Programowanie obiektowe

Wykład 10.

Marcin Młotkowski

26 kwietnia 2024

Plan wykładu

1. Typowanie w Ruby
2. Introspekcje (refleksje)
   - Duck typing w praktyce
   - Refleksje w praktyce: problem trwałości obiektów
   - Inne mechanizmy refleksji
   - Dynamiczne modyfikowanie istniejących klas
3. Wzorce projektowe Template i Strategy
   - Wzorzec szablon
   - Wzorzec strategia

Plan wykładu

Przypomnienie

Typowanie w w Ruby jest dynamiczne.

Duck typing (kacze typowanie)

Jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką.

Zastosowanie w programowaniu obiektowym

Jeśli obiekt ma odpowiednie metody, to jest taki jak trzeba.

Przykład w Javie

```
interface Kaczka
{
    String kwacz();
}

class Gęgawa implements Kaczka
{
  ...
}
```

Przykład w Ruby

```ruby
class Cyraneczka
    def kwacz
        puts "kwa kwa"
    end
end
```

Przykład w Ruby

```ruby
class Cyraneczka
    def kwacz
        puts "kwa kwa"
    end
end

def kwakanie(ptak)
    ptak.kwacz if ptak.respond_to? :kwacz
end
```

Przykład w Ruby

```ruby
class Cyraneczka
    def kwacz
        puts "kwa kwa"
    end
end

def kwakanie(ptak)
    ptak.kwacz if ptak.respond_to? :kwacz
end

kwakanie(Cyraneczka.new)
kwakanie(5)
```

Plan wykładu

Dynamiczne typowanie

- brak deklaracji typów zmiennych;
- dynamiczna (tj. w czasie wykonania programu) kontrola typów;
- *duck–typing*.

Wady i zalety statycznego typowania

Pod uwagę brane są tylko formalne typy obiektów.

Zalety

- kompilacja może uchronić przed elementarnymi błędami;
- możliwość wielu optymalizacji kodu wynikowego.

Wady i zalety statycznego typowania

Pod uwagę brane są tylko formalne typy obiektów.

Zalety

- kompilacja może uchronić przed elementarnymi błędami;
- możliwość wielu optymalizacji kodu wynikowego.

Wady

- konieczna jest kompilacja;
- kontrola typów może odrzucić programy poprawne;
- systemy typów mogą być błędne;
- czasem i tak konieczna jest kontrola typów.

Wady i zalety dynamicznego typowania

Ważne są nie typy, tylko implementowane operacje.

Zalety

- Szybkie uruchomienie prototypu;
- większe możliwości programistyczne

Wady i zalety dynamicznego typowania

Ważne są nie typy, tylko implementowane operacje.

Zalety

- Szybkie uruchomienie prototypu;
- większe możliwości programistyczne

Wady

- Wymaga większej staranności w pisaniu;
- czasem konieczna jest dynamiczna kontrola typów.

Praktyka stosowania dynamicznego typowania

Zadanie

Lista zadań TODO z możliwością dodawania nowych zadań.

Implementacja prosta

```
class Zadanie
  def initialize(data, tresc)
    @data = data
    @tresc = tresc
  end

  def data_opis
    return @data, @tresc
  end
end
```

Implementacja listy

```ruby
class Todo
  def initialize
    @lista = ""
  end

  def <<(zadanie)
    data, opis = zadanie.data_opis
    @lista = @lista + data + ":" + opis + "\n"
  end
end
```

Scenariusz użycia

```
todoList = Todo.new
todoList << Zadanie.new("Dzisiaj", "Lista z Ruby")

todoList << Zadanie.new("Pilne", "Zajrzeć na SKOS")

todoList << "Jutro: zakupy"
```

Scenariusz użycia

```
todoList = Todo.new
todoList << Zadanie.new("Dzisiaj", "Lista z Ruby")

todoList << Zadanie.new("Pilne", "Zajrzeć na SKOS")

todoList << "Jutro: zakupy"
```

refleksje.rb:18:in ‘<<’: undefined method ‘data_opis’ for "Jutro: zakupy":String (NoMethodError)

Wersja bezpieczniejsza

```
class Todo
  def initialize
    @lista = ""
  end

  def <<(doZrobienia)
    unless doZrobienia.kind_of?(Zadanie)
        puts "Zignorowano " + doZrobienia.to_s
        return
    rend
    data, opis = doZrobienia.data_opis
    @lista = @lista + data + ":" + opis + "\n"
  end
end
```

Wersja bezpieczniejsza

```ruby
class Todo
  def initialize
    @lista = ""
  end

  def <<(doZrobienia)
    unless doZrobienia.kind_of?(Zadanie)
        puts "Zignorowano " + doZrobienia.to_s
        return
    rend
    data, opis = doZrobienia.data_opis
    @lista = @lista + data + ":" + opis + "\n"
  end
end
```

```
todoList << "Jutro: zakupy"
Zignorowano Jutro: zakupy
```

Wady rozwiązania

```ruby
class Egzamin
  def initialize(data, egzamin)
    @data = data
    @egzamin = egzamin
  end
  def data_opis
    return @data, @egzamin
  end
end
```

Wady rozwiązania

```
class Egzamin
  def initialize(data, egzamin)
    @data = data
    @egzamin = egzamin
  end
  def data_opis
    return @data, @egzamin
  end
end
```

```
todoList << Egzamin.new("Niedługo", "Programowanie")
Zignorowano #<Egzamin:0xb743c5e4>
```

Źródło kłopotu

Kontrolujemy typ danej, a nie jej funkcjonalność.

Duck typing

Sprawdźmy, czy umie kwakać, a nie czy jest kaczką.

Ponowna implementacja

```
class Todo
    def initialize
        @lista = ""
    end

    def <<(zadanie)
        unless zadanie.kind_of?(Zadanie)
            puts "Zignorowano " + zadanie.to_s
            return
        end
        data, opis = zadanie.data_opis
        @lista = @lista + data + ":" + opis + "\n"
    end
end
```

Ponowna implementacja

```
class Todo
    def initialize
        @lista = ""
    end

    def <<(zadanie)
        unless zadanie.respond_to?("data_opis")
            puts "Zignorowano " + zadanie.to_s
            return
        end
        data, opis = zadanie.data_opis
        @lista = @lista + data + ":" + opis + "\n"
    end
end
```

Definicja problemu

Obiekty trwałe

Obiekty, których stan bieżący jest zapisywany pomiędzy kolejnymi uruchomieniami aplikacji.

Zagadnienia

- odczytanie wartości pól obiektu i zapisanie np. w relacyjnej bazie danych;
- odczytanie wartości z relacyjnej bazy danych i utworzenie na tej podstawie obiektu.

Czego potrzebujemy

lista pól obiektu	`obiekt.instance_variables`
odczyt wartości pola obiektu	`obiekt.instance_variable_get(pole)`
nadanie nowej wartości polu	`obiekt.instance_variable_set(pole, wartość)`

Rozwiązanie 1.

Implementacja klasy `Saveable` implementującej metody `save` i `restore`.

Rozwiązanie 1.

Implementacja klasy Saveable implementującej metody save i restore.

Przykład

```
class Ksiazka < Saveable
end

obj = Ksiazka.new
obj.restore( {"autor" => "Orzeszkowa",
              "tytul" => "Nad Niemnem" })
```

Rozwiązanie 2: zarządca

```ruby
module Zarządca
  def fabryka(klasa, zródło)
    case klasa
      when "Ksiazka"
        obj = Ksiazka.new
      when "Figura"
        obj = Figura.new
      else obj = Pusty.new
    end
    for varname in zródło.keys
      obj.instance_variable_set(varname, zródło[varname])
    end
    return obj
  end
```

Ciąg dalszy

```
  def save(obj)
    puts "Obiekt klasy #{obj.class}\n"
    puts "Zmienne #{obj.instance_variables}"
    for var in obj.instance_variables:
      puts "#{var}: #{obj.instance_variable_get(var)}"
    end
  end
end
```

Rozwiązanie 3: mix-ins!!!

```
module Persistence
  def save
    puts "Obiekt klasy #{self.class}\n"
    puts "Zmienne #{self.instance_variables}"
    for var in self.instance_variables:
      puts "#{var}: #{self.instance_variable_get(var)}"
    end
  end
  def restore(source)
    for key in source.keys
      self.instance_variable_set("@" + key, source[key])
    end
  end
end
```

Zastosowanie

```
class Ksiazka
  include Persistence
  def initialize(store)
    self.restore(store)
  end
end

obj = Ksiazka.new({"autor" => "Breza",
                   "tytul" => "Urząd" })
```

Odczyt z bazy danych

```
SQLite3::Database.open dbname do | db |
  db.results_as_hash = true
  db.query "SELECT * FROM Ksiazki" do | res |
    res.each do | row |
      ksiega = Ksiazka.new(row)
      puts ksiega.to_s
    end
  end
end
```

Zalety programowania dynamicznego

Przykład w Javie

```
class Ksiazka {
    public string tytul;
    public string autor;
}
```

W bazie danych mamy

tytul	autor
Brama	Breza
Ciotka	Bryll
Castorp	Huelle

Zmiana wymagań klienta

Przykład w Javie

```
class Ksiazka {
    public string tytul;
    public string autor;
}
```

W bazie danych mamy

tytul	autor	wydanie
Brama	Breza	3
Ciotka	Bryll	1
Castorp	Huelle	4

Zmiany

- zmiana schematu bazy danych;
- zmiana implementacji wszystkich aplikacji korzystających z tej bazy danych.

Zapis wszystkich obiektów

Jak zapisać wszystkie obiekty aplikacji?

```
ObjectSpace.each_object do | o |
    o.save if o.respond_to?("save")
end
```

Rozrzerzanie klas

Już zadeklarowane klasy można rozszerzać

Rozrzerzanie klas

Już zadeklarowane klasy można rozszerzać

Przykład

```
class Integer
    def nastepnik
        self + 1
    end
end
```

Plan wykładu

Zapisywanie i odtwarzanie obiektu — przypomnienie

Wersja z dziedziczeniem

```
class Saveable
  def save
  end
  def restore
  end
end
```

Wersja z Zarządcą

```
class Zarzadca
  def save(obj)
  end
  def restore
  end
end
```

Wersja mix-inowa (tylko Ruby i kilka innych języków)

```
module Persistence
end

class Ksiazka
  include Persistence
end
```

Programy konsolowe

Schemat

```
done = false
while not done
  msg = gets.chomp!
  puts "Podałeś #{msg}"
  done = true if info == "koniec"
end
puts "Koniec"
```

Wzorzec *Template* w Ruby

```ruby
class SzkieletAplikacji
  def run
    self.init
    while not @juz_koniec
      self.petla
    end
    self.zakoncz
  end
end
```

Wzorzec *Template* w Ruby – zastosowanie

```ruby
class PrawdziwaAplikacja < SzkieletAplikacji
  def init
    puts "Zaczynamy"
    @juz_koniec = false
  end

  def petla
    msg = gets.chomp!
    puts "Podałeś #{msg}"
    @juz_koniec = true if info == "koniec"
  end

  def zakoncz
  end
end
myapp = PrawdziwaAplikacja.new
myapp.run
```

Wzorzec *Template* w Javie

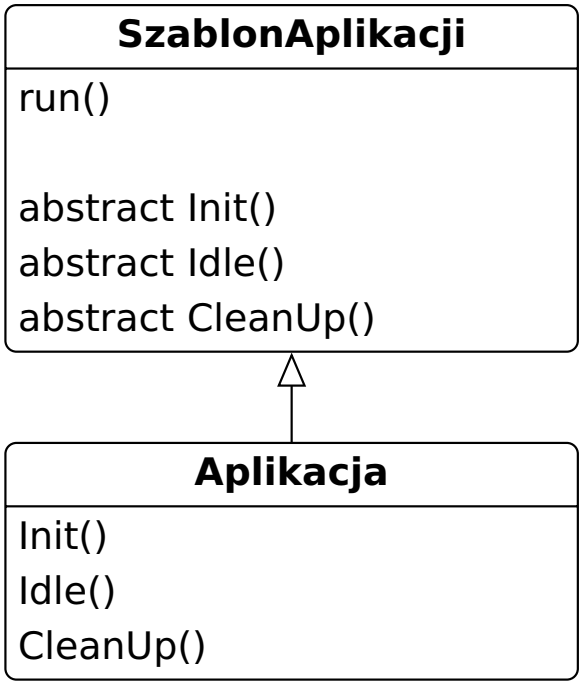
```
public abstract class Aplikacja
{
    protected bool isDone = false
    public void Run()
    {
        Init();
        while (!isDone)
            Idle();
        CleanUp();
    }
}
```

Wzorzec *Template* w Javie

```
public abstract class Aplikacja
{
    protected bool isDone = false
    protected abstract void Init();
    protected abstract void Idle();
    protected abstract void CleanUp();

    public void Run()
    {
        Init();
        while (!isDone)
            Idle();
        CleanUp();
    }
}
```

Schemat rozwiązania



Wzorzec *Template* w Javie

```
public class PrawdziwaAplikacja extends Aplikacja
{
  public static void Main()
  {
    new PrawdziwaAplikacja().Run();
  }
```

Wzorzec *Template* w Javie

```
public class PrawdziwaAplikacja extends Aplikacja
{
  public static void Main()
  {
    new PrawdziwaAplikacja().Run();
  }
  protected void Init() { ... }
  protected void Idle() { ... }
  protected void CleanUp() { ... }
}
```

Refleksja nad rozwiązaniem

Czy to uproszczenie czy skomplikowanie problemu?

Inne podejście — Strategia

```
class AplikacjaStrategia

  def initialize(app)
    @app = app
  end

  def run
    @app.init
    while not @app.juz_koniec
      @app.petla
    end
    @app.zakoncz
  end
end
```

Wzorzec *Strategy* w Ruby

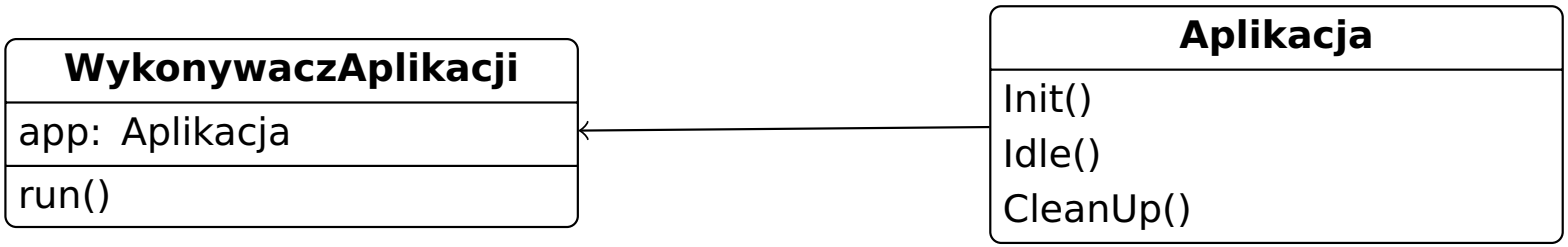
```ruby
class PrawdziwaAplikacja
  def init
    puts "Zaczynamy"
    @juz_koniec = false
  end
  def petla
    msg = gets.chomp!
    puts "Podałeś #{msg}"
    @juz_koniec = true if info == "koniec"
  end
  def zakoncz
    puts "Koniec"
  end
end

myapp = AplikacjaStrategia.new(PrawdziwaAplikacja.new)
myapp.run
```

Wzorzec Strategia w Javie

```
public class WykonywaczAplikacji
{
  private Aplikacja app;
  public WykonywaczAplikacji(Aplikacja app) {
    this.app = app;
  }
  public void run()
  {
    this.app.Init();
    while (!this.app.Done())
      this.app.Idle();
    this.app.CleanUp();
  }
}
```

Schemat rozwiązania



Co to jest Aplikacja

```
public interface Aplikacja
{
  void Init();
  void Idle();
  void CleanUp();
  boolean Done();
}
```

Ocena rozwiązań

- wzorzec Template Method jest prostszy;
- wzorzec Strategy jest elastyczniejszy;
- we wzorcu Strategia mniej interesują nas szczegóły klasy konkretnej.

Przypomnienie: implementacja wątków w Javie

Dziedziczenie (wzorzec *Template*)

```
public class Aplikacja extends Threads {
  public void run() { ... }
}
Aplikacja app = new Aplikacja();
app.start();
```

Składanie (wzorzec *Strategy*)

```
public class Aplikacja implements Runnable {
  public void run() { ... }
}
Thread app = new Thread(new Aplikacja());
app.start();
```