

EPI: Interfejs Graficzny 2009/2010

Programowanie obiektowe w Rubim

Aleksander Pohl

20 października 2010

OO – Object Orientedness

– Obiektowe zorientowanie

„Object oriented is a catchy phrase. To call anything object oriented can make you sound pretty smart.”

– Matz, Ruby User's Guide

języki proceduralne – Fortran, C, Pascal

program = kolekcja struktur danych i procedur

języki zorientowane obiektowo – Smalltalk, Java, Ruby

program = kolekcja obiektów

Motywacja

późne lata 60-te: **kryzys oprogramowania**

- ▶ sprzęt staje się coraz mocniejszy
- ▶ można zajmować się coraz bardziej skomplikowanymi problemami
- ▶ programy stają się większe
- ▶ kod staje się niezrozumiały

Edsger Dijkstra 1972:

"To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

Przykład – proceduralny licznik kilometrów

Zbudujmy model samochodowego licznika kilometrów – urządzenia, które śledzi dystans jaki przebył samochód od momentu, w którym licznik był ostatnio resetowany
prosta implementacja w C:

```
float distance; //trzyma aktualnie zarejestrowany dystans

void increment_distance(float d){
    distance+=d;
}

void reset(){
    distance=0;
}

void bogus_procedure(){
    distance-=100;//zdecydowanie niezamierzone!
}
```

Problem: każdy może manipulować stanem licznika!



Zupełnie inne spojrzenie na programowanie

Wysokopoziomowe spojrzenie na modelowaną domenę:

zamiast myśleć w kategoriach sekwencji procedur –

zidentyfikuj 'graczy', elementy, które są modelowane – **obiekty!**

Spójrz na **rzeczywiste obiekty!**

Przykład:

- ▶ domena: Alicja w Krainie Czarów
- ▶ obiekty: Alicja, Biały Królik, Królowa, Kot z Cheshire, Kryjówka Królika

Podejście obiektowe

- ▶ jakie **własności** posiadają? → atrybuty
- ▶ jakie **działania** podejmują? → metody

Obiekt „Alicja”:

- ▶ atrybuty: wzrost, położenie
- ▶ metody: powiększ, pomniejsz, powiedz_wiersz

Obiekt „Kot z Cheshire”:

- ▶ atrybuty: widoczność, lokalizacja
- ▶ metody: zniknij, pojaw_się, uśmiechnij_się

Klasy i obiekty

Jeśli wiemy jak zbudować jeden obiekt – możemy budować ich wiele, stosując ten sam **wzorzec**.

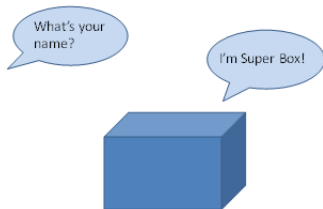
Ten wzorzec nazywany jest **klasą**.

- ▶ **klasa** definiuje abstrakcyjną charakterystykę rzeczy
np. klasa *Osoba*: posiada imię i nazwisko, potrafi się przedstawić
- ▶ **obiekt** jest instancją klasy, posiada stan
np. konkretna osoba, której imieniem jest "Jan", a nazwiskiem "Kowalski"

Przekazywanie wiadomości

Obiekty komunikują się przekazując wiadomości – wywołują nawzajem swoje **metody**.

Metafora **czarnej skrzynki**:



Pozostałe obiekty nie powinny być zainteresowane tym co znajduje się **wewnątrz** czarnej skrzynki – muszą jedynie znać komunikaty, które mogą być do niej przesłane w celu wysłania/odebrania danych!

Klasy i obiekty w Rubim

```
class Person
  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end

  def to_s
    "#{@first_name} #{@last_name}"
  end

  def introduce
    "My name is #{self.to_s}"
  end
end

john = Person.new("John", "Smith")
puts john.introduce
```

Klasy i obiekty w Rubim

- ▶ **new** – tworzy nowy obiekt danej klasy
- ▶ **initialize** – wywoływana, kiedy tworzony jest nowy obiekt (tzn. jako pochodna wywołania „new”)
- ▶ **self** – referencja do aktualnego obiektu; zazwyczaj może być pominięta
- ▶ **@** znak oznaczający zmienną instancyjną – atrybut obiektu

Przykład – obiektowy licznik kilometrów

```
class Odometer
  def initialize
    @distance = 0.0
  end

  def increment_distance(distance)
    @distance += distance if distance > 0
  end

  def reset
    @distance = 0.0
  end
end

o = Odometer.new
o.increment_distance(10.0)
o.distance -= 100

#=> NoMethodError: undefined method 'distance' for
#=> #<Odometer:0xb7cd3378 @distance=0.0>
```



Zasięg zmiennych

- ▶ **\$my_var** – zmienna globalna – dostępna w każdym kontekście
- ▶ **@@my_var** – zmienna klasowa – dostępna w klasie i wszystkich jej obiektach
- ▶ **@my_var** – zmienna instancyjna – dostępna tylko w obiekcie, do którego należy
- ▶ **my_var** – zmienna lokalna – zasięg leksykalny

Zasięg zmiennych

```
class Greeter
  @@hello_counter = 0
  def initialize(name)
    @name = name
  end

  def say_hello
    @@hello_counter += 1
    puts "Hello #{@@hello_counter} "+@name+' !'
  end
end

g1 = Greeter.new('Joe')
g1.say_hello
#=> Hello 1 Joe!
g2 = Greeter.new('Mary')
g2.say_hello
#=> Hello 2 Mary!
```

Obiekt/klasa atrybuty/metody

	Definiowanie metod klasowych	
	Obiekt	Klasa
Atrybuty	@attr np. @name – nazwa konkretnego obiektu	@@attr np. @@count – liczba obiektów danej klasy
Metody	Definicja: def introduce end Wywołanie: obiekt.metoda np. john.introduce	Definicja: def Person.new end Wywołanie: klasa.metoda np. Person.new

Tablica: Atrybuty i metody

Odbiorca

Odbiorca jest elementem, który odpowiada na wywołanie metody – jest tym „co stoi na lewo od **kropki**”

np. $\underbrace{\text{person}}_{\text{odbiorca}} \overset{\text{kropka}}{\cdot} \underbrace{\text{introduce}}_{\text{metoda}}$

- ▶ W przypadku metod instancyjnych – odbiorcą jest instancja
- ▶ W przypadku metod klasowych – odbiorcą jest klasa

Jeśli odbiorca jest pominięty – domyślnie jest:

- ▶ aktualnym **obiektem** lub **self** – w przypadku metod instancyjnych
- ▶ aktualną **klasą** – w przypadku metod klasowych

Dostęp do atrybutów

Atrybuty obiektu są domyślnie prywatne – ale można je udostępnić poprzez zdefiniowane akcesorów tzw. **getterów** i **setterów**.
Można to zrobić „ręcznie”:

```
class Colour
  def initialize(rgb)
    @rgb = rgb
  end

  def rgb
    @rgb
  end

  def rgb=(new_rgb)
    @rgb = new_rgb
  end
end
```

Dynamiczne akcesory

Lub zwięźlej:

```
class Colour
  attr_accessor :rgb
  def initialize(rgb)
    @rgb = rgb
  end
end
```

Obie definicje pozwolą na wykonywanie następujących operacji:

```
c = Colour.new('#000000')
c.rgb
c.rgb = '#FFFFFF'
```

W celu uczynienia atrybutu tylko odczytywalny/zapisywalny, używamy odpowiednio `attr_reader` i `attr_writer`.

Dostępność metod

W Rubim występują trzy poziomy dostępności metod:

- ▶ **public** (publiczny) – metody tego rodzaju mogą być wywoływane bez ograniczeń; domyślnie wszystkie metody są publiczne, z wyjątkiem `initialize`, która jest zawsze prywatna
- ▶ **protected** (chroniony) – metody mogą być wywoływane tylko przez obiekty danej klasy i jej podklas
- ▶ **private** (prywatny) – metody nie mogą być wywoływane z jawnym odbiorcą; innymi słowy mogą być wywoływane wyłącznie w obrębie danego obiektu

Modyfikatory dostępu są umieszczane w nowej linii i dotyczą wszystkich metod, które po nich następują, aż do napotkania kolejnego modyfikatora (o ile występuje).



Modyfikatory dostępu

```
class MyClass
  def method1      # domyślnie dost. ,,publiczny''
    #...
  end

  protected      # kolejne metody będą ,,chronione''
  def method2     # jest ,,chroniona''
    #...
  end

  private        # kolejne metody będą ,,prywatne''
  def method3     # jest ,,prywatna''
    #...
  end

  public         # kolejne metody będą ,,publiczne''
  def method4     # jest ,,publiczna''
    #...
  end
end
```



Dziedziczenie

Pojęcia, która są odpowiednikami klas w OOP, zazwyczaj tworzą hierarchię typu: pojęcie bardziej *ogólne* – pojęcie bardziej *specyficzne*. Przykład: *kot* jest rodzajem *ssaka*.

W Rubim zapisujemy to następująco:

```
class Mammal  
end
```

```
class Cat < Mammal  
end
```

W językach obiektowych mówimy, że:

- ▶ Ssak jest **nadklasą** kota
- ▶ Kot jest **podklasą** ssaka
- ▶ Podklasa **dziedziczy** własności (atrybuty i metody) nadklasy.

Dziedziczenie – przykład

```
class Mammal
  def breathe
    puts 'wdycham i wydycham'
  end
end

class Cat < Mammal
  def speak
    puts 'Miau!'
  end
end

tama = Cat.new
tama.breathe #=> wdycham i wydycham
tama.speak  #=> Miau!
```

Modyfikowanie zachowania

Co zrobić w wypadku, gdy podklasa nie powinna dziedziczyć wszystkich metod swojej nadklasy?

```
class Bird
  def lay_egg
    puts 'Składam jajko!'
  end

  def fly
    puts "Latam!"
  end
end

class Penguin < Bird
  def fly
    fail "Przykro mi, ja tylko pływam!"
  end
end
```

Redefiniowanie metod nadklasy

Podklasa może całkowicie zmodyfikować metodę nadklasy:

```
class Person
  def identify
    puts "Jestem osobą!"
  end
end
class Student < Person
  def identify
    puts "Jestem studentem!"
  end
end
```

lub ją rozszerzyć:

```
class Student < Person
  def identify
    super
    puts "Jestem studentem!"
  end
end
```

Wielodziedziczenie

W naturze wielodziedziczenie jest całkowicie naturalne:

- ▶ muł – jest rodzajem osła jak i konia

Większość języków programowania pozwala jedynie na dziedziczenie jednobazowe – Ruby również.

Niemniej jednak, klasy Rubiego mogą dziedziczyć metody od wielu rodziców – poprzez **dziedziczenie mieszane** (dziedziczenie implementacji), do którego wykorzystywane są **moduły**.

Moduły

Moduły są jak klasy, **ale**:

- ▶ nie mogą posiadać instancji
- ▶ nie mogą posiadać podklas (podmodułów)
- ▶ są definiowane za pomocą słów kluczowych `module ... end`

Moduł może być **dołączony** do klasy – poprzez **wmiksowanie** (wmieszanie) jego metod.

Przykład mixinu (wmiksowania)

```
class Player
  attr_accessor :name, :surname, :rank
  include Comparable

  def initialize(name, surname, rank)
    @name = name
    @surname = surname
    @rank = rank
  end

  def <=>(other)
    self.rank <=> other.rank
  end
end

p1 = Player.new("Andrea", "Agassi", 100)
p2 = Player.new("Roger", "Federer", 1)
p2 < p1 #=> true
```

Materiały źródłowe

- ▶ „Object Oriented thinking” w Ruby User’s Guide
(<http://www.rubyist.net/~slagell/ruby/oothinking.html>)
- ▶ Rozdział 3 „Programowanie w języku ruby”
(http://www.rubycentral.com/pickaxe/tut_classes.html)
- ▶ Rozdział „Obiektość” w „Ruby intro”
(<http://apohllo.pl/dydaktyka/ruby/intro/klasy/>)