SOLID Rails: A Phantom Pain

Yulia Oletskaya
Nov 30, 2018 · 7 min read

You probably heard of object-oriented design principles. Yes, the ones defined by the SOLID acronym created by Bob Martin and Michael Feathers. SOLID aims to help engineers to write easily maintainable code. Although those principles have long proven their effectiveness, sometimes it's hard to follow them. Or even impossible. Especially, if you're developing a Rails application.

It's complicated by the fact that engineers usually interpret SOLID principles in a slightly different way, and moreover, each one (me included) has their own ideas of the cases where and when the best object-oriented design practices do not make sense to be applied. After all, the principles aren't strict rules, they are simply the guidelines.

I have faced with the complexity and versatility of a decision which is the most convenient design solution not so long ago myself. I was attending a job interview and I was asked to refactor a piece of code similar to this one:

```ruby
class Agency < ApplicationRecord
  # ...

  # Credit
  #
  validates :credit, { greater_than_or_equal_to: 0 }

  def credit_increase(percentage)
   self.credit = credit + percentage * credit / 100.0
    # ...
  end

  def credit_decrease(percentage)
    # ...
  end

  def credit_recount
    # ...
  end

  # ...
end
```

example_int_gist.rb hosted with ❤ by GitHub view raw

My very first idea was to implement a decorator class and to move all the logic related to the credit there. In this way, we could decorate the class only in those places where this business logic is required, achieving lousy coupling, skinny models, etc, etc.

```ruby
class CreditDecorator < SimpleDelegator
  def initialize(agency)
    @agency = agency
    super
  end

  def credit_increase(percentage)
    # ...
  end

  def credit_decrease(percentage)
    # ...
  end

  def credit_recount
    # ...
  end
end

# Usage

agency = Agency.find(id)
decorated_agency = CreditDecorator.new(agency)
decorated_agency.increase_credit(0.1)
```

example_int_gist_2.rb hosted with ❤ by GitHub view raw

But although this was accepted as a possible option, the interviewer pointed out that the usage of the decorated class requires too much code on the “client-side”. And suggested the next solution.

```ruby
class CreditHandler
  attr_reader :agency

  def initialize(agency)
    @agency = agency
  end

  def increase(percentage)
```

```ruby
    def increase(percentage)
      # ...
    end

    def decrease(percentage)
      # ...
    end

    def recount
      # ...
    end

    def value
      agency.credit
    end
  end

  class Agency < ApplicationRecord
    # ...

    # Credit
    #

    def credit
      @credit ||= CreditHandler.new(self)
    end
    # ...
  end

  # Usage

  agency = Agency.find(id)
  agency.credit.increase(0.1)
```

example_int_gist_3.rb hosted with ❤ by GitHub view raw

The “client-side” usage appears to be more elegant. And this led us to quite an interesting conversation.

Int — *With the CreditHandler implemented as the above we can treat the credit attribute as an object and benefit from the OOP perspective.*
Me — *This solution does not benefit from the OOP perspective. It breaks SOLID principles, namely Dependency Inversion. Dependence on abstraction, not a specific implementation. Dependencies need to be passed either through the constructor or through the property, we shouldn’t hard-code classes into each other.*
Int — *Well, you can go for a dependency injection. And implement it as*

```
Agency.new(credit_handler: CreditHandler)
def credit; @credit ||= credit_handler.new(self); end
```

Int — *It does not change the essence. Looks like overengineering and premature optimization for me. The original proposed implementation indeed breaks SOLID, but those 5 are the design principles, it does not break the OOP rules itself (4 of them). In my opinion, follow SOLID means not to apply the practices immediately, but to apply them when necessary; that is, as soon as it becomes necessary to use different flows for the credit attribute in the example.*

And it made me think. How does one make a decision what is an overengineering and what is a reasonable solution? Should the engineers follow the best practices from the beginning? Or it is an overcomplication and it's best to apply only a limited set of recommendations? How do I make those decisions and how often I follow the best practices? How often I don't? And, gradually, it led me to a realization, that I'm, as a Ruby on Rails developer, break the SOLID recommendations every single day! Well, at least some of them. And it's while trying hard to follow the best practices whenever I see an opportunity. If you're a Ruby developer then most likely you break the guidelines on a daily basis too.

Let's list the SOLID principles to take a closer look at them before we'll try to understand when and how do we violate them:

- S — Single responsibility principle (SRP). *A class should have only one reason to change.*
- O — Open/closed principle (OCP). *Software entities should be open for extension, but closed for modification*
- L — Liskov substitution principle (LSP). *If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.*
- I — Interface segregation principle (ISP). *No client should be forced to depend on methods it does not use.*
- D — Dependency inversion principle (DIP). *High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

SRP

That principle is often being misinterpreted.
Almost every Ruby on Rails developer at least once said something like: “Ah, single responsibility principle? Yeah, ActiveRecord breaks this one, it has way too many responsibilities”. Active Record indeed is an example of a God-object anti-pattern, it knows too much and does too much. But it has nothing to do with SRP from SOLID.
The problem lays in the interpretation of “*reason*” word from “*A class should have only one reason to change.*”. Developers tend to think that each meaningful function of an object is some kind of a “*reason*”, while SRP itself refers to the business layer of things. ActiveRecord is responsible for validations, database adapters, caching, combining SQL queries, and many other things, but from a business logic perspective, ActiveRecord’s User model is responsible only for storing users in a database correctly.
Let’s take a look at the next example to make it a bit more clear.

```ruby
class Employee < ApplicationRecord
  # ...
end

class Manager < Employee
  # ...
end

class Lawyer < Employee
  # ...
end

class Accountant < Employee
  # ...
end

class PaymentCalculator
  # ...

  def salary_for(employee, time_range)
    # ...
    calculate
  end

  def benefit_for(employee)
    # ...
    calculate
  end

```

```ruby
    private

    def calculate
      # ...
    end
  end
```

srp_1.rb hosted with ❤ by GitHub — view raw

There are several types of employees and several types of payments. From the business point of view, the private method `calculate` has to handle too much — it calculates salaries and benefits for lawyers, managers, and accountants.
Let's say at some point the logic of calculation of benefits for lawyers specifically should be changed, while the rest of the calculations should stay the same. Can you imagine how much confusion and possible errors can potentially bring this kind of business requirement? Ouch.
Shortly speaking, in most cases the business domain stays behind the violation of SRP.

OCP

The Ruby itself has a very vague concept of a "closed". And while everything is open for extension, everything is open for modification as well.
I used to think that it's out of my concerns as long as I do not directly monkey-patch other classes. But let's take a closer look at the next example.

```ruby
describe AuthService do
  # ...
  before { allow_any_instance_of(Dnsruby::Resolver).to receive(:query).and_return(messag

  # ...

  before { allow(AuthService).to receive(:sso_url).and_return(sso_auth_url) }
  # ...
end
```

ocp_1.rb hosted with ❤ by GitHub — view raw

In the example, the actual result is that the classes in the namespace are being replaced by Rspec's mocks and by this — modified. I'm a happy user of Rspec and it simplifies my life dramatically. But the fact is each time I'm using its mocks I'm modifying the upper class in the hierarchy by the lower one, which is supposed to depend on the upper one and is supposed only to extend it without any modifications.

I'm not saying we must stop using Rspec, the gem is absolutely great and perfectly fits for testing needs, I just need to admit that it violates the OCP in SOLID.

LSP

LSP aims to ensure that the inheritance is used correctly. This principle stands out from others within this article, as it has nothing special in terms of the Ruby world. No common misusages, no misconceptions, seems like at least LSP is usually respected among Rubyists. You can read more about LSP on this article, I have nothing to add.

ISP

The ISP states that a client should not be forced to depend on methods that it does not use. Interfaces Segregation is considered to be a problem of a programming language, not an architecture. Ruby together with all other dynamically typed languages cannot violate ISP at any mean.

DIP

This is the one which started this article. High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. That's the principle which is violated the most in Ruby on Rails community. And also Sinatra community. And well, I think in many other Ruby communities independently of a framework they use, web-centered or not.
Let's take a look at the next example.

```ruby
class UsersController < ApplicationController
  # ...
  before_action :auth_user

  # ...

  def show
    @user = User.find(params[:id])
    @customers_portals = UserCustomersPortalsQuery.new(@user).all
  end

  private

  def auth_user
    return true if current_user.present?

    redirect_to AuthService.sso_url({
```

```
      callback_url: new_session_url,
      original_url: safe_url
    })
  end

  # ...
end
```

dip_1.rb hosted with ❤ by GitHub — view raw

`UsersController` — a high-level class depends on multiple low-level classes — `User`, `UserCustomersPortalsQuery`, `AuthService`.

The band of `UsersController` and `User` model is not such a bad thing though. The abuse of Dependency Injection is considered to be an anti-pattern itself. Some classes meant to be coupled, and in most cases `UsersController` does not make sense without `User` model. But we all can agree that the coupling of the controller with the `AuthService` does not feel so right.

In the enterprise world of .NET, this problem is usually solved by applying Inversion of Control and Dependency Injection patterns.
IoC implies that within your application there is an explicit point of a request entry, where you can explicitly call a constructor for a specific controller and explicitly pass all the classes it needs through Dependency Injection.

As a benefit — developers can easily substitute dependencies by other classes in the IoC. And by this, for example, easily replace the real classes with mocks in the test env.

It's quite fun, as in the Ruby world we usually do not substitute real classes with mocks via DI. We're already using Rspec and mocking the actual classes by the violation of OCP.

Just in order to help to get a very basic idea of what IoC looks like here's a bit of ASP.NET MVC code. The example is quick and simple, and hopefully can be understood by people with zero experience in .NET (like me).

```
// Within the Application_Start() event handler in the Global.asax.cs

ControllerBuilder.Current.SetControllerFactory(new DependencyControllerFactory());

ObjectFactory.Initialize(registry => registry
        .ForRequestedType<IContactRepository>()
         TheDefaultIsConcreteType<ContactManagerLingRepository>());
```

```
8
9  // The repository class which is passed through DI to the controller
10
11 public class ContactManagerEntityRepository : IContactRepository
12 {
13   public IList<Contact> GetContacts()
14   {
15     ContactManagerEntities entities = new ContactManagerEntities();
16     return entities.ContactSet.ToList();
17   }
18 }
19
20 // The IoC
21
22 using System;
23 using System.Web.Mvc;
24 using StructureMap;
25
26 namespace ContactManager.IoC
27 {
28   public class DependencyControllerFactory : DefaultControllerFactory
29   {
30     protected override IController GetControllerInstance(Type controllerType)
31     {
32         return ObjectFactory.GetInstance(controllerType) as Controller;
33     }
34   }
35 }
36
37 // The controller itself
38
39 public class ContactController : Controller
40 {
41   private readonly IContactRepository iRepository;
42
43   public ContactController(IContactRepository repository)
44   {
45     iRepository = repository;
46   }
47
48   public ActionResult Index()
49   {
50     return View(iRepository.GetContacts());
51   }
52 }
53
```

Controllers are not the root of all evil. I'm sure more or less the same violation of DIP can be found in, for example, ActiveJob classes.

Conclusion

It should be mentioned, that Bob Martin himself noted that the recommendations are not so strict in dynamic languages.

Violation of DIP helps to write code fast. Violation of OCP allows to reliably test the code. Although, the realization of mocks is tricky and as I can imagine is quite hard to maintain. Anyway, from some point of view, it proves that it's possible to write maintainable software without a strict following of SOLID.

Yet, I believe that Ruby developers should stick more to the object-oriented design practices. Use Dependency Injection more often and write maintainable easy-extendable loose-couped code. It simplifies life a lot when properly handled. Cheers.

Software Development | Ruby | Ruby on Rails | Object Oriented | Object Oriented Design