Rails As She Is Spoke

(By Giles Bowkett)

How Rails Breaks Object-Oriented Theory, And Why It Works Anyway

CHAPTER 1

Get Ready To Understand Rails

This book clears up confusion which creeps into any discussion about Rails. We're going to shoot down lies, tear apart mistaken assumptions, shatter illusions, and generally cause havoc. When you finish this book, you'll have a much clearer understanding of Rails, and in those rare cases where it's necessary, you'll be able to make judgement calls about modifying the framework, or straying from its conventions, with much greater clarity and security.

Get Ready To Emulate Rails

Many languages now feature Rails-like frameworks, ranging from outright ports to original code in its own right which nonetheless incorporates ideas from Rails. None of this code has seen a similar level of success. Once you understand which features of Rails drive its adoption, you'll know how to thrill developers and delight your customers, clients, co-workers, or employers.

Conventions

All Rails code samples come from Rails 3.2.9, but I've been writing Rails apps for a long time, so the discussion references earlier versions here and there as well. All links to Rails code point to the `3-2-stable` branch; this may result in individual line number links going stale, and if so I apologize, but it's probably better to risk a little link rot than to be too tied to any given moment in Rails's long history.

Links to blogs, presentations, and other resources appear throughout the book.

My language is often NSFW, but I've made an effort to tone it down a little for this book. Note please that toning it down a little is different from turning it off completely.

Where I refer to people who have written about code or ideas, I typically refer to them by last name. This is unusual among programmers, and especially unusual by Silicon Valley standards, but I think it's the most ethical way to write a book like this.

Here's the background: for a year, I went to a very unusual college in Santa Fe, New Mexico, called St. John's College. The school uses a nearly all-discussion format, and in discussions, you're required to refer to other participants formally. For instance, in a conversation with Barack Obama and Mitt Romney, you would be required to refer to them as Mr. Obama and Mr. Romney (as Nate Silver does in his blog on stats and politics).

The purpose of the rule is to show respect and encourage civil conversation. The convention rests on the assumption that arguments should be about ideas, not people. I feel that Ruby discussions online don't always make this distinction clear enough, although we don't seem to have this problem at conferences. I've probably been part of this problem in the past, so I want to be part of the solution now.

My goal is to make it clear that if I disagree with anyone, it's about their ideas. Using last names is more formal, but it's important to create an atmosphere of respect. I will, however, make one exception.

My Name Is Not Zed*

* Fuck that guy.

TL;DR:

- I don't hate Rails.
- I don't hate anyone who wrote it.
- I'm not trying to start a fight in the playground after school.

This book calls out several flaws within the Rails code base, but I have no hostile intentions. I am not trying to ruin anyone's life, and if you read this and then tell me "come at me, bro," I will not come at you, bro, whatever that is intended to mean. (Somebody actually said that to me over Twitter, after I wrote a controversial blog post about the Rails 3 refactor.)

With this book, I'm hoping to avoid immature reactions and instead attain a grownup conversation about imperfections and pitfalls in an otherwise amazing and useful toolkit. Rails transformed my life for the better, permanently raised the bar in terms of what open source can accomplish, and has been my primary source of income for many years now. My goal here is to help people who work with Rails, and help people who work on Rails as well.

So don't shoot the messenger, but I do have bad news.

Rails Contains Legacy Code

This shouldn't be a shock. David Heinemeier Hansson first released Rails as open source in July 2004; it hit 1.0 in December of 2005. Anything that old will contain legacy code.

You have to go into what's fucked up with Rails in order to figure out the difference between where Rails cheats on OOP, but shouldn't, and where Rails cheats on OOP, but totally gets away with it. Identifying those differences are crucial if you want to figure out what it is that Rails gets right but OOP theory gets wrong. And that's the real question driving this book.

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.

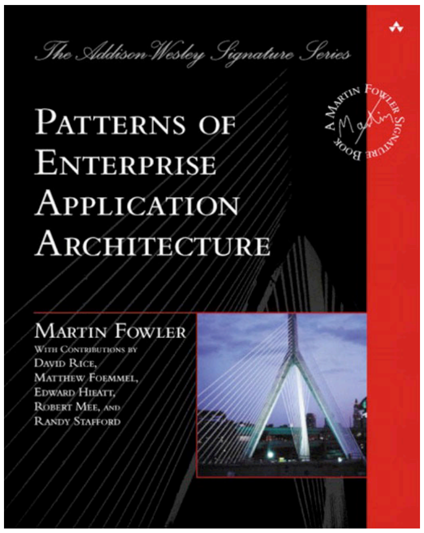
James Hague wrote on his blog:

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.

If he's correct, and I think he is, then locating this "arbitrary complexity threshold" becomes a very important thing, and I believe Rails is very good at doing that.

Rails Misinforms You About Rails

However, there are things that Rails is not good at. One very important example: Rails is not good at telling you what Rails is doing. We're going to see several places where Rails misinforms you, in both its code and its documentation, about its own design. Rails loves the Factory design pattern but hates using the word "Factory" for some reason. Rails calls its powerful default ORM `ActiveRecord`, after the design pattern Active Record, from Martin Fowler's book *Patterns Of Enterprise Architecture*. However, many extremely good Rails developers use and love`ActiveRecord` the library while abhorring and disavowing Active Record, the design pattern, which `ActiveRecord` believes itself to be based on, but is in fact only named after.


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler
With Contributions by David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford
A Martin Fowler Signature Book

The discussion around moving away from Active Record (the design pattern) in Rails apps does not usually mention the pattern by name, not because developers don't know about it, but because Rails has confused the issue with a bad naming decision. If `ActiveRecord` the library had a more accurate name, Rails developers would be free to have discussions about Active Record the design pattern without fear of confusion. We'll look at that in detail in a later chapter.

The question that matters here is "where and why does Rails break OOP and get away with it?" But to get to the answer, we need to have a clear, articulate discussion, which means we have to dispell inaccuracies about Rails which Rails itself has propagated.

CHAPTER 2

BAD NAMING WITHIN RAILS: A BROKEN WINDOW

Piotr Solnica wrote a great series of blog posts about refactoring called "Get Rid Of That Code Smell." In one post, he tackles the code smell widely known as Primitive Obsession. Ward Cunningham's c2.com wiki describes Primitive Obsession:

Primitive Obsession is the name of a code smell that occurs when we use primitive data types to represent domain ideas.

In his blog post, Solnica uses the example of a Date being represented as a String:

would you use a string to represent a date? You could, right? Just create a string, let's say "2012-06-25" and you've got a date! Well, no, not really — it's a string. It doesn't have semantics of a date, it's missing a lot of useful methods that are available in an instance of Date class.

To illustrate the refactoring which gets rid of this code smell, he works from the example of a class with an `@attributes` hash, but in the example, the class needs to know not only about its own attributes, but also the attributes of its parent class. So rather than adding parent-investigating functionality to `Hash`, which would be utterly irrelevant to some concept of `Hash-`

ness, he creates a new class, called `AttributeSet`. This class incorporates the very small amount of `Hash` functionality which the use case requires, but it exists as its own object, and also has some knowledge of its parent's attributes.

Compare this to the Rails class `HashWithIndifferentAccess`.

The comment at the top of the file says it all:

```
# This class has dubious semantics and we only have it
# so that people can write <tt>params[:key]</tt>
# instead of <tt>params['key']</tt>
# and they get the same value for both keys.
```

`HashWithIndifferentAccess` is indeed a `Hash` which erases the semantic distinction between `:foo` and `"foo"`, allowing you to use `params[:foo]` and `params['foo']` as equivalent in your app.

Rails extends Ruby in many ways. Here it erases the distinction between `Symbols` and `Strings` for the sake of convenience when handling parameters in a controller. You could argue for or against this decision, and you could argue for or against its implementation. I want to draw your attention to the variable name itself. The functionality and implementation decisions are a judgement call, but the Rails *naming* decision is just *wrong*.

Compare `HashWithIndifferentAccess` to `AttributeSet`. One of these names describes implementation; the other describes purpose. `AttributeSet` is a good name; `HashWithIndifferentAccess` is not. If the class only exists in order to enable `params [:foo]` to equal `params['foo']`, then it's all about `params`, and `Params` would be a better name. You could argue that `params` itself is a bad name, an example of the Variable Name Same As Type code smell, but any programmer can instantly guess the relationship between a `params` variable and a `Params` class. It would at least be an improvement.

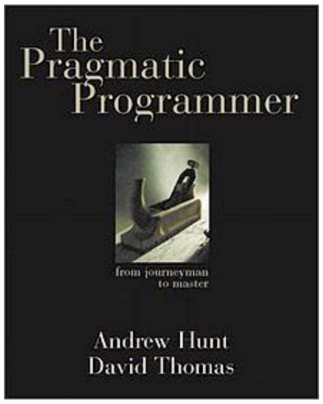
Rails has this precisely backwards: you know how the class works from its name, but only find out what it's for when you read the file.

In good TDD and BDD, you never test implementation. It becomes impossible to refactor, for one thing, because you can have code which achieves the desired behavior, but breaks your tests, since the implementation's changed. More importantly, though, your tests should describe your system, so anyone who reads the tests can figure out what your system is doing, and so when any given test breaks, you can read the test to determine what your system *isn't* doing, but should be.

Most rules of thumb for writing tests work equally well as rules of thumb for choosing names. This one is a great example. Names should reflect purpose, not implementation. Go through any Rails app and change the name of the `User` class to `ThingWhichLogsIn` and see how much better your life gets. Hint: it might get worse.

Consider again these comments:

```
# This class has dubious semantics and we only have it
# so that people can write <tt>params[:key]</tt>
# instead of <tt>params['key']</tt>
# and they get the same value for both keys.
```



They sound a little tired and frustrated, don't they? That's the sound of legacy code. Specifically, that's the sound of a broken window, the term introduced in *The Pragmatic Programmer* to describe problems which people know they should fix but don't. The metaphor refers to a curious discovery in civic politics, that the easiest and cheapest way to prevent bad neighborhoods from getting worse is to fix broken windows in abandoned buildings. The easiest and cheapest way to prevent project failure is to clean up even the tiniest minor annoyances in your code base.

Speaking of bases, consider a name every Rails developer is familiar with:

```
class User < ActiveRecord::Base
```

A `User` is not a type of `Base`. You're not going to surround it with a barbed-wire fence. You can't land airplanes in it or keep your generals there for convenient storage. It's not a base.

From an OOP perspective, `ActiveRecord::Base` has more serious problems than bad naming, so we'll return to the issue later. For now, the important thing to realize is that `Base` describes implementation, rather than purpose. It's a little like saying `class Child < Parent`.

This is the type of thing which could drive an OOP zealot nuts, but nobody really minds it in practice, because it's not a big deal. You don't typically interact with `ActiveRecord::Base` at all. You interact with your `ActiveRecord` models. In fact, if you use Rails's handy generators to create models for you, you might never actually type the words `ActiveRecord::Base` in your entire career as a Rails developer.

However, it's also a broken window. So yes, better to fix it. But also, one of the main things we're looking for in this book is how come Rails works well despite its broken windows. Both of these factors I just named — code generation and ease of use — are more significant than they might seem, and will become relevant in the later discussion.

If you want to understand where the boundaries of usefulness are for OOP theory, look at the stuff which drives OOP zealots crazy without mattering to anyone else. Those little disconnects help reveal the places where theory fails to accurately describe reality.

They also involve places where Rails fails to accurately describe Rails, but before we get deeper into that, let's consider a few more fundamentals of dealing with legacy code.

Tests, Names, And Comments

In most circumstances, I favor tests over comments, because tests make verifiable assertions, while comments do not. If a test tells you something false about your system, the test fails. If a comment tells you something false about your system, a developer gets confused, but nothing else happens. The code still runs.

As a code base ages, it will change. Most programmers change the code before they change the comments; sometimes programmers *only* change the code, and leave the comments as-is. That's why the best way to deal with comments in truly nasty legacy code is simply to delete them unread.

Names are like comments, in that the computer doesn't care, but bad names cause even more confusion than bad comments. You also can't delete names unread.

Naming errors inhibit your ability to reason about your system. So the easiest way to fix legacy code is to fix naming errors. It's a big payoff for minimal effort.

CHAPTER 3

IMPLICIT OBJECT MODELS

I once worked on a project with some truly terrible legacy code. It wasn't fun, but it taught me a valuable lesson.

The language was JavaScript. Most of the objects in the system were hand-coded literals in raw JavaScript. And I do mean objects; I don't mean classes or prototypes. I'm talking about gigantic blocks of code, hundreds of lines in length, which specified objects as JavaScript literals, without using constructors or any of JavaScript's object-oriented features. People had apparently built the entire system by cutting and pasting these literal objects, and then tweaking only a few pieces. Entire 100-line blocks were repeated verbatim, or very near verbatim,throughout the code base.

I was able to shrink this code base by literally thousands of lines. It was very, very easy to do. Say you've got a 100-line code block which appears 30 times in a code base. You create a function which contains that code block. Every time you see another appearance of that block, you replace all 100 lines with one line referencing the function. Boom. You just reduced 3,000 lines of code down to 130. In the worst-case scenario, you'd have to add in the ability to parameterize the function, or to pass it a callback for some custom behavior, but it was usually straightforward, and when it wasn't, it was instructive.

What I learned was how to recognize implicit object models. The simplest form is the Data Clumps code smell; to quote the c2 wiki:

DataClumps is when two or more variables are always used in group and it wouldn't make sense to use one of the variables by itself. This group of variables should be extracted into a class.

Here's a contrived example, with an imaginary business domain:

```
Ninja.chop({ style: 'Dragon Fist',
             target: 'windpipe',
             followupMove: Ninja.chop(
           { style: 'Dragon Fist',
             target: 'windpipe',
             followupMove: Ninja.chop(
           { style: 'Dragon Fist',
             target: 'windpipe',
             followupMove: null })})});
```

The code I was dealing with in real life wasn't every bit as stupid as this, but it came extremely close. Here's a refactor:

```
throatPunch = { style: 'Dragon Fist',
                target: 'windpipe' };
```

```
Ninja.chop(throatPunch);
Ninja.chop(throatPunch);
Ninja.chop(throatPunch);
```

That breaks DRY, but it's way easier to read. This is such an incredibly simple example of refactoring that you don't technically even need objects. Refactoring to variables is an improvement here. But refactoring to objects doesn't take much more effort.

```
ThroatPunch = function() {
  return { style: 'Dragon Fist', target: 'windpipe' }
}

Ninja.throatPunch = function() {
  this.chop(new ThroatPunch());
}

Ninja.tripleThreat = function() {
  for(i = 0; i < 3; i++) {
    this.throatPunch();
  }
}

Ninja.tripleThreat();
```

It's a violent metaphor, but a very simple process to understand. The combination of `style` and `target` represents an implicit object model, so you package that up into an object.

CHAPTER 4

URL_FOR AND LINK_TO

With that in mind, let's take a look at an interesting pattern of repetition within the Rails code base.

There are 5 methods in the Rails code base named `url_for`:

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_controller/metal.rb#L171

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_dispatch/http/url.rb#L23

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_dispatch/routing/route_set.rb#L577

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_dispatch/routing/url_for.rb#L110

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_view/helpers/url_helper.rb#L40

When you're putting links in a Rails view, you're using the version in `ActionView`. I believe that the version in `actionpack/lib/action_dispatch/routing/url_for.rb` is the one which you use when you're writing a Rails controller. I also believe that developers building conventional Rails apps are unlikely to see the others at any given time. Every version except one takes a hash; I'm not sure if that hash looks the same every time, but my guess is almost. The version in `ActionView` claims to use the same API as the version in `ActionController::Base`, but there is no version in `ActionController::Base` any more.

```
# Returns the URL for the set of +options+ provided. This takes the
# same options as +url_for+ in Action Controller (see the
# documentation for <tt>ActionController::Base#url_for</tt>).
```

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_view/helpers/url_helper.rb#L40

The version in `actionpack/lib/action_controller/metal.rb` takes a string and returns that string. That method contains absolutely no logic; it looks like it's an artifact of incomplete refactoring, likely to change in future, which only remains in place for backwards compatibility, although I want to make it clear that I'm guessing.

I'm really not sure how this code works at all. But I can say definitively that the string `Url.new` does not appear anywhere in the Rails `3-2-stable` code base as of this writing, and under the circumstances, I think that's pretty strange. It's pretty easy to make the argument that a `Url` plays a central role in the domain logic of a Web application development framework. Discounting the no-op string version of `url_for`, it looks to me like there are four factory methods here which return an object, but that object is not actually defined anywhere in the code base. It looks to me like there's an implicit object model we can surface here. It's possible that if all this logic were contained in a class called `Url`, the methods themselves would be much easier to read.

It seems even more likely when you look at the documentation for the related view helper `link_to`:

```
# ==== Options
# The +options+ hash accepts the same options as +url_for+.
```

https://github.com/rails/rails/blob/3-2-stable/actionpack/lib/action_view/helpers/url_helper.rb#L270

The caching APIs do something similar, throughout the files in the https://github.com/rails/rails/tree/3-2-stable/actionpack/lib/action_controller/caching dir, which again suggests an implicit object model. It's nowhere near as obvious or as painful as the `Ninja.chop()` example earlier, but when you see a consistent API used with methods with names like `url_for` and `link_to`, which sound a lot like they want to return objects, it seems a pretty safe bet to say Rails would be better off with a `Url` class.

When you dive deeper into these methods, you start to see things like `polymorphic_url` and `polymorphic_path`. These appear to be the methods which make it possible to name a route and then just refer to the URL that route involves by name — for instance, to create a home page URL simply by typing `home_page_url` in a Rails view.

Compare that to Wikipedia's definition of Factory objects:

In object-oriented computer programming, a factory is an object for creating other objects. It is an abstraction of a constructor, and can be used to implement various allocation schemes...

A factory object typically has a method for every kind of object it is capable of creating. *These methods optionally accept parameters defining how the object is created, and then return the created object.*

http://www.flickr.com/photos/mlnoone/4502977248/

We have a Factory design pattern here, which is unaware of its own Factory-ness.

The reason Rails gets away with this is really simple: the in-page API looks great in the simple case, which is `link_to "home", home_page_url`. Typing that, and getting back what you expect, is very nice. It's only when you throw a bunch of details and options at it that you glimpse the implicit object model and wonder why the internal code isn't more readable.

Nonetheless, the implicit object model is definitely there. In OO terms, it is impossible to explain this consistency — always the same parameters to supply, in varying contexts, to more than one method — without reference to 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:

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.

I think the in-page API for app developers stays on the right side of this threshold. Consider how ridiculous our code would look if we threw away `link_to "home", home_page_url` and demanded a pure OO approach instead:

```
<%= HomePageLinkFactory.create_link %>
```

Even the simpler version, replacing `url_for` with a constructor, would look stupid:

```
redirect_to HomePageLink.new
```

But internally, behind the scenes, within methods like `link_to` and the several different versions of `url_for`? I think it's pretty obvious when you have five methods which do almost the same thing and have almost the same API, and share that API with other methods in the caching system, you are not operating below the threshold of complexity for wisely using OO.

Once upon a time, I worked on a project where I "rediscovered" the Command pattern by accident. It was probably ten years after I had read the original Gang of Four book *Design Patterns*, and I had some very verbose, clunky code. I took the code apart, rebuilt it to make it more readable, and when I had a whole bunch of small methods and focused classes, a co-worker pointed out to me that I had used the pattern. I'm proud of that, because I think that's what it looks like when you use patterns well. Your subconscious does it, while you focus on writing good code, and afterwards you spot the pattern.

Design Patterns recommends using patterns to guide design, but I think that's hubris. I think it's much smarter to match your code to the problem space and then notice when design patterns emerge within your work. To paraphrase Yoda, a Jedi uses design patterns for knowledge and defense, never to attack. If your code naturally matches an existing design pattern, you want to be aware of the fact; it's fair to call that a form of programmer literacy. When you recognize the similarity, you can use it to inform your decisions. But nearly every time I've seen where the process started out with somebody grabbing a design pattern and setting out to implement it, they've built something else instead. In many cases — I believe `ActiveRecord` is one of them — we build something else instead, without even noticing we've built something else instead.

Speaking of `ActiveRecord`, I have a question.

CHAPTER 5

What The Fuck Is The Deal With ActiveRecord?

A few years back I explored the world of affiliate marketing and figured out an easy way to pull in a few hundred dollars per year with an affiliate marketing site. I didn't do anything more than build a simple proof-of-concept site at first, because it would take hundreds of such sites to create significant income, but recently I started building an automated publishing system to semi-automatically generate the sites for me. (This was not a spamblogs project, although I realize it might sound like one.)

Anyway, I was making good headway when I tried to use a standard object-oriented technique with `ActiveRecord` models. The effort failed, because the standard object-oriented feature which I attempted to use is apparently one which `ActiveRecord` models do not possess. I'm of the opinion that's a design flaw in Rails, but others might disagree. Regardless, the entire thing was so mind-numbing and disheartening that I set aside the entire project out of dismay. I'll likely rebuild it without Rails in the future, but that's tangential to this discussion.

What matters here is that the standard OO feature I was unable to use with `ActiveRecord` was **inheritance**. I'm not kidding.

The code involved single-table inheritance (STI) and association classes. If you've been working with Rails for a while you probably know that Rails STI often confuses people, but I'm going to have to admit that I hadn't used it in a long enough time that I forgot all about this aspect. I just leapt right into the fray, tried to use it like normal inheritance, and got burned.

I don't want to give away the internal mechanics of my awe-inspiring get-rich-quick scheme, so we'll just pretend it's an app about horses and friendships instead. In this app, you can have a `Friendship`.

A `Friendship.has_many :horses`, and a `Horse.belongs_to :friendship`. Meanwhile, `Horse` has subclasses, such as `Pony`, `Unicorn`, and `Pegasus`.

I tried to do this:

```
class Horse < ActiveRecord::Base
  def frolic!
    # frolicking happens here
  end
end
```

```
class Pony < Horse
  def frolic!
    # frolic in a pony-specific way
  end
end

class Pegasus < Horse
  def fly!
    # flying happens here
  end

  alias :frolic! :fly!
end

class Unicorn < Horse
  def use_magic!
    # unicorns get their magic on here
  end

  alias :frolic! :use_magic!
end


Friendship.first.horses.each do |horse|
  horse.frolic!
end
```

When I ran this code, I expected a broad spectrum of frolicking to occur all over the place, including general frolicking, pony-specific frolicking, aerial frolicking, and magical frolicking, from each hooved creature according to its own proclivities. Unfortunately, every `Pony`, `Pegasus`, and `Unicorn` in my database was a `Horse` as far as `ActiveRecord` could tell while running this code. General frolicking happened. Pony-specific frolicking did not, and neither did flying or the use of magic.

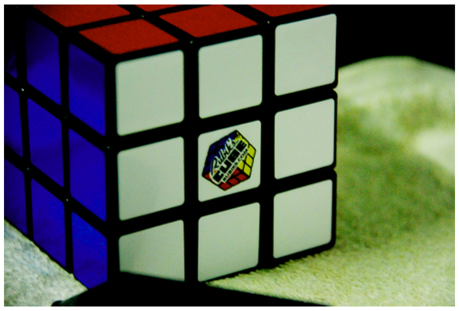
http://www.flickr.com/photos/66766724@N07/7469820870/

Rails makes some weird fucking choices when it 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.

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


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

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?

Meeting a talking rhino with Rubik's Cubes for balls is just so fucking weird that you don't even know where to begin, and that's what this unusual hypothetical event has in common with the relationship `ActiveRecord` has to inheritance. "Idiosyncratic" really just doesn't even cut it.

Imagine how much more uncomfortable the moment gets if you're a champion Rubik's Cube solver, and the cubes are unsolved. Better hope that tidbit doesn't come up in conversation. Could be awkward. Could involve a discussion of personal boundaries you might not feel prepared to have with a rhino who you just met. That's kind of what it's like when you're well-versed in OO theory and you look at the code for `ActiveRecord`.*

* By the way, this analogy is an example of Homeric simile, a rhetorical device common in classical Greek epic poetry but tragically underused in modern English. Let me get all serious for a second: I think I'm better at explaining technical matters to non-technical people than almost any other programmer I've ever met, and I believe that studying Homeric simile is why. There's ego and pretension in this claim, obviously, but Homeric simile means exploiting a metaphor in detail, and in a sequence of several different ways. To construct a Homeric simile, you have to align your metaphor with several different aspects of whatever it is you're using the metaphor to describe. This is also pretty much the only way to communicate technical ideas to non-technical people. It might be the only effective way to communicate technical ideas at all. If you want to become a better blogger, I recommend developing skills in Homeric simile. And you don't have to learn classical Greek, or even read Homer, to do it. George Carlin's 1980s comedy contains, in my opinion, the best Homeric similes in the English language.

And please remember that inheritance could hardly even be a more fundamental element of object-oriented programming. It really is a mystery where to even begin with this shit. But I have to start somewhere, so I'll start with STI.

Single-Table Inheritance

As I mentioned earlier, and will explain later, I don't believe that `ActiveRecord` necessarily implements the Active Record pattern. However, when implementing the Active Record pattern, you basically have two options for representing inheritance: single-table inheritance, and class table inheritance. Rails, opinionated framework that it is, chose single-table inheritance and stuck with it. With single-table inheritance, you store both `Horse` and `Pony` in a table called `horses`, along with a metadata column that tells you the class of each object — in Rails, surprisingly, the convention is not to name that column `klass` but `type` — while class table inheritance works in a much simpler way: just make a table for each class.

To quote Martin Fowler:

relational databases don't support inheritance. You want database structures that map clearly to the objects and allow links anywhere in the inheritance structure. Class Table Inheritance supports this by using one database table per class in the inheritance structure.

My little pony problem comes down to how `ActiveRecord` handles association classes. Since every `Pony` represents data which lives in a table called `horses`, and you have to check its `type` column to determine if it's a `Pony` or a `Horse`, a method which retrieves this data and reifies it into objects, but does not look for or at a `type` column, will perceive every `Pony` as a `Horse`. The method receives a bunch of SQL rows from the database adapter, and then turns every SQL row into a so-called `Horse`, and then it's done.

I'm going to have to apologize here, though, because it's been a while since I encountered this bug, and I'm under-describing its complexity. I'm also having a hard time reproducing it. Rails *does* actually check for the `type` column and reify it, in some cases, but it definitely doesn't do it everywhere, or all the time. For instance, I got different results from querying the parent class directly (e.g., `Horse.all`) than I did from association classes which I knew for a fact contained all the horses in the system (e.g., `Friendship.first.horses`). I'm also not sure which version of Rails I saw this problem in, and it could take me all day to find out.

It's entirely possible what I'm about to say is completely wrong. I definitely hope so. However, at the time, I found reasonably credible sources on the Web which seemed to indicate that the preferred, official solution was to use the `after_initialize` callback to have the object inspect its own `type` column, and then use the `becomes` method to update the object's class accordingly.

```
class Horse
  after_initialize :becomes_something_other_than_horse_if_not_horse

  def becomes_something_other_than_horse_if_not_horse
    becomes self.type if self.type != 'Horse'
  end
end
```

This is, quite simply, the stupidest method name which has ever happened. I can't even acknowledge that method name as something I wrote. I'm just going to call it something which happened. The implementation also would of course break for any `Horse` rows in the database which had been created without first loading the subclasses, since Rails does not always realize its own STI system is active, and in that case, it would have saved my `Horse` data without anything in the `type` column at all.

This would work better, if "better" is even a word that can enter the discussion at this point:

```
becomes self.type if self.type && self.type != 'Horse'
```

Or "better" yet:

```
becomes self.type if self.type && self.type != self.class.to_s
```

You could take out most of the `self` variables, and alias `type` as `klass`, for a more readable version:

```
becomes klass if klass && klass != self.class.to_s
```

That code is clearly just totally fucking awesome.

The theme of this book is all the places where Rails breaks OOP and gets away with it, but this is really a place where Rails breaks OOP and falls flat on its face. STI has been baffling the shit out of newbies, and indeed even experienced Rails devs, since the days when the *only* experienced Rails devs all worked at 37Signals. Every new release of Rails seems to have a different set of STI bugs. It's painful in 2012, it was painful in 2005, and it was painful in between. Rails's single-table inheritance support sprays bugs in every direction the same way a hippo, spinning its tail like a propeller while it defecates, sprays its own feces in every direction to mark its territory.

Rails modifies Ruby itself in many places, enough so that it might be more reasonable from a pure linguistic perspective to call Rails a dialect of Ruby, rather than a framework which merely uses it. Convenience methods like `3.weeks.from_now` look utterly ridiculous if you view them through the prism of object-oriented theoretical purity, because they disregard the Single Responsibility Principle* completely, but in practice they make writing code simpler, easier, and an utter joy. Setting up an entire custom inheritance system for just one class, however, does not.

* For background on SRP, see page 55.

And this is just the tip of the iceberg in `ActiveRecord` inheritance weirdness. Rails does not **only** set up an entire custom inheritance system for `ActiveRecord` models, to enable STI. It **also** sets up a completely **different** custom inheritance system for `ActiveRecord::Base`, the alleged parent class of all these models.

And when I say alleged, I mean alleged. I do not mean alleged with credibility, or alleged in a plausible way. I simply mean alleged. It's going to become very obvious, very soon, that `ActiveRecord::Base` isn't just a bad name because of its non-specificity, as I mentioned earlier. `ActiveRecord::Base` should be called `ActiveRecord::Factory`, because that's what it is. There's no barracks and no soldiers, but it is a thing which systematically churns out product.

Before we get into that, I want to just wrap up this section on STI with some words from Xavier Shay:

Three Reasons Why You Shouldn't Use Single Table Inheritance

It creates a cluttered data model. *Why don't we just have one table called objects and store everything as STI? STI tables have a tendency to grow and expand as an application develops, and become intimidating and unwieldy as it isn't clear which columns belong to which models.*

It forces you to use nullable columns. *A comic book must have an illustrator, but regular books don't have an illustrator. Subclassing Book with Comic using STI forces you to allow illustrator to be null at the database level (for books that aren't comics), and pushes your data integrity up into the application layer, which is not ideal.*

***It prevents you from efficiently indexing your data.** Every index has to reference the type column, and you end up with indexes that are only relevant for a certain type...*

What you should be doing is using Class Table Inheritance. Rails doesn't "support it natively", but that doesn't particularly mean much since it's a simple pattern to implement yourself, especially if you take advantage of named scopes and delegators. Your data model will be much easier to work with, easier to understand, and more performant.

My take on this: the man is right. Do what he says.

BY FAR THE WEIRDEST THING ABOUT `ACTIVERECORD` INHERITANCE IS THE CUSTOM `INHERITED` METHOD.

The missing `Link` or `Url` objects which `url_for` and `link_to` imply, and the methods which generate URLs without generating `Url` objects, are not the only example of Rails using the Factory design pattern without realizing it.

We've already seen how Rails goes nuts if you do this:

```
class Pony < Horse
```

Let's look at what happens when you do this:

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

First of all, every Ruby class has an `#inherited` method. To quote ruby-doc.org:

[Class#inherited] is a singleton method (per class) invoked by Ruby when a subclass... is created. The new subclass is passed as a parameter.

Here's the `#inherited` implementation from `ActiveRecord::Base`:

```
def inherited(child_class) #:nodoc:
  child_class.initialize_generated_modules
  super
end
```

Raising of course the obvious question, what happens inside `#initialize_generated_modules`?

```
def initialize_generated_modules #:nodoc:
  @attribute_methods_mutex = Mutex.new

  # force attribute methods to be higher in
  # inheritance hierarchy than other generated methods
  generated_attribute_methods
  generated_feature_methods
end
```

The `Mutex`, a mutually exclusive thread lock*, just makes sure that no other threads get in the way while the `ActiveRecord` "subclass" is being defined. Other than that, this method just invokes two other methods.

* http://www.ruby-doc.org/core-1.9.3/Mutex.html

`generated_feature_methods` is the very next method in the file:

```
def generated_feature_methods
  @generated_feature_methods ||= begin
```

```
    mod = const_set(:GeneratedFeatureMethods, Module.new)
    include mod
    mod
  end
end
```

However, `generated_attribute_methods` lives in a completely different location. It's not even part of `ActiveRecord`, but instead `ActiveModel`. Hopefully the ongoing refactoring efforts mean that this is not an example of tight coupling. I enjoy tight coupling in my leisure time, of course, but it's not something you want to see in code. Let's not even go into the topic of tight tripling.

Anyway, here's the code from `ActiveModel`.

```
def generated_attribute_methods #:nodoc:
  @generated_attribute_methods ||= begin
    mod = Module.new
    include mod
    mod
  end
end
```

In each case, all either method does is either create a Module and store it an instance variable, or just give you that instance variable, if it already exists. Later in the process of "inheriting" from`ActiveRecord`, Rails populates these instance variables with methods which it grabs from throughout the `ActiveRecord` and `ActiveModel` code bases.

Compare this to Wikipedia's definition of the Abstract Factory pattern:

The abstract factory pattern is a software creational design pattern that provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes... This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

When you write this code...

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

...Rails activates an Abstract Factory which, for some reason, it keeps inside the code bases of two separate libraries, `ActiveRecord` and `ActiveModel`.

This Abstract Factory is one of the greatest implementations of the pattern in the entire history of object-oriented code. It implements a powerful, complex toolkit for describing object relationships — the `ActiveRecord` DSL for association classes can model nearly every relationship in the gigantic, academic class description language UML — and is able to **infer** the design of the so-called "subclass" from the structure of its database table. This is a classic example of why Rails gets away with such insane object orientation: the Abstract Factory at the heart of `ActiveRecord` rocks so hard that nobody cares that this code essentially stuck itself up its own ass. It makes no sense at all for it be located where it is, yet it remains so staggeringly useful that even after almost 10 years, it has very few credible competitors.

The Abstract Factory which powers `ActiveRecord` is a beautiful thing. But what the ever-loving fuck is it doing inside an inheritance mechanism?

I believe the answer, quite simply, is that it looks cool:

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

It doesn't look as cool as Rails thinks it does, because `Base` is an appallingly generic name, but it does look cool. It's clean, it's simple, it's direct, and the minute you see it you know what to do next. When most people think of an Abstract Factory, they probably imagine an implementation like this:

```
ActiveRecord::AbstractFactory.generate_activerecord_class
(named: 'Horse')
```

I can understand why people might not want to see that kind of code ever in their lives. But let's look for an alternative. We won't have to look very far. Consider the `Struct` class in Ruby, which features an Abstract Factory called `Struct.new`. This method allows you to generate new class parametrically, and exists to provide simple, convenient data objects. This code creates a `Struct` which represents a horse as something with a particular hair color:

```
Horse = Struct.new(:hair_color)
```

To then instantiate a black horse, do this:

```
Horse.new("black")
```

And if you decide to turn your horse purple for some reason:

```
@horse.hair_color = "purple"
```

Does that use case look familiar at all?

Consider this alternative syntax for an Abstract Factory which creates `ActiveRecord` objects:

```
Horse = ActiveRecord.new(table: "horses")
```

This API is more truthful, and it allows you to pull that Abstract Factory out into its own class.

Adding functionality or multiple attributes to a `Struct` is easy:

```
Horse = Struct.new(:hair_color, :name) do
  def frolic
    puts "frolicking!"
  end
end
```

And the same syntax would work just fine for `ActiveRecord`.

```
Horse = ActiveRecord.new(table: "horses") do
  belongs_to :friendship
end
```

This API would also allow us to escape one staggeringly aggravating side effect of Rails's utterly unnecessary and deeply eccentric decision about where to put its wonderful Abstract Factory: you can't really subclass `ActiveRecord` at all.

Once I did an interesting database project, which you'll hear more about in a later chapter. My very first, ultra-naïve implementation attempted to subclass `ActiveRecord`, but of course failed, because `ActiveRecord` does such weird things with inheritance that it makes it impossible (or at least extremely time-consuming) to create true `ActiveRecord` subclasses.

The lesson here, basically, is that if you've got something really fantastic, you can stick it anywhere. That sounds kind of dirty, though, so I'm glad I'm not saying it in a presentation at a Ruby conference. I've met a lot of the people who will be reading this book, and I know how mature you aren't. Anyway, I think it's obvious this code could become clearer, and more logically located.

The other lesson, of course, is that people hate the kind of overspecified, verbose code associated with Java and C++ — and, through guilt by association, the *Design Patterns* school of thought — that they will tie their code in fucking knots before they will crack open a book on patterns. `ActiveRecord` wins despite its OO fail here not just on sheer usefulness, but also on the very smooth and clean user experience for any programmer who doesn't do anything particularly unusual or creative with it. It might be convoluted interally, but it's very easy to get started with.

However, it's probably obvious that I think it would win harder without the OO fail.

CHAPTER 6

Rails Controllers Are Much Stranger Than They Seem At First Glance

The MVC implementation in Rails moves so far away from what the term originally meant that it's more reasonable to say Rails appropriated the term than that it implemented the pattern. I remember reading about MVC on Wikipedia when I first discovered Rails in 2005. The Wikipedia page described MVC in its classic GUI incarnation: the version first developed by Xerox Parc and still to this day used in OS X and iOS development. Today, Wikipedia describes the Rails interpretation, with the original version treated as a historical footnote.

Let's delve into that footnote. The c2.com wiki keeps history alive for us:

> *Model-View-Controller is the concept introduced by Smalltalk's inventors (Trygve Reenskaug and others) of encapsulating some data together with its processing (the model) and isolate it from the manipulation (the controller) and presentation (the view) part that has to be done on a UserInterface.*

In other words, imagine you're building an iPad app where you press a button and you hear a fart noise. The model would basically just be `Fart.honk()` — there's not much more to it than that, in this case. The controller process your taps, swipes, and pokes. The view controls displaying data, in the same way a Rails "view" simply puts data on a screen. But the view also controls any user interface elements which animate or present visual effects in any way — any time something slides around the screen, appears, disappears, or suddenly turns purple, view code makes it happen.

We don't use Ruby for that at all. We use JavaScript. A Rails view does not even slightly resemble a View in the classic MVC sense, and until recently, JavaScript user interface code was never object-oriented. It's difficult, to say the least, to write UI code for a Web app on the server side. You can only ever write UI View code using Ruby in a Rails "view" if you write Ruby code which compiles to JavaScript, like Rails JS helpers, which are often not worth the effort. If you write classic MVC Views in a Rails app, you're probably writing them in Backbone, or some similar JavaScript MVC library.

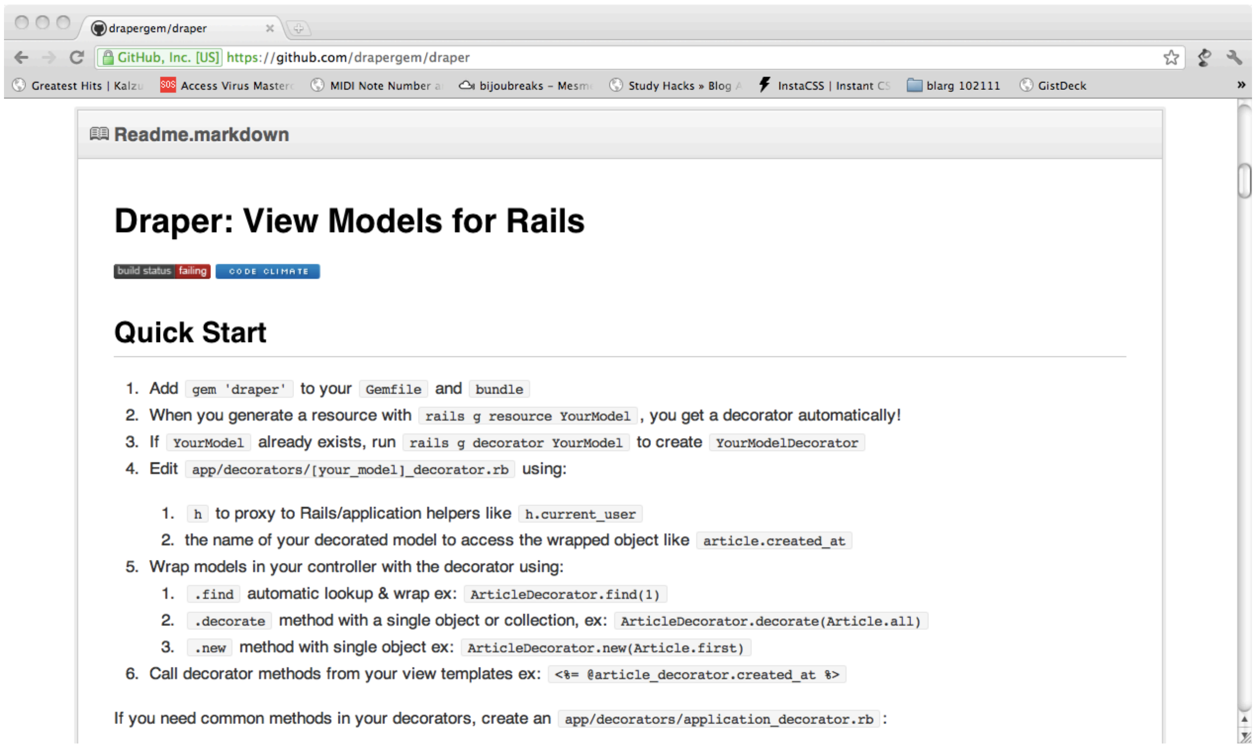
A Rails view is better understood as a template, and this causes problems when you have views sufficiently complex to warrant their own View objects. Consider this CodeClimate blog post, *7 Ways To Refactor Fat ActiveRecord Models*:

If logic is needed purely for display purposes, it does not belong in the model. Ask yourself, "If I was implementing an alternative interface to this *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.



Readme.markdown

Draper: View Models for Rails

Quick Start

1. Add `gem 'draper'` to your `Gemfile` and `bundle`
2. When you generate a resource with `rails g resource YourModel`, you get a decorator automatically!
3. If `YourModel` already exists, run `rails g decorator YourModel` to create `YourModelDecorator`
4. Edit `app/decorators/[your_model]_decorator.rb` using:
   1. `h` to proxy to Rails/application helpers like `h.current_user`
   2. the name of your decorated model to access the wrapped object like `article.created_at`
5. Wrap models in your controller with the decorator using:
   1. `.find` automatic lookup & wrap ex: `ArticleDecorator.find(1)`
   2. `.decorate` method with a single object or collection, ex: `ArticleDecorator.decorate(Article.all)`
   3. `.new` method with single object ex: `ArticleDecorator.new(Article.first)`
6. Call decorator methods from your view templates ex: `<%= @article_decorator.created_at %>`

If you need common methods in your decorators, create an `app/decorators/application_decorator.rb`:

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 quirky intepretation of MVC, really, with "views" and "helpers," Rails acts on the assumption that PHP was right when it came to the idea that web pages with dynamic code embedded were the way to go — so in some ways, writing Rails is like writing PHP at a level where you don't need much code, but writing OO stuff at the level where you do.

Rather than implement MVC, what Rails really gives us is M-PHP-C.

This sounds horrible, but PHP is popular for a reason. If all you want to do is display the date on a web page, for example, you're working well below the complexity threshold which justifies OOP.

Rails is a multi-paradigm framework in the same way that Ruby is a multi-paradigm langauge. There's the front end, where you say "PHP was right," and you basically just write PHP, and then there's the backend, where you say "PHP was wrong," and instead you use ideas from the world of object-oriented programming.

This compromise actually works really well, but it's a pain in the ass when you try to test your controllers. Testing controllers is hard for a reason: they're objects, but Rails uses them not just as objects, but also in a completely different way.

Here's Steve Klabnik speaking about this mechanism on his blog:

ActionController relies on instance variables to pass information from the controller to the view. Have you ever seen a 200 line long controller method? I have. Good luck teasing out which instance variables actually get set over the course of all those nested ifs.

The whole idea is kinda crazy: Yeah, it looks nice, but we literally just say 'increase the scope of variables to pass data around.' If I wrote a post saying "Don't pass arguments to methods, just promote your data to a global" I'd be crucified. Yet we do the same thing (albeit on a smaller scale) every time we write a Rails application.

Keep in mind that tiny shell scripts which use global variables work just fine without namespacing. Namespacing is unnecessary within extremely small programs. Rails gets away with this for the same reason.

David Bryant Copeland summarized it on his blog:

In a Rails controller, you create ivars to communicate data to the view.

The blog post itself isn't entirely relevant, but I like the phrasing, because it's brief, concise, and highlights something important: the role that instance variables play in a Rails controller has absolutely nothing to do with being an instance variable. It's the role a *data store* usually plays. It really blurs the line between an object in your system and **Redis**, which is a 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.

In OOP, instance variables represent the internal state of objects. Rails uses them as a way to transmit data. Instance variables are not Smalltalk message-sending. But I just argued, earlier in this book, that you can spot an implicit object model in the arguments for the `url_for` and `link_to` methods. If you can look at code which passes around a list of keywords and values and see that the consistency of those keywords and values strongly hints at an implicit class, then it's pretty easy to see that storing data in instance variables works the same way, but in reverse. With `url_for` and `link_to`, Rails uses a Hash which really has the consistent structure which might normally characterize an object. So say you have a PHP-like system, and you want to send some data to it, and each piece of data has to be associated with a name. Why use a Hash for that when you can use an object?

Yes, I realize that's a weird question. But think about it.

In Java or C++, it's very reasonable to say that if data has a consistent structure, you should reify that consistent structure in a class. But Ruby is not Java or C++; there's no actual requirement that objects of a class be identical in their structure. In fact, there's technically no requirement that objects of the same class even be *similar* in their structure. I believe Rails should change the arguments for `url_for` and `link_to` to surface and reify those implicit objects. It'll make the documentation and the implementation clearer, which means it'll be easier for newbies to learn, and it won't involve any significant change in functionality whatsoever.

But I'm not going to claim here that Rails shouldn't be using controllers to store data, at least not for the reason that the data doesn't have a consistent structure, and therefore doesn't seem like an object. That's not a valid objection in object-oriented programming. It's only a valid argument if you fail to differentiate object-oriented programming from class-oriented programming. Just because it's useful to construct classes of objects doesn't change the fact that it's also useful to treat objects as individually unique.

However, even when you understand the rationale, it's pretty strange to use instance variables as little carrier pigeons carrying data from the controller out into the template, or alleged "view," especially since that data leaves on a one-way trip. But that is what we use instance variables for in Rails controllers. And it's nothing to do with the requirements of ERb; ERb can easily use local variables instead of instance variables. So why do we do it?

We use instance variables in controllers for two reasons. First, Rails does some complicated instance-variable-copying behind the scenes. Local variables don't get "saved" to an object's internal state when you exit a method, while instance variables do.

Second, instance variables are a different color in most text editors, and it looks cool. I'm not kidding, I really think that's the other reason. You can look at a page of code and instantly see what data you'll have available when you switch into PHP mode. It's a very convenient, simple, intuitive, and productive user experience for app developers. This is not the first time we've seen Rails make really strange decisions about its internals which result in a really good experience for app developers.

So, in the same way that ActiveRecord builds an incredibly sophisticated class factory inside `ActiveRecord::Base` the class, when that factory could in fact go pretty much anywhere, Rails controllers treat their own internal state as a data store. And it's not really a big deal at this point, since many of us have been writing Rails apps for well over 7 years now.

We all know you can do any crazy shit you want when you use a language which can assign any number of instance variables (or indeed methods) to an object in real time. Ruby, in its flexibility, echoes the old Lisp idea that code is data and data is code. An object using its own in-memory representation and internal structure as a data store might seem incomprehensibly bizarre from a Java perspective, but it's so effortless in Ruby that you might never even notice how unusual it is.

Rails actually exposes major problems with OO dogma. First, very few people who speak of object-oriented programming actually do object-oriented programming. The overwhelming majority do class-oriented programming. Morphing an object's structure in real time can be useful in OO languages like JavaScript and Ruby, or certain of their predecessors like Smalltalk and Self, but is virtually impossible in Java or C++. And that's where most OO dogma comes from.

The majority of OO discussion overrates techniques which were developed in the context of extremely inflexible languages, because of the sheer number of people who come to the discussion from that perspective. Every OO idea from the worlds of Java or C++ is unnecessarily static, timid, and clunky from a Ruby perspective. And most OO dogma reflects a fundamental, total, and yet utterly unnecessary separation of data and code.

It's like when you go to a fancy restaurant and they expect you to eat different types of food with different forks. That distinction must have meant something, at some point in human history, but now it's just incomprehensible and meaningless. If code is data, and data is code, why not just put the data in the code?

Rails is very good at identifying compromises between paradigms and creating social fictions which paper over the rough edges. In both these examples, the solutions pretend to be very simple, familiar mechanisms (instance variables and inheritance) while in reality doing something very different and much more complex. Rails is a multi-paradigm system which allows you to write PHP-like code on the front end and rigorous OO-style code on the back end. It also stuffs its factory patterns into inexplicable locations just for the hell of it. Rails is actually pretty fucking weird.

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*

This is more responsibilities than a single object should have, if we decide to use the Single Responsibility Principle. But it's not a list of all the responsibilities for a single object. It's all the responsibilities for a single method.

Copeland then applies Jamis Buck's "Fat Model, Skinny Controller" rule, and moves some of the responsibilities to the model:

```
class UsersController < ApplicationController
  def create
    @user = User.create(params[:user])
  end
end

class User < ActiveRecord::Base

  after_create :deliver_welcome_email

private

  def deliver_welcome_email
    if self.valid?
      UserMailer.deliver_welcome_email(@user)
    end
  end
end
```

The controller now only has three responsibilities:

- Creates a new User instance from form parameters
- Saves the new User to the database
- Renders the view

And putting the fourth responsibility in the model creates problems later on, as Copeland points out:

All w've done is move the problem somewhere else. We've also made testing our application a huge pain, because everywhere we create a User instance for a test, we'll fire off the UserMailer, so we'd need to stub that out or otherwise arrange for that code not to run, except when we test that code…

We've mixed up the concerns of creating instances of `User` objects with creating new users of our application. The distinction might be subtle, but it's important.

I want to rephrase that last part:

We've mixed up the concerns of creating database rows to represent our user data with the other work involved in creating new users of our application.

Copeland recommends using `ActiveRecord` for persistence only, which we'll cover later, with an `ApplicationUserCreator` class. The more classic OO name would be `ApplicationUserFactory`.

This simple use case, of alerting a user with notifications of some kind upon signup, presents a surprisingly thorny object modelling problem in Rails. Many people in the Rails community have discussed this use case. David Heinemeier Hansson, who created Rails, proposed a solution which leverages `after_create` callbacks in `ActiveRecord` models, like Copeland's does above. The code was elegant, but in my opinion

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?"

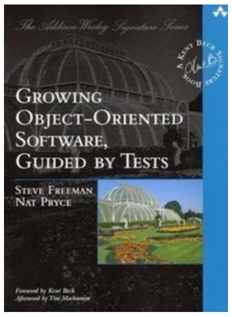
Let's look at Martin's original definition of the Single Responsibility Principle:

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 the unusual nature and scope of what controllers really do, as a bridge between real object-oriented code and a sort of a less-hideous PHP Lite™.

In my opinion, this means "Fat Model, Skinny Controller" is still a good rule of thumb, and that it's a good idea to populate your model with smaller objects which handle a lot of the other, associated responsibilities. Copeland's idea of an `ApplicationUserFactory`, for example, could make a lot of sense in a sufficiently complicated application, and would certainly make the use case in question much easier to test. Using `ActiveRecord` for persistence only is the topic of an upcoming chapter, but first, let's look at Rails tests.

CHAPTER 8

Rails Unit Tests Are Not Unit Tests

This is too obvious, and widely known, to go into a great deal of detail, so I'm going to be quick.

Unit tests cannot touch the database or the network or the filesystem. Rails tests do.

Michael Feathers, author of *Working Effectively With Legacy Code*, explains:

A test is not a unit test if:

- *It talks to the database*
- *It communicates across the network*
- *It touches the file system*
- *It can't run at the same time as any of your other unit tests*
- *You have to do special things to your environment (such as editing config files) to run it.*

He adds immediately thereafter:

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes... When you pull the database, sockets, or file system access into your unit tests, they aren't really about those methods any more; they are about the integration of your code with that other software.

He wrote this back in 2005, and the discussion at the time centered around Java, but the points remain valid. I want to especially highlight one sentence:

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.

Since then, practically every Rails or Ruby developer who has expressed any opinion on TDD or BDD has expressed this opinion, in so many different presentations, blog posts, tweets, books, podcasts, and videos that any attempt to list them all would not only fail but be totally pointless.

I will say, however, that I think the best materials on TDD and BDD in Ruby (or any language) are Gary Bernhardt's *Destroy All Software* screencasts, which you can find at destroyallsoftware.com.

However, Rails made TDD mainstream. Before Rails, most programmers saw TDD as an exotic or eccentric technique. Rails made writing apps without TDD a mark of unprofessionalism and incompetence, which in the long run was fantastic for programmers.

Rails tests work because if the reason you don't write unit tests is because you don't think TDD 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.

With test-driven development, the goal is not really test-driven development at all, but test-driven design. With test-first development, you just write the tests first and build the code around that. It's very close to the same thing, but with test-first development, and integration tests which pretend to be unit tests, it's a lot harder to let the tests drive your design.

For a Rails developer, the obvious big takeaway here is probably something you're already doing: ignore everything Rails itself tells you about tests, and instead concentrate on the consensus of the Ruby community, and especially the work of Gary Bernhardt, Corey Haines, and others.

But there's a much more useful lesson here as well. Before Rails, very few people used TDD, despite the fact that people had already been doing TDD for years, and frameworks, books, and conferences all supported it. Rails made testing the norm. If you ever work in any large company, ever, you'll have the opportunity to discover something like TDD — something everybody could do, and everybody should do, but which most people aren't doing. If you ever go to a conference or participate in open source in any way, you have the same opportunity.

When you find something everybody should be doing but isn't, what do you do? The correct move to make might be to copy Rails, and make an oversimplified implementation incredibly easy to get into. From day one, Rails made it really easy to write tests. It included example tests, it automated running tests all the way down to typing one single word — `rake` — and automatically generated your tests for you as well. Rails provided an incredibly smooth user experience for writing tests. It took years for alternative testing systems to match that convenience and simplicity. Providing a frictionless user experience is a much more effective way to get people to do things than asking them to read a bunch of books and re-invent the way they write code.

This is a lesson to learn which makes Rails look awesome. Unfortunately, there's still another lesson to learn from this, and it's a lesson which makes Rails look stupid. This is because Rails is both awesome and stupid*.

* Come on, you knew that already.

Writing your own test library is almost a rite of passage for Ruby devs. It's really easy to understand why. We're all expected to write "unit tests," yet we very quickly discover the limitations of Rails "unit tests." Tying a test suite to the database makes it slow, clunky, and painful. The correct solution is to use real unit tests, but many people are already using that term to describe integration tests that sort of adhere to most of the basic ideas of unit testing, sort of. And so we hammer out countless original approaches to the problem, because we've polluted our vocabulary. If the entire tribe has agreed to refer to a square object as a "wheel," every caveman and cavewoman worth his or her salt is guaranteed to re-invent the alleged wheel sooner or later.

We'll see this again in the next chapter.

CHAPTER 9

ACTIVERECORD VS ACTIVE RECORD

`ActiveRecord` takes its name from the Active Record pattern, from Martin Fowler's *Patterns Of Enterprise Architecture*:

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data... An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database. Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.

However, many of the best Rails developers will tell you they use `ActiveRecord` but not Active Record. It's becoming very popular to separate the business logic from the persistence responsibilities of the ORM.

In fact, I consider this an emerging convention, with good odds of becoming a standard among all serious Rails developers. To my knowledge, James Golick was the first to advocate it, back in 2010:

The current best-practice for writing rails code dictates that your business logic belongs in your model objects. Before that, it wasn't uncommon to see business logic scattered all over controller actions and even view code. Pushing business logic in to models makes apps easier to understand and test...

[But] coupling all of your business logic to your persistence objects can have weird side-effects. In our application, when something is created, an after_create callback generates an entry in the logs, which are used to produce the activity feed. What if I want to create an object without logging â€" say, in the console? I can't. Saving and logging are married forever and for all eternity.

When we deploy new features to production, we roll them out selectively. To achieve this, both versions of the code have to co-exist in the application. At some level, there's a conditional that sends the user down one code path or the other. Since both versions of the code typically use the same tables in the database, the persistence objects have to be flexible enough to work in either situation.

If calling #save triggers version 1 of the business logic, then you're basically out of luck. The idea of creating a database record is inseparable from all the actions that come before and after it...

A simplified explanation of the problem is that we violated the Single Responsibility Principle. So, we're going to use standard object oriented techniques to separate the concerns of our model logic...

To decouple the logging from the creation of the database record, we're going to use something called a service object. A service object is typically used to coordinate two or more objects; usually, the service object doesn't have any logic of its own (simplified definition).

To separate the saving functionality from the logging functionality, Golick recommends wrapping calls to `ActiveRecord` in service objects:

```
class UserCreationService
  def initialize(user_klass = User, log_klass = Log)
    @user_klass = user_klass
    @log_klass  = log_klass
  end
  def create(params)
    @user_klass.create(params).tap do |u|
      @log_klass.new_user(u)
    end
  end
end
```

He acknowledges that `UserCreationService.create` looks weirder (and more Java-tastic) than `User.create`, but argues that the benefits to your tests, and design, make it worthwhile.

Corey Haines advocated something similar, in his Golden Gate RubyConf 2011 presentation "Fast Rails Tests," available on YouTube.

The code looks like this:

```
class ShoppingCart < ActiveRecord::Base
  include ShoppingCartExtensions::CalculatesTotalPrice

  has_many :shopping_cart_products, dependent: :destroy
  has_many :products, :through => :shopping_cart_products
end

module ShoppingCartExtensions
  module CalculatesTotalPrice
    def total_price
      products.map(&:price).inject(0,&:+)
    end
  end
end
```

Adam Keys presented a similar idea on his blog:

When modeling how our domain objects map to what is stored in a database, an object-relational mapper often comes into the picture. And then, the angst begins. Bad queries are generated, weird object models evolve, junk-drawer objects emerge, cohesion goes down and coupling goes up.

It's not that ORMs are a smell. They are genuinely useful things that make it easier for developers to go from an idea to a working, deployable prototype. But its easy to fall into the habit of treating them as a top-level concern in our applications.

What if our domain models weren't built out from the ORM? Some have suggested treating the ORM, and the persistence of our objects themselves, as mere implementation details. What might that look like?

In this post, he gives a code example where he creates a main `User` class for domain logic, and creates a related persistence object:

```
class User::Model < ActiveRecord::Base
```

It works well, although I would definitely prefer

```
class User::Record < ActiveRecord::Base
```

Or for that matter

```
class User::Record = ActiveRecord.new(table: "users")
```

He also further decouples the persistence mechanism from the domain logic, to the extent that when his app migrates from SQL to Cassandra, to scale up to enormous numbers of users, the switch is relatively painless.

Avdi Grimm, Greg Moeck, and others have also recommended similar approaches.

Steve Klabnik tweeted:

We need something better. Persistence and logic are two separate responsibilities that every rails app combines.

Piotr Solnica blogged:

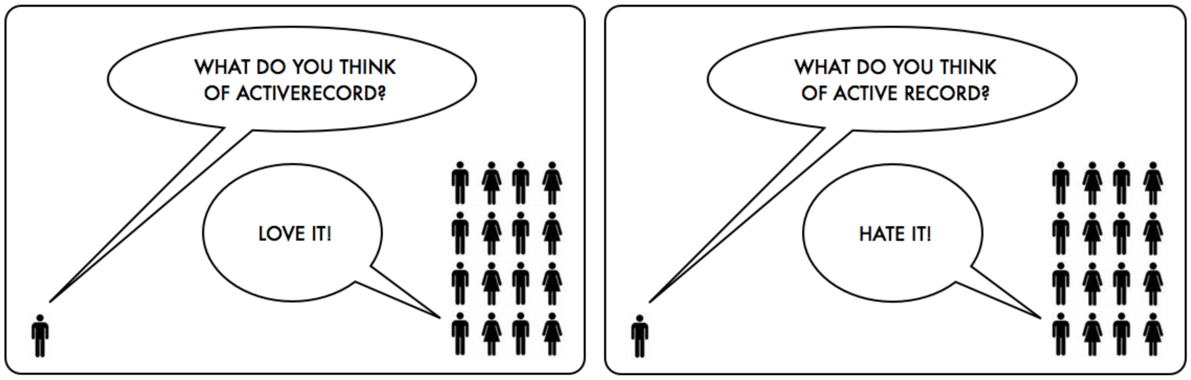
When we say "model" we usually think about ActiveRecord. In Ruby on Rails world this is how we established things. "M" in the MVC means app/models with a bunch of ActiveRecord model files. This is where the domain logic of our applications lives. I think we should stop thinking like that...

The way your Domain Model behaves and the way your data are persisted are two separate concerns. ActiveRecord objects represent your data. They give you a low level interface to access your data. Yes, low level. If you mix domain specific behavior into ActiveRecord models you will create classes with too many responsibilities. By violating Single Responsibility Principle model code becomes difficult to extend, maintain and test. I have seen it many times, I'm pretty sure you have too.

The interesting part for the purposes of this book: many of the best Rails developers in the world feel totally comfortable using `ActiveRecord`, the library, while simultaneously uniting in their opposition to Active Record, the design pattern.

In other words, many very experienced and skilled Rails developers share a consensus which differentiates `ActiveRecord`, the library, from Active Record, the design pattern. This same large number of very knowledgeable developers share the opinion that using the design pattern is a bad idea, even though using the library is a good idea.

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


WHAT DO YOU THINK OF ACTIVERECORD?
LOVE IT!
WHAT DO YOU THINK OF ACTIVE RECORD?
HATE IT!

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!")

Programming is a pop culture, as Alan Kay said, so, as we'll see in the next section, programmers have simply revived an OO idea called DCI to give them the vocabulary which `ActiveRecord` took away with its reckless naming.

The important thing is that every time we see Rails bullshit its way through OOP theory and create something better than that theory ever anticipated, we learn something about the deficiencies of the theory. The most important lesson of `ActiveRecord` is that a powerful, flexible, well-maintained DSL for generating cross-platform SQL is an incredible, life-changing convenience. (I'll go into more detail on this later in the book.)

The inaccurate naming does become a problem, however, when we face a complex situation where we need to retrace our steps, because the confusion leads us astray. Rails is extremely customized for a very specific type of application, one where adding a few methods to an object which represents a database table is enough to qualify as an implementation of Active Record, and when you use it for anything else, you have to work harder.

The ideas put forward by Golick, Haines, Klabnik, Grimm, and Solnica are evolving. In particular, Andrzej Krzywda, Mike Pack, and Jim Gay are advocating a formalism called DCI.

CHAPTER 10

DCI

With the Active Record design pattern, it's easy to see where your business logic goes: it goes on the same class which also represents your data. But what guidelines do you use when your domain model (e.g., `User`) and your persistence manager (e.g., `User::Record`) are going to be different things?

DCI is a set of guidelines for making these decisions. It stands for Data, Context, and Interactions. I hope it's pretty obvious that in the forthcoming entertainment, `User::Record` will be playing the part of Data. It's a role `User::Record` was born to play. "Data" in DCI means a data object.

Interactions and Context, respectively, represent the different things your domain model might need to do, and the different situations in which it might need to do them. One way people implement this is with a concept of Roles; when you perform certain Interactions within particular Contexts, that's a little like the way human beings will play different roles in different situations.

Let's put Roles on Rails. Jim Gay is writing an entire book on DCI called *Clean Ruby*; he also wrote a great introduction to the topic on his blog, where he uses the example of a user on a social network who might want to approve friend requests.

```
module FriendApprover
  def approve(friend_request)
    friend_request.approved = true
    friend_request.save
    increment_friends
    notify_new_buddy(friend_request.user_id)
  end

  def increment_friends
    friend_count += 1
    save
  end

  def notify_new_buddy(buddy_id)
    BuddyMailer.notify_buddy(buddy_id,
      "We're officially friends!")
  end
end
```

In this example, `FriendApprover` is a Role, and in the Context of approving friend requests, the Interaction is approving the request.

I'd namespace this for consistency:

```
class User

class User::Record < ActiveRecord::Base

module User::FriendApprover
```

Anyway, when it's time for your user to approve a friend request, you `extend` the Module.

```
current_user.extend FriendApprover
current_user.approve(friend_request)
```

This allows you to decouple business logic from persistence. It's a huge win. You still use `ActiveRecord`, but you wind up with an `ActiveRecord` "model" (really a data object in this case) which only manages persistence. Corey Haines has said:

I have a general rule that things outside of ActiveRecord — the actual ActiveRecord class — are not allowed to call ActiveRecord methods. They're only allowed to call scopes or hand-built scopes.

Not coincidentally, he said this right after a presentation in which he demo-ed gigantic spec suites which run in seconds, including one which ran 100 specs in less than a second. That's easy to achieve with Ruby, but extraordinarily difficult with Rails. By pulling Rails out of the specs whenever possible, you can make your specs much faster, and thereby achieve massive boosts in productivity. Anything which takes you out of the zone costs you extra time and mental energy because of context switching, so eliminating slow tests is actually worth a lot of money.

You also get all the benefits of James Golick's approach, without the relatively clunky syntax of something like

```
UserCreationService.create
```

And if you discover this method in a code base, it's pretty easy to guess where friend-approving is implemented, when the first line of code in the process looks like this:

```
current_user.extend FriendApprover
```

That's a lot easier to deal with than looking for specific methods in a billion-line `User` model.

The person who invented DCI, a Norwegian computer scientist named Trygve Reenskaug, also came up with the idea of MVC, but while MVC has dominated GUI thinking from day one, DCI appears to have lain dormant for many years, in terms of people not using it or knowing about it. As I mentioned before, a lot of OO dogma comes from a period during which first C ++ and later Java were the dominant languages in the OO culture. Neither Java nor C++ is a particularly good OO language, and more to the point, neither one has the flexibility to implement DCI easily. Reenskaug developed both MVC and DCI in Smalltalk, a language he helped invent, and Ruby is of course very similar to a Smalltalk which runs on Unix.

DCI provides a wonderful example of the difference between object-oriented programming and class-oriented programming. In the DCI paradigm, you do not ever need to categorically extend each and every `User` with friend-approving functionality. You only grant that power to individual `User` objects when they actually need it, finally solving the nightmare of absurdly gigantic `User`models which has plagued Rails since the beginning.

Basically, I think this shit is dope as hell. But Jim Gay is writing a whole book on this, and I haven't even bought it yet, because I knew if I bought it, I'd turn this book into another whole book on the topic as well. This book is about various crazy ways Rails implements OOP ideas, and what we can learn from that, so I'm just giving you the brief overview here. Check out *Clean Ruby* if you want to know more.

For the purposes of our discussion, it's enough to note that a lot of great Rails developers are bringing back a classic OO idea, and getting a lot of mileage out of it.

CHAPTER 11

Convention Over Configuration Often Remains Your Best Bet

I don't want to issue a blanket endorsement for wild deviations from Rails's conventions. It's certainly true that you need to experiment in order to develop new and better conventions in the first place. But DCI is an emerging convention with an impeccable historical pedigree and years of experimentation among great Rails developers to back it up. Generally, I echo the Rails mantra of convention over configuration. Deviating from the obvious happy path in Rails can have painful consequences.

I'm going to give you some examples from real-world projects, but in order to avoid violating non-disclosure agreements or embarassing anybody, including myself, I'm going to blur the details, and use fake names for each project. In the interest of adding a completely unnecessary sense of doom and bleakness, each project will get its fake name from a novel by Charles Dickens.

A Tale Of Two Cities was a startup with Silicon Valley talent and Hollywood ambitions. We anticipated (and received) very significant traffic, so for scaling purposes, we used

ActiveResource instead of ActiveRecord, and we had to build some of ActiveRecord's features back into ActiveResource to make it work. This was not painful, but it did have some unanticipated complexities, and we were working against a deadline. In particular, tons of view code broke because Rails view helpers assume methods which exist by default on ActiveRecord, but not ActiveResource.

I did very similar work at Hard Times, and after a while it got really time-consuming. This was before Rails switched from its own idiosyncratic plugins system to using gems like it does today, so I produced several Rails plugins that ported various ActiveRecord conveniences and features over to ActiveResource. This got painful and tedious at times.

Great Expectations had a very messy architecture. Like A Tale Of Two Cities, they drafted a custom architecture around Rails's REST implementation* to handle scaling, but unlike A Tale Of Two Cities, I don't believe they ever saw the traffic to support it. For persistence, the apps at Great Expectations (and there were several, and they all had to be running on your machine simultaneously) used MongoDB, Redis, REST, and Postgres; it was confusing and slow in the extreme.

* Rails's REST implementation nearly made it into this book, but for now I'm just going to point you to this presentation by Steve Klabnik.

On each of these projects, deviating from Rails conventions cost me time and energy. Wasting developer time is obviously expensive, but burning away developer enthusiasm is actually much, much more expensive. This is especially the case if you're a consultant, freelancer or entrepreneur, because in that context, your ability to code drives your entire business, and if you burned out, your whole business would be gone.

I think every developer who gets too used to Rails should wander off the reservation a little. When I worked on refactoring bad JavaScript, or even just on Rails projects which deviated in a few, small idiosyncratic ways from Rails conventions, it felt like travelling back in time to an era of primitive technology, or the difference between using OS X and using Windows. A dismissive attitude towards user experience is the hallmark of crappy technology. The more you use Rails, I think, the less likely you are to fully appreciate its incredible convenience and elegance.

I was lucky enough, on one project, to have a limousine bring me to and from work every day. After a few weeks or months of that, driving myself around town felt horrific and stressful. I feel silly even saying that now, but there's one thing about my limo experience that you might not see coming: since my fellow consultants and

I used our time in the limo to plan for the day ahead in the morning, and review our work in conversation at the end of the day, most of that time was billable, so even with the extravagant cost, we still turned a profit. Driving yourself to work for free is actually less **profitable** than hiring a limo to drive you around instead while you have business meetings. It's crazy but it's true. As spoiled and self-indulgent as this limo story must sound, it was actually a great business decision.

That's kind of what it's like when you suddenly can't use Rails's more indulgent pieces of automation, like the command-line model generator — a totally unnecessary extravagance for anyone who's built any Rails project, ever. We typically think of these conveniences as pointless frills, because we're big important people doing serious engineering, but "luxuries" which prevent burnout and maximize productivity are a great business decision.

Archaeopteryx

Rails seems to have a secret philosophy that API design is user experience design. I severely underestimated the importance of this in Rails success in 2008, when I went to every Ruby, Rails, or computer music conference and/or user group I could find to promote an AI breakbeat improviser I had written in Ruby called

Archaeopteryx. With Archaopeteryx, you could describe a detailed template for the type of breakbeat that you wanted to hear, and the library would then create an infinite stream of breakbeats which matched that template.

When Rails first debuted, DHH promoted the hell out of it, with brilliant presentations and screencasts, as well as controversial blog posts and shameless trolling. I did pretty much the same thing with Archaeopteryx, and my presentations provoked widespread praise and excitement — even standing ovations and awards — but my project still didn't see anywhere near the level of adoption that Rails did.

Obviously, part of this has to do with usefulness; the average developer needs a kickass web framework much more badly than they need a kickass artificial intelligence drum-and-bass loop generator. But I think another very important difference was that I never put in the time to make working with (or on) Archaeopteryx a pleasant or rewarding experience. In fact, I think that's the most important difference. Rails makes good user experience its primary goal, where "user" means the app developer.

Rails is like an iPad or a limousine; it's crafted to exacting specifications, designed for comfort, and is full of little flourishes that may appear to many as shameless luxury. But whenever I see people at a café using iPads, they're working. When I hired a limo with several other consultants, we had serious conversations about work in there*. The biggest lesson in all the bizarre OOP deviations in the Rails code base is that when you prioritize programmer happiness, you prevent burnout and you maximize productivity.

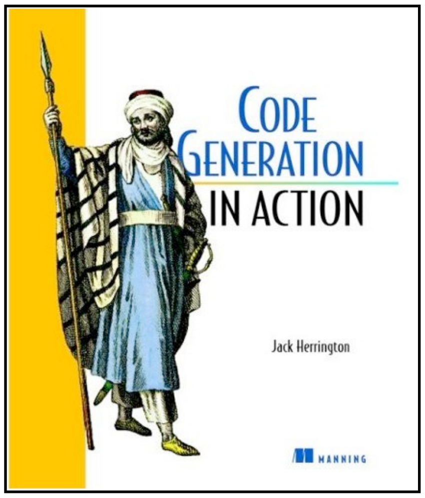
* We also drank top-shelf liquor in the limo from time to time, of course, but we didn't bill the client for that, and we had serious conversations more often. And by the way, if this story is making you jealous, or dubious, I'll readily admit that it doesn't happen every day.

If there's truth to all those things people say about great developers being 10 or 100 times more productive than average developers, then developer productivity is one of the greatest advantages you can have, so anything which maximizes your productivity as a developer is a good thing.

CHAPTER 12

Code Generation In Action

Many of these productivity-maximizing "frivolities" (which in reality are anything but) share a core strategy: code generation.



When Rails was still new, DHH recommended a code generation book on his blog, so I picked it up. The title was *Code Generation In Action*, the publisher was Manning, and the author was Jack Herrington. It's out of print today, although I think you can still buy the PDF on Manning's site. This is the single most underrated book I have ever read on any topic even remotely related to computers.

The book starts with a case study:

In this section, we examine a case study that shows the potential for code generation in modern application development. Our case study examines a corporate accounting application. Because of the complex nature of the accounting work, the initial schema has 150 database tables...

Our implementation of the EJB architecture specifies five classes and two interfaces per table. This is not an unusual object/relational mapping in the EJB model. Each class or interface is in its own file. These seven files make up a single EJB "entity." Seven files multiplied by 150 tables tells us that we are already looking at 1,050 files for the EJB entities. It's a big number, but it is not unusual for a J2EE system with this many tables...

Herrington goes on to build a code generator for this problem, rather than building these files and their tables by hand:

Our estimation of how long it would have taken to hand-code all 1,050 classes is three man-years; it takes us one man-month to build the generator. This represents a significant reduction in our schedule. The resulting EJBs are much more consistent than any we would have handwritten, and they contain all of the functionality we could want. In addition, if large-scale changes are required we can alter the templates and roll them out across the entire code base in seconds...

...we've turned several man-years of effort into a couple of man-months.

For context, the book was written in 2003. Rails first appeared in 2004.

CHAPTER 13

CODE GENERATION AND (SEMI-) AUTOMATED REFACTORING

Earlier in this book I mentioned a project where, by refactoring extremely bad JavaScript code to objects, I reduced the size of the code base by thousands of lines without changing its functionality at all. Code generation made it possible.

My system started with a spreadsheet. I investigated, by hand, different places which used very similar code.* I described them in the spreadsheet, mapping keys and values onto the spreadsheet to determine exactly how similar and how different the different copy/paste sites were from each other.

* Later, I built systems in JavaScript and Ruby with the capacity to automatically identify highly similar JavaScript functions. The Ruby system is the more sophisticated of the two. I never really finished it, and it's kind of messy, but I still think it's pretty cool: https://github.com/gilesbowkett/wheatley

So for example, using the silly domain of ninjas and massive, obvious oversimplification, I might see two copy/paste sites like this:

```
// filenameA.js
Ninja.jump({ backflip: true, style: 'Shadow' });

// filenameB.js
Ninja.jump({ backflip: false, style: 'Shadow' });
```

The spreadsheet would look like this:

	backflip	**style**
filenameA.js	true	Shadow
filenameB.js	false	Shadow

Then I would build a JavaScript object expressing the common elements:

```
Ninja.shadowJump = function(backflip) {
  this.jump({ backflip: backflip, style: 'Shadow' });
}
```

Then I built code generators driven by the data in the spreadsheets to automatically strip out the repetitive code and replace it with `Ninja.shadowJump(true)` and `Ninja.shadowJump(false)`. The result: a code base thousands of lines slimmer, and infinitely more readable.

I never would have been able to do this if I hadn't read *Code Generation In Action* — and Rails would be infitely less powerful without code generation as well. Code generation powers database migrations, the `rails new` command, the Abstract Factory at the heart of `ActiveRecord::Base`, and many other facets of Rails. But the funny thing is, code generation remains Rails's secret weapon, because most people really just do not understand how incredibly powerful it is.

CHAPTER 14

Code Generation Plus Great User Experience

This epic semi-automated refactoring pleased my client, to say the least, but I created something much better a little later on, when I began to mimic not just Rails's use of code generation, but its committment to user experience.

I worked on a project where I built an ORM-like DSL for defining MongoDB objects, and a system which used that DSL to convert those Mongo collections into SQL tables. I'm planning to open-source it if/when I have time. For now, here's a basic pseudocode example:

```
class UserTranslation < MongoTranslation
  string :name, :email
  has_one :payment_method
end
```

Although the most obvious use case is regretful companies which want to junk their Mongo installs and move back to SQL, that wasn't the actual purpose of the project. The client wanted to plug into an analytics system which operated against SQL databases.

I built this system, it worked, everything was cool, but there were occasional requests to implement the DSL in trivial ways, which annoyed me because they were boring. I'm a busy man with a lot of kung fu movies to watch and a lot of weed to smoke. I have allocated virtually no moments in my schedule for either the jibber or the jabber. So rather than play the role of a maintenance programmer, I decided to write a maintenance programmer to do the work for me.

http://www.flickr.com/photos/49663222@N08/5512582918/

In other words, for the use case where the client wanted to add a new Mongo collection to the process which converted Mongo data into SQL data for analytics purposes, I wrote a few Rake tasks which automatically analyzed Mongo collections, automatically created a model of their probable representation in SQL, automatically expressed that relationship using the DSL, auto-generated a Rails database migration to create the relevant tables and indexes, and auto-generated the translator files. For example, the system could look at a `sushi` Mongo collection like this:

```
{ "_id" => "ljhasdljkhfap987908712314jkhlkjsadf987",
  "klass" => "Nigiri",
  "fish" => "Salmon" }
```

and auto-generate code like this:

```
class SushiTranslation < MongoTranslation
  string :klass, :fish
end
```

I didn't get much further than that, but it still impressed the shit out of my main contact at the client. The rest of the story is a little ironic. My goal here was a Rails-y power and ease of use, based on the realization that Rails "luxuries" are in actuality not luxuries at all, but massive, almost godly productivity boosts.* The result was not just easier to use, however; it also really seemed like I had written a maintenance programmer to do the boring parts of my consulting engagement for me. I have long believed that programmers are actually much cheaper to write than they are to hire, and I believe this new feature set conclusively demonstrated the fact.

* Seriously. The code is a mess in places, and even some of the core ideas are bizarrely twisted, but the design is just genius, especially in terms of the priorities it establishes.

However, if you're a busy man with a lot of kung fu movies to watch*, you have to be careful about how much you impress your clients. My main contact referred to this impressive feature as "fucking awesome," and immediately asked me to switch onto a more high-priority project. I cheerfully agreed, and ended up with not enough time to maintain my "fucking awesome" system. Its ownership passed along to a young data scientist who had very little experience in Ruby and re-wrote the whole thing in Python

* In the 1990s, I watched so many Hong Kong action and kung fu movies — particularly films by Yuen Wu-Ping and John Woo — that I actually started picking up a few very basic words of Cantonese from context. Who needs a social life when there's kung fu?

and bash. I believe it lost its magical, Rails-meets-AI feature set in the process.

Nonetheless, an excessively-impressed client is a wonderful problem to have. The important thing for the purposes of this discussion is not the eventual fate of the feature set, nor the fact that I figured out a way to work some AI hacking (a personal hobby) into a Ruby contract. The important thing for the purposes of this discussion is that I knew other programmers would be using this code, I did some real thinking about what their user experience would be like, and I put in serious work to make it much, much better than the user experience they would have using any other comparable system.

This, I think, is the most important thing you can learn from Rails: Make Steve Jobs your role model when you design your APIs, and the world will be your oyster.

http://www.flickr.com/photos/acaben/541334636/

Appendix

English As She Is Spoke

The Incomplete Book Of Failures, The Official Handbook of the Not-Terribly-Good Club of Great Britain, first introduced me to *English As She Is Spoke*, a Portuguese-English conversational guide from 1883, with this excerpt from the phrasebook's introduction:

We expect then, who the little book (for the care what we wrote him, and for her typographical correction) that may be worth the acceptation of the studious persons, and especially of the youth, at which we dedicate him particularly.

I feel the same way.

English As She Is Spoke contained virtually no accurate information about English at all. The author did not speak English, and probably created his book by combining a French-English dictionary with a Portugese-French phrasebook.

Some example English phrases which *English As She Is Spoke* recommends using in casual conversation:

- *Dress your hairs.*
- *This hat go well.*

- *Exculpate me by your brothers.*
- *She make the prude.*
- *After the paunch comes the dance.*
- *That pond it seems me many multiplied of fishes. Let us amuse rather to the fishing.*
- *I jest of them; my vessel is armed in man of war, i have a vigilant and courageous equipage, and the ammunitions don't want me its.*
- *The stone as roll not heap up not foam.*
- *He has tost his all good.*
- *Go us more fast never I was seen a so much bad beast; she will not nor to bring forward neither put back.*

Mark Twain had this to say about *English As She Is Spoke*:

Nobody can add to the absurdity of this book, nobody can imitate it successfully, nobody can hope to produce its fellow; it is perfect.

Nonetheless, I hope *Rails As She Is Spoke* can sit side-by-side with this classic.