Addison-Wesley Professional Ruby Series

THE RAILS 3 WAY

Foreword by **David H. Hansson**
creator of Ruby on Rails

OBIE FERNANDEZ

Library of Congress Cataloging-in-Publication Data
Fernandez, Obie.
The rails 3 way / Obie Fernandez.
p. cm.
Rev. ed. of: The Rails way / Obie Fernandez. 2008.
Includes index.
ISBN 0-321-60166-1 (pbk. : alk. paper)
1. Ruby on rails (Electronic resource) 2. Object-oriented programming (Computer science)
3. Ruby (Computer program language) 4. Web site development. 5. Application
software–Development. I. Fernandez, Obie. Rails way. II. Title.
QA76.64.F47 2010
005.1'17–dc22 2010038744

ISBN-13: 978-0-321-60166-7
ISBN-10: 0-321-60166-1

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, December 2010

Editor-in-Chief
Mark Taub

Executive Acquisitions Editor
Debra Williams Cauley

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Carol Loomis

Indexer
Valerie Haynes Perry

Proofreader
Erica Orloff

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Chuti Prasertsith

Compositor
Glyph International

Contents

Foreword

Rails is more than a programming framework for creating web applications. It's also a framework for thinking about web applications. It ships not as a blank slate equally tolerant of every kind of expression. On the contrary, it trades that flexibility for the convenience of "what most people need most of the time to do most things." It's a designer straightjacket that sets you free from focusing on the things that just don't matter and focuses your attention on the stuff that does.

To be able to accept that trade, you need to understand not just how to do something in Rails, but also why it's done like that. Only by understanding the why will you be able to consistently work with the framework instead of against it. It doesn't mean that you'll always have to agree with a certain choice, but you will need to agree to the overachieving principle of conventions. You have to learn to relax and let go of your attachment to personal idiosyncrasies when the productivity rewards are right.

This book can help you do just that. Not only does it serve as a guide in your exploration of the features in Rails, it also gives you a window into the mind and soul of Rails. Why we've chosen to do things the way we do them, why we frown on certain widespread approaches. It even goes so far as to include the discussions and stories of how we got there—straight from the community participants that helped shape them.

Learning how to do Hello World in Rails has always been easy to do on your own, but getting to know and appreciate the gestalt of Rails, less so. I applaud Obie for trying to help you on this journey. Enjoy it.

— David Heinemeier Hansson
Creator of Ruby on Rails

This page intentionally left blank

Foreword

From the beginning, the Rails framework turned web development on its head with the insight that the vast majority of time spent on projects amounted to meaningless sit-ups. Instead of having the time to think through your domain-specific code, you'd spend the first few weeks of a project deciding meaningless details. By making decisions for you, Rails frees you to kick off your project with a bang, getting a working prototype out the door quickly. This makes it possible to build an application with some meat on its bones in a few weekends, making Rails the web framework of choice for people with a great idea and a full-time job.

Rails makes some simple decisions for you, like what to name your controller actions and how to organize your directories. It also gets pretty aggressive, and sets development-friendly defaults for the database and caching layer you'll use, making it easy to change to more production-friendly options once you're ready to deploy.

By getting so aggressive, Rails makes it easy to put at least a few real users in front of your application within days, enabling you to start gathering the requirements from your users immediately, rather than spending months architecting a perfect solution, only to learn that your users use the application differently than you expected.

The Rails team built the Rails project itself according to very similar goals. Don't try to overthink the needs of your users. Get something out there that works, and improve it based on actual usage patterns. By all accounts, this strategy has been a smashing success, and with the blessing of the Rails core team, the Rails community leveraged the dynamism of Ruby to fill in the gaps in plugins. Without taking a close look at Rails, you might think that Rails' rapid prototyping powers are limited to the 15-minute blog demo, but that you'd fall off a cliff when writing a real app. This has never been true. In fact, in Rails 2.1, 2.2 and 2.3, the Rails team looked closely at common usage patterns

reflected in very popular plugins, adding features that would further reduce the number of sit-ups needed to start real-life applications.

By the release of Rails 2.3, the Rails ecosystem had thousands of plugins, and applications like Twitter started to push the boundaries of the Rails defaults. Increasingly, you might build your next Rails application using a non-relational database or deploy it inside a Java infrastructure using JRuby. It was time to take the tight integration of the Rails stack to the next level.

Over the course of 20 months, starting in January 2008, we looked at a wide range of plugins, spoke with the architects of some of the most popular Rails applications, and changed the way the Rails internals thought about its defaults.

Rather than start from scratch, trying to build a generic data layer for Rails, we took on the challenge of making it easy to give any ORM the same tight level of integration with the rest of the framework as Active Record. We accepted no compromises, taking the time to write the tight Active Record integration using the same APIs that we now expose for other ORMs. This covers the obvious, such as making it possible to generate a scaffold using DataMapper or Mongoid. It also covers the less obvious, such as giving alternative ORMs the same ability to include the amount of time spent in the model layer in the controller's log output.

We brought this philosophy to every area of Rails 3: flexibility without compromise. By looking at the ways that an estimated million developers use Rails, we could hone in on the needs of real developers and plugin authors, significantly improving the overall architecture of Rails based on real user feedback.

Because the Rails 3 internals are such a departure from what's come before, developers building long-lived applications and plugin developers need a resource that comprehensively covers the philosophy of the new version of the framework. *The Rails™ 3 Way* is a comprehensive resource that digs into the new features in Rails 3 and perhaps more importantly, the rationale behind them.

—Yehuda Katz
Rails Core

Introduction

As I write this new introduction in the spring of 2010, the official release of Rails 3.0 is looming, and what a big change it represents. The "Merb-ification" of Rails is almost complete! The new Rails is quite different from its predecessors in that its underlying architecture is more modular and elegant while increasing sheer performance significantly. The changes to Active Record are dramatic, with Arel's query method chaining replacing hashed `find` parameters that we were all used to.

There is a lot to love about Rails 3, and I do think that eventually most of the community will make the change. In most cases, I have not bothered to cover 2.x ways of doing things in Rails if they are significantly different from the Rails 3 way—hence the title change. I felt that naming the book "The Rails Way (Second Edition)" would be accurate, but possibly misleading. This new edition is a fully new book for a fully new framework. Practically every line of the book has been painstakingly revised and edited, with some fairly large chunks of the original book not making the new cut. It's taken well over a year, including six months of working every night to get this book done!

Even though Rails 3 is less opinionated than early versions, in that it allows for easy reconfiguration of Rails assumptions, this book is more opinionated than ever. The vast majority of Rails developers use `RSpec`, and I believe that is primarily because it is a superior choice to `Test::Unit`. Therefore, this book does not cover `Test::Unit`. I firmly believe that `Haml` is vastly, profoundly, better than ERb for view templating, so the book uses `Haml` exclusively.

0.1 About This Book

This book is not a tutorial or basic introduction to Ruby or Rails. It is meant as a day-to-day reference for the full-time Rails developer. The more confident reader might be able to get started in Rails using just this book, extensive online resources, and his or her wits, but there are other publications that are more introductory in nature and might be a wee bit more appropriate for beginners.

Every contributor to this book works with Rails on a full-time basis. We do not spend our days writing books or training other people, although that is certainly something that we enjoy doing on the side.

This book was originally conceived for myself, because I hate having to use online documentation, especially API docs, which need to be consulted over and over again. Since the API documentation is liberally licensed (just like the rest of Rails), there are a few sections of the book that reproduce parts of the API documentation. In practically all cases, the API documentation has been expanded and/or corrected, supplemented with additional examples and commentary drawn from practical experience.

Hopefully you are like me—I really like books that I can keep next to my keyboard, scribble notes in, and fill with bookmarks and dog-ears. When I'm coding, I want to be able to quickly refer to both API documentation, in-depth explanations, and relevant examples.

0.1.1 Book Structure

I attempted to give the material a natural structure while meeting the goal of being the best-possible Rails reference book. To that end, careful attention has been given to presenting holistic explanations of each subsystem of Rails, including detailed API information where appropriate. Every chapter is slightly different in scope, and I suspect that Rails is now too big a topic to cover the whole thing in depth in just one book.

Believe me, it has not been easy coming up with a structure that makes perfect sense for everyone. Particularly, I have noted surprise in some readers when they notice that Active Record is not covered first. Rails is foremost a web framework and, at least to me, the controller and routing implementation is the most unique, powerful, and effective feature, with Active Record following a close second.

0.1.2 Sample Code and Listings

The domains chosen for the code samples should be familiar to almost all professional developers. They include time and expense tracking, auctions, regional data management, and blogging applications. I don't spend pages explaining the subtler nuances of the

business logic for the samples or justify design decisions that don't have a direct relationship to the topic at hand. Following in the footsteps of my series colleague Hal Fulton and *The Ruby Way*, most of the snippets are not full code listings—only the relevant code is shown. Ellipses (. . .) denote parts of the code that have been eliminated for clarity.

Whenever a code listing is large and significant, and I suspect that you might want to use it verbatim in your own code, I supply a listing heading. There are not too many of those. The whole set of code listings will not add up to a complete working system, nor are there 30 pages of sample application code in an appendix. The code listings should serve as inspiration for your production-ready work, but keep in mind that they often lack touches necessary in real-world work. For example, examples of controller code are often missing pagination and access control logic, because it would detract from the point being expressed.

Some of the source code for my examples can be found at `http://github.com/obie/tr3w_time_and_expenses`. Note that it is not a working nor complete application. It just made sense at times to keep the code in the context of an application and hopefully you might draw some inspiration from browsing it.

0.1.3 Concerning Third-Party RubyGems and Plugins

Whenever you find yourself writing code that feels like plumbing, by which I mean completely unrelated to the business domain of your application, you're probably doing too much work. I hope that you have this book at your side when you encounter that feeling. There is almost always some new part of the Rails API or a third-party RubyGem for doing exactly what you are trying to do.

As a matter of fact, part of what sets this book apart is that I never hesitate in calling out the availability of third-party code, and I even document the RubyGems and plugins that I feel are most crucial for effective Rails work. In cases where third-party code is better than the built-in Rails functionality, we don't cover the built-in Rails functionality (pagination is a good example).

An average developer might see his or her productivity double with Rails, but I've seen serious Rails developers achieve gains that are much, much higher. That's because we follow the Don't Repeat Yourself (DRY) principle religiously, of which Don't Reinvent The Wheel (DRTW) is a close corollary. Reimplementing something when an existing implementation is good enough is an unnecessary waste of time that nevertheless can be very tempting, since it's such a joy to program in Ruby.

Ruby on Rails is actually a vast ecosystem of core code, official plugins, and third-party plugins. That ecosystem has been exploding rapidly and provides all the raw

technology you need to build even the most complicated enterprise-class web applications. My goal is to equip you with enough knowledge that you'll be able to avoid continuously reinventing the wheel.

0.2 Recommended Reading and Resources

Readers may find it useful to read this book while referring to some of the excellent reference titles listed in this section.

Most Ruby programmers always have their copy of the "Pickaxe" book nearby, *Programming Ruby* (ISBN: 0-9745140-5-5), because it is a good language reference. Readers interested in really understanding all of the nuances of Ruby programming should acquire *The Ruby Way, Second Edition* (ISBN: 0-6723288-4-4).

I highly recommend Peepcode Screencasts, in-depth video presentations on a variety of Rails subjects by the inimitable Geoffrey Grosenbach, available at `http://peepcode.com`

Ryan Bates does an excellent job explaining nuances of Rails development in his long-running series of free webcasts available at `http://railscasts.com/`

Last, but not least, this book's companion website at `http://tr3w.com` is the first place to look for reporting issues and finding additional resources, as they become available.

Regarding David Heinemeier Hansson, a.k.a. DHH

I had the pleasure of establishing a friendship with David Heinemeier Hansson, creator of Rails, in early 2005, before Rails hit the mainstream and he became an International Web 2.0 Superstar. My friendship with David is a big factor in why I'm writing this book today. David's opinions and public statements shape the Rails world, which means he gets quoted a lot when we discuss the nature of Rails and how to use it effectively.

David has told me on a couple of occasions that he hates the "DHH" moniker that people tend to use instead of his long and difficult-to-spell full name. For that reason, in this book I try to always refer to him as "David" instead of the ever-tempting "DHH." When you encounter references to "David" without further qualification, I'm referring to the one-and-only David Heinemeier Hansson.

There are a number of notable people from the Rails world that are also referred to on a first-name basis in this book. Those include:

- *Yehuda* Katz
- *Jamis* Buck
- *Xavier* Noria

0.3 Goals

As already stated, I hope to make this your primary working reference for Ruby on Rails. I don't really expect too many people to read it through end to end unless they're expanding their basic knowledge of the Rails framework. Whatever the case may be, over time I hope this book gives you as an application developer/programmer greater confidence in making design and implementation decisions while working on your day-to-day tasks. After spending time with this book, your understanding of the fundamental concepts of Rails coupled with hands-on experience should leave you feeling comfortable working on real-world Rails projects, with real-world demands.

If you are in an architectural or development lead role, this book is not targeted to you, but should make you feel more comfortable discussing the pros and cons of Ruby on Rails adoption and ways to extend Rails to meet the particular needs of the project under your direction.

Finally, if you are a development manager, you should find the practical perspective of the book and our coverage of testing and tools especially interesting, and hopefully get some insight into why your developers are so excited about Ruby and Rails.

0.4 Prerequisites

The reader is assumed to have the following knowledge:

- Basic Ruby syntax and language constructs such as blocks
- Solid grasp of object-oriented principles and design patterns
- Basic understanding of relational databases and SQL
- Familiarity with how Rails applications are laid out and function
- Basic understanding of network protocols such as HTTP and SMTP
- Basic understanding of XML documents and web services
- Familiarity with transactional concepts such as ACID properties

As noted in the section "Book Structure," this book does not progress from easy material in the front to harder material in the back. Some chapters do start out with fundamental, almost introductory material and push on to more advanced coverage. There are definitely sections of the text that experienced Rails developer will gloss over. However, I believe that there is new knowledge and inspiration in every chapter, for all skill levels.

This page intentionally left blank

CHAPTER 9
Advanced Active Record

Active Record is a simple object-relational mapping (ORM) framework compared to other popular ORM frameworks, such as Hibernate in the Java world. Don't let that fool you, though: Under its modest exterior, Active Record has some pretty advanced features. To really get the most effectiveness out of Rails development, you need to have more than a basic understanding of Active Record—things like knowing when to break out of the one-table/one-class pattern, or how to leverage Ruby modules to keep your code clean and free of duplication.

In this chapter, we wrap up this book's comprehensive coverage of Active Record by reviewing callbacks, observers, single-table inheritance (STI), and polymorphic models. We also review a little bit of information about metaprogramming and Ruby domain-specific languages (DSLs) as they relate to Active Record.

9.1 Scopes

Scopes (or "named scopes" if you're old school) allow you define and chain query criteria in a declarative and reusable manner.

```
class Timesheet < ActiveRecord::Base
  scope :submitted, where(:submitted => true)
  scope :underutilized, where('total_hours < 40')
```

To declare a scope, use the `scope` class method, passing it a name as a symbol and some sort of query definition. If your query is known at load time, you can simply use Arel criteria methods like `where`, `order`, and `limit` to construct the definition as shown in the example. On the other hand, if you won't have all the parameters for your query until runtime, use a lambda as the second parameter. It will get evaluated whenever the scope is invoked.

```
class User < ActiveRecord::Base
  scope :delinquent, lambda { where('timesheets_updated_at < ?',
1.week.ago)}
```

Invoke scopes as you would class methods.

```
>> User.delinquent
=> [#<User id: 2, timesheets_updated_at: "2010-01-07 01:56:29"...>]
```

9.1.1 Scope Parameters

You can pass arguments to scope invocations by adding parameters to the lambda you use to define the scope query.

```
class BillableWeek < ActiveRecord::Base
  scope :newer_than, lambda { |date| where('start_date > ?', date) }
```

Then pass the argument to the scope as you would normally.

```
BillableWeek.newer_than(Date.today)
```

9.1.2 Chaining Scopes

One of the beauties of scopes is that you can chain them together to create complex queries from simple ones:

```
>> Timesheet.underutilized.submitted
=> [#<Timesheet id: 3, submitted: true, total_hours: 37 ...
```

Scopes can be chained together for reuse within scope definitions themselves. For instance, let's say that we always want to constrain the result set of underutilized to submitted timesheets:

```
class Timesheet < ActiveRecord::Base
  scope :submitted, where(:submitted => true)
  scope :underutilized, submitted.where('total_hours < 40')
```

9.1.3 Scopes and has_many

In addition to being available at the class context, scopes are available automatically on has_many association attributes.

```
>> u = User.find 2
=> #<User id: 2, login: "obie"...>

>> u.timesheets.size
=> 3
>> u.timesheets.underutilized.size
 => 1
```

9.1.4 Scopes and Joins

You can use Arel's `join` method to create cross-model scopes. For instance, if we gave our recurring example `Timesheet` a `submitted_at` date attribute instead of just a boolean, we could add a scope to `User` allowing us to see who is late on their timesheet submission.

```
scope :tardy, lambda {
  joins(:timesheets).
  where("timesheets.submitted_at <= ?", 7.days.ago).
  group("users.id")
}
```

Arel's `to_sql` method is useful for debugging scope definitions and usage.

```
>> User.tardy.to_sql
=> "SELECT users.* FROM users
    INNER JOIN timesheets ON timesheets.user_id = users.id
    WHERE (timesheets.submitted_at <= '2010-07-06 15:27:05.117700')
    GROUP BY users.id"  # query formatted nicely for the book
```


Note that as demonstrated in the example, it's a good idea to use unambiguous column references (including table name) in cross-model scope definitions so that Arel doesn't get confused.

9.1.5 Scope Combinations

Our example of a cross-model scope violates good object-oriented design principles: it contains the logic for determining whether or not a `Timesheet` is submitted, which is code that properly belongs in the `Timesheet` class. Luckily we can use Arel's merge method (aliased as `&`) to fix it. First we put the late logic where it belongs, in `Timesheet`:

```
scope :late, lambda { where("timesheet.submitted_at <= ?", 7.days.ago) }
```

Then we use our new `late` scope in `tardy`:

```
scope :tardy, lambda {
  joins(:timesheets).group("users.id") & Timesheet.late
}
```

If you have trouble with this technique, make absolutely sure that your scopes' clauses refer to fully qualified column names. (In other words, don't forget to prefix column names with tables.) The console and `to_sql` method is your friend for debugging.

9.1.6 Default Scopes

There may arise use cases where you want certain conditions applied to the finders for your model. Consider our timesheet application has a default view of open timesheets—we can use a default scope to simplify our general queries.

```
class Timesheet < ActiveRecord::Base
  default_scope :where(:status => "open")
end
```

Now when we query for our `Timesheets`, by default the open condition will be applied:

```
>> Timesheet.all.map(&:status)
=> ["open", "open", "open"]
```

Default scopes also get applied to your models when building or creating them, which can be a great convenience or a nuisance if you are not careful. In our previous example, all new `Timesheets` will be created with a status of "open."

```
>> Timesheet.new
=> #<Timesheet id: nil, status: "open">
>> Timesheet.create
=> #<Timesheet id: 1, status: "open">
```

You can override this behavior by providing your own conditions or scope to override the default setting of the attributes.

```
>> Timesheet.where(:status => "new").new
=> #<Timesheet id: nil, status: "new">
>> Timesheet.where(:status => "new").create
=> #<Timesheet id: 1, status: "new">
```

There may be cases where at runtime you want to create a scope and pass it around as a first class object leveraging your default scope. In this case, Active Record provides the `scoped` method.

```
>> timesheets = Timesheet.scoped.order("submitted_at DESC")
=> [#<Timesheet id: 1, status: "open"]
>> timesheets.where(:name => "Durran Jordan")
=> []
```

There's another approach to scopes that provides a sleeker syntax, `scoping`, which allows the chaining of scopes via nesting within a block.

```
>> Timesheet.order("submitted_at DESC").scoping do
>>   Timesheets.all
>> end
=> #<Timesheet id: 1, status: "open">
```

That's pretty nice, but what if we *don't* want our default scope to be included in our queries? In this case Active Record takes care of us through the `unscoped` method.

```
>> Timesheet.unscoped.order("submitted_at DESC")
=> [#<Timesheet id: 2, status: "submitted">]
```

Similarly to overriding our default scope with a relation when creating new objects, we can supply `unscoped` as well to remove the default attributes.

```
>> Timesheet.unscoped.new
=> #<Timesheet id: nil, status: nil>
```

9.1.7 Using Scopes for CRUD

You have a wide range of Active Record's CRUD methods available on scopes, which gives you some powerful abilities. For instance, let's give all our underutilized timesheets some extra hours.

```
>> u.timesheets.underutilized.collect(&:total_hours)
=> [37, 38]

>> u.timesheets.underutilized.update_all("total_hours = total_hours + 2")
=> 2

>> u.timesheets.underutilized.collect(&:total_hours)
=> [37, 38]  # whoops, cached result

>> u.timesheets(true).underutilized.collect(&:total_hours)
=> [39]  # results after telling association to reload
```

Scopes including a where clause using hashed conditions will populate attributes of objects built off of them with those attributes as default values. Admittedly it's a bit difficult to think of a plausible use case for this feature, but we'll show it in an example. First, we add the following scope to `Timesheet`:

```
scope :perfect, submitted.where(:total_hours => 40)
```

Now, building an object on the `perfect` scope should give us a submitted timesheet with 40 hours.

```
> Timesheet.perfect.build
 => #<Timesheet id: nil, submitted: true, user_id: nil, total_hours: 40
...>
```

As you've probably realized by now, the new Arel underpinnings of Active Record are tremendously powerful and truly elevate the Rails 3 platform.

9.2 Callbacks

This advanced feature of Active Record allows the savvy developer to attach behavior at a variety of different points along a model's life cycle, such as after initialization, before database records are inserted, updated or removed, and so on.

Callbacks can do a variety of tasks, ranging from simple things such as logging and massaging of attribute values prior to validation, to complex calculations. Callbacks can halt the execution of the life-cycle process taking place. Some callbacks can even modify the behavior of the model class on the fly. We'll cover all of those scenarios in this section, but first let's get a taste of what a callback looks like. Check out the following silly example:

```
class Beethoven < ActiveRecord::Base
  before_destroy :last_words

  protected

  def last_words
    logger.info "Friends applaud, the comedy is over"
  end
end
```

So prior to dying (ehrm, being `destroy`'d), the last words of the `Beethoven` class will always be logged for posterity. As we'll see soon, there are 14 different opportunities to add behavior to your model in this fashion. Before we get to that list, let's cover the mechanics of registering a callback.

9.2.1 Callback Registration

Overall, the most common way to register a callback method is to declare it at the top of the class using a typical Rails macro-style class method. However, there's a less verbose way to do it also. Simply implement the callback as a method in your class. In other words, I could have coded the prior example as follows:

```
class Beethoven < ActiveRecord::Base

  protected

  def before_destroy
    logger.info "Friends applaud, the comedy is over"
  end
end
```

This is a rare case of the less-verbose solution being bad. In fact, it is almost always preferable, dare I say it is the Rails way, to use the callback macros over implementing

callback methods, for the following reasons:

- Macro-style callback declarations are added near the top of the class definition, making the existence of that callback more evident versus a method body potentially buried later in the file.
- Macro-style callbacks add callback methods to a queue. That means that more than one method can be hooked into the same slot in the life cycle. Callbacks will be invoked in the order in which they were added to the queue.
- Callback methods for the same hook can be added to their queue at different levels of an inheritance hierarchy and still work—they won't override each other the way that methods would.
- Callbacks defined as methods on the model are always called last.

9.2.2 One-Liners

Now, if (and only if) your callback routine is really short,[1] you can add it by passing a block to the callback macro. We're talking one-liners!

```
class Napoleon < ActiveRecord::Base
  before_destroy { logger.info "Josephine..." }
  ...
end
```

As of Rails 3, the block passed to a callback is executed via `instance_eval` so that its scope is the record itself (versus needing to act on a passed in record variable). The following example implements "paranoid" model behavior, covered later in the chapter.

```
class Account < ActiveRecord::Base
  before_destroy { update_attribute(:deleted_at, Time.now); false }
  ...
```

9.2.3 Protected or Private

Except when you're using a block, the access level for callback methods should always be protected or private. It should never be public, since callbacks should never be called from code outside the model.

1. If you are browsing old Rails source code, you might come across callback macros receiving a short string of Ruby code to be evaluated in the binding of the model object. That way of adding callbacks was deprecated in Rails 1.2, because you're always better off using a block in those situations.

Believe it or not, there are even more ways to implement callbacks, but we'll cover those techniques further along in the chapter. For now, let's look at the lists of callback hooks available.

9.2.4 Matched before/after Callbacks

In total, there are 14 types of callbacks you can register on your models! Twelve of them are matching `before/after` callback pairs, such as `before_validation` and `after_validation`. (The other two, `after_initialize` and `after_find`, are special, and we'll discuss them later in this section.)

List of Callbacks

This is the list of callback hooks available during a `save` operation. (The list varies slightly depending on whether you're saving a new or existing record.)

- `before_validation`
- `before_validation_on_create`
- `after_validation`
- `after_validation_on_create`
- `before_save`
- `before_create` (for new records) and `before_update` (for existing records)
- (Database actually gets an `INSERT` or `UPDATE` statement here)
- `after_create` (for new records) and `after_update` (for existing records)
- `after_save`

Delete operations have their own two callbacks:

- `before_destroy`
- (Database actually gets a `DELETE` statement here)
- `after_destroy` is called after all attributes have been frozen (read-only)

Additionally transactions have callbacks as well, for when you want actions to occur after the database is guaranteed to be in a permanent state. Note that only "after" callbacks exist here because of the nature of transactions—it's a bad idea to be able to interfere with the actual operation itself.

- after_commit
- after_commit_on_create
- after_commit_on_update
- after_commit_on_destroy
- after_rollback
- after_rollback_on_create
- after_rollback_on_update
- after_rollback_on_destroy

9.2.5 Halting Execution

If you return a boolean `false` (not `nil`) from a callback method, Active Record halts the execution chain. No further callbacks are executed. The `save` method will return `false`, and `save!` will raise a `RecordNotSaved` error.

Keep in mind that because the last expression of a Ruby method is returned implicitly, it is a pretty common bug to write a callback that halts execution unintentionally. If you have an object with callbacks that mysteriously fails to save, make sure you aren't returning `false` by mistake.

9.2.6 Callback Usages

Of course, the callback you should use for a given situation depends on what you're trying to accomplish. The best I can do is to serve up some examples to inspire you with your own code.

Cleaning Up Attribute Formatting with `before_validation_on_create`

The most common examples of using `before_validation` callbacks have to do with cleaning up user-entered attributes. For example, the following `CreditCard` class cleans up its `number` attribute so that false negatives don't occur on validation:

```
class CreditCard < ActiveRecord::Base

  ...

  def before_validation_on_create
    # Strip everything in the number except digits
    self.number = number.gsub(/[^0-9]/, "")
  end
end
```

Geocoding with `before_save`

Assume that you have an application that tracks addresses and has mapping features. Addresses should always be geocoded before saving, so that they can be displayed rapidly on a map later.[2]

As is often the case, the wording of the requirement itself points you in the direction of the `before_save` callback:

```
class Address < ActiveRecord::Base
  include GeoKit::Geocoders

  before_save :geolocate
  validates_presence_of :street, :city, :state, :zip
  ...

  def to_s
    "#{street} #{city}, #{state} #{zip}"
end

  protected

  def geolocate
    res = GoogleGeocoder.geocode(to_s)
    self.latitude = res.lat
    self.longitude = res.lng
  end
end
```

Before we move on, there are a couple of additional considerations. The preceding code works great if the geocoding succeeds, but what if it doesn't? Do we still want to allow the record to be saved? If not, we should halt the execution chain:

```
def geolocate
  res = GoogleGeocoder.geocode(to_s)
  return false if not res.success   # halt execution

  self.latitude = res.lat
  self.longitude = res.lng
end
```

The only problem remaining is that we give the rest of our code (and by extension, the end user) no indication of why the chain was halted. Even though we're not in a validation routine, I think we can put the `errors` collection to good use here:

```
def geolocate
  res = GoogleGeocoder.geocode(to_s)
```

2. I recommend the excellent GeoKit for Rails plugin available at `http://geokit.rubyforge.org/`.

```
  if res.success
    self.latitude = res.lat
    self.longitude = res.lng
  else
    errors[:base] << "Geocoding failed. Please check address."
    return false
  end
end
```

If the geocoding fails, we add a base error message (for the whole object) and halt execution, so that the record is not saved.

Exercise Your Paranoia with `before_destroy`

What if your application has to handle important kinds of data that, once entered, should never be deleted? Perhaps it would make sense to hook into Active Record's destroy mechanism and somehow mark the record as deleted instead?

The following example depends on the `accounts` table having a `deleted_at` datetime column.

```
class Account < ActiveRecord::Base

  ...

  def before_destroy
    update_attribute(:deleted_at, Time.now)
    false
  end

end
```

I chose to implement it as a callback method so that I am guaranteed it will execute last in the `before_destroy` queue. It returns `false` so that execution is halted and the underlying record is not actually deleted from the database.[3]

It's probably worth mentioning that there are ways that Rails allows you to unintentionally circumvent `before_destroy` callbacks:

- The `delete` and `delete_all` class methods of `ActiveRecord::Base` are almost identical. They remove rows directly from the database without instantiating the corresponding model instances, which means no callbacks will occur.

3. Real-life implementation of the example would also need to modify all finders to include `deleted_at is NULL` conditions; otherwise, the records marked deleted would continue to show up in the application. That's not a trivial undertaking, and luckily you don't need to do it yourself. There's a Rails plugin named ActsAsParanoid by Rick Olson that does exactly that, and you can find it at `http://svn.techno-weenie.net/projects/plugins/acts_as_paranoid`.

- Model objects in associations defined with the option `:dependent => :delete_all` will be deleted directly from the database when removed from the collection using the association's `clear` or `delete` methods.

Cleaning Up Associated Files with `after_destroy`

Model objects that have files associated with them, such as attachment records and uploaded images, can clean up after themselves when deleted using the `after_destroy` callback. The following method from Rick Olson's old AttachmentFu[4] plugin is a good example:

```
# Destroys the file.  Called in the after_destroy callback
def destroy_file
  FileUtils.rm(full_filename)
  ...
rescue
  logger.info "Exception destroying   #{full_filename ... }"
  logger.warn $!.backtrace.collect { |b| " > #{b}" }.join("\n")
end
```

9.2.7 Special Callbacks: `after_initialize` and `after_find`

The `after_initialize` callback is invoked whenever a new Active Record model is instantiated (either from scratch or from the database). Having it available prevents you from having to muck around with overriding the actual `initialize` method.

The `after_find` callback is invoked whenever Active Record loads a model object from the database, and is actually called before `after_initialize`, if both are implemented. Because `after_find` and `after_initialize` are called for each object found and instantiated by finders, performance constraints dictate that they can only be added as methods, and not via the callback macros.

What if you want to run some code only the first time that a model is ever instantiated, and not after each database load? There is no native callback for that scenario, but you can do it using the `after_initialize` callback. Just add a condition that checks to see if it is a new record:

```
def after_initialize
  if new?
    ...
  end
end
```

4. Get AttachmentFu at `http://svn.techno-weenie.net/projects/plugins/attachment_fu`.

In a number of Rails apps that I've written, I've found it useful to capture user preferences in a serialized hash associated with the `User` object. The `serialize` feature of Active Record models makes this possible, since it transparently persists Ruby object graphs to a text column in the database. Unfortunately, you can't pass it a default value, so I have to set one myself:

```
class User < ActiveRecord::Base
  serialize :preferences # defaults to nil
  ...

  protected

    def after_initialize
      self.preferences ||= Hash.new
    end
end
```

Using the `after_initialize` callback, I can automatically populate the `preferences` attribute of my user model with an empty hash, so that I never have to worry about it being `nil` when I access it with code such as `user.preferences[:show_help_text] = false`.

Ruby's metaprogramming capabilities combined with the ability to run code whenever a model is loaded using the `after_find` callback are a powerful mix. Since we're not done learning about callbacks yet, we'll come back to uses of `after_find` later on in the chapter, in the section "Modifying Active Record Classes at Runtime."

9.2.8 Callback Classes

It is common enough to want to reuse callback code for more than one object that Rails gives you a way to write callback classes. All you have to do is pass a given callback queue an object that responds to the name of the callback and takes the model object as a parameter.

Here's our paranoid example from the previous section as a callback class:

```
class MarkDeleted
  def self.before_destroy(model)
     model.update_attribute(:deleted_at, Time.now)
     return false
  end
end
```

The behavior of `MarkDeleted` is stateless, so I added the callback as a class method. Now you don't have to instantiate `MarkDeleted` objects for no good reason. All you

do is pass the class to the callback queue for whichever models you want to have the mark-deleted behavior:

```
class Account < ActiveRecord::Base
  before_destroy MarkDeleted
  ...
end

class Invoice < ActiveRecord::Base
  before_destroy MarkDeleted
  ...
end
```

Multiple Callback Methods in One Class

There's no rule that says you can't have more than one callback method in a callback class. For example, you might have special audit log requirements to implement:

```
class Auditor
  def initialize(audit_log)
    @audit_log = audit_log
  end

  def after_create(model)
    @audit_log.created(model.inspect)
  end

  def after_update(model)
    @audit_log.updated(model.inspect)
  end

  def after_destroy(model)
    @audit_log.destroyed(model.inspect)
  end
end
```

To add audit logging to an Active Record class, you would do the following:

```
class Account < ActiveRecord::Base
  after_create Auditor.new(DEFAULT_AUDIT_LOG)
  after_update Auditor.new(DEFAULT_AUDIT_LOG)
  after_destroy Auditor.new(DEFAULT_AUDIT_LOG)
  ...
end
```

Wow, that's ugly, having to add three `Auditor`s on three lines. We could extract a local variable called `auditor`, but it would still be repetitive. This might be an opportunity to take advantage of Ruby's open classes, the fact that you can modify classes that aren't part of your application.

Wouldn't it be better to simply say `acts_as_audited` at the top of the model that needs auditing? We can quickly add it to the `ActiveRecord::Base` class, so that it's available for all our models.

On my projects, the file where "quick and dirty" code like the method in Listing 9.1 would reside is `lib/core_ext/active_record_base.rb`, but you can put it anywhere you want. You could even make it a plugin (as detailed in Chapter 19, "Extending Rails with Plugins").

Listing 9.1 A quick-and-dirty ''acts as audited'' method

```
class ActiveRecord::Base
  def self.acts_as_audited(audit_log=DEFAULT_AUDIT_LOG)
    auditor = Auditor.new(audit_log)
    after_create auditor
    after_update auditor
    after_destroy auditor
  end
end
```

Now, the top of `Account` is a lot less cluttered:

```
class Account < ActiveRecord::Base
  acts_as_audited
```

Testability

When you add callback methods to a model class, you pretty much have to test that they're functioning correctly in conjunction with the model to which they are added. That may or may not be a problem. In contrast, callback classes are super-easy to test in isolation.

```
def test_auditor_logs_created
  (model = mock).expects(:inspect).returns('foo')
  (log = mock).expects(:created).with('foo')
  Auditor.new(log).after_create(model)
end
```

9.3 Calculation Methods

All Active Record classes have a `calculate` method that provides easy access to aggregate function queries in the database. Methods for `count`, `sum`, `average`, `minimum`, and `maximum` have been added as convenient shortcuts.

Options such as `conditions`, `:order`, `:group`, `:having`, and `:joins` can be passed to customize the query.

There are two basic forms of output:

Single aggregate value The single value is type cast to `Fixnum` for `COUNT`, `Float` for `AVG`, and the given column's type for everything else.

Grouped values This returns an ordered hash of the values and groups them by the `:group` option. It takes either a column name, or the name of a `belongs_to` association.

The following options are available to all calculation methods:

`:conditions` An SQL fragment like `"administrator = 1"` or `[ "user_name = ?", username ]`. See conditions in the intro to ActiveRecord::Base.

`:include` Eager loading, see Associations for details. Since calculations don't load anything, the purpose of this is to access fields on joined tables in your conditions, order, or group clauses.

`:joins` An SQL fragment for additional joins like `"LEFT JOIN comments ON comments.post_id = id"`. (Rarely needed). The records will be returned read-only since they will have attributes that do not correspond to the table's columns.

`:order` An SQL fragment like `"created_at DESC, name"` (really only used with `GROUP BY` calculations).

`:group` An attribute name by which the result should be grouped. Uses the `GROUP BY` SQL-clause.

`:select` By default, this is `*` as in `SELECT * FROM`, but can be changed if you, for example, want to do a join, but not include the joined columns.

`:distinct` Set this to true to make this a distinct calculation, such as `SELECT COUNT(DISTINCT posts.id) ...`

The following examples illustrate the usage of various calculation methods.

```
  Person.calculate(:count, :all) # The same as Person.count

  # SELECT AVG(age) FROM people
  Person.average(:age)

  # Selects the minimum age for everyone with a last name other than
'Drake'
  Person.minimum(:age).where('last_name <> ?', 'Drake')
```

```
# Selects the minimum age for any family without any minors
Person.minimum(:age).having('min(age) > 17').group(:last_name)
```

9.3.1 average(column_name, *options)

Calculates the average value on a given column. The first parameter should be a symbol identifying the column to be averaged.

9.3.2 count(column_name, *options)

Count operates using three different approaches. Count without parameters will return a count of all the rows for the model. Count with a `column_name` will return a count of all the rows for the model with the supplied colum present. Lastly, count using :options will find the row count matched by the options used. In the last case you would send an options hash as the only parameter. 213

```
total_contacts = person.contacts.count(:from => "contact_cards")
```

Options are the same as with all other calculations methods with the additional option of :from which is by default the name of the table name of the class, however it can be changed to a different table name or even that of a database view. Remember that Person.count(:all) will not work because :all will be treated as a condition, you should use Person.count instead.

9.3.3 maximum(column_name, *options)

Calculates the maximum value on a given column. The first parameter should be a symbol identifying the column to be calculated.

9.3.4 minimum(column_name, *options)

Calculates the minimum value on a given column. The first parameter should be a symbol identifying the column to be calculated.

9.3.5 sum(column_name, *options)

Calculates a summed value in the database using SQL. The first parameter should be a symbol identifying the column to be summed.

9.4 Observers

The single responsibility principle is a very important tenet of object-oriented programming. It compels us to keep a class focused on a single concern. As you've learned in the previous section, callbacks are a useful feature of Active Record models that allow us to hook in behavior at various points of a model object's life cycle. Even if we pull that extra behavior out into callback classes, the hook still requires code changes in the model class definition itself. On the other hand, Active Record gives us a way to hook in to models that is completely transparent: Observers.

Here is the functionality of our old `Auditor` callback class as an observer of `Account` objects:

```
class AccountObserver < ActiveRecord::Observer
  def after_create(model)
    DEFAULT_AUDIT_LOG.created(model.inspect)
  end

  def after_update(model)
    DEFAULT_AUDIT_LOG.updated(model.inspect)
  end

  def after_destroy(model)
    DEFAULT_AUDIT_LOG.destroyed(model.inspect)
  end
end
```

9.4.1 Naming Conventions

When `ActiveRecord::Observer` is subclassed, it breaks down the name of the subclass by stripping off the "Observer" part. In the case of our `AccountObserver` in the preceding example, it would know that you want to observe the `Account` class. However, that's not always desirable behavior. In fact, with general-purpose code such as our `Auditor`, it's positively a step backward, so it is possible to overrule the naming convention with the use of the `observe` macro-style method. We still extend `ActiveRecord::Observer`, but we can call the subclass whatever we want and tell it explicitly what to observe using the `observe` method, which accepts one or more arguments.

```
class Auditor < ActiveRecord::Observer
  observe Account, Invoice, Payment

  def after_create(model)
    DEFAULT_AUDIT_LOG.created(model.inspect)
  end
```

```
  def after_update(model)
    DEFAULT_AUDIT_LOG.updated(model.inspect)
  end

  def after_destroy(model)
    DEFAULT_AUDIT_LOG.destroyed(model.inspect)
  end
end
```

9.4.2 Registration of Observers

If there weren't a place for you to tell Rails which observers to load, they would never get loaded at all, since they're not referenced from any other code in your application. Register observers with the following kind of code in an initializer:

```
# Activate observers that should always be running
ActiveRecord::Base.observers = Auditor
```


9.4.3 Timing

Observers are notified after the in-object callbacks are triggered.[5] It's not possible to act on the whole object from an observer without having the object's own callbacks executed first.

Durran says...

> For those of us who love to be organized, you can now put your observers in a separate directory under `app` if your heart desires. You won't need to perform custom loading anymore since Rails now loads all files under the `app` directory automatically.

9.5 Single-Table Inheritance (STI)

A lot of applications start out with a `User` model of some sort. Over time, as different kinds of users emerge, it might make sense to make a greater distinction between them. `Admin` and `Guest` classes are introduced, as subclasses of `User`. Now, the shared behavior can reside in `User`, and subtype behavior can be pushed down to subclasses. However, all user data can still reside in the `users` table—all you need to do is introduce a `type` column that will hold the name of the class to be instantiated for a given row.

5. `https://rails.lighthouseapp.com/projects/8994/tickets/230` contains an interesting discussion about callback execution order.

To continue explaining single-table inheritance, let's turn back to our example of a recurring `Timesheet` class. We need to know how many `billable_hours` are outstanding for a given user. The calculation can be implemented in various ways, but in this case we've chosen to write a pair of class and instance methods on the `Timesheet` class:

```
class Timesheet < ActiveRecord::Base
  ...

  def billable_hours_outstanding
    if submitted?
      billable_weeks.map(&:total_hours).sum
    else
      0
    end
  end

  def self.billable_hours_outstanding_for(user)
    user.timesheets.map(&:billable_hours_outstanding).sum
  end

end
```

I'm not suggesting that this is good code. It works, but it's inefficient and that `if/else` condition is a little fishy. Its shortcomings become apparent once requirements emerge about marking a `Timesheet` as paid. It forces us to modify `Timesheet`'s `billable_hours_outstanding` method again:

```
def billable_hours_outstanding
  if submitted? && not paid?
    billable_weeks.map(&:total_hours).sum
  else
    0
  end
end
```

That latest change is a clear violation of the open-closed principle,[6] which urges you to write code that is open for extension, but closed for modification. We know that we violated the principle, because we were forced to change the `billable_hours_outstanding` method to accommodate the new `Timesheet` status. Though it may not seem like a large problem in our simple example, consider the amount of conditional code that will end up in the `Timesheet` class once we start having to implement functionality such as `paid_hours` and `unsubmitted_hours`.

6. `http://en.wikipedia.org/wiki/Open/closed_principle` has a good summary.

So what's the answer to this messy question of the constantly changing conditional? Given that you're reading the section of the book about single-table inheritance, it's probably no big surprise that we think one good answer is to use object-oriented inheritance. To do so, let's break our original `Timesheet` class into four classes.

```
class Timesheet < ActiveRecord::Base
  # non-relevant code ommitted

  def self.billable_hours_outstanding_for(user)
    user.timesheets.map(&:billable_hours_outstanding).sum
  end
end

class DraftTimesheet < Timesheet
  def billable_hours_outstanding
    0
  end
end

class SubmittedTimesheet < Timesheet
  def billable_hours_outstanding
    billable_weeks.map(&:total_hours).sum
  end
end
```

Now when the requirements demand the ability to calculate partially paid timesheets, we need only add some behavior to a `PaidTimesheet` class. No messy conditional statements in sight!

```
class PaidTimesheet < Timesheet
  def billable_hours_outstanding
    billable_weeks.map(&:total_hours).sum - paid_hours
end
end
```

9.5.1 Mapping Inheritance to the Database

Mapping object inheritance effectively to a relational database is not one of those problems with a definitive solution. We're only going to talk about the one mapping strategy that Rails supports natively, which is single-table inheritance, called STI for short.

In STI, you establish one table in the database to holds all of the records for any object in a given inheritance hierarchy. In Active Record STI, that one table is named after the top parent class of the hierarchy. In the example we've been considering, that table would be named `timesheets`.

Hey, that's what it was called before, right? Yes, but to enable STI we have to add a `type` column to contain a string representing the type of the stored object. The following migration would properly set up the database for our example:

```
class AddTypeToTimesheet < ActiveRecord::Migration
  def self.up
    add_column :timesheets, :type, :string
  end

  def self.down
    remove_column :timesheets, :type
  end
end
```

No default value is needed. Once the type column is added to an Active Record model, Rails will automatically take care of keeping it populated with the right value. Using the console, we can see this behavior in action:

```
>> d = DraftTimesheet.create
>> d.type
=> 'DraftTimesheet'
```

When you try to find an object using the `find` methods of a base STI class, Rails will automatically instantiate objects using the appropriate subclass. This is especially useful in polymorphic situations, such as the timesheet example we've been describing, where we retrieve all the records for a particular user and then call methods that behave differently depending on the object's class.

```
>> Timesheet.find(:first)
=> #<DraftTimesheet:0x2212354...>
```

Sebastian says . . .

The word "type" is a very common column name and you might have plenty of uses for it not related to STI—which is why it's very likely you've experienced an `ActiveRecord::SubclassNotFound` error. Rails will read the "type" column of your `Car` class and try to find an "SUV" class that doesn't exist.The solution is simple: Tell Rails to use another column for STI with the following code:

```
set_inheritance_column "not_sti"
```

Note

Rails won't complain about the missing column; it will simply ignore it. Recently, the error message was reworded with a better explanation, but too many developers skim error messages and then spend an hour trying to figure out what's wrong with their models. (A lot of people skim sidebar columns too when reading books, but hey, at least I am doubling their chances of learning about this problem.)

9.5.2 STI Considerations

Although Rails makes it extremely simple to use single-table inheritance, there are a few caveats that you should keep in mind.

To begin with, you cannot have an attribute on two different subclasses with the same name but a different type. Since Rails uses one table to store all subclasses, these attributes with the same name occupy the same column in the table. Frankly, there's not much of a reason why that should be a problem unless you've made some pretty bad data-modeling decisions.

More importantly, you need to have one column per attribute on any subclass and any attribute that is not shared by all the subclasses must accept `nil` values. In the recurring example, `PaidTimesheet` has a `paid_hours` column that is not used by any of the other subclasses. `DraftTimesheet` and `SubmittedTimesheet` will not use the `paid_hours` column and leave it as null in the database. In order to validate data for columns not shared by all subclasses, you must use Active Record validations and not the database.

Third, it is not a good idea to have subclasses with too many unique attributes. If you do, you will have one database table with many null values in it. Normally, a tree of subclasses with a large number of unique attributes suggests that something is wrong with your application design and that you should refactor. If you have an STI table that is getting out of hand, it is time to reconsider your decision to use inheritance to solve your particular problem. Perhaps your base class is too abstract?

Finally, legacy database constraints may require a different name in the database for the `type` column. In this case, you can set the new column name using the class method `set_inheritance_column` in the base class. For the `Timesheet` example, we could do the following:

```
class Timesheet < ActiveRecord::Base
  set_inheritance_column 'object_type'
end
```

Now Rails will automatically populate the `object_type` column with the object's type.

9.5.3 STI and Associations

It seems pretty common for applications, particularly data-management ones, to have models that are very similar in terms of their data payload, mostly varying in their behavior and associations to each other. If you used object-oriented languages prior to Rails, you're probably already accustomed to breaking down problem domains into hierarchical structures.

Take for instance, a Rails application that deals with the population of states, counties, cities, and neighborhoods. All of these are places, which might lead you to define an STI class named `Place` as shown in Listing 9.2. I've also included the database schema for clarity:[7]

Listing 9.2 The places database schema and the place class

```
# == Schema Information
#
# Table name: places
#
#  id  :integer(11)  not null, primary key
#  region_id  :integer(11)
#  type  :string(255)
#  name  :string(255)
#  description  :string(255)
#  latitude  :decimal(20, 1)
#  longitude  :decimal(20, 1)
#  population  :integer(11)
#  created_at  :datetime
#  updated_at  :datetime

class Place < ActiveRecord::Base
end
```

`Place` is in essence an abstract class. It should not be instantiated, but there is no foolproof way to enforce that in Ruby. (No big deal, this isn't Java!) Now let's go ahead

7. For autogenerated schema information added to the top of your model classes, try Dave Thomas's annotate_models plugin at `http://svn.pragprog.com/Public/plugins/`

and define concrete subclasses of `Place`:

```
class State < Place
  has_many :counties, :foreign_key => 'region_id'
end

class County < Place
  belongs_to :state, :foreign_key => 'region _id'
  has_many :cities, :foreign_key => 'region _id'
end

class City < Place
  belongs_to :county, :foreign_key => 'region _id'
end
```

You might be tempted to try adding a `cities` association to `State`, knowing that `has_many :through` works with both `belongs_to` and `has_many` target associations. It would make the `State` class look something like this:

```
class State < Place
  has_many :counties, :foreign_key => 'region_id'
  has_many :cities, :through => :counties
end
```

That would certainly be cool, if it worked. Unfortunately, in this particular case, since there's only one underlying table that we're querying, there simply isn't a way to distinguish among the different kinds of objects in the query:

```
Mysql::Error: Not unique table/alias: 'places': SELECT places.* FROM
places INNER JOIN places ON places.region_id = places.id WHERE
((places.region_id = 187912) AND ((places.type = 'County'))) AND
((places.`type` = 'City' ))
```

What would we have to do to make it work? Well, the most realistic would be to use specific foreign keys, instead of trying to overload the meaning of `region_id` for all the subclasses. For starters, the `places` table would look like the example in Listing 9.3.

Listing 9.3 The places database schema revised

```
# == Schema Information
#
# Table name: places
#
#  id  :integer(11)  not null, primary key
#  state_id  :integer(11)
#  county_id  :integer(11)
#  type  :string(255)
#  name  :string(255)
#  description  :string(255)
```

```
#  latitude  :decimal(20, 1)
#  longitude  :decimal(20, 1)
#  population  :integer(11)
#  created_at  :datetime
#  updated_at  :datetime
```

The subclasses would be simpler without the `:foreign_key` options on the associations. Plus you could use a regular `has_many` relationship from `State` to `City`, instead of the more complicated `has_many :through`.

```
class State < Place
  has_many :counties
  has_many :cities
end

class County < Place
  belongs_to :state
  has_many :cities
end

class City < Place
  belongs_to :county
end
```

Of course, all those null columns in the places table won't win you any friends with relational database purists. That's nothing, though. Just a little bit later in this chapter we'll take a second, more in-depth look at polymorphic `has_many` relationships, which will make the purists positively hate you.

9.6 Abstract Base Model Classes

In contrast to single-table inheritance, it is possible for Active Record models to share common code via inheritance and still be persisted to different database tables. In fact, every Rails developer uses an abstract model in their code whether they realize it or not: `ActiveRecord::Base`.[8]

The technique involves creating an abstract base model class that persistent subclasses will extend. It's actually one of the simpler techniques that we broach in this chapter. Let's take the `Place` class from the previous section (refer to Listing 9.3) and revise it to

8. `http://m.onkey.org/2007/12/9/namespaced-models`

be an abstract base class in Listing 9.4. It's simple really—we just have to add one line of code:

Listing 9.4 The abstract place class

```
class Place < ActiveRecord::Base
  self.abstract_class = true
end
```

Marking an Active Record model abstract is essentially the opposite of making it an STI class with a type column. You're telling Rails: "Hey, I don't want you to assume that there is a table named `places`."

In our running example, it means we would have to establish tables for states, counties, and cities, which might be exactly what we want. Remember though, that we would no longer be able to query across subtypes with code like `Place.all`.

Abstract classes is an area of Rails where there aren't too many hard-and-fast rules to guide you—experience and gut feeling will help you out.

In case you haven't noticed yet, both class and instance methods are shared down the inheritance hierarchy of Active Record models. So are constants and other class members brought in through module inclusion. That means we can put all sorts of code inside `Place` that will be useful to its subclasses.

9.7 Polymorphic has_many Relationships

Rails gives you the ability to make one class `belong_to` more than one type of another class, as eloquently stated by blogger Mike Bayer:

> The "polymorphic association," on the other hand, while it bears some resemblance to the regular polymorphic union of a class hierarchy, is not really the same since you're only dealing with a particular association to a single target class from any number of source classes, source classes which don't have anything else to do with each other; i.e., they aren't in any particular inheritance relationship and probably are all persisted in completely different tables. In this way, the polymorphic association has a lot less to do with object inheritance and a lot more to do with aspect-oriented programming (AOP); a particular concept needs to be applied to a divergent set of entities which otherwise are not directly related. Such a concept is referred to as a cross-cutting concern, such as, all the entities in your domain need to support a history log of all changes to a common logging table. In the AR example, an Order and a User object are illustrated to both require links to an Address object.[9]

9. `http://techspot.zzzeek.org/?p=13`

In other words, this is not polymorphism in the typical object-oriented sense of the word; rather, it is something unique to Rails.

9.7.1 In the Case of Models with Comments

In our recurring Time and Expenses example, let's assume that we want both `BillableWeek` and `Timesheet` to have many comments (a shared `Comment` class). A naive way to solve this problem might be to have the `Comment` class belong to both the `BillableWeek` and `Timesheet` classes and have `billable_week_id` and `timesheet_id` as columns in its database table.

```
class Comment < ActiveRecord::Base
  belongs_to :timesheet
  belongs_to :expense_report
end
```

I call that approach is naive because it would be difficult to work with and hard to extend. Among other things, you would need to add code to the application to ensure that a `Comment` never belonged to both a `BillableWeek` and a `Timesheet` at the same time. The code to figure out what a given comment is attached to would be cumbersome to write. Even worse, every time you want to be able to add comments to another type of class, you'd have to add another nullable foreign key column to the comments table.

Rails solves this problem in an elegant fashion, by allowing us to define what it terms polymorphic associations, which we covered when we described the `:polymorphic => true` option of the `belongs_to` association in Chapter 7, Active Record Associations.

The Interface

Using a polymorphic association, we need define only a single `belongs_to` and add a pair of related columns to the underlying database table. From that moment on, any class in our system can have comments attached to it (which would make it commentable), without needing to alter the database schema or the `Comment` model itself.

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end
```

There isn't a `Commentable` class (or module) in our application. We named the association `:commentable` because it accurately describes the interface of objects that will be

associated in this way. The name `:commentable` will turn up again on the other side of the association:

```
class Timesheet < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class BillableWeek < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

Here we have the friendly `has_many` association using the `:as` option. The `:as` marks this association as polymorphic, and specifies which interface we are using on the other side of the association. While we're on the subject, the other end of a polymorphic `belongs_to` can be either a `has_many` or a `has_one` and work identically.

The Database Columns

Here's a migration that will create the `comments` table:

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
      t.text :body
      t.integer :commentable
      t.string       :commentable_type
    end
  end
end
```

As you can see, there is a column called `commentable_type`, which stores the class name of associated object. The Migrations API actually gives you a one-line shortcut with the `references` method, which takes a `polymorphic` option:

```
create_table :comments do |t|
  t.text :body
  t.references :commentable, :polymorphic => true
end
```

We can see how it comes together using the Rails console (some lines ommitted for brevity):

```
>> c = Comment.create(:text => "I could be commenting anything.")
>> t = TimeSheet.create
>> b = BillableWeek.create
>> c.update_attribute(:commentable, t)
=> true
>> "#{c.commentable_type}: #{c.commentable_id}"
=> "Timesheet: 1"
```

```
>> c.update_attribute(:commentable, b)
=> true
>> "#{c.commentable_type}: #{c.commentable_id}"
=> "BillableWeek: 1"
```

As you can tell, both the `Timesheet` and the `BillableWeek` that we played with in the console had the same id (1). Thanks to the `commentable_type` attribute, stored as a string, Rails can figure out which is the correct related object.

`has_many :through` and Polymorphics

There are some logical limitations that come into play with polymorphic associations. For instance, since it is impossible for Rails to know the tables necessary to join through a polymorphic association, the following hypothetical code, which tries to find everything that the user has commented on, will not work.

```
class Comment < ActiveRecord::Base
  belongs_to :user # author of the comment
  belongs_to :commentable, :polymorphic => true
end

class User < ActiveRecord::Base
  has_many :comments
  has_many :commentables, :through => :comments
end

>> User.first.comments
ActiveRecord::HasManyThroughAssociationPolymorphicError: Cannot have
a has_many :through association 'User#commentables' on the polymorphic
object 'Comment#commentable'.
```

If you really need it, `has_many :through` is possible with polymorphic associations, but only by specifying exactly what type of polymorphic associations you want. To do so, you must use the `:source_type` option. In most cases, you will also need to use the `:source` option, since the association name will not match the interface name used for the polymorphic association:

```
class User < ActiveRecord::Base
  has_many :comments
  has_many :commented_timesheets, :through => :comments,
           :source => :commentable, :source_type => 'Timesheet'
  has_many :commented_billable_weeks, :through => :comments,
           :source => :commentable, :source_type => 'BillableWeek'
end
```

It's verbose, and the whole scheme loses its elegance if you go this route, but it works:

```
>> User.first.commented_timesheets
=> [#<Timesheet ...>]
```

9.8 Foreign-key Constraints

As we work toward the end of this book's coverage of Active Record, you might have noticed that we haven't really touched on a subject of particular importance to many programmers: foreign-key constraints in the database. That's mainly because use of foreign-key constraints simply isn't the Rails way to tackle the problem of relational integrity. To put it mildly, that opinion is controversial and some developers have written off Rails (and its authors) for expressing it.

There really isn't anything stopping you from adding foreign-key constraints to your database tables, although you'd do well to wait until after the bulk of development is done. The exception, of course, is those polymorphic associations, which are probably the most extreme manifestation of the Rails opinion against foreign-key constraints. Unless you're armed for battle, you might not want to broach that particular subject with your DBA.

9.9 Using Value Objects

In Domain Driven Design[10] (DDD), a distinction is drawn between Entity Objects and Value Objects. All model objects that inherit from ActiveRecord::Base could be considered Entity Objects in DDD. An Entity object cares about identity, since each one is unique. In Active Record uniqueness is derived from the primary key. Comparing two different Entity Objects for equality should always return false, even if all of its attributes (other than the primary key) are equivalent.

Here is an example comparing two Active Record Addresses:

```
>> home = Address.create(:city => "Brooklyn", :state => "NY")
>> office = Address.create(:city => "Brooklyn", :state => "NY")
>> home == office
=> false
```

In this case you are actually creating two new Address records and persisting them to the database, therefore they have different primary key values.

Value Objects on the other hand only care that all their attributes are equal. When creating Value Objects for use with Active Record you do not inherit from

10. http://www.domaindrivendesign.org/

`ActiveRecord::Base`. Instead you make them part of a parent model using the `composed_of` class method. This is a form of composition, called an Aggregate in DDD. The attributes of the Value Object are stored in the database together with the parent object and `composed_of` provides a means to interact with those values as a single object.

A simple example is of a Person with a single Address. To model this using composition, first we need a Person model with fields for the Address. Create it with the following migration:

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.string :name
      t.string :address_city
      t.string :address_state
    end
  end
end
```

The Person model looks like this:

```
class Person < ActiveRecord::Base
  composed_of :address, :mapping => [%w(address_city city),
%w(address_state state)]
end
```

We'd need a corresponding Address object which looks like this:

```
class Address
  attr_reader :city, :state

  def initialize(city, state)
    @city, @state = city, state
  end

  def ==(other_address)
    city == other_address.city && state == other_address.state
  end
end
```

Note that this is just a standard Ruby object that does not inherit from `ActiveRecord::Base`. We have defined reader methods for our attributes and are assigning them upon initialization. We also have to define our own == method for use in comparisons. Wrapping this all up we get the following usage:

```
>> gary = Person.create(:name => "Gary")
>> gary.address_city = "Brooklyn"
>> gary.address_state = "NY"
>> gary.address
=> #<Address:0x20bc118 @state="NY", @city="Brooklyn">
```

Alternately you can instantiate the address directly and assign it using the address accessor:

```
>> gary.address = Address.new("Brooklyn", "NY")
>> gary.address
=> #<Address:0x20bc118 @state="NY", @city="Brooklyn">
```

9.9.1 Immutability

It's also important to treat value objects as immutable. Don't allow them to be changed after creation. Instead, create a new object instance with the new value instead. Active Record will not persist value objects that have been changed through means other than the writer method.

The immutable requirement is enforced by Active Record by freezing any object assigned as a value object. Attempting to change it afterwards will result in a `ActiveSupport::FrozenObjectError`.

9.9.2 Custom Constructors and Converters

By default value objects are initialized by calling the `new` constructor of the value class with each of the mapped attributes, in the order specified by the `:mapping` option, as arguments. If for some reason your value class does not work well with that convention, `composed_of` allows a custom constructor to be specified.

When a new value object is assigned to its parent, the default assumption is that the new value is an instance of the value class. Specifying a custom converter allows the new value to be automatically converted to an instance of value class (when needed).

For example, consider the `NetworkResource` model with `network_address` and `cidr_range` attributes that should be contained in a `NetAddr::CIDR` value class.[11] The constructor for the value class is called create and it expects a CIDR address string as a parameter. New values can be assigned to the value object using either another `NetAddr::CIDR` object, a string or an array. The `:constructor` and `:converter` options are used to meet the requirements:

```
class NetworkResource < ActiveRecord::Base
  composed_of :cidr,
              :class_name => 'NetAddr::CIDR',
              :mapping => [ %w(network_address network), %w(cidr_range
bits) ],
              :allow_nil => true,
```

11. Actual objects from the NetAddr gem available at `http://netaddr.rubyforge.org`

```
              :constructor => Proc.new { |network_address, cidr_range|
NetAddr::CIDR.create("#{network_address}/#{cidr_range}") },
              :converter => Proc.new { |value|
NetAddr::CIDR.create(value.is_a?(Array) ? value.join('/') : value) }
end

# This calls the :constructor
network_resource = NetworkResource.new(:network_address => '192.168.0.1',
:cidr_range => 24)

# These assignments will both use the :converter
network_resource.cidr = [ '192.168.2.1', 8 ]
network_resource.cidr = '192.168.0.1/24'

# This assignment won't use the :converter as the value is already an
instance of the value class
network_resource.cidr = NetAddr::CIDR.create('192.168.2.1/8')

# Saving and then reloading will use the :constructor on reload
network_resource.save
network_resource.reload
```

9.9.3 Finding Records by a Value Object

Once a `composed_of` relationship is specified for a model, records can be loaded from the database by specifying an instance of the value object in the conditions hash. The following example finds all customers with `balance_amount` equal to 20 and `balance_currency` equal to `"USD"`:

```
Customer.where(:balance => Money.new(20, "USD"))
```

The Money Gem

A common approach to using `composed_of` is in conjunction with the `money` gem.[12]

```
class Expense < ActiveRecord::Base
  composed_of :cost,
  :class_name => "Money",
  :mapping => [%w(cents cents), %w(currency currency_as_string)],
  :constructor => Proc.new do |cents, currency|
    Money.new(cents || 0, currency || Money.default_currency)
  end
end
```

Remember to add a migration with the 2 columns, the integer `cents` and the string `currency` that money needs.

12. `http://github.com/FooBarWidget/money/`

```
class CreateExpenses < ActiveRecord::Migration
  def self.up
    create_table :expenses do |table|
      table.integer :cents
      table.string :currency
    end
  end
  def self.down
    drop_table :expenses
  end
end
```

Now when asking for or setting the cost of an item would use a `Money` instance.

```
>> expense = Expense.create(:cost => Money.new(1000, "USD"))
>> cost = expense.cost
>> cost.cents
=> 1000
>> expense.currency
=> "USD"
```


9.10 Modules for Reusing Common Behavior

In this section, we'll talk about one strategy for breaking out functionality that is shared between disparate model classes. Instead of using inheritance, we'll put the shared code into modules.

In the section "Polymorphic `has_many` Relationships," we described how to add a commenting feature to our recurring sample Time and Expenses application. We'll continue fleshing out that example, since it lends itself to factoring out into modules.

The requirements we'll implement are as follows: Both users and approvers should be able to add their comments to a `Timesheet` or `ExpenseReport`. Also, since comments are indicators that a timesheet or expense report requires extra scrutiny or processing time, administrators of the application should be able to easily view a list of recent comments. Human nature being what it is, administrators occasionally gloss over the comments without actually reading them, so the requirements specify that a mechanism should be provided for marking comments as "OK" first by the approver, then by the administrator.

Again, here is the polymorphic `has_many :comments, :as => :commentable` that we used as the foundation for this functionality:

```
class Timesheet < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

```
class ExpenseReport < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end
```

Next we enable the controller and action for the administrator that list the 10 most recent comments with links to the item to which they are attached.

```
class Comment < ActiveRecord::Base
  scope :recent, order('created_at desc').limit(10)
end

class CommentsController < ApplicationController
  before_filter :require_admin, :only => :recent
  expose(:recent_comments) { Comment.recent }
end
```

Here's some of the simple view template used to display the recent comments.

```
%ul.recent.comments
  - recent_comments.each do |comment|
    %li.comment
      %h4= comment.created_at
      = comment.text
      .meta
        Comment on:
        = link_to comment.commentable.title, comment.commentable Yes, this
 would result in N+1 selects.
```

So far, so good. The polymorphic association makes it easy to access all types of comments in one listing. In order to find all of the unreviewed comments for an item, we can use a named scope on the Comment class together with the comments association.

```
class Comment < ActiveRecord::Base
  scope :unreviewed, where(:reviewed => false)
end

>> timesheet.comments.unreviewed
```

Both `Timesheet` and `ExpenseReport` currently have identical `has_many` methods for comments. Essentially, they both share a common interface. They're commentable!

To minimize duplication, we could specify common interfaces that share code in Ruby by including a module in each of those classes, where the module contains the code common to all implementations of the common interface. So, mostly for the sake of

example, let's go ahead and define a `Commentable` module to do just that, and include it in our model classes:

```
module Commentable
  has_many :comments, :as => :commentable
end

class Timesheet < ActiveRecord::Base
  include Commentable
end

class ExpenseReport < ActiveRecord::Base
  include Commentable
end
```

Whoops, this code doesn't work! To fix it, we need to understand an essential aspect of the way that Ruby interprets our code dealing with open classes.

9.10.1 A Review of Class Scope and Contexts

In many other interpreted OO programming languages, you have two phases of execution—one in which the interpreter loads the class definitions and says "this is the definition of what I have to work with," followed by the phase in which it executes the code. This makes it difficult (though not necessarily impossible) to add new methods to a class dynamically during execution.

In contrast, Ruby lets you add methods to a class at any time. In Ruby, when you type `class MyClass`, you're doing more than simply telling the interpreter to define a class; you're telling it to "execute the following code in the scope of this class."

Let's say you have the following Ruby script:

```
1  class Foo < ActiveRecord::Base
2    has_many :bars
3  end
4  class Foo < ActiveRecord::Base
5    belongs_to :spam
6  end
```

When the interpreter gets to line 1, you are telling it to execute the following code (up to the matching end) in the context of the `Foo` class object. Because the `Foo` class object doesn't exist yet, it goes ahead and creates the class. At line 2, we execute the statement `has_many :bars` in the context of the `Foo` class object. Whatever the `has_many` method does, it does right now.

When we again say class `Foo` at line 4, we are once again telling the interpreter to execute the following code in the context of the `Foo` class object, but this time, the

interpreter already knows about class `Foo`; it doesn't actually create another class. Therefore, on line 5, we are simply telling the interpreter to execute the `belongs_to :spam` statement in the context of that same `Foo` class object.

In order to execute the `has_many` and `belongs_to` statements, those methods need to exist in the context in which they are executed. Because these are defined as class methods in `ActiveRecord::Base`, and we have previously defined class `Foo` as extending `ActiveRecord::Base`, the code will execute without a problem.

However, when we defined our `Commentable` module like this:

```
module Commentable
  has_many :comments, :as => :commentable
end
```

. . . we get an error when it tries to execute the `has_many` statement. That's because the `has_many` method is not defined in the context of the `Commentable` module object.

Given what we now know about how Ruby is interpreting the code, we now realize that what we really want is for that `has_many` statement to be executed in the context of the including class.

9.10.2 The `included` Callback

Luckily, Ruby's `Module` class defines a handy callback that we can use to do just that. If a `Module` object defines the method `included`, it gets run whenever that module is included in another module or class. The argument passed to this method is the module/class object into which this module is being included.

We can define an `included` method on our `Commentable` module object so that it executes the `has_many` statement in the context of the including class (`Timesheet`, `ExpenseReport`, and so on):

```
module Commentable
  def self.included(base)
    base.class_eval do
      has_many :comments, :as => :commentable
    end
  end
end
```

Now, when we include the Commentable module in our model classes, it will execute the `has_many` statement just as if we had typed it into each of those classes' bodies.

The technique is common enough, within Rails and plugins, that it was added as a first-class concept in the Rails 3 ActiveSupport API. The above example becomes shorter and easier to read as a result:

```
module Commentable
  extend ActiveSupport::Concern
  included do
    has_many :comments, :as => :commentable
  end
end
```

Whatever is inside of the `included` block will get executed in the class context of the class where the module is included.

```
has_many :comments, :as => :commentable, :extend => Commentable
```

Courtenay says . . .

There's a fine balance to strike here. Magic like `include Commentable` certainly saves on typing and makes your model look less complex, but it can also mean that your association code is doing things you don't know about. This can lead to confusion and hours of head-scratching while you track down code in a separate module. My personal preference is to leave all associations in the model, and extend them with a module. That way you can quickly get a list of all associations just by looking at the code.

9.11 Modifying Active Record Classes at Runtime

The metaprogramming capabilities of Ruby, combined with the `after_find` callback, open the door to some interesting possibilities, especially if you're willing to blur your perception of the difference between code and data. I'm talking about modifying the behavior of model classes on the fly, as they're loaded into your application.

Listing 9.5 is a drastically simplified example of the technique, which assumes the presence of a `config` column on your model. During the `after_find` callback, we get a handle to the unique singleton class[13] of the model instance being loaded. Then we execute the contents of the `config` attribute belonging to this particular `Account` instance, using Ruby's `class_eval` method. Since we're doing this using the singleton class for this instance, rather than the global `Account` class, other account instances in the system are completely unaffected.

13. I don't expect this to make sense to you, unless you are familiar with Ruby's singleton classes, and the ability to evaluate arbitrary strings of Ruby code at runtime. A good place to start is `http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html`.

Listing 9.5 Runtime metaprogramming with **`after_find`**

```
class Account < ActiveRecord::Base

  ...

  protected

  def after_find
    singleton = class << self; self; end
    singleton.class_eval(config)
  end
end
```

I used powerful techniques like this one in a supply-chain application that I wrote for a large industrial client. A lot is a generic term in the industry used to describe a shipment of product. Depending on the vendor and product involved, the attributes and business logic for a given lot vary quite a bit. Since the set of vendors and products being handled changed on a weekly (sometimes daily) basis, the system needed to be reconfigurable without requiring a production deployment.

Without getting into too much detail, the application allowed the maintenance programmers to easily customize the behavior of the system by manipulating Ruby code stored in the database, associated with whatever product the lot contained.

For example, one of the business rules associated with lots of butter being shipped for Acme Dairy Co. might dictate a strictly integral product code, exactly 10 digits in length. The code, stored in the database, associated with the product entry for Acme Dairy's butter product would therefore contain the following two lines:

```
validates_numericality_of :product_code, :only_integer => true
validates_length_of       :product_code, :is => 10
```

9.11.1 Considerations

A relatively complete description of everything you can do with Ruby metaprogramming, and how to do it correctly, would fill its own book. For instance, you might realize that doing things like executing arbitrary Ruby code straight out of the database is inherently dangerous. That's why I emphasize again that the examples shown here are very simplified. All I want to do is give you a taste of the possibilities.

If you do decide to begin leveraging these kinds of techniques in real-world applications, you'll have to consider security and approval workflow and a host of other important concerns. Instead of allowing arbitrary Ruby code to be executed, you might

feel compelled to limit it to a small subset related to the problem at hand. You might design a compact API, or even delve into authoring a domain-specific language (DSL), crafted specifically for expressing the business rules and behaviors that should be loaded dynamically. Proceeding down the rabbit hole, you might write custom parsers for your DSL that could execute it in different contexts—some for error detection and others for reporting. It's one of those areas where the possibilities are quite limitless.

9.11.2 Ruby and Domain-Specific Languages

My former colleague Jay Fields and I pioneered the mix of Ruby metaprogramming, Rails, and internal[14] domain-specific languages while doing Rails application development for clients. I still occasionally speak at conferences and blog about writing DSLs in Ruby.

Jay has also written and delivered talks about his evolution of Ruby DSL techniques, which he calls Business Natural Languages (or BNL for short[15]). When developing BNLs, you craft a domain-specific language that is not necessarily valid Ruby syntax, but is close enough to be transformed easily into Ruby and executed at runtime, as shown in Listing 9.6.

Listing 9.6 Example of business natural language

```
employee John Doe
compensate 500 dollars for each deal closed in the past 30 days
compensate 100 dollars for each active deal that closed more than
365 days ago
compensate 5 percent of gross profits if gross profits are greater than
1,000,000 dollars
compensate 3 percent of gross profits if gross profits are greater than
2,000,000 dollars
compensate 1 percent of gross profits if gross profits are greater than
3,000,000 dollars
```

The ability to leverage advanced techniques such as DSLs is yet another powerful tool in the hands of experienced Rails developers.

14. The qualifier internal is used to differentiate a domain-specific language hosted entirely inside of a general-purpose language, such as Ruby, from one that is completely custom and requires its own parser implementation.
15. Googling BNL will give you tons of links to the Toronto-based band Barenaked Ladies, so you're better off going directly to the source at `http://bnl.jayfields.com`.

9.12 Conclusion

With this chapter we conclude our coverage of Active Record. Among other things, we examined how callbacks and observers let us factor our code in a clean and object-oriented fashion. We also expanded our modeling options by considering single-table inheritance, abstract classes and Active Record's distinctive polymorphic relationships.

At this point in the book, we've covered two parts of the MVC pattern: the model and the controller. It's now time to delve into the third and final part: the view.

Courtenay says . . .

DSLs suck! Except the ones written by Obie, of course. The only people who can read and write most DSLs are their original authors. As a developer taking over a project, it's often quicker to just reimplement instead of learning the quirks and exactly which words you're allowed to use in an existing DSL.In fact, a lot of Ruby metaprogramming sucks, too. It's common for people gifted with these new tools to go a bit overboard. I consider metaprogramming, `self.included`, `class_eval`, and friends to be a bit of a code smell on most projects.If you're making a web application, future developers and maintainers of the project will appreciate your using simple, direct, granular, and well-tested methods, rather than monkeypatching into existing classes, or hiding associations in modules.That said, if you can pull it off . . . your code will become more powerful than you can possibly imagine.

Index

B

C

R

S

T

This page intentionally left blank