Wykład nr 4
Programowanie obiektowe

Aleksander Pohl

Elektroniczne Przetwarzanie Informacji

13 listopada 2013

Motywacja

Około 2/3 – 3/4 kosztów posiadania oprogramowania związanych jest z jego utrzymaniem[1]:

- dodawanie nowych funkcjonalności (zachowanie przewagi konkurencyjnej)
- dostosowanie do zmieniających się warunków zewnętrznych (np. regulacje prawne, nowe platformy użytkowe, etc.)
- usuwanie błędów
- poprawa wydajności

Im bardziej złożony kod źródłowy, tym większe koszty jego utrzymania.

[1] http://www.galorath.com/index.php/software_maintenance_cost

Paradygmaty programowania

- programowanie proceduralne
- programowanie modularne
- programowanie abstrakcyjnych typów danych
- programowanie obiektowe

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
– Martin Fowler

Programowanie proceduralne

funckcja/procedura – jednostka abstrakcji algorytmów

Zdecyduj, jakie chcesz mieć procedury; stosuj najlepsze algorytmy jakie możesz znaleźć. – B. Stroustrup

```
void sort(int * table, int size){
  int i,j,temp;
  for(i=0;i<size-1;i++){
    for(j=i+1;j<size;j++){
      if(table[i] > table[j]){
        temp = table[i];
        table[i] = table[j];
        table[j] = temp;
      }
    }
  }
}
```

Programowanie modularne

moduł – jednoska organizacji kodu

Zdecyduj jakie chcesz mieć moduły; podziel program w taki sposób aby ukryć dane w modułach. – B. Stroustrup

interfejs modułu – `stack.h`

```
void push(char c);
char pop();
const int size = 100;
```

implementacja modułu – `stack.c`

```
#include "stack.h"

static char stack[size];
static char * pointer;
void push(char c){
  //implementacja z użyciem 'pointer'
}
char pop(){
  //implementacja z użyciem 'pointer'
}
```

Programowanie modularne – cd.

użycie modułu – program.c

```
#include <stdio.h>
#include "stack.h"

int main(){
  push('c');
  char character = pop();
  if(character != 'c'){
    pritnf("Coś nie tak!\n");
  } else {
    pritnf("Wszystko ok.\n");
  }
}
```

Abstrakcyjne typy danych

abstrakcyjne typy (struktury) danych – jednostka organizacji danych i stowarzyszonych operacji

Zdecyduj się jakie chcesz mieć typy; dla każdego typu dostarcz pełny zbiór operacji. – B. Stroustrup

interfejs abstrakcyjnego typu danych – `complex.h`

```
class complex {
  double re, im;
  complex(double r, double i){re = r; im = i};
  complex operator+(complex a, complex b);
  // inne operacje
}
```

implementacja abstrakcyjnego typu danych – `complex.c`

```
#include "complex.h"

complex operator+(complex a, complex b){
  return comlex(a.re + b.re,a.im + b.im);
}
```

Abstrakcyjne typy danych – cd.

użycie abstrakcyjnego typu danych – program.c

```
#include "complex.h"

int main(){
  complex c1 = new complex(1,0);
  complex c2 = new complex(0,1);
  complex c3 = c1 + c2;
}
```

OOP – Object Oriented Programming – Programowanie zorientowanie obiektowo

klasa – podstawowa jednostka abstrakcji
Zdecyduj jakie chcesz mieć klasy; dla każdej klasy dostarcz pełny zbiór operacji; korzystając z mechanizmu dziedziczenia, jawnie wskaż to co jest wspólne. – B. Stroustrup

- dynamiczne wywoływanie metod
- enkapsulacja
- dziedziczenie
- polimorfizm
- testy jednostkowe

Podejście obiektowe

Wysokopoziomowe spojrzenie na modelowaną domenę:

zamiast myśleć w kategoriach sekwencji procedur –

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

Przykład:

- domena: informacje o pogodzie
- obiekty: 20°C, zachmurzenie całkowite, opady 100mm, wiatr umiarkowany w kierunku północnym, wilgotność 85%, Kraków, 20 sierpnia 2011, godz. 11:00, czas środkowoeuropejski

Podejście obiektowe – cd.

Obiekty:

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

Obiekt: 20°C

- atrybuty: wartość, skala
- metody: dodaj, odejmnij, porównaj, konwertuj, wyświetl

Obiekt: 20 sierpnia 2011, godz. 11:00, czas środkowoeuropejski

- atrybuty: rok, miesiąc, dzień, godzina, minuta, strefa czasowa
- metody: dodaj, odejmnij, porównaj, konwertuj, wyświetl

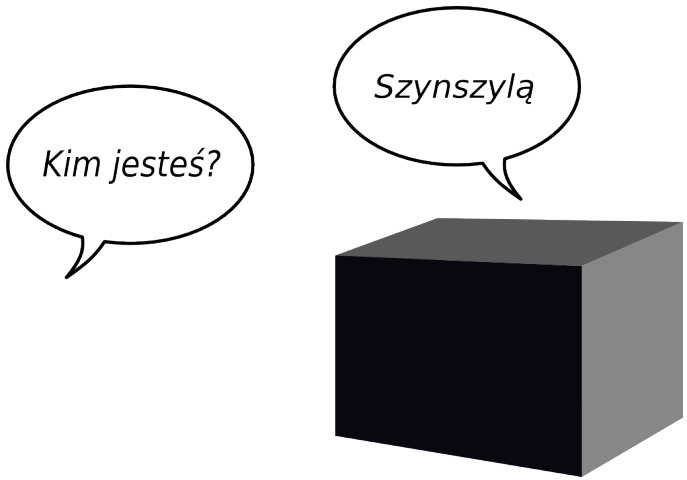
Klasy i obiekty

Jeśli wiemy jak zbudować jeden obiekt – możemy budować ich wiele, stosują ten sam **wzorzec**. Ten wzorzec nazywany jest **klasą**.

- **klasa** definiuje abstrakcyjną charakterystykę rzeczy
- np. klasa **Temperature** – posiada wartość i skalę, może być skonwertowana, porównana, etc.
- **obiekt** jest instancją klasy, posiada stan
- np. konkretna temperatura: **36.6°C** – wartość: 36.6, skala: Celsjusza

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 (interfejs/API).

Przykład – temperatura

Funkcjonalności:

- określanie temeratury w dowolnej skali
- konwersja temperatury do napisu
- konwersja z jednej skali do innej
- definicja „zera bezwzględnego”
- dodawanie temperatur
- zwielokrotnianie temperatury w danej skali
- porównywanie temperatur

Implementacja proceduralna

temperature.h

```
const int CELSIUS = 0x001;
const int KELVIN = 0x002;
const int FAHRENHEIT = 0x003;
const float ZERO = -273.15;
const float CELSIUS_TO_FAHRENHEIT_FACTOR = 9.0/5;
const float CELSIUS_TO_FAHRENHEIT_OFFSET = 32.0;

float temperature_convert(float value, int from, int to);
float temperature_add(float value1,int type1, float value2, int type2);
char * temperature_to_string(float value,int type);
float temperature_multiply(float value,int type,float factor);
int temperature_compare(float value1,int type1, float value2,int type2);
```

Implementacja proceduralna – cd.

temperature.c

```
float temperature_convert(float value, int from, int to){
  float in_kelvins;
  switch(from){
    case CELSIUS : in_kelvins = value + ZERO; break;
    case KELVIN : in_kelvins = value; break;
    case FAHRENHEIT :
      in_kelvins = value * CELSIUS_TO_FAHRENHEIT_FACTOR +
        CELSIUS_TO_FAHRENHEIT_OFFSET + ZERO; break;
  }
  switch(to){
    case CELSIUS : return in_kelvins - ZERO;
    case KELVIN : return in_kelvins;
    case FAHRENHEIT :
      return (value - ZERO - CELSIUS_TO_FAHRENHEIT_OFFSET) /
        CELSIUS_TO_FAHRENHEIT_FACTOR;
  }
}
```

Implementacja proceduralna – problemy

program.c

```c
#include "temperature.h"
int main(){
  float in_celsius = 36.6;
  float in_kelvins = temperature_convert(in_celsius,CELSIUS,KELVIN);
  // powtórnie musimy określić jednostkę
  float in_fahrenheits = temperature_convert(in_celsius,CELSIUS,FAHRENHEIT);
  // nieisniejące jednostki
  float in_what = temperature_convert(in_celsius,CELSIUS,5);
  // w jakich jednostkach?
  float multiplied = temperature_multiply(in_celsius,CELSIUS,10.0);
  // niepoprawne dane
  in_kelvins = -300.0;
  // niepoprawna operacja
  in_fahrenheits = in_celsius + in_kelvins;
  // tak szybciej... choć błędnie
  in_kelvins = in_celsius + ZERO;
  // można... ale po co?
  if(in_kelvins > in_celsius){ }
}
```

Przykład nazw funkcji łańcuchowych w PHP

- addcslashes
- convert_cyr_string
- convert_uudecode
- count_chars
- implode
- lcfirst
- nl2br
- sha1
- str_getcsv
- str_shuffle
- strcasecmp
- strchr
- strrchr
- strlen

Implementacja proceduralna – problemy

- rozwlekłość (`temperature_add`, `temperature_convert`, ...)
- bałagan w globalnej przestrzeni nazw funkcji
- brak kontroli danych/każdorazowa kontrola danych
- brak powiązania wartości z innymi wartościami (można rozwiązać dzięki strukturom)
- brak powiązania wartości z operacjami
- brak wsparcia dla reużytkowania kodu
- w dużych systemach kod przestaje być zrozumiały

Cel – implementacja obiektowa

```
in_celsius = Temperature.new(36.6,:c)
in_celsius.value                          #=> 36.6
in_celsius.scale                          #=> :c
in_celsius.scale = :z                     # niemożliwe - wewnętrzna
                                          # struktura jest ukryta
in_kelvins = in_celsius.to_kelvin         # jednostka trzymana jest
                                          # "wewnątrz"
invalid = Temperature.new(36.6,:z)        # można np. rzucić wyjątek
multiplied = in_celsius * 10              # zachowuje skalę
invalid = Temperature.new(-300,:c)        # można np. rzucić wyjątek
sum = in_celsius + in_kelvins             # da poprawny wynik
if(in_celsius > in_kelvins)               # da poprawny wynik
  # ...
end
zero = Temperature.absolute_zero          # posiada określoną jednostkę
puts zero                                 # można określić domyślną
                                          # reprezentację np. '0 K'
```

Definicja klasy – przykład

```ruby
class Temperature
  # Initialize the temperature with +value+ and +scale+.
  def initialize(value,scale)
    @value = value
    @scale = scale
  end

  # Default string representation of the temperature.
  def to_s
    "%.2f %s" % [@value, @scale.to_s.upcase]
  end
end

temperature = Temperature.new(36.6,:c)
puts temperature                          #=> 36.60 C
```

Definicja klasy

- definicja rozpoczyna się słowem kluczowym `class` a kończy słowem `end`
- nazwa klasy jest stałą w notacji wielbłądziej
- wszystko co znajduje się pomiędzy `class` a odpowiadającym mu `end` definiowane jest w **przestrzeni nazw** klasy
- **new** – tworzy nowy obiekt danej klasy
- **initialize** (konstruktor) – wywoływana, kiedy tworzony jest nowy obiekt (tzn. jako pochodna wywołania `new`)
- **to_s** – automatycznie wywoływana przy konwersji do łańcucha znaków
- **@** znak oznaczający zmienną instancyjną – prywatny atrybut obiektu

Atrybuty

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

```
class Temperature
  # Get the +value+ of the temperature.
  def value
    @value
  end

  # Get the +scale+ of the temperature.
  def scale
    @scale
  end

  # Set the +value+ of temperature.
  def value=(new_value)
    @value = new_value
  end
end
```

Atrybuty – dynamiczne akcesory

Lub zwięźlej:

```
class Temperature
  attr_accessor :value
  attr_reader :scale
end
```

Obie definicje pozwolą na wykonywanie następujących operacji:

```
temperature = Temperatura.new(10, :c)
temperature.value                    #=> 10
temperature.value = 20
temperature.value                    #=> 20
temperature.scale                    #=> :c
temperature.scale = :z               # niemożliwe!
```

Jest również wywołanie `attr_writer`, które powoduje, że atrybut jest tylko modyfikowalny (ale nieodczytywalny).

Metody instancyjne – przykład

```
class Temperature
  # Converts the temeperature to Kelvin scale.
  def to_kelvin
    Temperature.new(convert(@scale,:k,@value),:k)
  end

  # Converts the temeperature to Celsius scale.
  def to_celsius
    Temperature.new(convert(@scale,:c,@value),:c)
  end

  # Converts the temeperature to Fahrenheit scale.
  def to_fahrenheit
    Temperature.new(convert(@scale,:f,@value),:f)
  end
end
```

Metody instancyjne

- określane są tak samo jak funkcje (słowo kluczowe `def`, nazwa, parametry)
- definiują zachowanie obiektów klasy – określają komunikaty, na które reagują obiekty klasy
- umieszczane są w ciele klasy – pomiędzy słowem kluczowym `class` a słowem kluczowym `end` kończącym definicję klasy
- należą do przestrzeni nazw klasy – odrębne klasy mogą mieć metody o tych samych nazwach i różnych implementacjach
- zakończone *wykorzyknikiem* określają metody modyfikujące obiekt, dla których istnieje wersja niemodyfikująca – *konwencja*
- zakończone *pytajnikiem* określają predykaty (zwracają wartości prawda/fałsz) – *konwencja*

Metody modyfikujące – przykład

```
class Temperature
  # Converts (in place) the temeperature to Kelvin scale.
  def to_kelvin!
    @value = convert(@scale,:k,@value)
    @scale = :k
    self
  end
  # Converts (in place) the temeperature to Celsius scale.
  def to_celsius!
    @value = convert(@scale,:c,@value)
    @scale = :c
    self
  end
  # Converts (in place) the temeperature to Fahrenheit scale.
  def to_fahrenheit!
    @value = convert(@scale,:f,@value)
    @sclae = :f
    self
  end
end
```

Metody chronione – przykład

```
class Temperature
  ABSOLUTE_ZERO = -273.15 # in Celsius
  CELSIUS_FACTORS = {:c => 1.0, :f => 1.8, :k => 1.0}
  CELSIUS_OFFSETS = {:c => 0, :f => 32, :k => -ABSOLUTE_ZERO}

  protected
  def convert(from,to,value)
    factor(from,to) * value + offset(from,to)
  end
  def factor(from,to)
    return CELSIUS_FACTORS[to] if from == :c
    return (1 / CELSIUS_FACTORS[from]) if to == :c
    factor(from,:c) * factor(:c,to)
  end
  def offset(from,to)
    return CELSIUS_OFFSETS[to] if from == :c
    return (- CELSIUS_OFFSETS[from] * factor(from,to)) if to == :c
    offset(from,:c) * factor(:c,to) + offset(:c,to)
  end
end
```

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).

Dostępność metod – zasięg

```
class Temperature
  def to_kelvin  # domyślny dostęp - publiczny
    #...
  end

  protected      # kolejne metody będą chronione
  def convert(from,to,value)
    #...
  end

  def factor(from,to)
    #...
  end

  public         # kolejne metody będą publiczne
  def to_kelvin!
    #...
  end
end
```

Metody i zmienne klasowe – przykład

```ruby
class Temperature
  @@absolute_zero = Temperature.new(0,:k)

  # Returns true if the temperature is an absolute zero.
  def zero?
    self == @@absolute_zero
  end

  # The absolute zero.
  def Temperature.absolute_zero
    @@absolute_zero
  end
end

zero = Temperature.absolute_zero
puts zero                          #=> 0 K
puts zero.to_celsius               #=> -273.15 C
puts zero.zero?                    #=> true
puts Temperature.new(10,:c).zero?  #=> false
```

Obiekt/klasa atrybuty/metody

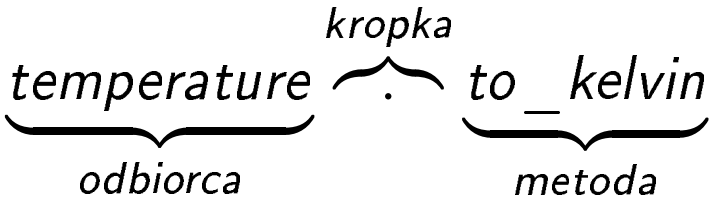
	Obiekt	Klasa
Atrybuty	`@attr` np. `@value` – wartość temperatury	`@@attr` np. `@@absolute_zero` – zero absolutne

Metody	**Definicja:**
	`def method`
	`end`
	Wywołanie:
	`object.method`
	np. `temperature.to_s`
		Definicja:
		`def Class.method`
		`end`
		Wywołanie:
		`Class.method`
		np. `Temperature.new`

Tablica : Atrybuty i metody

Odbiorca komunikatu

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



- W przypadku metod instancyjnych – odbiorcą jest obiekt
- W przypadku metod klasowych – odbiorcą jest klasa

Jeśli odbiorca jest pominięty – domyślnie jest nim:

- aktualny **obiekt** (**self**) – w przypadku metod instancyjnych
- aktualna **klasa** – w przypadku metod klasowych

Zasięg zmiennych

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

Dziedziczenie

Pojęcia, która są odpowiednikami klas, zazwyczaj tworzą hierarchię: pojęcie bardziej *ogólne* – pojęcie bardziej *specyficzne*.

Przykład: *temperatura* jest rodzajem *miary fizycznej*.

W Rubim zapisujemy to następująco:

```
class Measure
end

class Temperature < Measure
end
```

W językach obiektowych:

- *miara fizyczna* jest **nadklasą** *temperatury*
- *temperatura* jest **podklasą** *miary fizycznej*
- podklasa **dziedziczy** własności (atrybuty i metody) nadklasy

Dziedziczenie – przykład

```ruby
class Measure
  # Initialize the measure with +value+ and +scale+.
  def initialize(value,scale)
    @value = value
    @scale = scale
  end
  # Default string representation of the measure.
  def to_s
    "%.2f %s" % [@value, @scale]
  end
end

class Temperature < Measure
  # Converts the temeperature to Kelvin scale.
  def to_kelvin
    Temperature.new(convert(@scale,:k,@value),:k)
  end
end

temperature = Temperature.new(10,:c)
puts temperature.to_kelvin.to_s        #=> 283.15 k
```

Zastępowanie metod nadklasy

```ruby
class Temperatura < Measure
  VALID_SCALES = [:c,:k,:f]

  # Initialize the temperature with +value+ and +scale+.
  # Raise exception if the +scale+ is invalid.
  def initialize(value,scale)
    @value = value
    @scale = scale
    unless VALID_SCALES.include?(scale)
      raise "Invalid scale '#{scale}'."
    end
  end
end

Measure.new(10,:m)          #=> 10.00 m
Temperature.new(36.6,:m)    #=> RuntimeError: Invalid scale 'm'
Temperature.new(36.6,:c)    #=> 36.60 c
```

Modyfikowanie metod nadklasy

```ruby
class Temperatura < Measure
  VALID_SCALES = [:c,:k,:f]

  # Initialize the temperature with +value+ and +scale+.
  # Raise exception if the +scale+ is invalid.
  def initialize(value,scale)
    super(value,scale)
    unless VALID_SCALES.include?(scale)
      raise "Invalid scale '#{scale}'."
    end
  end
end
```

Wielodziedziczenie

W języku naturalnym wielodziedziczenie jest dość powszechne:

- wóza strażacki – rodzaj pojazdu uprzywilejowanego i samochodu ciężarowego

Większość języków programowania pozwala jedynie na dziedziczenie tylko z jednej klasy nadrzędnej (jednobazowe) – Ruby również.

Niemniej jednak, klasy Rubiego, poza dziedziczeniem, mają inny mechanizm współdzielenia implementacji metod – poprzez **dziedziczenie mieszane** (dziedziczenie implementacji), do którego wykorzystywane są **moduły**.

Moduły

Moduły są jak klasy[2], **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.

[2] Tak naprawdę wszystkie klasy są modułami, ale nie wszystkie moduły – klasami; klasa `Class` dziedziczy z klasy `Module`.

Przykład mixinu (wmiksowania modułu)

```ruby
module Comparable
  def <(other)
    (self <=> other) < 0
  end
  def >(other)
    (self <=> other) > 0
  end
  def ==(other)
    (self <=> other) == 0
  end
end

class Temperature
  include Comparable
  def <=>(other)
    self.to_kelvin.value <=> other.to_kelvin.value
  end
end

Temperature.new(10,:c) < Temperature.new(20,:f)        #=> false
Temperature.new(-273.15,:c) == Temperature.new(0,:k)   #=> true
```

Materiały

- „Object Oriented thinking” w Ruby User's Guide
  `www.rubyist.net/~slagell/ruby/oothinking.html`
- Programowanie obiektowe w „książce z kilofem”
  `ruby-doc.org/docs/ProgrammingRuby/html/tut_classes.html`
- Programowanie obiektowe w przewodniku po Rubim
  `apohllo.pl/dydaktyka/ruby/intro/klasy/`

Pytania

PYTANIA?

Podziękowania dla:

- Agnieszki Figiel, za udostępnienie prezentacji w postaci plików źródłowych
- Marka Kowalcze oraz Jakuba Kuźmy z grupy SRUG (srug.pl), za pomoc przy kolorowaniu składniu w Latex'u