Programowanie i projektowanie obiektowe

Metody i dziedziczenie

Paweł Daniluk

Wydział Fizyki

Jesień 2014

Metody

Przypomnienie

Obiekty odpowiadają za przechowywanie informacji.
W klasach definiuje się odpowiedzialność za funkcjonalności.

Metoda to funkcja zdefiniowana w klasie i operująca na obiekcie, dla którego została wywołana (i dodatkowych argumentach).

Metody

Przypomnienie

Obiekty odpowiadają za przechowywanie informacji.
W klasach definiuje się odpowiedzialność za funkcjonalności.

Metoda to funkcja zdefiniowana w klasie i operująca na obiekcie, dla którego została wywołana (i dodatkowych argumentach).

Uwaga

W niektórych językach programowania używa się pojęcia komunikatu.

Metody c.d.

Co mogą robić metody?

- zmiana i pobieranie właściwości obiektu (kapsułkowanie)

Metody c.d.

Co mogą robić metody?

- zmiana i pobieranie właściwości obiektu (kapsułkowanie)
- pobieranie przetworzonych właściwości obiektu

Metody c.d.

Co mogą robić metody?

- zmiana i pobieranie właściwości obiektu (kapsułkowanie)
- pobieranie przetworzonych właściwości obiektu
- zmiana właściwości w wyniku obliczenia

Metody c.d.

Co mogą robić metody?

- zmiana i pobieranie właściwości obiektu (kapsułkowanie)
- pobieranie przetworzonych właściwości obiektu
- zmiana właściwości w wyniku obliczenia
- obliczenie funkcji z dodatkowymi argumentami wynikającymi ze stanu obiektu

Metody c.d.

Co mogą robić metody?

- zmiana i pobieranie właściwości obiektu (kapsułkowanie)
- pobieranie przetworzonych właściwości obiektu
- zmiana właściwości w wyniku obliczenia
- obliczenie funkcji z dodatkowymi argumentami wynikającymi ze stanu obiektu
- wykonanie operacji na innym obiekcie

Metody c.d.

Co mogą robić metody?

- zmiana i pobieranie właściwości obiektu (kapsułkowanie)
- pobieranie przetworzonych właściwości obiektu
- zmiana właściwości w wyniku obliczenia
- obliczenie funkcji z dodatkowymi argumentami wynikającymi ze stanu obiektu
- wykonanie operacji na innym obiekcie
- kaskada wywołań/zdarzeń

Operacje na właściwościach

Kapsułkowanie

Square
–side
+set_side(side) +get_side()

Quadratic
–a,b,c
+set_abc(a,b,c) +get_abc()

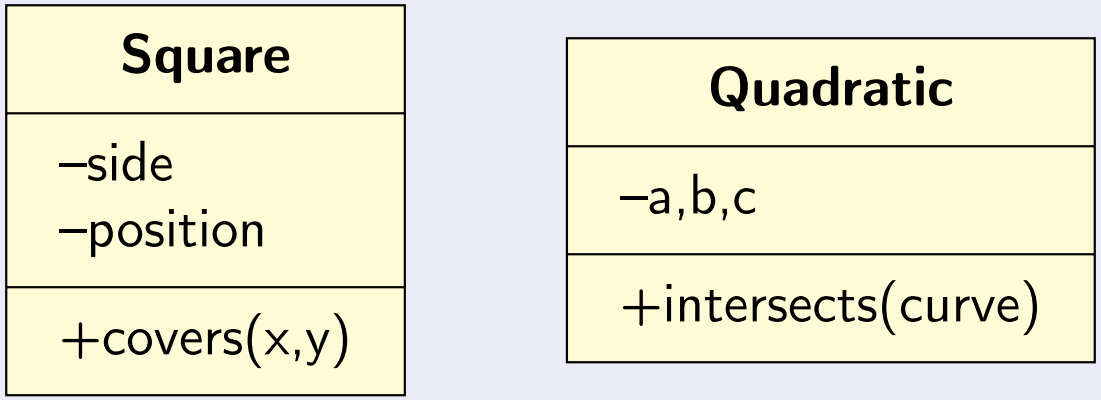
Przetworzone właściwości

Square
+get_area() +get_perimeter()

Quadratic
+get_solutions()

Obliczenia

Funkcje



Operacje na innym obiekcie

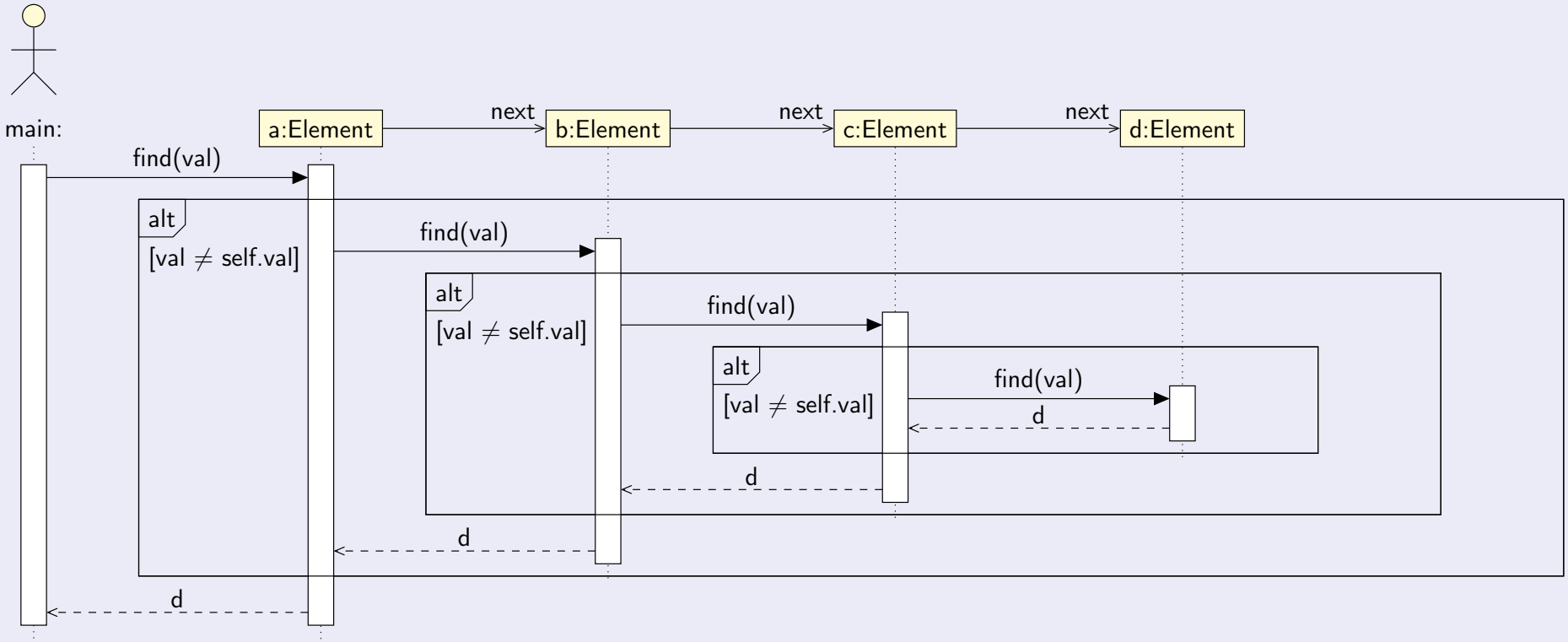
Wywołanie metody obiektu może powodować wywołanie metod innych (np. przekazanych jako argumenty) obiektów.

Przykład

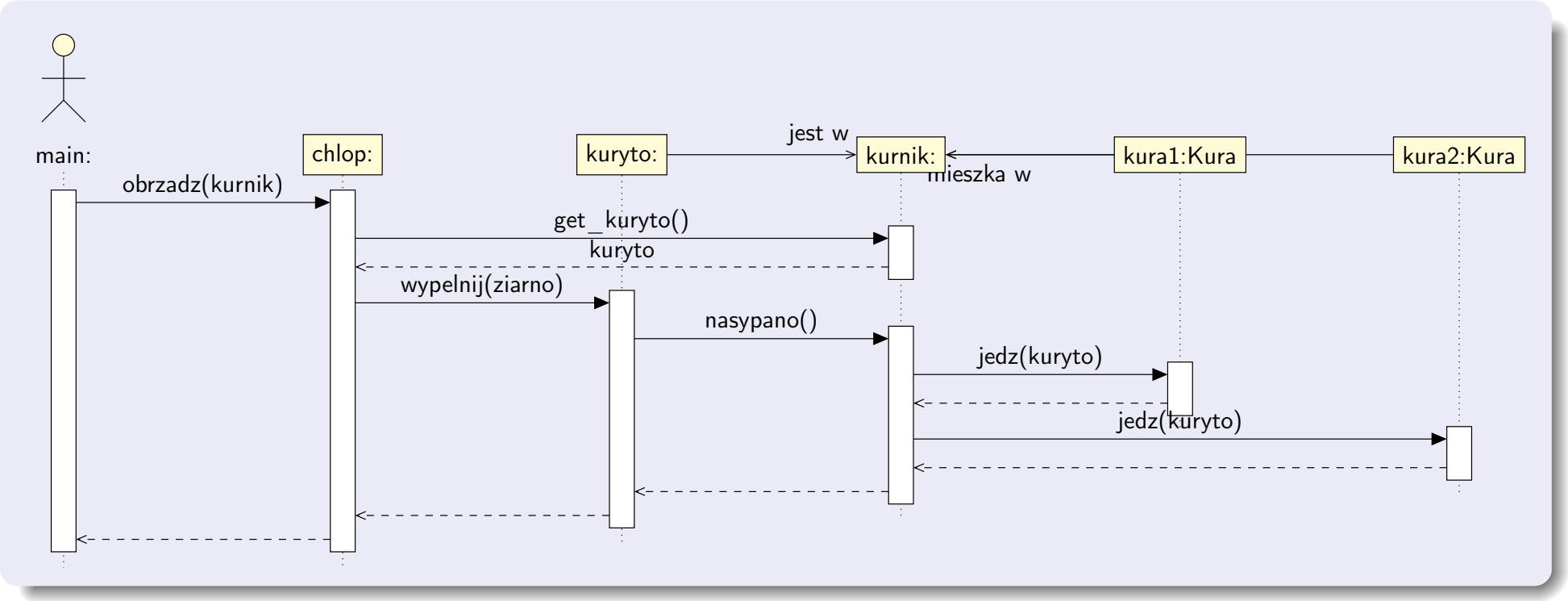
```
class Treser:
    def nakarm(self, lew):
        pasza=self.przygotuj_pasze()
        lew.jedz(pasza)

class Lew:
    def jedz(self, pasza):
        print "Mniam, pyszna ", pasza
```

Kaskada wywołań

Wywołanie metody może pociągać za sobą zaplanowaną serię kolejnych wywołań.

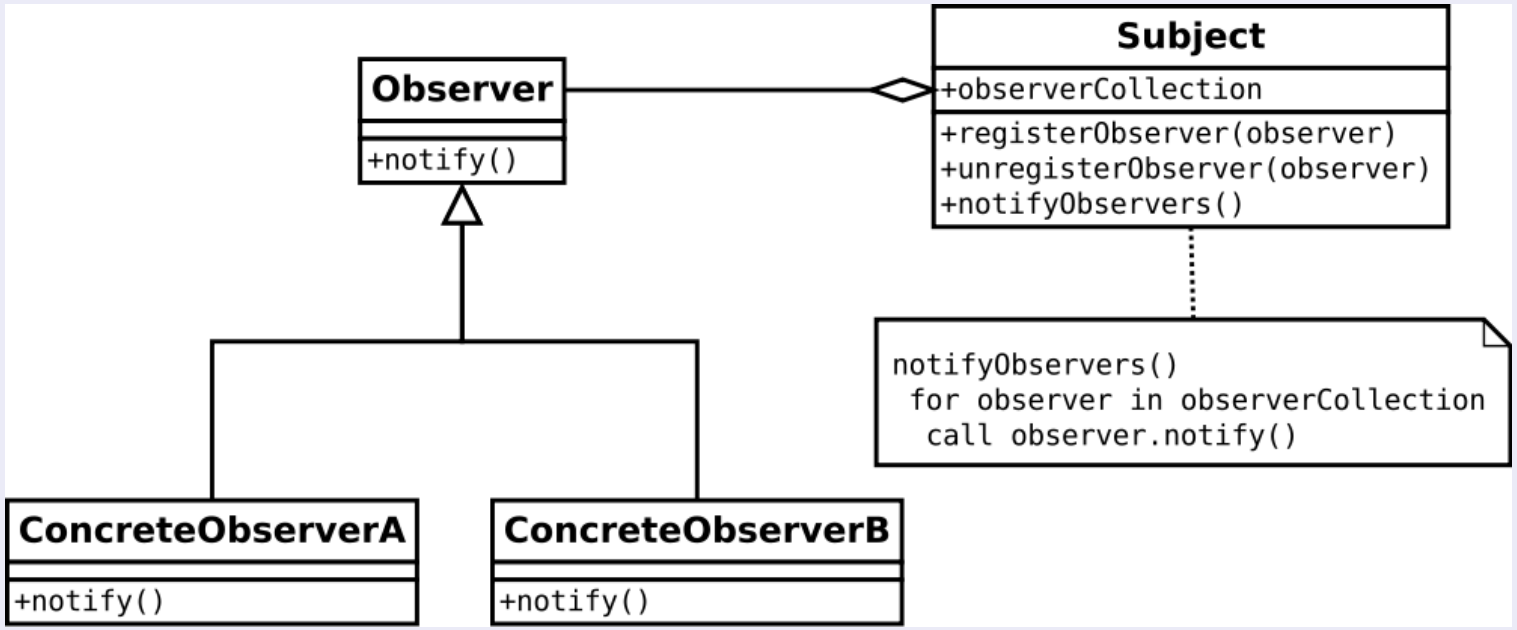


Przykład


main:
chlop:
kuryto:
jest w
kurnik:
mieszka w
kura1:Kura
kura2:Kura
obrzadz(kurnik)
get_kuryto()
kuryto
wypelnij(ziarno)
nasypano()
jedz(kuryto)
jedz(kuryto)

Observer

Przedstawiony na poprzednim slajdzie schemat jest nienaturalny. Kury powinny obserwować koryto. Wzorzec *Observer* pozwala na rejestrowanie obiektów, które powinny być informowane (klasa *Observer*) w obiekcie obserwowanym (klasa *Subject*). Wywołanie metody `notifyObservers` powoduje wywołanie metody `notify` u wszystkich obserwatorów.



Dziedziczenie

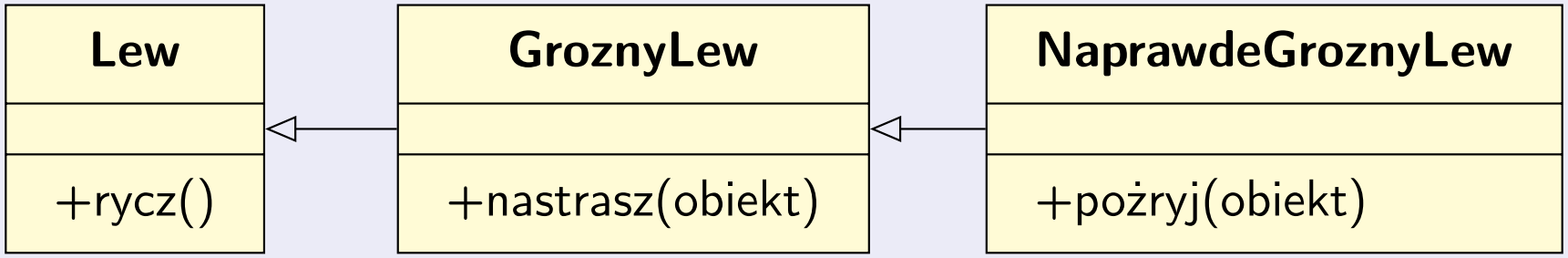
Podklasa dziedziczy składowe nadklasy. Jeśli jest to konieczne, może je przedefiniować.

Zastosowania

- rozszerzenie istniejącej klasy o nowe funkcjonalności (uszczegółowienie)
- grupowanie klas o wspólnych funkcjonalnościach
- oznaczenie przynależności do wspólnej, nadrzędnej kategorii

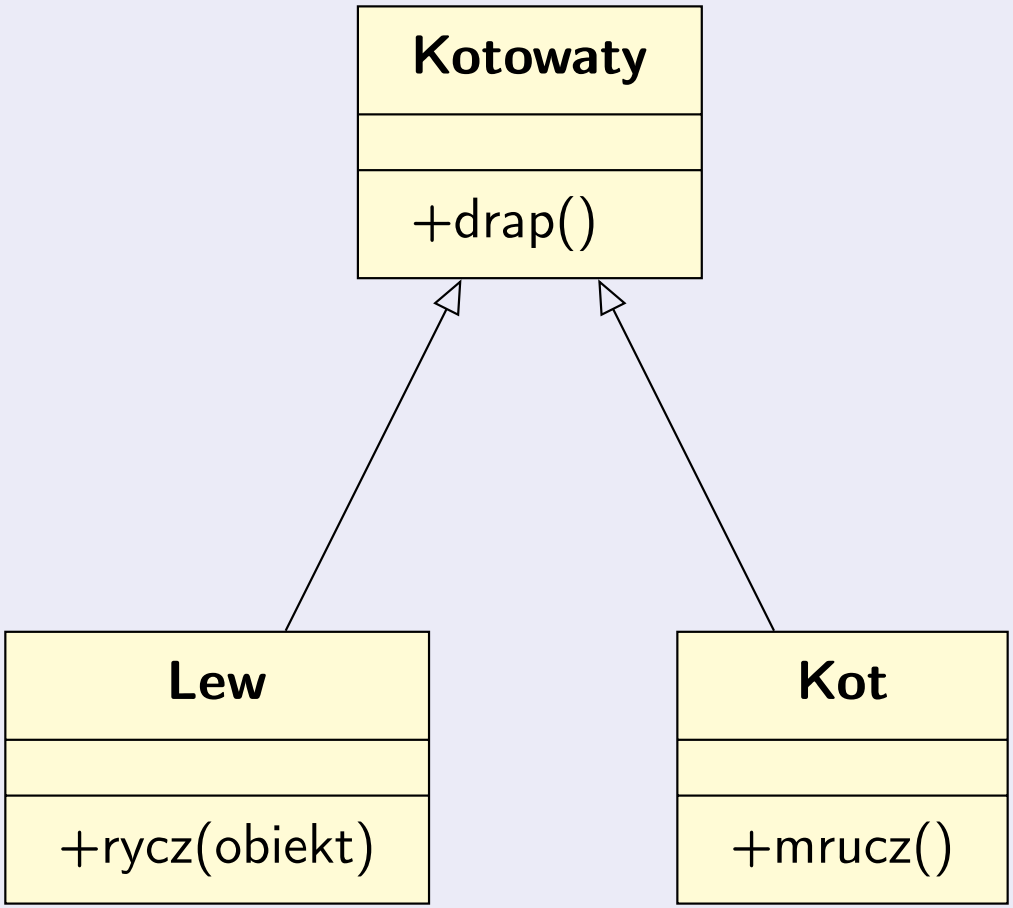
Rozszerzenie o nowe funkcjonalności

Przykład



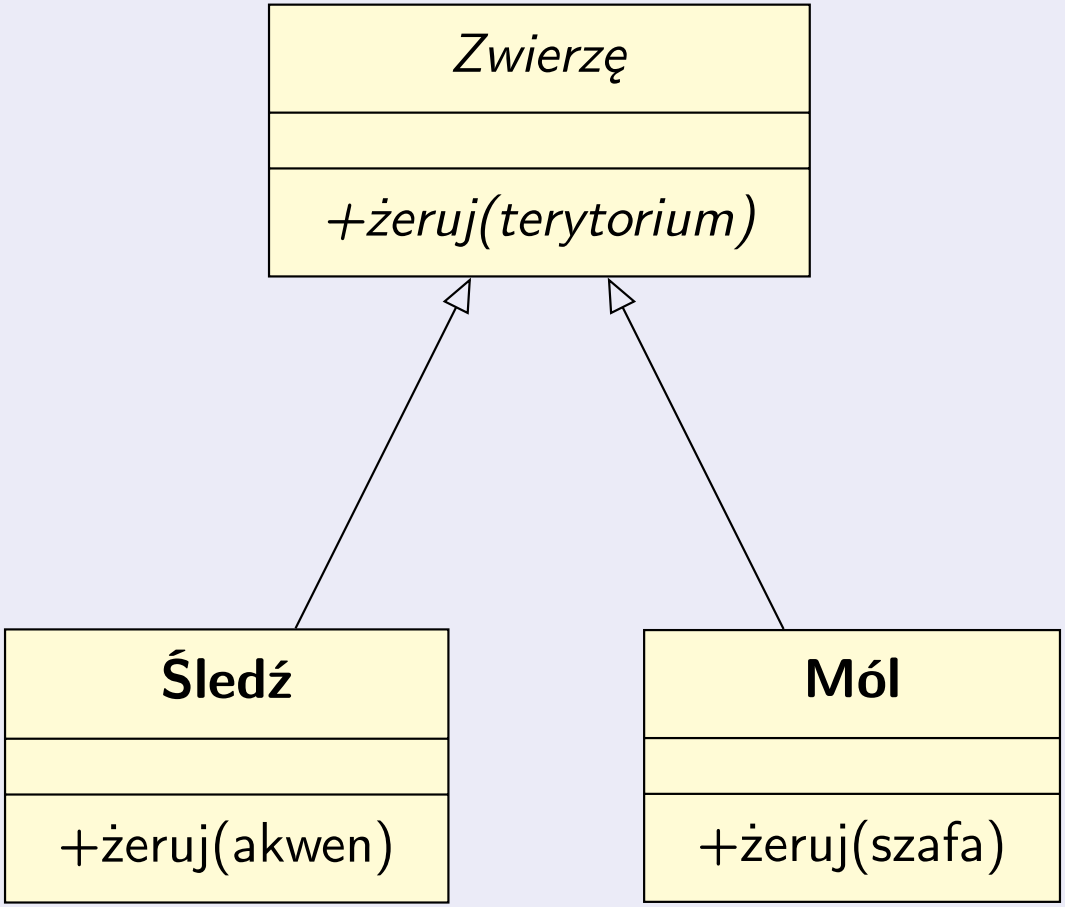
Grupowanie ze względu na funkcjonalności

Przykład



Przynależność do jednej kategorii

Przykład



Mówimy, że klasa *Zwierzę* jest klasą abstrakcyjną.

Uwaga o typowaniu

Typowanie statyczne (Java, C++)

Stosowanie klas abstrakcyjnych jest konieczne, jeżeli ma istnieć możliwość operowania w tym samym kontekście na obiektach różnych klas, które nie mają naturalnej (obejmującej wspólną funkcjonalność) nadklasy. Występowanie wymaganych składowych jest weryfikowane na etapie kompilacji.

Typowanie dynamiczne (duck-typing) (Python, Smalltalk)

Klasy abstrakcyjne nie są konieczne, ponieważ dowolne obiekty mogą występować we wszystkich kontekstach. Jeżeli nie posiadają wymaganych składowych zostanie to wykryte dopiero podczas wykonania programu. Jednakowoż stosowanie klas abstrakcyjnych ułatwia projektowanie i późniejsze utrzymanie kodu.

Przysłanianie metod

Metoda zdefiniowana w podklasie przesłania tę zdefiniowaną w nadklasie.

Możliwe są dwie strategie:

- metoda z nadklasy jest nieużywana w podklasie
- metoda z nadklasy jest wywoływana w podklasie

Python

Metody, których nazwa jest poprzedzona prefiksem __ nie są przysłanialne.

Przysłanianie metod c.d.

Przykład

```
class Student:
    def odpowiedz(self, pytanie):
        self.zrob_karpia()
        self.dukaj(pytanie)

class DobryStudent(Student):
    def odpowiedz(self, pytanie):
        self.odpowiedz_spiewajaco(pytanie)
```

Przysłanianie metod c.d.

Przykład

```
class Lew:
    def pozryj(self, antylopa):
        print "Mniam."

class GroznyLew(Lew):
    def pozryj(self, antylopa):
        self.rycz()
        Lew.pozryj(self, antylopa)
```

Klasy abstrakcyjne

Czasem zachodzi potrzeba zdefiniowania metody, która musi zostać przysłonięta (w językach z typowaniem statycznym jest to standardem). Obiekty klasy, która zawiera taką metodą, nie powinny być tworzone.

Klasy abstrakcyjne c.d.

W Pythonie

```
class Animal():
  def speak(self):
    raise NotImplementedError #abstract

class Dog(Animal):
  def speak(self):
    return "bark"

class MuteAnimal(Animal):
  pass
```

Nieme zwierzę można stworzyć. Próba wywołania metody speak zakończy się wyjątkiem.

Klasy abstrakcyjne c.d.

W Pythonie

```
from abc import ABCMeta, abstractmethod

class Animal():
  __metaclass__ = ABCMeta

  @abstractmethod
  def speak(self):
    pass

class MuteAnimal(Animal):
  pass

class Dog(Animal):
  def speak(self):
    print "Bark"
```

```
>>> Animal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Animal with abstract methods speak
>>> MuteAnimal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MuteAnimal with abstract methods speak
>>> Dog()
<__main__.Dog object at 0x10ade20d0>
```

Kiedy warto rozszerzyć klasę?

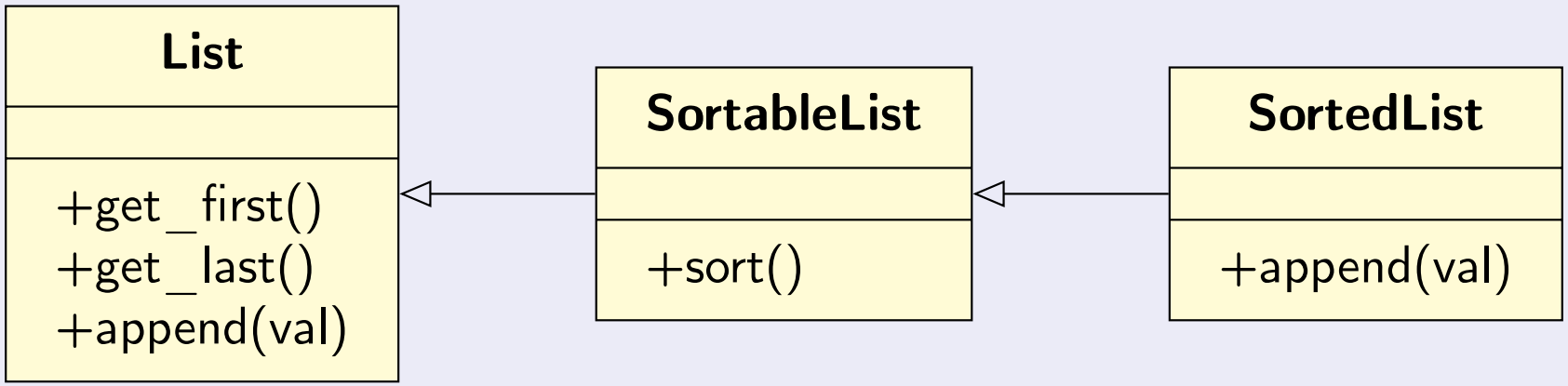
Podklasę tworzymy, jeżeli:

- wszystkie składowe nadklasy zostaną wykorzystane lub przysłonięte w użyteczny sposób,
- obiekt podklasy ma udawać (być używany tak samo jak) obiekt nadklasy.

Jeżeli żaden z powyższych warunków nie jest spełniony (czyli np. zachodzi konieczność przysłonięcia wielu metod w sposób trywialny), należy stworzyć wspólną nadklasę.

Kiedy warto rozszerzyć klasę? c.d.

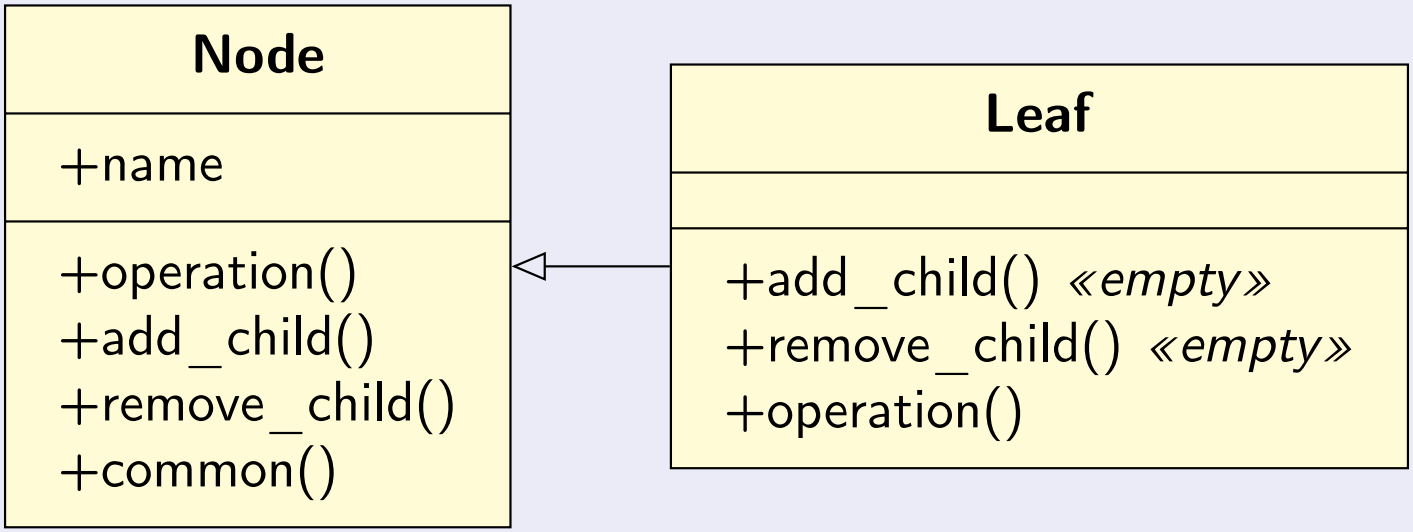
Dobrze



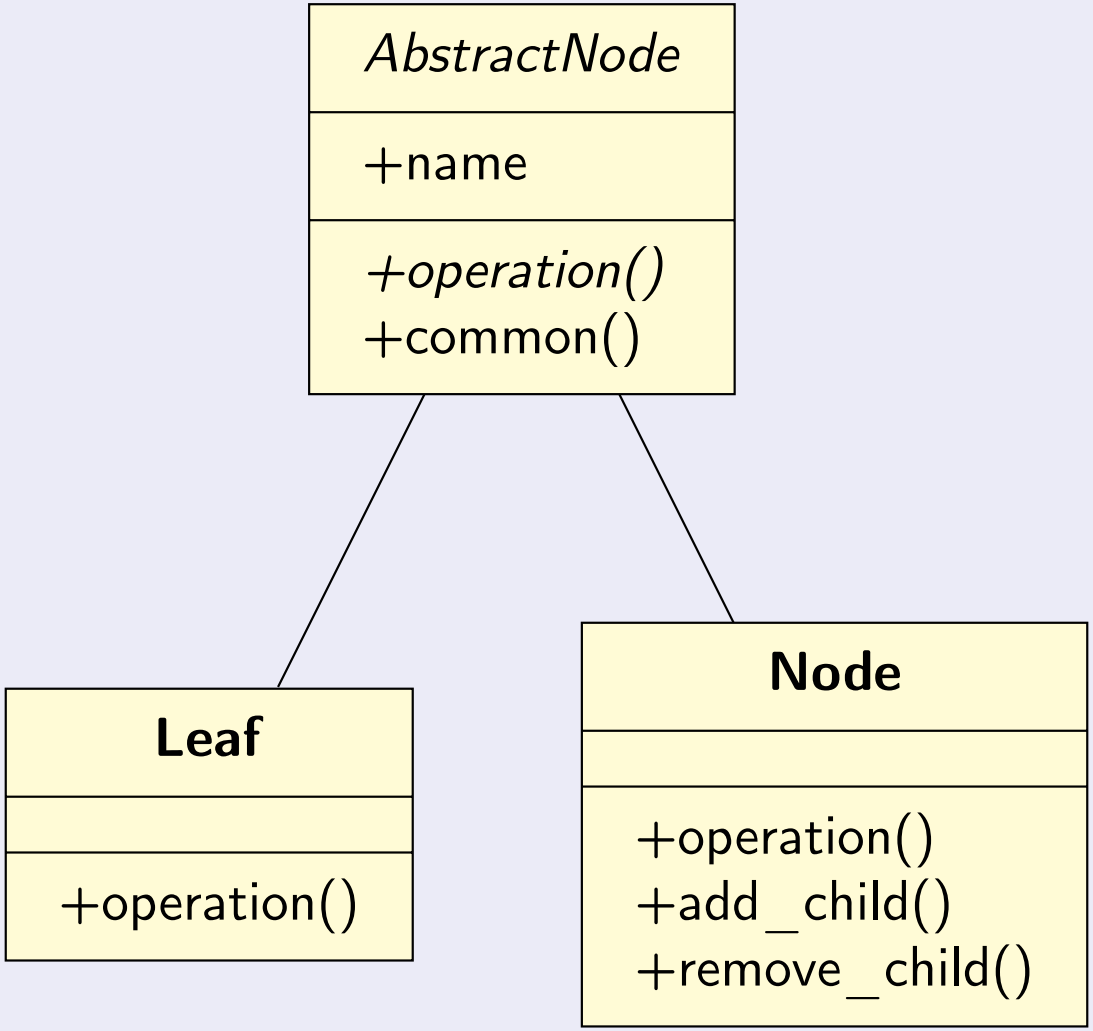
Metoda `sort` w klasie *SortedList* mogłaby być prywatna.

Kiedy warto rozszerzyć klasę? c.d.

Źle



Dobrze

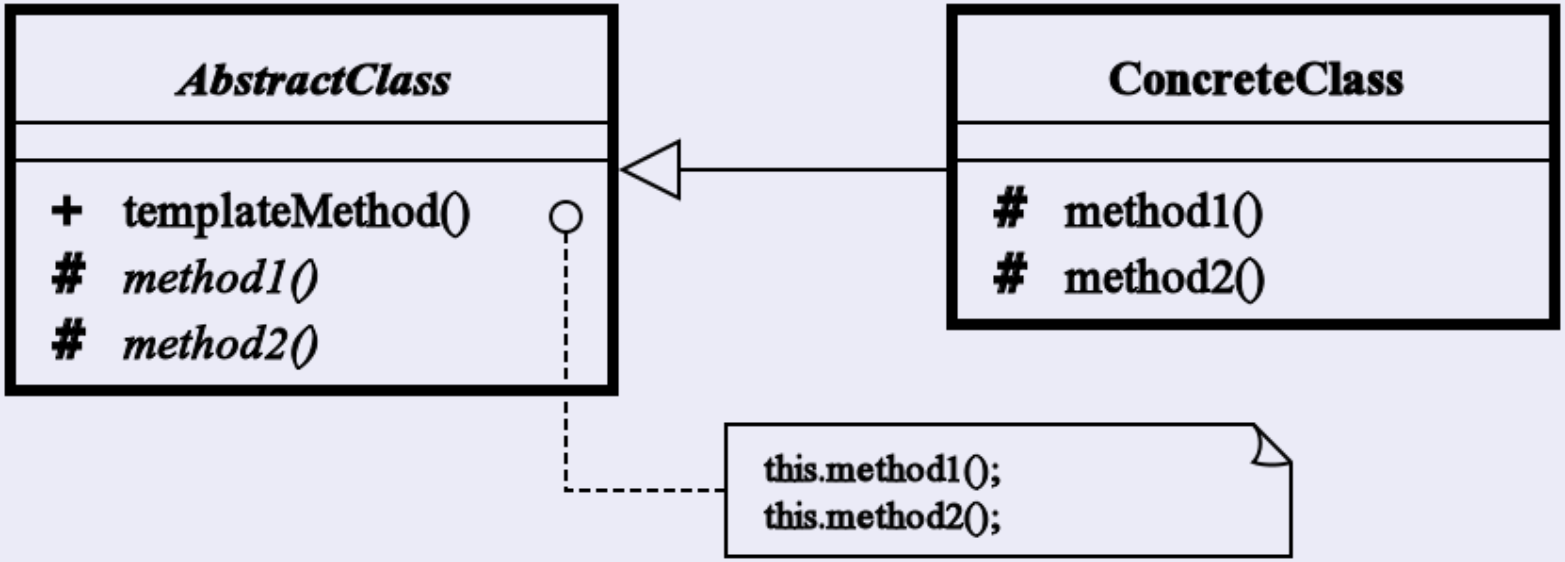

AbstractNode
+name
+operation()
+common()
Leaf
+operation()
Node
+operation()
+add_child()
+remove_child()

Jak projektować hierarchie klas

- Klasy dziedziczące po sobie powinny spełniać warunki podane wcześniej.
- Warto wstawiać klasy abstrakcyjne tam, gdzie występują wspólne funkcjonalności.
- Metoda podklasy może wywoływać więcej niż jedną metodę nadklasy.
- Jeżeli klasy mają wspólną funkcjonalność, ale nie mogą mieć wspólnej nadklasy, należy rozważyć stworzenie klasy pomocniczej.

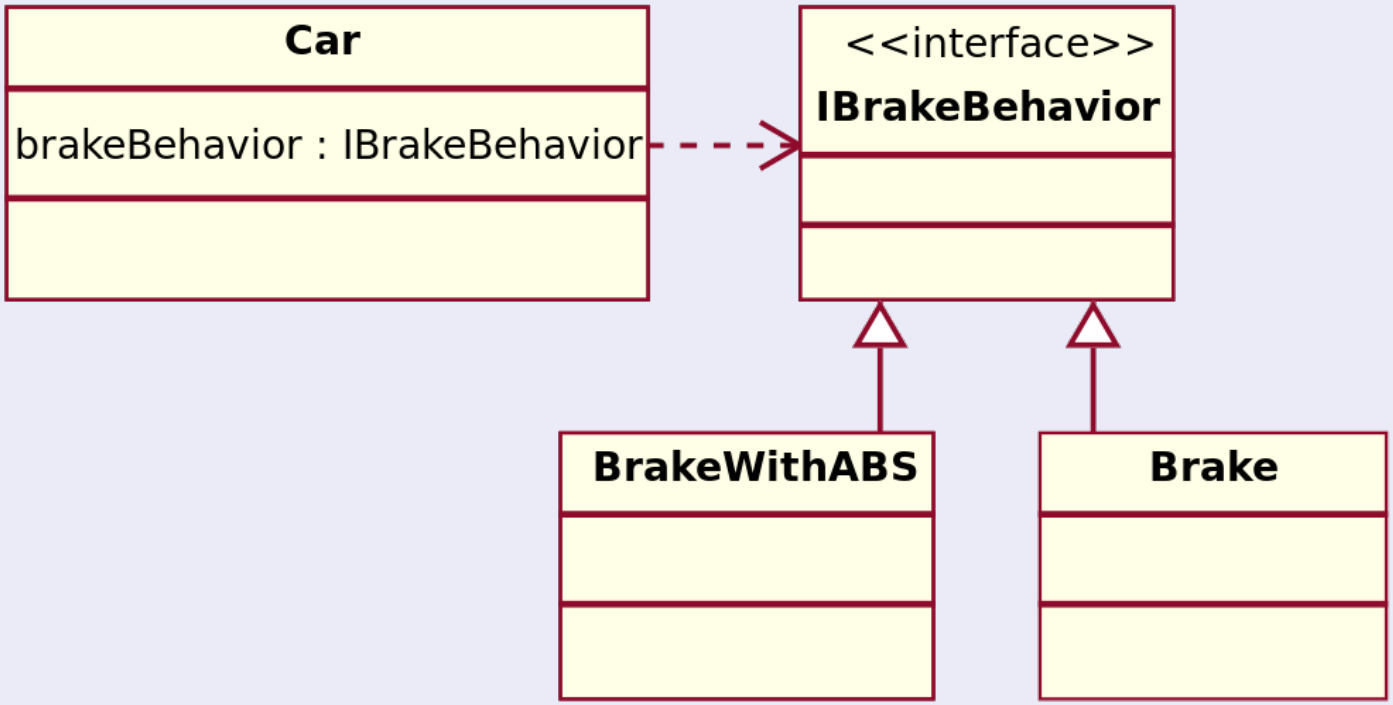
Template method – metoda szablonowa

Często ogólny algorytm jest wspólny dla wielu klas. Zmieniają się jedynie niektóre jego fragmenty.



Strategy

Jeżeli istnieje kilka algorytmów, które mogą być stosowane w danym kontekście, można je opakować w klasy i przekazywać odpowiedni obiekt.



State

Jeżeli funkcjonalność obiektu powinna zmieniać się w czasie, trudno jest zdefiniować i umieścić w hierarchii hybrydową klasę implementującą odpowiednie metody. Lepiej zastosować klasę pomocniczą.

```
class Cursor:
    def __init__(self):
        self.usePenTool()

    def moveTo(self, point):
        return self.current_tool.moveTo(point)

    def moveDown(self, point):
        return self.current_tool.moveDown(point)

    def moveUp(self, point):
        return self.current_tool.moveUp(point)

    def usePenTool(self):
        self.current_tool = PenTool()

    def useSelectionTool(self):
        self.current_tool = SelectionTool()
```

Zadanie 1 – Wzorzec Observer i kaskady wywołań

Zadanie

Zaimplementuj klasy *Kurnik*, *Kura* i *Kuryto* o poniższej funkcjonalności:

- Kury mieszkają w kurniku.
- Kuryta znajdują się w kurniku.
- Kury mogą obserwować dowolnie wiele kuryt.
- Do kuryta można nasypać określoną liczbę porcji paszy.
- Kury obserwujące kuryto rzucają się na paszę po jej nasypaniu i próbują zjeść po jednej porcji.

Zadanie 2 – Figury geometryczne

Zadanie

Utwórz hierarchię klas służącą do przechowywania informacji o figurach geometrycznych (kwadrat, prostokąt, koło, trójkąt) pozwalającą na wykonywanie następujących operacji (tam gdzie to możliwe):

- obliczanie obwodu i pola,
- obliczanie długości najdłuższego boku,
- obliczanie promienia okręgu opisanego na figurze,
- wypisywanie informacji.