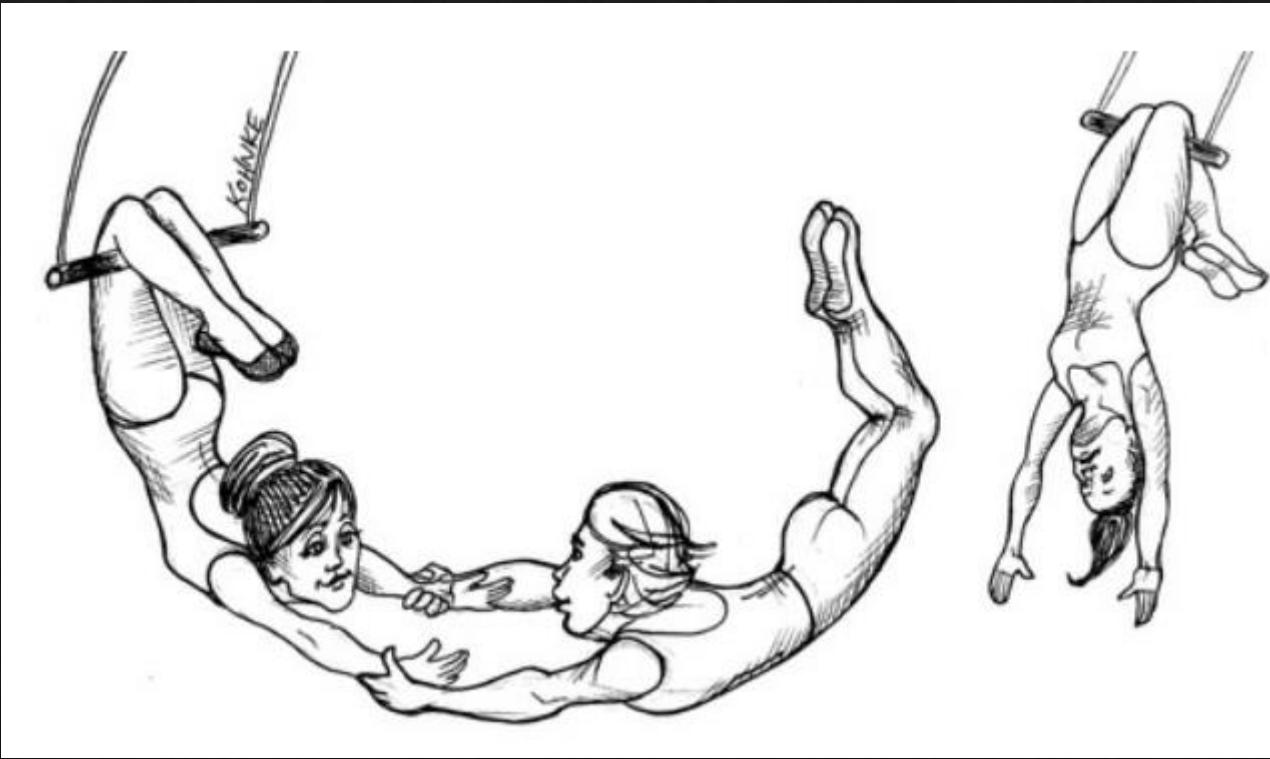
Podklasy powinny być w stanie zastąpić swoich rodziców bez wywoływania błędu w programie.

# I – Interface Segregation Principle



Wiele różnych interfejsów jest lepsze niż jeden interfejs typu do-it-all

# D – Dependency Inversion Principle



Powinniśmy programować za pomocą abstrakcji, nie implementacji. Implementacje mogą się zmieniać, abstrakcje nie powinny

# SOLID Principles of Object Oriented Design

**S**

Reguła jednej odpowiedzialności

SRP (Single Responsibility Principle)

**O**

Reguła otwarte-zamknięte

OCP (Open-Close Principle)

**L**

Zasada podstawień Barbary Liskov

LSP (Liskov Substitution Principle)

**I**

Zasada rozdzielenia interfejsów

ISP (Interface Segregation Principle)

**D**

Zasada odwrócenia zależności

DIP (Dependency Inversion Principle)

# 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

Wprowadzone  
przez PEP 3119  
za pomocą klas  
abstrakcyjnych

Pierwszy raz  
pojawiły się w  
Pythonie 2.6 i 3

# Definiowanie abstrakcyjnych klas bazowych

Moduł abc

```
import abc
```

Klasa abstrakcyjna

```
class MyABC(metaclass=abc.ABCMeta):  
    """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"""
```

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 ?

TypeError: Can't instantiate abstract class BadClass with abstract methods  
do\_something, some\_property

# Wzorce projektowe

# Wzorce projektowe

## List of Design Patterns - 23

---

- Creational Patterns
  - Singleton
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

# 1. Strategia (Strategy)



Complexity: ★★★

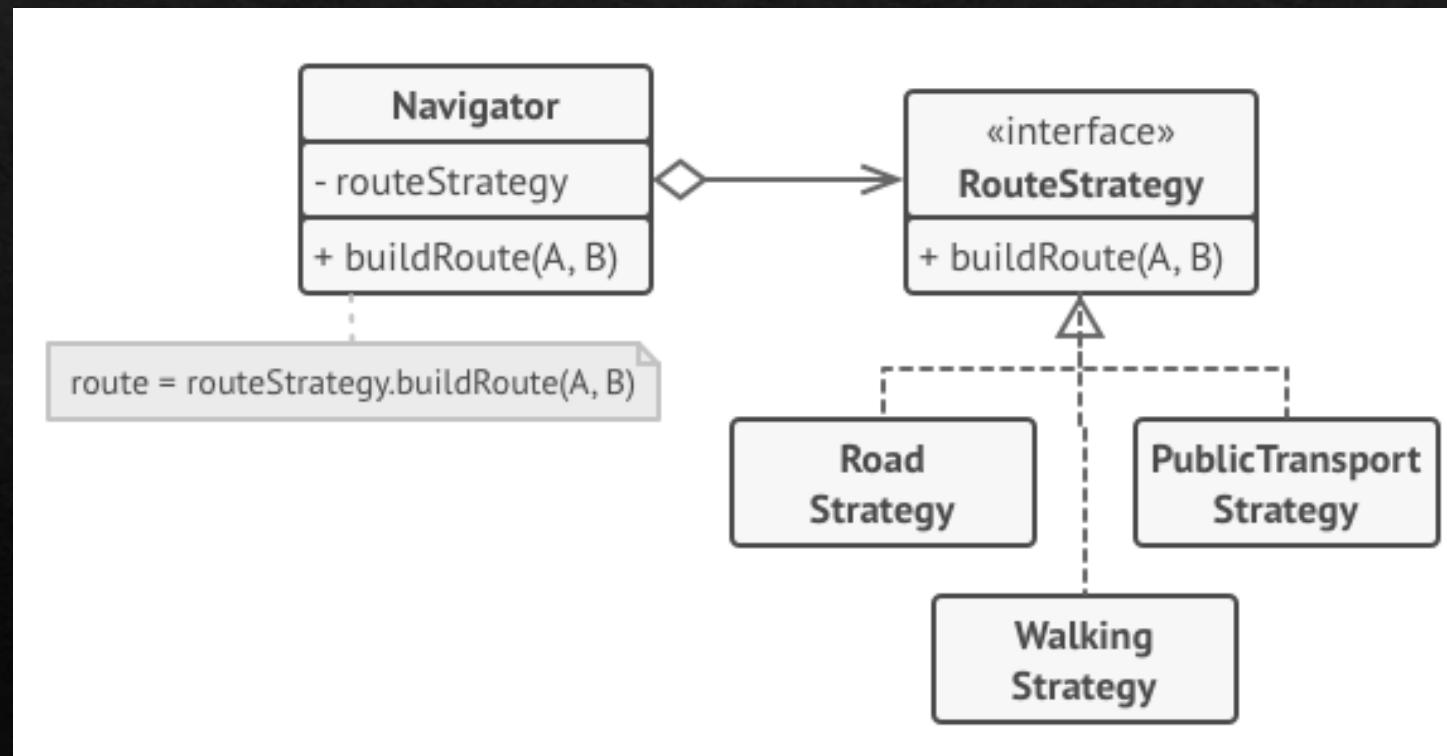
Popularity: ★★★★

Czynnościowy wzorzec projektowy, który zamienia zbiór czynności (rodzinę wymiennych algorytmów) w obiekty i czynni je zmiennymi wewnętrznymi korzystającego z nich obiektu.

# 1. Strategia (Strategy)

Rozwiązanie:

Strategia definiuje rodzinę algorytmów (tutaj wytyczania trasy), kapsułkuje każdy z nich w oddzielnej klasie i zapewnienie wymienność powstałych obiektów poprzez zastosowanie interfejsu.



# 1. Strategia (Strategy)

**Przykład praktyczny - Kalkulator kosztów przesyłki**

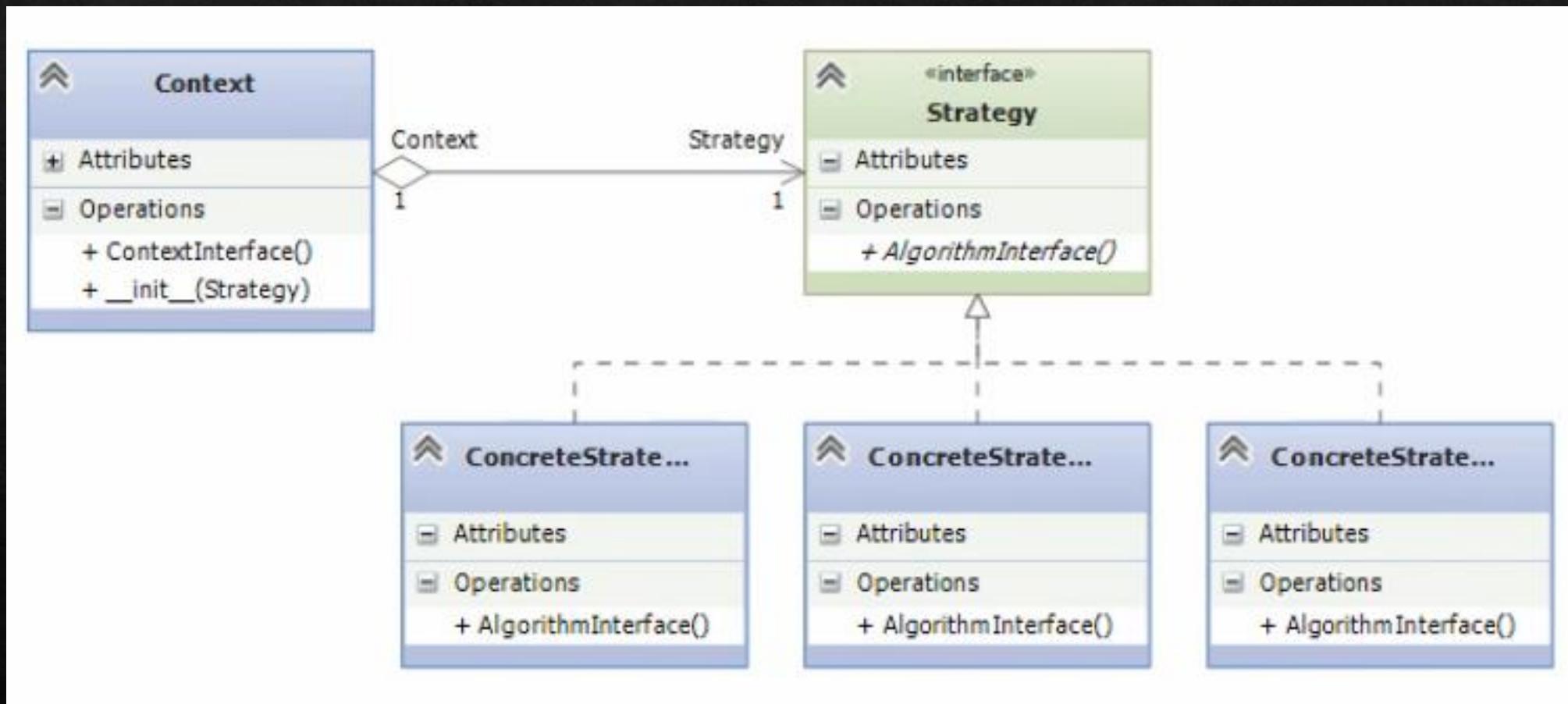
Specyfikacja.

Musi uwzględniać:

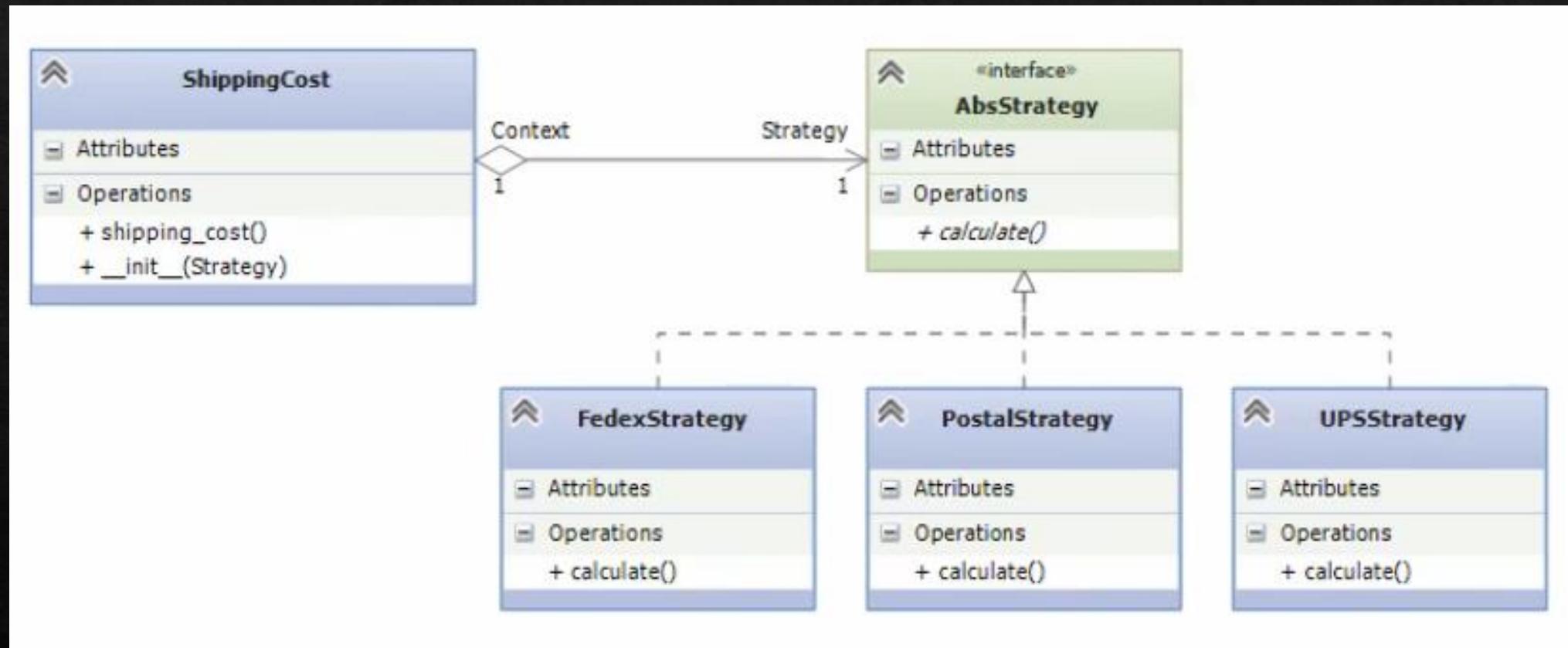
- Federal Express (FedEx)
- UPS
- Poczta Polska

Powinien zapewniać łatwą rozbudowę o nowe firmy kurierskie.

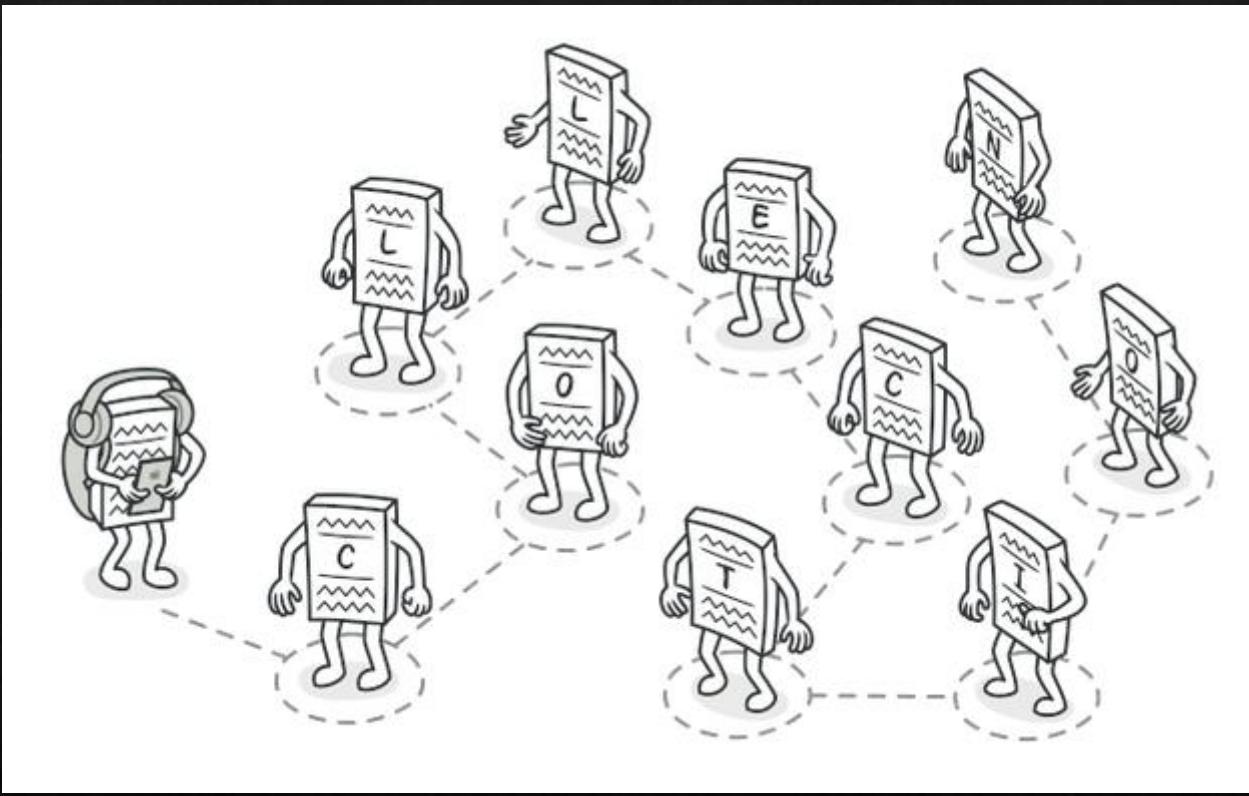
# Struktura wzorca Strategia (Strategy)



# Struktura wzorca Strategia (Strategy)

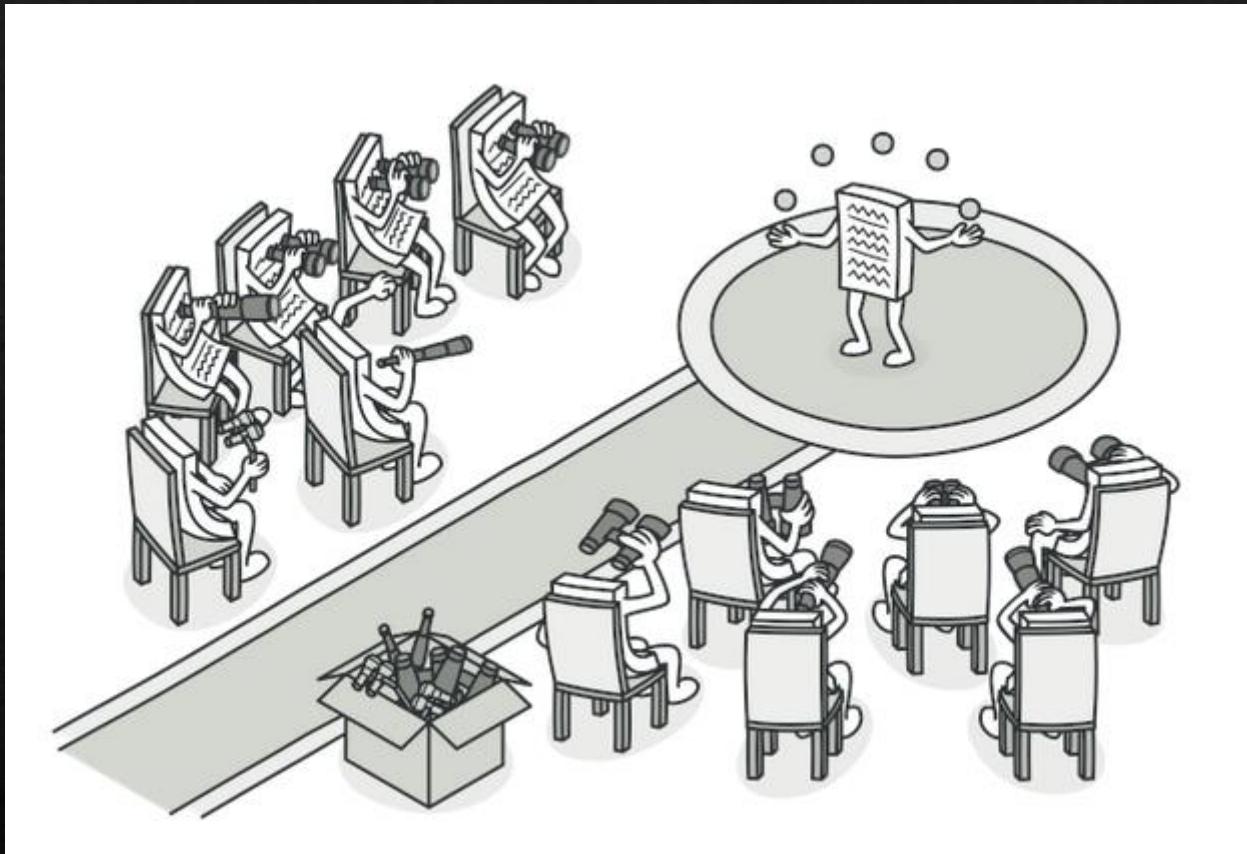


## 2. Iterator



Czynnościowy wzorzec projektowy, który pozwala na iterowanie po kolekcji bez konieczności eksponowania jej reprezentacji (lista, stos, drzewo, etc.)

### 3. Obserwator (Observer pattern aka Dependents pattern, Publish-Subscribe pattern)

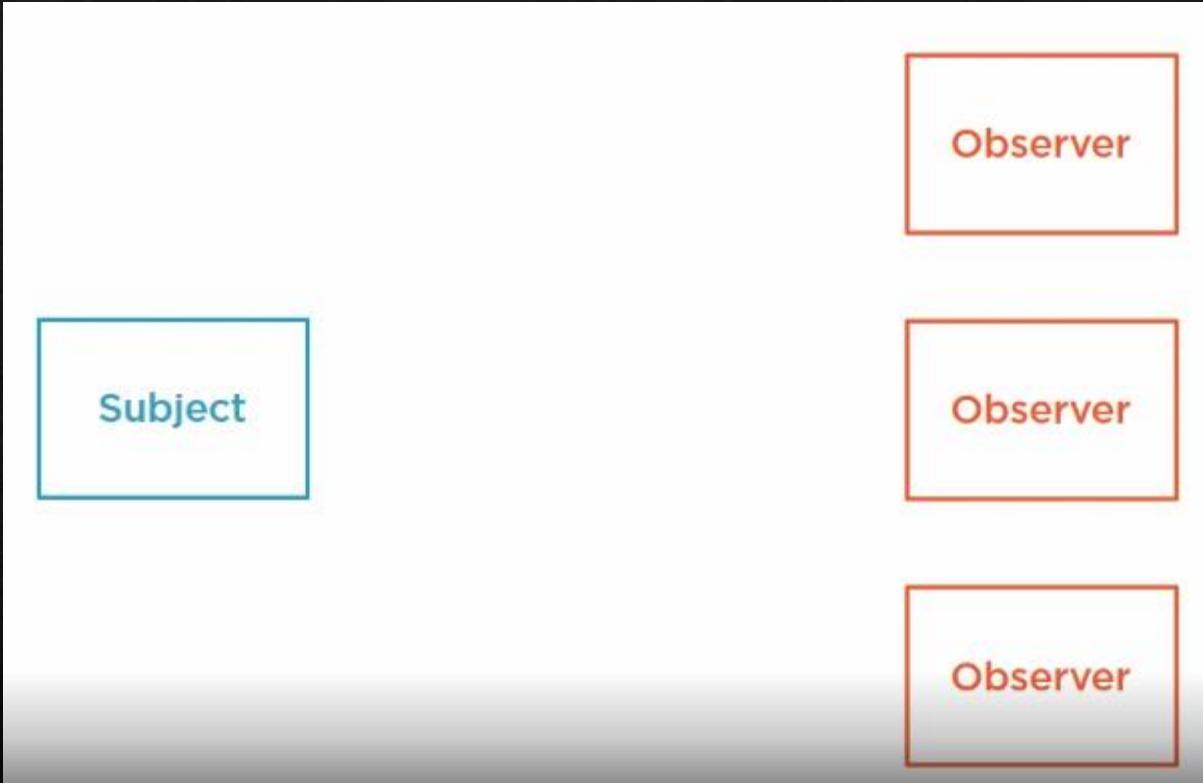


Czynnościowy wzorzec projektowy używany do powiadamiania zainteresowane obiekty o zmianie stanu pewnego innego obiektu

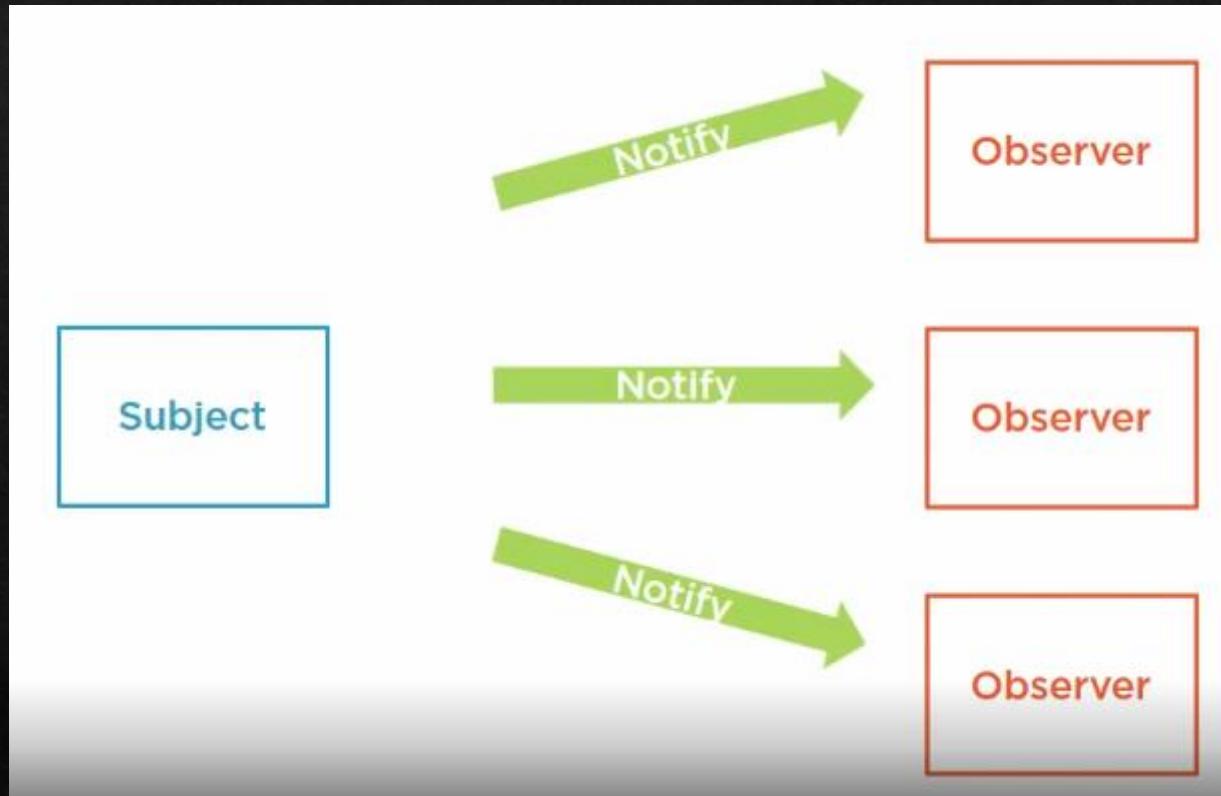
# Obserwator

Subject

# Obserwator



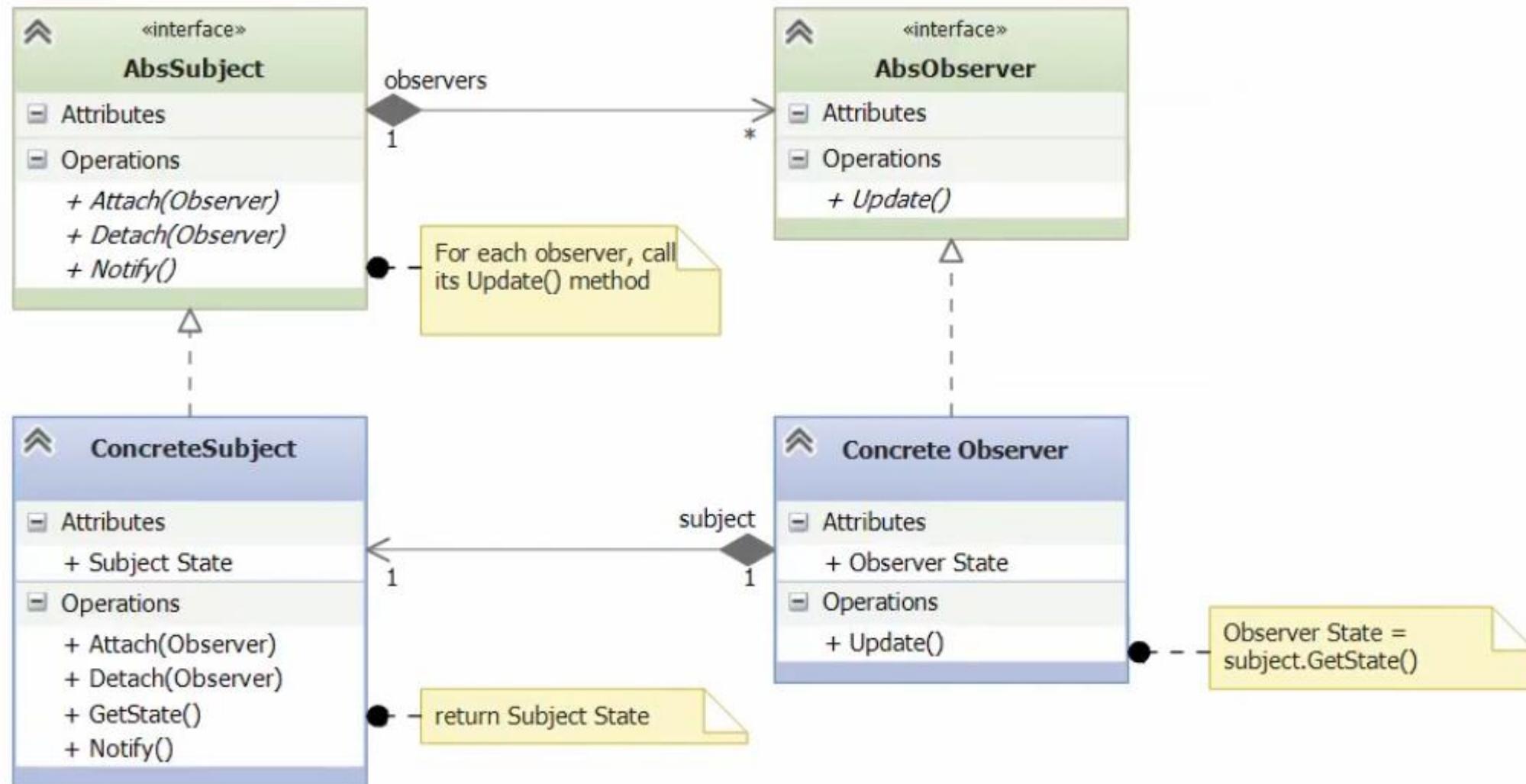
# Obserwator



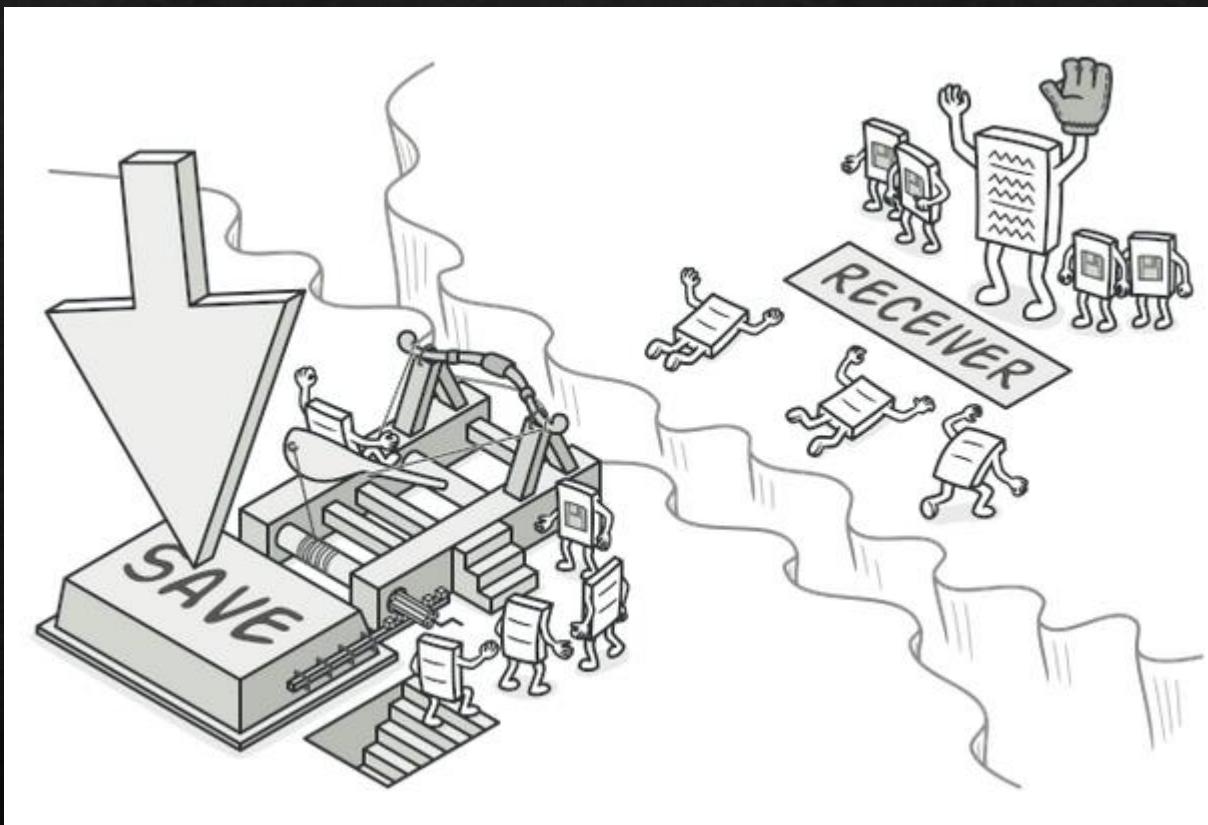
# Obserwator



# Obserwator



## 4. Polecenie (Command)



Czynnościowy wzorzec projektowy, traktujący żądanie wykonania określonej czynności jako obiekt, dzięki czemu mogą być one parametryzowane w zależności od rodzaju odbiorcy, a także umieszczane w kolejkach i dziennikach.

# 1. Polecenie (Command)

Napotkane problemy

Złamanie reguły pojedynczej odpowiedzialności (S)

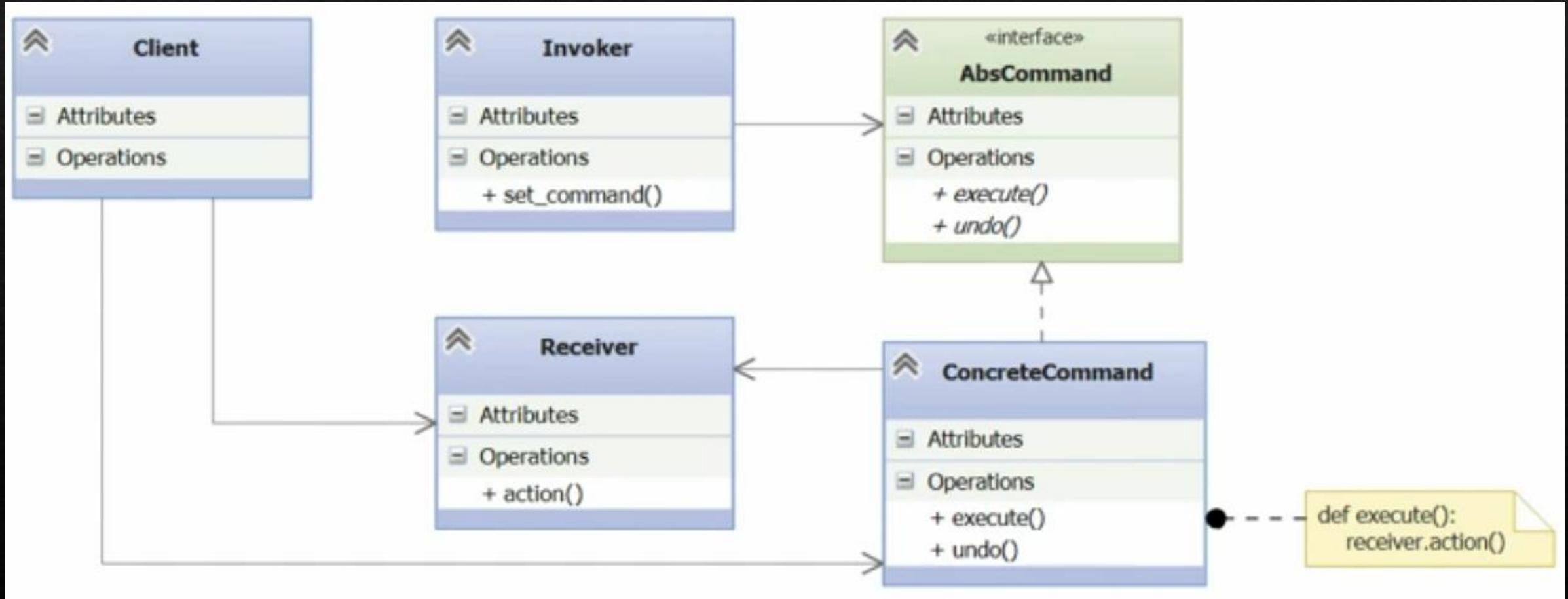
Złamanie reguły otwarte/zamknięte (O)

Złamanie reguły odwróconych zależności (D)

Długa lista if/elif/else

Naprawmy je

# Polecenie



# 1. Polecenie (Command)

Co zyskaliśmy

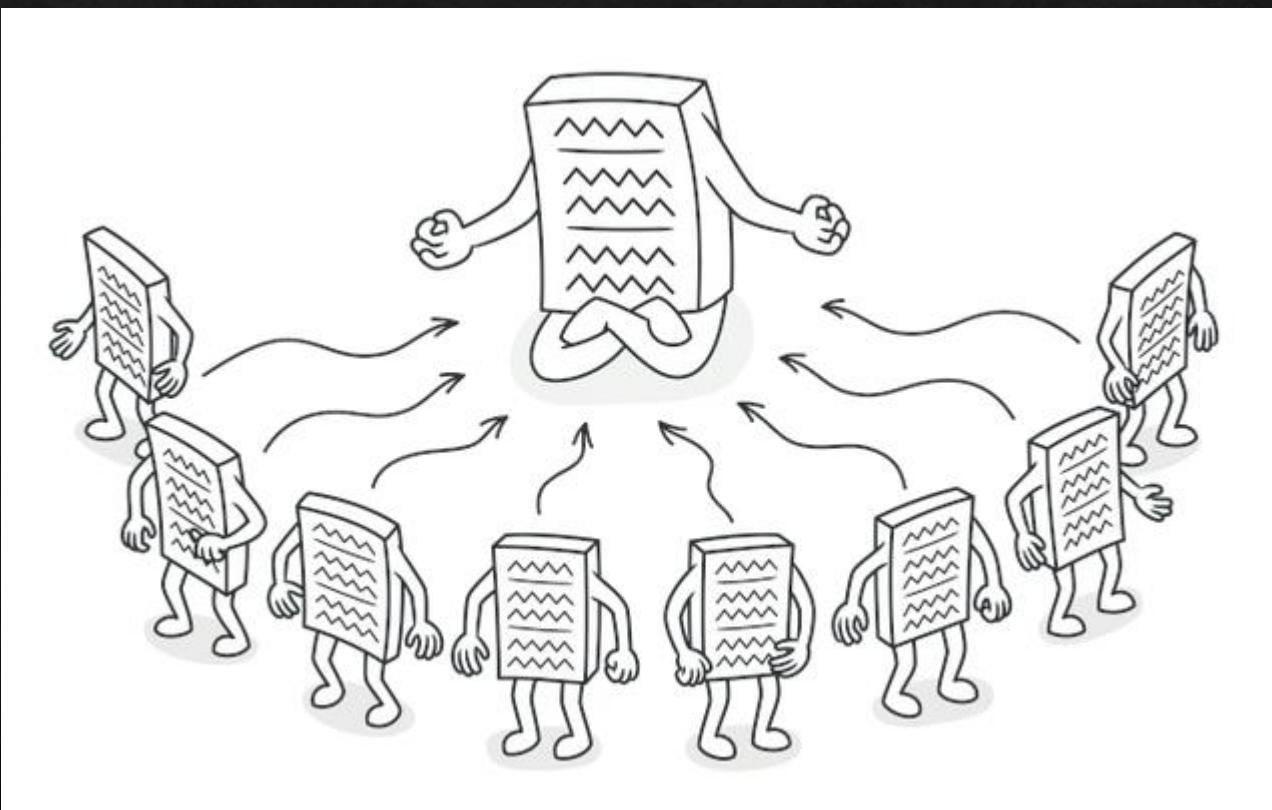
Enkapsulacje (separacje logiki polecenia od klienta)

Łatwość rozbudowy (validation, undo)

SOLID

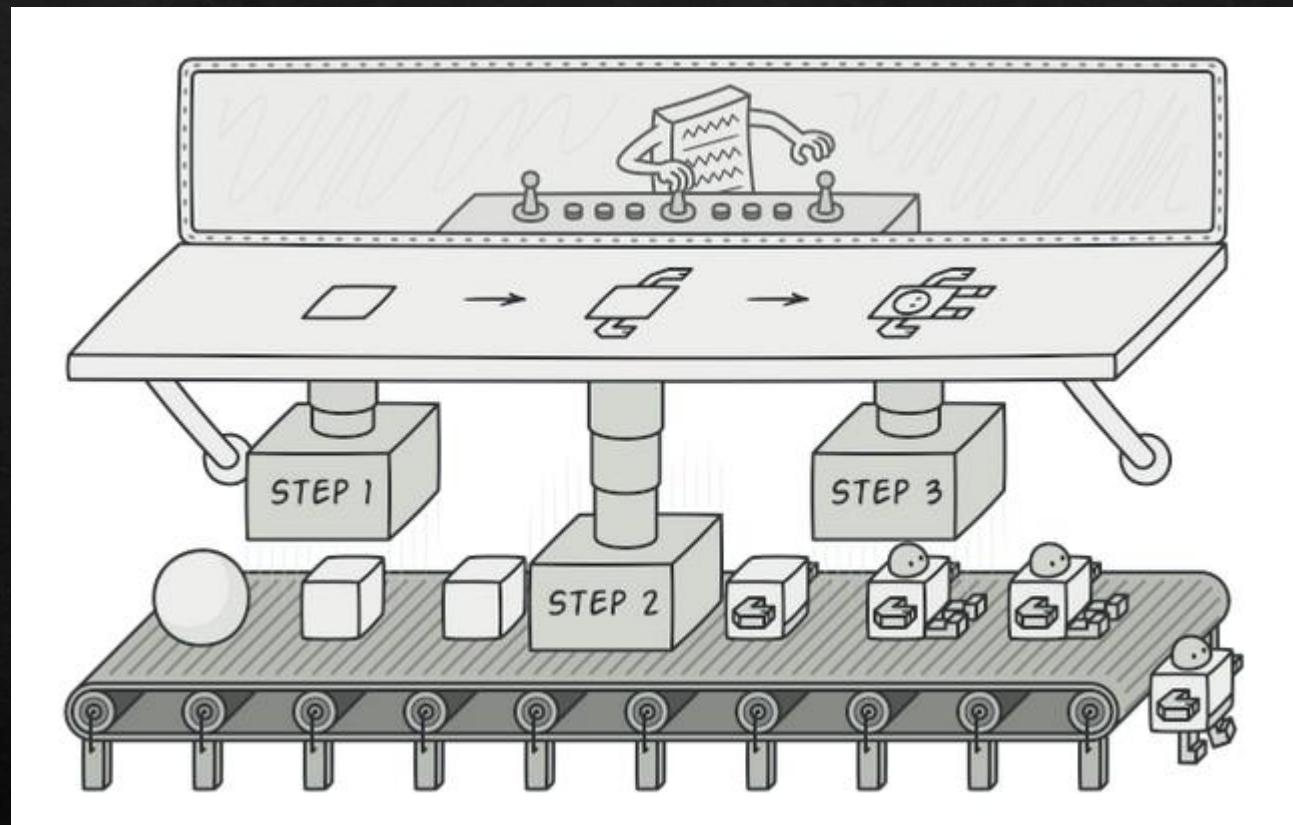
Pozbyliśmy się długiej listy if/elif/else

## 5. Singleton



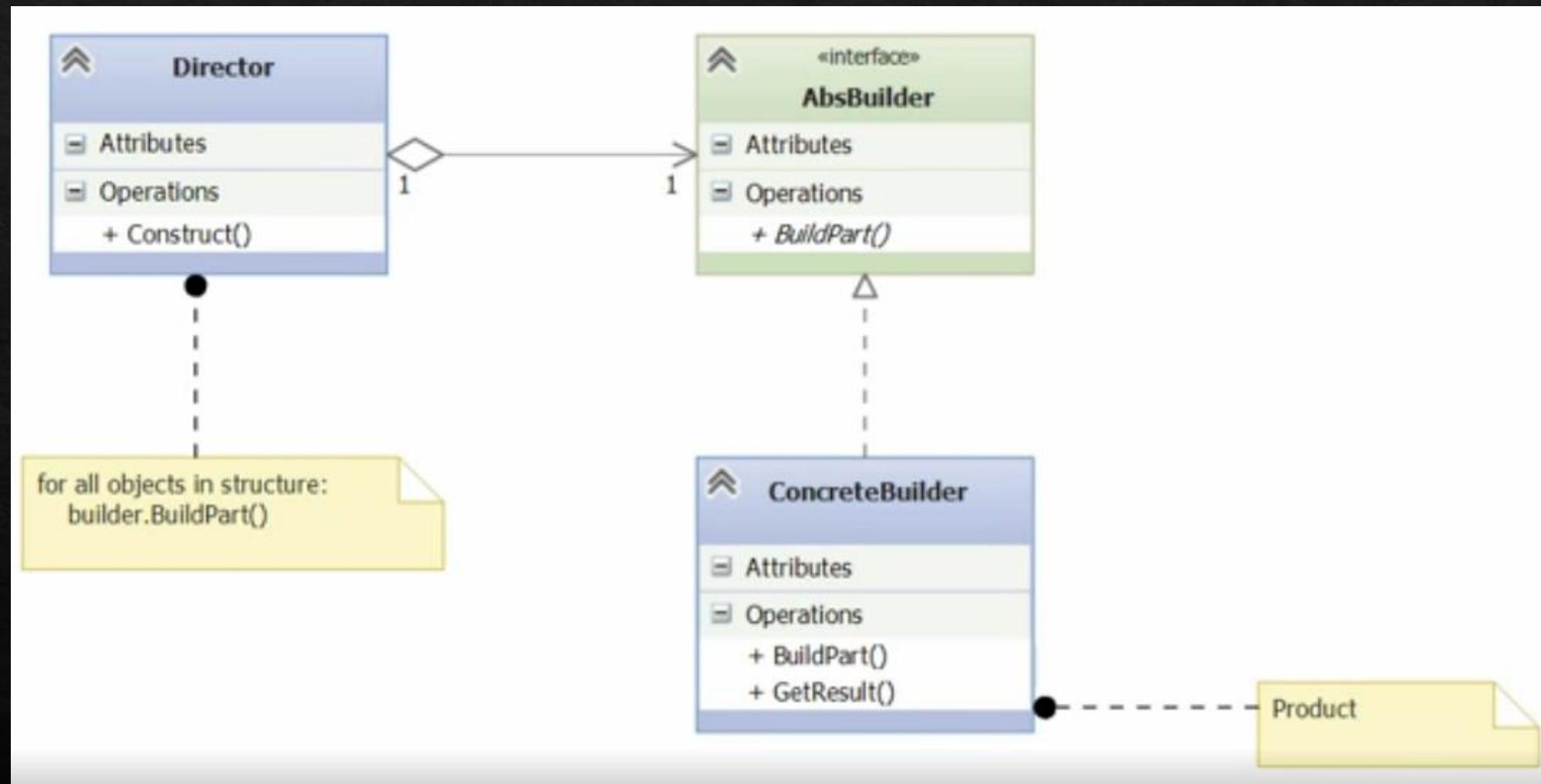
Kreacyjny wzorzec projektowy, którego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji oraz zapewnienie globalnego dostępu do stworzonego obiektu

## 6. Budowniczy (Builder)

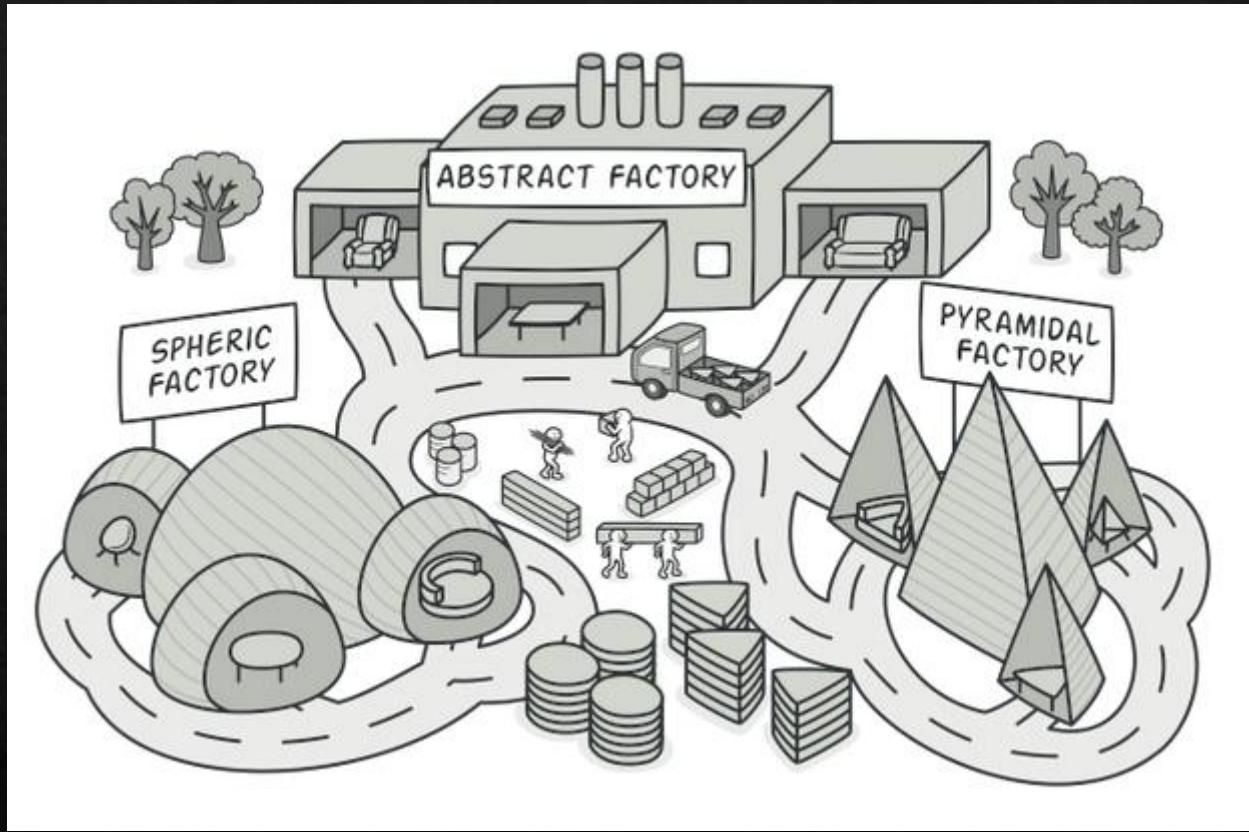


Kreacyjny wzorzec projektowy, którego celem jest rozdzielenie sposobu tworzenia obiektów od ich reprezentacji.

# Struktura wzorca Budowniczy (Builder)



## 7. Abstrakcyjna fabryka (Abstract Factory)



Kreacyjny wzorzec projektowy, którego celem jest dostarczenie interfejsu do tworzenia różnych obiektów jednego typu (tej samej rodziny) bez specyfikowania ich konkretnych klas.