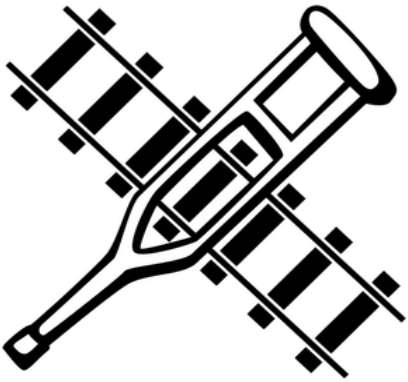
Stop being Rails-developer



pre-alpha

Stop being Rails developer

Ivan Nemytchenko

I think this book should have been written in 2011. But nobody had written it, and I still see a huge demand for this topic to be covered on conferences and inside people's code. So, this book is finally going to happen.

Moreover, it is good that I started to write in 2015. Because now we already have alternatives to Rails-way approach, like ROM.rb & Lotus.

Contents

Intro

There's a whole bunch of smart people telling us why we shouldn't use Ruby on Rails. The problem is that it seems those guys are too smart. They use very smart words and high-level concepts, assuming that everybody knows what it means.

But come on, if we would be that smart, we wouldn't even be considering working as another form creator in a million-first Rails app. We would be flying under the sky with gods, hacking around and moving mountains with the power of Clojure or Erlang.

Remember the day you switched from PHP to Rails. What a pleasure it was! The Chaos has been scattered with The Power of Mighty MVC. Everything had its own place and its own responsibility. What a wonderful feeling it was.

But something went wrong, right? Suddenly it becomes hard to maintain the application, that you crafted with love for months. And you starting to think that probably something is wrong with you. Probably you need to try harder to follow Rails way.

Or maybe it was just an accident. In your next Rails application, everything will be different. You're gonna show

those haters, that they are wrong! Rails is awesome! It simplifies my life as a developer!

But time passes, and history repeats. What the f*ck? Am I a shitty developer? I was doing my best and trying so hard!!

Calm down. You're not alone. There is a whole generation of Rails developers like me and you, who are in the same situation. Sit closer, brother. It is going to take some time.

What is wrong with being Rails developer

Principles misunderstanding

Let's assume for now, what maybe we don't completely understand something. And this little something makes us use Rails in a wrong way.

At the early days, we accepted few principles unconditionally because they sound like axioms. Here's some of them:

- DRY - "Do not repeat yourself"
- KISS - "Keep it simple, stupid"
- Convention over configuration
- Fat model, skinny controller
- Rails is not your application
- YAGNI - "You aren't gonna need it"

People use principles and acronyms to compress meanings. By doing so, there's a risk for them to be misunderstood. And I think this is exactly what's happened to with some people in Rails community.

So let's try to decompress them, and see what might be wrong with our understanding.

DRY - Do not repeat yourself

This is a simple one, isn't it? Just don't duplicate the code. Whenever you see duplication, you should immediately make a function or a class and reuse it as much as possible. Sounds smart, isn't it?

The problem here that it tells us nothing about when it is a good time to apply this principle. And if nothing told, we assume that it should be applied all the time.

And, maybe you forgot, but everything has its price. And the price of duplication removing is more relations.

Guess what's happening when you apply DRY principle at the early stages of application development? You're still not sure about what is your application is going to do, and how everything should be organized. But you already removed all possible duplications and made things related to each other.

But here's the trick: number of relations increases the complexity of a system.

And this is what Sandi Metz means when she says that duplication is much cheaper than wrong abstraction.

By reducing the number of duplications, you're introducing abstractions. Your code becomes dependent on those abstractions.

And if they were wrong, you'll be stumbling upon them every single time you need to change your code. And they are wrong if you introduced them when your system understanding was not yet mature enough.

This thing is also called *accidental complexity*. By contract with *essential complexity*, it has nothing in common with the complexity of your business domain. It happens because you decide to organize everything in a specific way.

For sure, you shouldn't be wanting that additional pile of complexity in your app. Let's just keep the stuff simple, right?

KISS - Keep it simple, stupid

> The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore simplicity should

> be a key goal in design and unnecessary complexity should be avoided

Sounds good, right? And we use it to advocate for Rails. Because Rails is so simple, right?

It lets you forget about databases and tons of other technical details which have nothing in common with your application business logic. That's convenient! We can concentrate on just one thing, and all the other stuff we have for free. Rails world is so cool, that we don't even had a whole class of problems, which millions of Java programmers struggling every day!

But… is it true?

Have you tried to take a look at the ActiveRecord source code? It is insane. It is like a horde of gnomes inside of a beautiful box with two buttons and couple of settings, who do all the kind of things to satisfy your request every time.

Here are some of the things, ActiveRecord gnomes had to do for you:

- input data coercion
- setting default values
- input data validations
- interaction with the database
- handling nested structures
- callbacks(before_save, etc...)

Look like tons of work. Doesn't look that simple, right? It just has a convenient interface, but inside it is crazy as hell.

But why should you care? Why not just let the Rails core team members suffer from that complexity, and just keep using it in order to keep things -simple- convenient for us?

Well, at some point when you'll have a pretty big system, you find yourself wanting to have unthinkable. Something like polymorphic STI model which belongs to another polymorphic model through another model, which also has some valuable JSON data stored in Postgres using `hstore`.

Of course, there's no reason to want something like this when you develop an application from scratch. But your app is mature and you already have all those associations and other stuff. And it doesn't contradict with Rails-way, quite the contrary. So why not?

If you ever try to do something like this, you quickly start to get all kinds of unpredictable errors and misbehavior.
It is just because those gnomes under the hood go crazy of the complexity they need to handle. Or in other words, nobody from core team supposed that it can be used in such an interesting way. And nobody cared to implement that specific case.

And at this point it becomes your personal problem, not Rails core team problem. And you have two options:

- jump into ActiveRecord sources, and try to be a hero, who covers that specific scenario and becomes *RAILS CORE CONTRIBUTOR*!!!
- reorganize your associations to make things little bit simpler

Both ways are usually pretty painful. Because of the complexity. Because **IT IS** complex. This complexity is just

hidden under the mask of convenience from you until some moment.

So let's admit it - Rails is not simple, it is convenient.

One of its conveniences is that we have so little things to configure.

Conventions over configurations

What could be wrong with conventions? We should praise Rails for giving us naming convention about database stuff, that makes sense.

But some developers take it too literally and go mad with it. They introduce their own extra conventions. And they are usually implemented with hardly readable metaprogramming code.

The bad thing about conventions is that they are implicit. So if there's too many of them, it can be hard to remember why everything is done in a way it is done.

Even default conventions might become too tight for you when your app grows. So just remember that the price of conventions is flexibility.

Fat model, skinny controller

This is a lovely one. I do not remember the author. But in the days we all were suffering from fat controllers,

it sounded like a great blessing for us all. Finally, we're figured it out! We shouldn't put all the business logic into controllers! Lets put all that crap into models, right?

But why do you think life with fat models is going to be easier? It turns out that just moving stuff from one place to another is not enough.

Rails is not your application

The problem with this is not that it is misunderstood. It is not understood at all.

It just doesn't make sense. Here's my Rails *application*. Here's `app` folder. What's wrong?

This is where the idea of *DDD* starts floating around. We'll cover it later, but for now let's not overcomplicate.

Why it is always so painful to upgrade to next version of Rails? Can you imagine your application without Rails? I mean your application has some business logic, right? Why can't you re-use it for some small command-line utility, for example?

It is because such code desperately dependent on Rails. Every single piece of usual Rails application code depends on it and assumes Rails is there to help.

And if you have business logic code intermixed with Rails framework, guess what happens when business logic changes or Rails needs to be upgraded?

The PAIN starts to appear from every single change which should be done!

Ideally you business logic code should have as little points of contact with web framework as possible. And it is a very good question how to achieve this.

Moreover, this is not necessarily one-to-one relation. Your app might want to interact with more than one framework, or there could be more than one app, interacting with the same instance of the framework.

YAGNI - You aren't gonna need it

It looks pretty clear:

> "...a programmer should not add functionality until deemed necessary" (wikipedia[1])

But wait, every ruby developer in the world who is about to build a new web app, taking Rails. Rails provides a lot of functionality, you might not need yet.

What if you don't need persistence layer yet? Wouldn't it be better if you could develop stuff, not thinking about how it is going to be stored in DB? And at one point implement persistence layer, and switch memory storage to MySQL of MongoDB or ElasticSearch or whatever.

1

When we take Rails by default, we make critical decisions about storage too early. And then we become slaves of the storage or even slaves of DB schema.

And what if you don't need web framework at all? Have you ever asked this question to yourself? :)

Prefer composition over inheritance

In ruby we have mixins. And once we learned that inheritance sucks, we tend to switch to modules, assuming it is any better.

I'll just put this tweet here:

Still not convinced? Try this in your console:

```
irb(main):001:0> A = Module.new
=> A
irb(main):002:0> B = Module.new
=> B
irb(main):003:0> C = Class.new
=> C
irb(main):004:0> D = Class.new(C) { include A, \
B }
=> D
irb(main):005:0> D < C
=> true
irb(main):006:0> D < A
=> true
irb(main):007:0> D < B
=> true
irb(main):008:0> D.ancestors
=> [D, A, B, C, Object, Kernel, BasicObject]
```

* * *

How are you so far? I like the way we started to question all the things. Let's continue in the same direction, but take a look at Rails itself.