Lecture 2
Object Oriented Programming in Ruby

Aleksander Smywiński-Pohl

Elektroniczne Przetwarzanie Informacji

10 kwietnia 2013

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

The only constant thing in the World is *change*.

How to write change anticipating programs?

OOP techniques

- encapsulation
- abstraction
- delegation
- inheritance
- polymorphism
- dependency injection

OOP principles

- duplication avoidance
- single responsibility principle
- loose coupling
- high cohesion
- Law of Demeter
- and more ...

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Encapsulation

Don't expose the implementation details of the class to the outside world.

- use accessors
- use protected and private methods

Accessors

```ruby
class Post
  attr_accessor :title
end
```

```ruby
class Post
  attr_writer :title
  def title
    translation(@title)
  end
end
```

```ruby
class Post
  def title
    translation(@title)
  end
  def title=(new_title)
    @title = new_title
    create_translations(new_title)
  end
```

```ruby
describe Post do
  subject { Post.new(title: "Ruby") }

  context "default language" do
    it "shows its title in English" do
      subject.title.should == "Ruby"
    end
  end

  context "language set to Polish" do
    it "shows its title in Polish" do
      subject.title.should == "Rubin"
    end
  end
end
```

Protected and private methods

```
class TodoList
  def toggle_task(index)
    raise IllegalArgument if index < 0 || index >= self.size
    @list[index].completed = ! @list[index].completed
  end

  def remove_task(index)
    raise IllegalArgument if index < 0 || index >= self.size
    @list.delete(index)
  end
end
```

Protected and private methods

```ruby
class TodoList
  def toggle_task(index)
    check_index(index)
    @list[index].completed = ! @list[index].completed
  end

  def remove_task(index)
    check_index(index)
    @list.delete(index)
  end

  protected
  def check_index(index)
    raise IllegalArgument if index < 0 || index >= self.size
  end
end
```

Rails pathology

```
# controller
def show
  @post = Post.find(params[:id])
end

# view
<%= @post.title %>
```

Alternative – Decent exposure

https://github.com/voxdolo/decent_exposure

```
# controller
class Controller
  expose(:post)

  def create
    if post.save
      redirect_to(post)
    else
      render :new
    end
  end
end

# view
<%= post.title %>
```

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Abstraction

Divide the system responsibilities into meaningful abstractions.

- classes
- methods
- modules
- design patterns
- services

Class abstraction

```ruby
class TodoList
  def completed?(index)
    @task_status[index]
  end

  def toggle_task(index)
    @task_status[index] = ! @task_status[index]
  end

  def task_name(index)
    @tasks[index]
  end

  def <<(task_name)
    @tasks << task_name
    @task_status << false
  end

  def delete(index)
    @tasks.delete(index)
    @task_status.delete(index)
  end
end
```

Class abstraction

```ruby
class TodoList
  def completed?(index)
    @tasks[index].completed?
  end
  def toggle_task(index)
    @tasks[index].toggle
  end
  def task_name(index)
    @tasks[index].name
  end
  def <<(task_name)
    @tasks << Task.new(task_name)
  end
  def delete(index)
    @tasks.delete(index)
  end
end
```

```ruby
class Task
  attr_reader :name

  def initialize(name)
    @name = name
    @completed = false
  end

  def completed?
    @completed
  end

  def toggle
    @completed = ! @completed
  end
end
```

Method abstraction

```
# show.html.erb
<%= post.user.first_name + " " + post.user.last_name %>

# index.html.erb
<% posts.each do |post| %>
  <%= post.user.first_name + " " + post.user.last_name %>
<% end %>
```

Method abstraction

```
# show.html.erb
<%= post.user.full_name %>

# index.html.erb
<% posts.each do |post| %>
  <%= post.user.full_name %>
<% end %>
```

```
class User
  def full_name
    self.first_name + " " +
      self.last_name
  end
end
```

Module abstraction

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

  def >(other)
    (self <=> other) > 0
  end
end

class Post
  include Comparable

  def <=>(other)
    self.pub_date <=> other.pub_date
  end
end
```

```ruby
post1 = Post.new(pub_date: Time.now)
post2 = Post.new(pub_date: 1.day.ago)
post1 < post2 # => false
```

Controller – design pattern example

Controller is a common abstraction in applications with GUI. It mediates between the View and the Business Model. It defines actions that operate on the Model, which for the outside world looks like cohesive resource. In fact it might use many model classes to achieve this goal, but for the outside world it doesn't matter.

Service abstraction

Service in SOA (Service Oriented Architecture) is a bunch of controllers that are accessible from the outside, that provide some cohesive set of features for the user of the service. E.g. the Wallet application could define the following services:

- Wallet – buy and sell currencies and stocks
- Banker – define and manage your bank accounts
- Exchanger – define and manage stock exchanges

Services are defined from the end-user perspective. They are not layers of the application (like DB service, computation service, etc.) but they divided the application vertically.

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Delegation means passing the message received by an object to another object, that is more suitable for performing the task.

It means that the class understands given message, but doesn't perform the work, that is associated with it. It helps in fulfilling the Law of Demeter.

Manual delegation

```ruby
class TodoList
  def initialize
    @items = []
  end

  def size
    @itmes.size
  end

  def empty?
    @items.empty?
  end

  def first
    @items.first
  end
end
```

SimpleDelegator

```ruby
class Task
  attr_accessor :title, :completed

  def initialize(title)
    @title = title
    @completed = false
  end

  def completed?
    self.completed
  end

  def complete
    @completed = true
  end
end
```

```ruby
require 'delegate'

class TextFormatter < SimpleDelegator
  def formatted
    state = self.completed? ? "x" : " "
    "[#{state}] #{self.title}"
  end
end

task = Task.new("Buy toilet paper")
task = TextFormatter.new(task)
task.formatted
#=> "[ ] Buy toilet paper"
task.title
#=> "Buy toilet paper"
task.completed?
#=> false
```

Forwardable

```ruby
require 'forwardable'

class TodoList
  def_delegators :@items, :size, :empty?, :first, :last

  def initialize(items=[])
    @items = items
  end
end

list = TodoList.new
list.size #=> delegates to @items.size
```

ActiveSupport Module extension

```ruby
class TodoList
  delegate :size, :empty?, :first, :last, :to => :@items
end

list = TodoList.new
list.size #=> @items.size

class Post < ActiveRecord::Base
  belongs_to :user

  delegate :name, :to => :user, :prefix => true, :allow_nil => true
end

user = User.new(name: "John")
post = Post.new(user: user)
post.user_name
#=> "John"

post = Post.new
post.user_name
#=> nil
```

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Inheritance allows for defining type hierarchies. Common behavior is defined in more abstract (parent) classes, while specific behavior in more concrete (children) classes.

Children share (by inheritance) the behavior defined in the parent class.

```ruby
class Animal
  def eat
    "kill some living"
  end
end

class Mammal < Animal
  def feed_children
    "use breast"
  end
end

class Dog < Mammal
  def make_sound
    "bark"
  end
end

class Cat < Mammal
  def make_sound
    "meow"
  end
```

```ruby
my_dog = Dog.new
my_dog.eat
#=> "kill some living"
my_dog.feed_children
#=> "use breast"
my_dog.make_sound
#=> "bark"

my_cat = Cat.new
my_cat.eat
#=> "kill some living"
my_cat.feed_children
#=> "use breast"
my_cat.make_sound
#=> "meow"
```

```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
```

ActiveRecord::Base

```ruby
class Post < ActiveRecord::Base
  # attributes title, body
  belongs_to :user
end

post = Post.new(:title => "Title", :body => "Some text")
post.title # => "Title"
post.body  # => "Some text"
post.user  # => nil

post.save
post.destroy
```

```ruby
class Task
  def initialize(name,description)
    @name = name
    @description = description
    @completed = false
  end

  def to_s
    @name
  end
  # or
  def to_s
    "#{@name}: #{@description[0..30]}"
  end
end

class FormattedTask < Task
  def to_s
    state = @completed ? "x" : " "
    "[#{state}] #{super}"
  end
end
```

```ruby
task = FormattedTask.
  new("Tesco","Buy toilet paper")

task.to_s
# [ ] Tesco
# or
# [ ] Tesco: Buy toilet paper
```

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Polymorphism

Polymophism is the property of objects, allowing them to respond differently for the same message, depending on their type.

In Ruby you will often hear the term **duck-typing**: if something walks like a duck and quacks like a duck it is treated as if it was a duck.

In Ruby polymorphism doesn't require inheritance.

Duck typing example

```ruby
def print_collection(collection)
  collection.each do |element|
    puts "- #{element}"
  end
end

print_collection([1,3,5])
# - 1
# - 3
# - 5
print_collection(1..3)
# - 1
# - 2
# - 3
```

```ruby
class Figure
  attr_accessor :x, :y
  def initialize(x,y)
    @x, @y = x, y
  end
  def move(x_delta,y_delta)
    @x += x_delta
    @y += y_delta
  end
end

def Apple < Figure
  def draw
    puts "*"
  end
end

def Snake < Figure
  def draw
    puts "----->"
  end
end
```

```ruby
apple = Apple.new(0,0)
snake = Sname.new(10,0)
objects = [apple,snake]

loop do
  snake.move(1,0)
  objects.each do |object|
    object.draw
  end
end
```

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Dependency Injection

Dependency injection allows for replacing the dependencies of the class with their alternatives during compilation or at run time.

It removes all hard-coded cooperators of the class.

Dependency example

```ruby
class TodoList
  def initalize
    @items = []
  end

  def <<(name)
    @items << Task.new(name)
  end
end
```

`Task` is a global (name), which can't be replaced. It is a hard-coded dependency of `TodoList`.

Dependency removal

```ruby
class TodoList
  def initialize(options={})
    @items = []
    @task_factory = options[:task_factory] || Task
  end

  def <<(name)
    @items << @task_factory.new(name)
  end
end
```

Alternative dependency

```ruby
require_relative 'spec_helper'
require_relative '../../lib/todo_list'

stub_class 'Task'

describe TodoList do
  subject(:list)      { TodoList.new(:task_factory => task_factory) }
  let(:task_factory) { stub!.new(task_name) { task }.subject }
  let(:task_name)     { "Buy toilet paper" }
  let(:task)          { Struct.new(:title,:completed).(task_name,false) }
end
```

Dependency setter

```ruby
class TodoList
  attr_writer :task_source

  def initalize
    @items = []
  end

  def <<(name)
    @items << task_source.call(name)
  end

  private
  def task_source
    @task_source ||= Task.public_method(:new)
  end
end

list = TodoList.new
fake_task_class = Struct.new(:title)
list.task_source = fake_task_class.public_method(:new)
```

Default dependency

```ruby
class Post
  def publish(clock=DateTime)
    self.pub_date = clock.now
    # ...
  end
end

class FixedClock
  def initialize(date)
    @date = date
  end

  def now
    DateTime.parse(@date)
  end
end

class DeleyedClock
  def now
    DateTime.now + 24.hours
  end
end
```

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

Duplication avoidance

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. *Wikipedia*

It is called DRY – Don't repeat yourself.

This applies both to **data** and **processing**.

DRY data

DB normalization – removal of duplicated data

name	price	price + VAT	VAT	VAT rate
starter	10	12.30	2.30	23
vegetarian dish	20	24.60	4.60	23
main dish	25	31.90	6.90	23

DRY data

name	price	VAT rate id
starter	10	1
vegetarian dish	20	1
main dish	25	1

id	VAT rate
1	23
2	5
3	0

DRY code

```
class TodoList
  def toggle_task(index)
    raise IllegalArgument if index < 0 || index >= self.size
    @list[index].completed = ! @list[index].completed
  end

  def remove_task(index)
    raise IllegalArgument if index < 0 || index >= self.size
    @list.delete(index)
  end
end
```

DRY code

```
class TodoList
  def toggle_task(index)
    check_index(index)
    @list[index].completed = ! @list[index].completed
  end

  def remove_task(index)
    check_index(index)
    @list.delete(index)
  end

  protected
  def check_index(index)
    raise IllegalArgument if index < 0 || index >= self.size
  end
end
```

Single Responsibility Principle

Single responsibility principle states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. *Wikipedia*

Responsibility is understood as **a reason for change**.
Robert C. Martin.

```ruby
class Product
  # Creates new product by parsing the XML
  # representation of the product found
  # under the +url+.
  def self.import(url)
  end

  # Stores the product in local database.
  def save
  end

  # Converts the product to HTML representation.
  def render(context)
  end
end
```

Responsibilites of the Product class

What can change?

- the organization and format of the imported file
  e.g. individual files might be replaced with aggregated documents
- the database used to store the product
  e.g. a relational DB might be replaced with document-based one
- the presentation of the product
  e.g. HTML might be replaced with JSON

```ruby
class ProductParser
  # Parses the product definition and
  # returns a struct containing the product
  # name and price.
  def parse(url)
  end
end

class Product
  # Stores the product in the database.
  def save
  end
end

class ProductPresenter
  # Renders the product as a list item.
  def render(context)
  end
end
```

```ruby
class TodoList
  def completed?(index)
    @task_status[index]
  end

  def toggle_task(index)
    @task_status[index] = ! @task_status[index]
  end

  def task_name(index)
    @tasks[index]
  end

  def <<(task_name)
    @tasks << task_name
    @task_status << false
  end

  def delete(index)
    @tasks.delete(index)
    @task_status.delete(index)
  end
end
```

Responsibilities of the TodoList class

What can change?

- how the list is persisted – in memory vs. via database
- task lifecycle – e.g. three states: fresh, in progress, finished

```ruby
class TodoList
  def [](index)
    @task[index]
  end

  def <<(task_name)
    @tasks << Task.new(task_name)
  end

  def delete(index)
    @tasks.delete(index)
  end
end
```

```ruby
class Task
  attr_reader :name

  def initialize(name)
    @name = name
    @completed = false
  end

  def completed?
    @completed
  end

  def toggle
    @completed = ! @completed
  end
end
```

High cohesion

Cohesion is a measure of how strongly-related or focused the responsibilities of a single module are. As applied to object-oriented programming, if the methods that serve the given class tend to be similar in many aspects, then the class is said to have high cohesion. *Wikipedia*

Loose coupling

A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. *Wikipedia*

Law of Demeter

For all classes C and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:

- The argument classes of M (including C).
- The instance variable classes of C.

Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.

Law of Demeter – pragmatic formulation

- Your method can call other methods in its class directly.
- Your method can call methods on its own fields directly (but not on the fields' fields).
- When your method takes parameters, your method can call methods on those parameters directly.
- When your method creates local objects, that method can call methods on the local objects.

Example

```ruby
class Post < ActiveRecord::Base
  belongs_to :user

  def user_full_name
    user.profiles.first.personal_data.full_name
  end
end
```

Classes used in `user_full_name`:

- `User`
- `Profile`
- `PersonalData`

```ruby
class Post < ActiveRecord::Base
  belongs_to :user

  def user_full_name
    user.full_name
  end
end
```

Only `User` class is used – it's ok, since the object is returned by `Post`'s own method.

```ruby
class User < ActiveRecord::Base
  has_many :posts
  has_many :profiles

  def full_name
    self.personal_data.full_name
  end

  def personal_data
    self.default_profile.personal_data
  end

  def default_profile
    self.profiles.first
  end
end
```

What about this?

```ruby
def format(line)
  line.chomp.strip.capitalize
end
```

It's ok, since we only have one class – `String`. `line`, the method parameter, is an instance of `String`.

LoD is more than dot counting.

Agenda

Principles of Object Oriented Programming

Encapsulation

Abstraction

Delegation

Inheritance

Polymorphism

Dependency Injection

Principles

References

References

- *Object Oriented Software Construction*, Bertrand Meyer
- *Growing Object-Oriented Software, Guided by Tests*, Steve Freeman
- *Clean code: A Handbook of Agile Software Craftsmanship*, Robert C. Martin
- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

References

- *Objects on Rails*, Avdi Grimm
- *Refactoring: Ruby Edition*, Jay Fields, Shane Harvie, Martin Fowler, Kent Beck
- *Refactoring in Ruby*, William C. Wake, Kevin Rutherford
- *Law of Demeter*, Avdi Grimm's blog