Theoretical Purity According To OO Dogma Is Not A Goal In Itself

In this book, I'm going to show you several places where Rails gets OOP wrong and works anyway. The goal is not to expose Rails's failures to conform to an infallible orthodoxy; the goal is to learn and understand. Doing OOP wrong works better for Rails than doing OOP right works for many other projects. If we figure out why, we can become better at choosing when to lean on OOP, and when to ignore it.

"The right tool for the job" is a catchphrase among developers, and the more you learn what tools are right for which jobs, the better you become as a developer. Ruby is a multi-paradigm language, and there are times when its functional features are the right tool for the job, and its object-oriented features are not. Every seasoned Ruby developer knows this. But something less obvious is that Rails is a multi-paradigm framework, and requires you to make similar tradeoffs.

James Hague wrote on his blog:

object-oriented programming isn't the fundamental particle of computing that some people want it to be. When blindly applied to problems below an arbitrary complexity threshold, OOP can be verbose and contrived.

the concept of a data object. Ruby is the most object-oriented of all the popular web languages. Why on Earth do `link_to` and the four different `url_for` methods not represent their incredibly consistent data structure as objects **internally**? The answer comes from functional programming; in a language which supports lambdas, there's literally nothing you can do with objects that you can't do with lists as well.* The API for `url_for` is basically just a list:

* Seriously. See: http://www.naildrivin5.com/blog/2012/07/17/adventures-in-functional-programming-with-ruby.html

```
url_for :controller => "foo",
        :action => "bar",
        :thing_which_logs_in => thing
```

That's basically Lisp:

```
(url_for ((controller, foo),
          ((action, bar),
           (thing_which_logs_in, thing)))
```

One of the reasons Rails gets away with disregarding OO dogma, when we're lucky, or completely misrepresenting it, when we're not — think "unit tests," which I'll get to — is that many of the moments where Rails strays from OO dogma are moments that look like Lisp. It's actually a good problem to have.

Let me remind you again of the idea that OO is only valuable above a particular threshold of complexity:

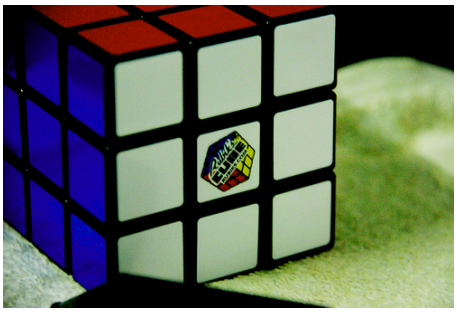
comes to `ActiveRecord` and inheritance. Not only is there an elaborate mechanism for single-table inheritance, there's also this whole thingamajig:

```
class Horse < ActiveRecord::Base
```

That shit is a whole lot weirder than it looks.

I mean seriously, apologies for all the time I spend wandering into the world of NSFW vocabulary in this book— I'm making an effort here and this book is incredibly not-NSFW by my standards — but that fucking shit is as weird as a talking rhinosceros with Rubik's Cubes instead of balls. I mean imagine this happens to you. Say you meet a talking rhinosceros, and he's male, and you notice he has Rubik's Cubes where most male rhinos have balls. Do you ask him why his balls are Rubik's Cubes, or does that only add a whole new layer of social awkwardness to what is already a strange moment? Is it more important to deal with all the unusual ways this rhinosceros varies from other rhinosceri in the first place, and make a strange new friend, and/or learn something about the world — or is it better to consider that rhinos are legendary for their bad tempers and capable of squashing SUVs, and that this particular rhino might be sensitive about his condition, and thus make it your priority to end the conversation peacefully, but quickly, and go somewhere else?

http://www.flickr.com/photos/yanov/2851768907/



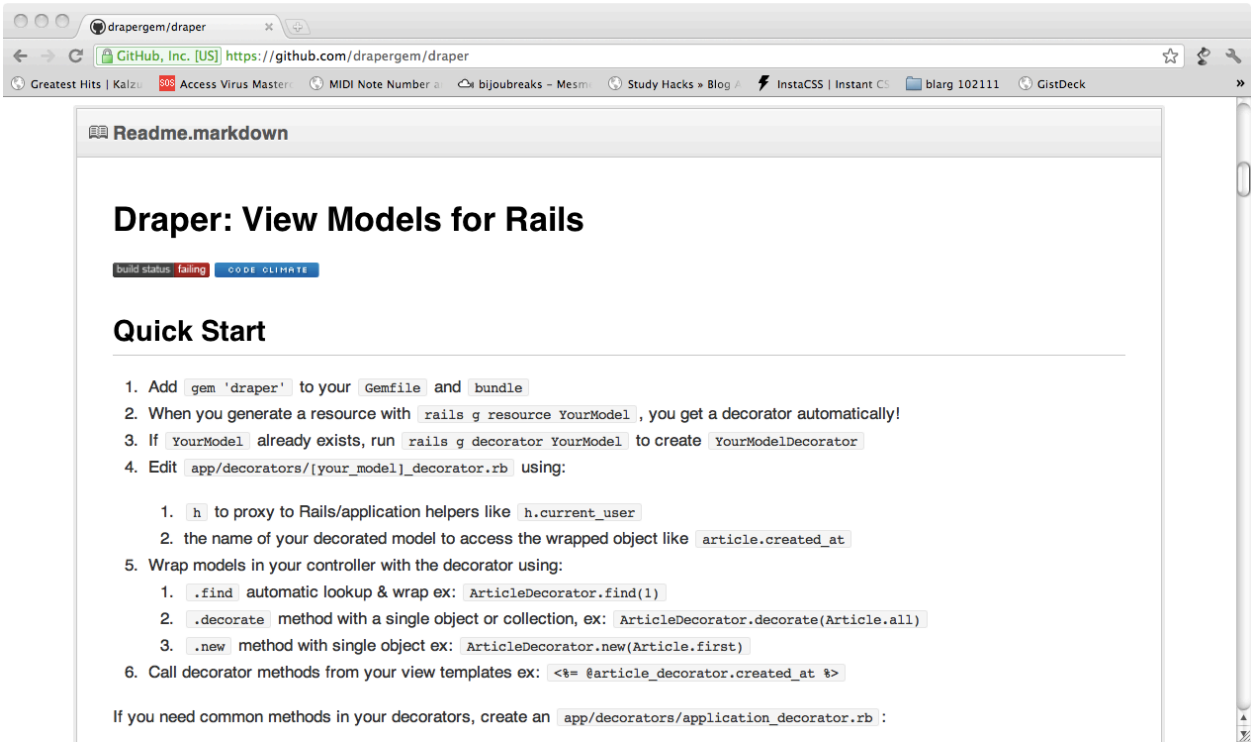
http://www.flickr.com/photos/lowjianwei/2312763892/

application, like a voice-activated UI, would I need this?". If not, consider putting it in a helper or (often better) a View object.

...Rails unfortunately uses the term "view" to describe what are otherwise known as "templates". To avoid ambiguity, I sometimes refer to these View objects as "View Models".

Basically, if you really want to write a classic MVC view in Ruby — and there are sometimes good reasons — you have to call it a View Model or a View Object to avoid confusing people, because when we usually say "view," we mean "template." *

* We have always been at war with Eurasia.



The thing that we call a "view" in Rails operates almost identically to straight-up PHP: a template lives at a particular URL and contains embedded code with no frills. With its own

pretty weird line to blur. It's a pretty weird line to draw in the first place, in fact.

Instance variables pass through from the controller to the so-called view like a spaceship passing through a wormhole into another dimension. You enter the gateway an instance variable on an object, and you exit it a global variable, as far as the "view" is concerned, in a system which looks just like PHP. It's like a fucking StarGate from the disciplined land of OO to the Wild West of unscoped (and unscope-able) variables.

http://www.flickr.com/photos/kretyen/3297935915/

Our real goal in this discussion is not just to reveal the multi-paradigm nature of Rails, nor to criticize its deeply strange perspective on OOP, but to identify the principles behind its compromises. One related goal is to determine when Rails goes OO and when it goes imperative, and why. Some of Rails's more unusual choices, like hiding a Factory inside an inheritance mechanism, almost pretend to be different kinds of code than they actually are, apparently just because it looks cooler.

In both the way that Rails views are not really views, and in the way ActiveRecord "subclasses" are not really subclasses at all, or subclassable, you have this thing where Rails presents a massively oversimplified interface to the app developer, but gets away with it.

CHAPTER 7

How Many Responsibilities Does A Rails Controller Have?

David Bryant Copeland wrote another blog post which opened my eyes to some very old and very total Single Responsibility Principle fail within Rails. The Single Responsibility Principle, in a nutshell: every object in a system should have only one responsibility. Robert C. Martin* introduced the term.

* Many people know Martin as 'Uncle Bob,' a nickname he uses. Apologies, but I'm not actually comfortable referring to him as Uncle Bob, because we're not related.

Copeland examined the idea of a fairly typical Rails controller, which sends a user a welcome note via email after they sign up:

```
class UsersController < ApplicationController
  def create
    @user = User.new(params[:user])
    respond_to do |format|
      if @user.save
        UserMailer.deliver_welcome_email(@user)
      end
    end
  end
end
```

He then pointed out the method's responsibilities:

- *Creates a new User instance from form parameters*
- *Saves the new User to the database*
- *Sends the user an email if the save was successful*
- *Renders the view*

Hansson overlooked the testing ramifications of his solution* and Copeland's is better.

* For an alternate take, see http://gmoeck.github.com/2012/07/09/dont-make-your-code-more-testable.html

Rails controllers turn themselves into data transmissions when their instance variables ship off to the "view." The pitfall here is that classic OO theory has no way to even explain this idea. If we apply the Single Responsibility Principle to a controller, what single responsibility could we possibly articulate? "A controller exists to create an entire microcosm of effectively-global variables within which 'view' code will operate?"

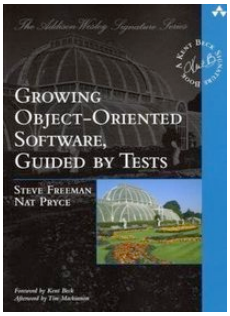
Let's look at Martin's original definition of the Single Responsibility Principle:

A class should have one, and only one, reason to change.

Unfortunately, I can't call a rule like that a rule of object-oriented design; I have to call it class-oriented. It doesn't mention objects, and it does mention classes. It's oriented to classes.

A Rails apologist could very easily argue that class-oriented design is an irrelevant legacy of the Java and C++ eras, and does not apply at all to work done in true object-oriented languages like Ruby and JavaScript. However, JavaScript's powerful prototypal object modelling is not enough to prevent relentless Internet drama if you say Java is not object-oriented but JavaScript is — despite the fact that this is

* With Ruby, you can think in terms of classes or in terms of objects. With JavaScript, you can think in terms of prototypes or in terms of objects. With Java and C++, classes are your only option.



obviously true* — and the overwhelming majority of Rails code uses classes much, much more than it uses objects. Controllers might be objects, but they're also classes, and as they currently exist, they have many reasons to change.

A better way to look at Single Responsibility Principle comes from the book *Growing Object-Oriented Software, Guided By Tests*, by Steve Freeman and Nat Pryce:

> *Think of a machine that washes both clothes and dishes — it's unlikely to do both well.*

Brandon Keepers quotes this in a presentation on YouTube from the RuLu conference, and adds: "That's called an ActiveRecord object."

I think the single responsibility I articulated above is just fine:

> *A controller exists to create an entire microcosm of effectively-global variables within which 'view' (template) code will operate.*

But I think it's only OK if we articulate it and acknowledge it. This is one of the places where Rails loses something in its lack of clarity. Single Responsibility Principle is crucial for clear thinking, and I don't think it's possible to think clearly about controllers without acknowledging

makes any sense, then you're just completely and utterly fucked — that gun you pointed at your feet *will* shoot your toes off — but if the reason you don't write unit tests is because you're too busy writing tons and tons of integration tests which adhere loosely to the principles of unit testing, then you're close enough, most of the time.

The only thing is, this part matters A LOT:

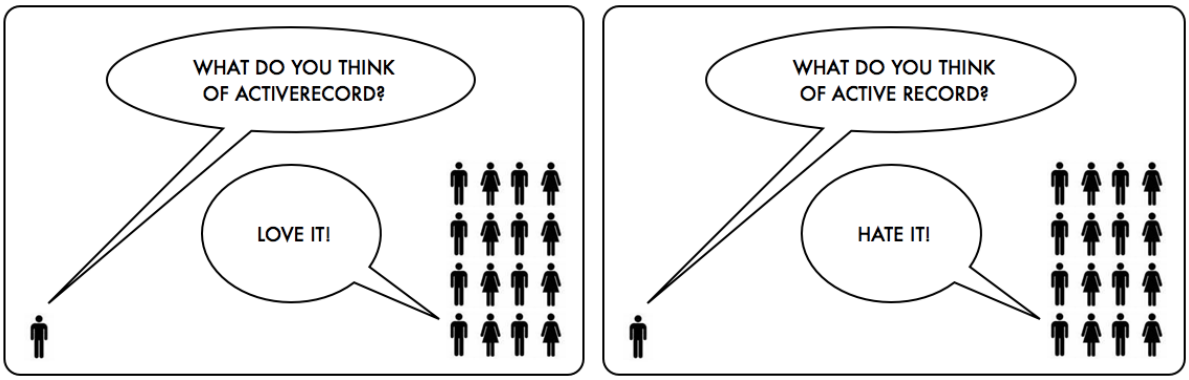
> *it is important to be able to separate [integration tests] from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.*

Again, the best thing to do is grab Gary Bernhardt's screencasts, and learn to write tests (and Rails apps) the way he does. I'm not saying every last word he utters is flawless truth; I'm just saying that if you don't know how to write tests like Gary Bernhardt, then you don't know how to do TDD.

As Corey Haines said, what Rails calls TDD is not test-driven development but test-first development. As with its misimplementation of so-called "unit tests," Rails achieves an 80/20 Rule* approximation of the real thing. It's probably much easier to mainstream an 80/20 Rule approximation than the real thing, but the real thing is still a lot more useful.

* The 80/20 Rule, or Pareto principle, comes from economics. Among other things, it refers to markets where 20% of the participants produce 80% of the profits, and to tasks where 20% of the work can produce 80% of the desired result. Be careful using this term with business people, as lower-quality MBAs interpret "80/20 Rule" as to half-ass something and get away with it, and with the exception of Stanford and the Ivy League, the overwhelming majority of MBAs are lower-quality MBAs. Here I use the term to mean judiciously cherry-picking the most important aspects of an idea, which is something Rails does a lot.

I think it is therefore extremely likely that `ActiveRecord` is really not an accurate name at all.



Rails of course is an opinionated framework, and you could argue that `ActiveRecord` makes a good name because it advocates a particular way of using the library, but I would argue for objectivity over opinion in this context. It is definitely possible to implement Active Record with `ActiveRecord`, but it is also pretty easy (and in my opinion a damn good idea for many applications) to use `ActiveRecord` without implementing Active Record. Rails name-drops OOP theory in the library's name, but there's only an aspirational or normative connection between the library and the design pattern it's named after.

This is bullshit, of course, but it's not the end of the world. The worst thing it does is turn architectural discussions into Abbott and Costello routines. ("Do you mean `ActiveRecord`? No, I mean Active Record!")