

# Programowanie i projektowanie obiektowe

## Powiązania i tworzenie obiektów

Paweł Daniluk

Wydział Fizyki

Jesień 2016



# Powiązania

## Jeden do jeden

Przez atrybut.

## Jeden do wielu

Przez atrybut po stronie “wielu”, lub listę/zbiór po stronie “jeden”.

## Wiele do wielu

Podobnie, jak w przypadku reprezentacji grafów.

Powiązania są skierowane. Musi być możliwość dojścia do każdego obiektu w systemie.

## Jeden do jednego

Można zastosować atrybuty w obydwu powiązanych klasach, ale wtedy trzeba dbać o ich aktualizację. Kapsułkowanie pomaga.

### Przykład

```
class Man:
    ...

    def set_wife(self, wife):
        self.wife = wife
        wife.set_husband(self)

class Woman:
    ...

    def set_husband(self, husband):
        self.husband = husband
        husband.set_wife(self)
```

## Jeden do jednego

Można zastosować atrybuty w obydwu powiązanych klasach, ale wtedy trzeba dbać o ich aktualizację. Kapsułkowanie pomaga.

### Przykład “poprawniej”

```
class Man:
    ...

    def set_wife(self, wife):
        if(self.wife == wife):
            return

        if self.wife is not None:
            self.wife.set_husband(None)

        self.wife = wife

        if wife is not None:
            wife.set_husband(self)
```

## Jeden do jednego

Można zastosować atrybuty w obydwu powiązanych klasach, ale wtedy trzeba dbać o ich aktualizację. Kapsułkowanie pomaga.

### Przykład poprawnie

```
class Man:
    ...
    def set_wife(self, wife):
        if (self.wife == wife):
            return

        oldwife=self.wife
        if oldwife is not None:
            self.wife = None
            oldwife.set_husband(None)

        self.wife = wife
        if wife is not None:
            wife.set_husband(self)
```

## Jeden do wielu

W Pythonie naturalne jest zastosowanie list.

```
class Mother:
    def __init__(self):
        self.children = []

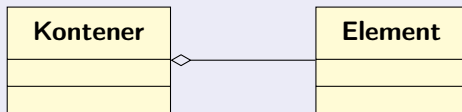
    def add_child(self, child):
        self.children.append(child)

    def remove_child(self, child):
        self.children.remove(child)
```

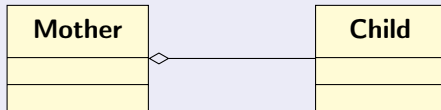
# Kontenery

Kontener to obiekt, który zawiera inne obiekty.

Obiekty typów sekwencyjnych w Pythonie są kontenerami.

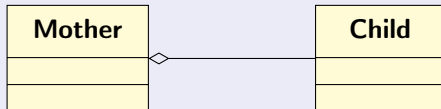


## Kontenery c.d.

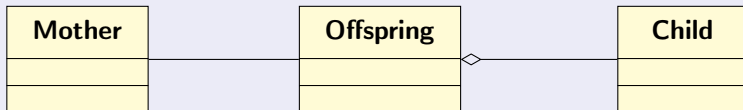




## Kontenery c.d.

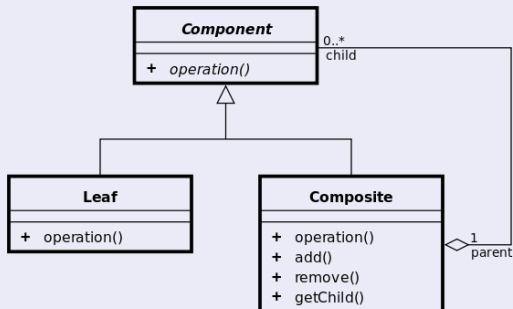


Kontener może być klasą pomocniczą w związku “jeden do jednego”.



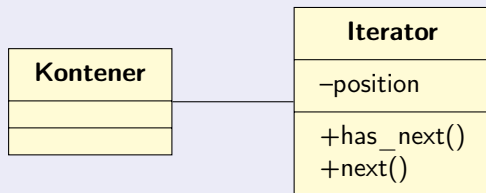
# Composite pattern

Wzorzec pozwalający na realizację hierarchicznej struktury obiektów.



# Iteratory

Iterator pozwala przeglądać obiekty znajdujące się w kontenerze.



# Iteratory w Pythonie

Python pozwala definiować specjalne metody, które pozwalają na używanie iterowalnych obiektów np. w instrukcji `for`.

```
class LiczbyIter:  
    def __init__(self, od, do):  
        self.pos = od  
        self.do = do  
  
    def next(self):  
        if self.pos <= self.do:  
            self.pos += 1  
            return self.pos - 1  
        else:  
            raise StopIteration  
  
    def __iter__(self):  
        return self
```

```
class Liczby:  
    def __init__(self, od, do):  
        self.od = od  
        self.do = do  
  
    def __iter__(self):  
        return LiczbyIter(self.od, self.do)
```

## Iteratory w Pythonie c.d.

```
>>> list(LiczbyIter(1,3))
[1, 2, 3]
>>> list(Liczby(1,3))
[1, 2, 3]
>>> l = Liczby(1,3)
>>> for i in l:
...     print i
...
1
2
3
```

Obiekt iterowalny może zostać przekazany do konstruktora typu sekwencyjnego.

## Iteratory w Pythonie c.d.

```
>>> for i in l:  
...     for j in l:  
...         print i,j  
...  
1 1  
1 2  
1 3  
2 1  
2 2  
2 3  
3 1  
3 2  
3 3
```

Pętle for mogą być zagnieżdżane. W każdej tworzony jest nowy iterator.

## Iteratory w Pythonie c.d.

```
>>> li = l.__iter__()  
>>> for i in li:  
...     print i  
...  
1  
2  
3  
>>> for i in li:  
...     print i  
...  
>>> li = l.__iter__()  
>>> for i in li:  
...     for j in li:  
...         print i,j  
...  
1 2  
1 3  
>>>
```

Iterowanie po iteratorze go “zużywa”.

# Konstruktory

Standardowo obiekty tworzymy wywołując konstruktor.

```
>>> Lew('Ziutek')  
<__main__.Lew instance at 0x10edfec20>
```



# Konstruktory

Standardowo obiekty tworzymy wywołując konstruktor.

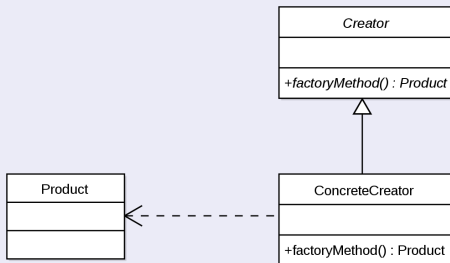
```
>>> Lew('Ziutek')  
<__main__.Lew instance at 0x10edfec20>
```

Nie zawsze takie podejście jest wystarczająco elastyczne. Na przykład jeżeli:

- 1 są różne algorytmy tworzenia/inicjalizacji obiektu,
- 2 nie wiadomo jakiej klasy obiekt chcemy stworzyć,
- 3 chcemy parametry inicjalizacji podawać na raty (np. wczytując je z pliku),
- 4 chcemy obiekty ewidencjonować.

# Factory method

Tworzy się klasę pomocniczą (Factory) odpowiedzialną za tworzenie obiektów. Może być wiele metod tworzących obiekty danej klasy.



## Factory method c.d.

```
class MazeGame:
    def __init__(self):
        room1 = self.make_room()
        room2 = self.make_room()
        room1.connect(room2)
        self.add_room(room1)
        self.add_room(room2)

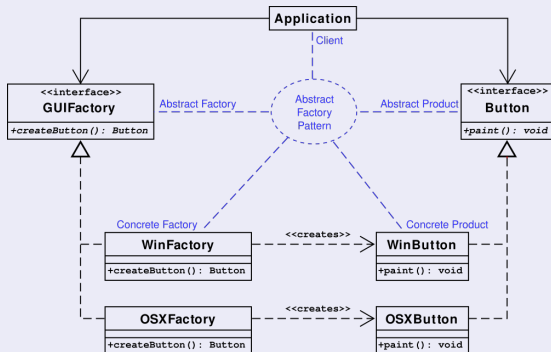
    def make_room(self):
        return OrdinaryRoom()

class MagicMazeGame(MazeGame):
    def make_room(self):
        return MagicRoom()
```

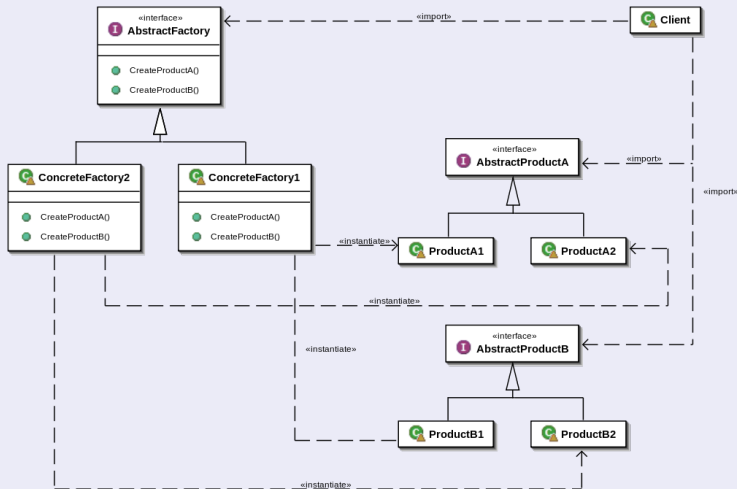
Metoda `make_room` tworzy pokój w labiryncie odpowiadający rodzajowi labiryntu.

# Abstract factory

Czasami zachodzi konieczność stosowania różnych fabryk w zależności od kontekstu. Wzorec *Abstract factory* jest uogólnieniem wzorca *Factory method*.



# Abstract factory



# Builder

Nie zawsze łatwo jest zdefiniować konstruktor/inicjalizator/metodę tworzącą obiekty ze względu na złożoność i różnorodność argumentów. W takiej sytuacji opłaca się tworzyć obiekt “na raty”.

```
class Car:
# Can have GPS, trip computer and a various number of seaters.
# Can be a city car, a sport car or a cabriolet.

class CarBuilder:
    def getResult(self):
#         output: a Car with the right options
#         Construct and return the car.

    def setSeats(self, number):
#         input: the number of seats the car may have.
#         Tell the builder the number of seats.

    def setCityCar(self):
        ...

    def setCabriolet():
        ...

    def setSportCar():
        ...

    def setTripComputer():
        ...
```

## Builder c.d.

### Użycie

```
carBuilder = CarBuilder()  
carBuilder.setSeaters(2)  
carBuilder.setSportCar()  
carBuilder.setTripComputer()  
carBuilder.unsetGPS()  
car = carBuilder.getResult()
```

## Inne przydatne wzorce projektowe

### Leniwa inicjalizacja

Obiekt lub jego fragmenty są tworzone dopiero w momencie pierwszego użycia.

### Singleton

Istnieje dokładnie jedna instancja danej klasy. Jeżeli klasy są obiektami to można je traktować jak singletony.

### Multiton

Istnieje ściśle określona liczba instancji danej klasy, zazwyczaj dostępnych za pośrednictwem słownika.