Aptana RadRails:

An IDE for Rails Development

Develop Ruby on Rails applications fast using
RadRails 1.0 Community Edition

A comprehensive guide to using RadRails to develop your Ruby
on Rails projects in a professional and productive manner

Javier Ramírez

Aptana RadRails:
An IDE for Rails Development

Develop Ruby on Rails applications fast using
RadRails 1.0 Community Edition

A comprehensive guide to using RadRails to develop
your Ruby on Rails projects in a professional and
productive manner

Javier Ramírez

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Aptana RadRails: An IDE for Rails Development

Develop Ruby on Rails applications fast using RadRails 1.0 Community Edition

Credits

Author

Javier Ramírez

Reviewer

Chris Williams

Acquisition Editor

Shayantani Chaudhuri

Adil Rizwan

Technical Editor

Bhupali Khule

Editorial Team Leader

Akshara Aware

Project Manager

Abhijeet Deobhakta

Indexer

Hemangini Bari

Proofreader

Chris Smith

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Javier Ramírez has been developing Web Applications since before the term Web Application was coined. Born in Zaragoza, Spain, in 1974, he started programming as a hobby around the age of 11 assisted by his older sister. A few years later, he got his first modem and became a regular of BBSes and Newsgroups. His interest in developing server applications that can be accessed remotely comes from those times.

He has learned—and forgotten—many programming languages, including Basic, dBase III, Cobol, Pascal, SQL, C, C++, ASP, TCL, JavaScript, PHP, and Java, the language on which he has focused for most of his career so far. He has held the positions of programmer, analyst, consultant, team leader, post-sales engineer, project manager, and software architect, totaling over 12 years in the IT business.

Having developed projects mainly for banks and other big corporations in Spain, Italy, and the US, he co-founded some years ago a small software development shop, which provided him with valuable experience about the difficulties and the joys of entrepreneurship. After two years, he left the company in pursuit of new professional challenges.

For the last two years, he has been proudly working for ASPgems, where he discovered Ruby on Rails, which soon became his framework of choice for developing Web Applications. He is one of the organizers of the Spanish Rails Conference, also participating as a Speaker in the two events held so far.

He has also been an instructor on Robotics, Java, FatWire Content Server, and Ruby on Rails, and a University Lecturer in the subjects of 'Software Engineering' and 'The Java Programming Language', which he currently teaches at Universidad Francisco de Vitoria, in Madrid.

Javier Ramírez holds a B.Sc. in Business Information Systems with First Class Honors and a degree in Ingeniería en Sistemas de Computación

About the Reviewer

Chris Williams has spent the last four years working on the Ruby Development Tools project to bring Ruby tooling to the Eclipse IDE, and has been the lead developer of RadRails since joining Aptana in the first half of 2007. Prior to joining Aptana, Chris has worked in R&D for both Paychex Inc. and the Xerox Corporation. Chris lives with his wife and dog in Rochester, New York.

Table of Contents

Preface

Coming from a background of developing in languages such as Java, one of the things that surprised me the most about the Ruby and Rails community, was the common practice of not using an Integrated Development Environment. Most of the members of the community, including the most relevant, were comfortable with just a programmer's editor.

At first I thought it was because, Ruby being a dynamic language, using a full IDE might be an overkill. But then I thought of the PHP community, in which several IDEs are popular, with PHP also being a dynamic language. So I still had to guess why using an IDE was not a common practice within the Ruby on Rails world.

Nowadays, there is a growing list of IDEs with support for Ruby on Rails, but two years ago the options were really scarce. Back then, I chose to use RadRails because it worked on top of the Eclipse IDE—which was the tool I was already using for other programming languages—and because it was the only free, open source, and portable option.

Truth is, the first version of RadRails I used was very promising, but still a bit too basic. It featured just a few specialized tools, Ruby syntax colorization, and a slow and faulty code-assistance. As a result, the difference between RadRails and a good programmer's editor was not really significant. However, as Ruby on Rails gained popularity, RadRails was vastly improved, and a lot of new features were added.

At the same time, several other IDEs started to provide support for Ruby too. Today, even if many Ruby on Rails developers still don't use an IDE, a growing number of them already.

During these two years, I've been developing projects almost exclusively with Ruby on Rails; and I developed all of them using RadRails. Of course I have been keeping an eye on every new IDE with Ruby support, just to see if there were any reasons for changing, but I still didn't find any.

To me, writing this book is a way of contributing back to the RadRails project. I hope this book will help the existing community of users of Aptana RadRails, and will also help new users to start working with this tool. Besides, thanks to the Packt Open Source Project Royalty Scheme, a part of the benefits will be directly paid as a royalty to the RadRails project, so by purchasing this book you are funding a bit of the Community Edition of Aptana RadRails.

What This Book Covers

This book will show you how to get the most of the Community Edition of Aptana RadRails for developing Ruby on Rails projects. Apart from the features provided by RadRails, the book will give you an overview of working with the Eclipse IDE, and will show you how to use the Eclipse functionalities that are relevant for Ruby and Rails development.

This book is not about the Ruby programming language or the Ruby on Rails framework. Even if you don't need to be an expert, you should already be familiar with the language and the framework to get the most from this book.

Chapters 1 and 2 will show you how to install and configure Aptana RadRails, and will help you find your way around the Eclipse IDE. If you have previous experience with Eclipse , and you have already installed Aptana RadRails, then you can proceed directly to Chapter 3.

Chapters 3 to 8 are a complete reference to each of the components of RadRails, including all the configuration options.

Finally, in Chapter 9 you will find documentation about some complementary plugins you can use for connecting to a database and for managing your source repositories.

You can find below a brief introduction to each of the chapters.

Chapter 1: This chapter will introduce you the concept of IDE and will give you a general overview of what you can expect from Aptana RadRails. You will also find instructions about how to install Aptana RadRails and the Eclipse IDE in your system. Even if you should already be familiar with the installation of Ruby and Rails, the chapter also provides a quick reference for installing Ruby and Ruby on Rails on Windows, Linux, and OSX.

Chapter 2: In most cases, Aptana RadRails will work directly out of the box. However, in some cases you will need to make a minimal configuration of the IDE. The first part of this chapter will show you the basic configuration of RadRails.

Chapter 3: Two of the basic tools RadRails provides are the Ruby Explorer and the Console View. With the Ruby explorer you will be able to browse the structure of your projects and perform any kind of file-related operations, including working with the local history of your files. The console view will display the output of most of the processes we will launch from RadRails. Apart from learning how to use these views, we will show how to use Generators and Rake Tasks from Aptana RadRails to create a simple demo application. You will also learn how to start and stop your servers and how to use the built-in browser to watch your application in action.

Chapter 4 explains in detail all the built-in capabilities of RadRails for developing Ruby code. You will learn to use the Ruby Editor to write your source code, to navigate between the different classes and files, and to get the most out of code completion and the code templates.

Chapter 5: One of the strong points of Aptana RadRails is the great support for the client-side of your application: JavaScript, HTML, and CSS. In this chapter you will learn how to write Rails views mixing together Ruby code with HTML or JavaScript and getting assistance for all of the languages.

Chapter 6: When an application grows large, it's always a good idea to have a way of debugging the potential errors. This chapter will show you how to use RadRails' built-in debugger for interacting with your code at run time. You will learn to start a server or a stand-alone script in debug mode, how to set breakpoints , and how to intercept any Ruby exceptions. The debugger will also allow you to walk through your code, to examine the values of any variables and expressions, and even to execute arbitrary code at run time by using the Display view.

Chapter 7: Apart from the coding and debugging, Aptana RadRails provides a number of specialized tools to make the development and management of your application easier. In the context of Eclipse, each of these tools is called a View. In this chapter, you will learn how to use the different views to browse the Ruby and Rails documentation, manage and monitor your servers, install gems and plugins, launch generators and rake tasks, use code annotations, keep track of warnings and to-do lists, evaluate regular expressions, and run your tests. If you prefer to use the command line, then you will learn how to take advantage of the built-in Rails Shell, in which you can get auto-completion for the most used Ruby and Rails commands directly at the command line. This chapter will also show you how to use your IDE to control external servers such as Apache or MySQL.

Chapter 8: Out of the box, Aptana RadRails provides a fully working environment. However, many of its components allow for some configuration. This chapter is a complete reference to all the preferences you can set to change the user experience when using RadRails.

Chapter 9: Aptana RadRails bundles together plenty of interesting features for the developer. However, since the focus is on Ruby on Rails, there are some general aspects of the development of a project that are not covered by RadRails. Fortunately, since the underlying platform is the Eclipse IDE, we have a virtually unlimited number of complementary plugins to choose from. This chapter will give you a general overview of the Eclipse plugins ecosystem, and will also explain in detail how to use two of the plugins you might want to use when developing. DBViewer is a plugin you can use to connect to your database from the IDE. This chapter will show you how to set up the plugin, and how to use it for examining and modifying your database structure and contents. Subclipse is a plugin to connect to Subversion repositories. By using Subclipse you will have repository access directly from your IDE. Besides, the built-in features of Subclipse will help you examine and merge changes in a much more comfortable way than using the Subversion command line.

What You Need for This Book

In order to install Aptana RadRails, you will need the following:

- Java Virtual Machine (version 1.5 or higher), preferably the Sun JVM is preferred.
- Ruby
- Ruby on Rails (version 2.0 or higher)
- The database of your choice, with the proper Ruby gems to establish a connection from Ruby on Rails
- Connection to the Internet to download/install the different components

As a part of the installation process, Aptana RadRails will automatically guide you through the installation of Ruby and Ruby on Rails if they are not available in your system. You will have to manually install at least a JVM and the database manager of your choice.

Even though the installation of all the required components is out of the scope of this book, you will find in chapter number one a quick guide to installing the JVM, Ruby and Ruby on Rails. This reference has been included for completeness and it's not intended to be exhaustive.

Who This Book Is For

This book is for Ruby on Rails developers who want to make the most of the framework by using an Integrated Development Environment.

Even though the book explains everything you need to follow the contents, the focus is on how to use the tool and not on the Rails framework itself, so previous working knowledge of Rails is highly advisable. Previous knowledge of Eclipse is not necessary.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "RadRails would allow me to introduce the `ModelName`, the `first_field`, and the `second_field`".

A block of code will be set as follows:

```
begin
  ${cursor}
  ${line_selection}
end
```

Any command-line input and output is written as follows:

\dev\ruby186\bin\mongrel_rails start --port 3100 -o 1

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "Take your time and when you are ready for the installation select the option **Go to Workbench**".

> Important notes appear in a box like this.

> Tips and tricks appear like this.

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/files/code/3988_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Getting Started

You already know how productive, addictive, and fun Ruby on Rails development can be. Yet you often find yourself using the command line, navigating through your source files to find method definitions or references, debugging by printing to the logs, going to the shell for launching or stopping your servers, and basically changing context many times as you are working.

Not to be picky, but when I work with Java or C++ I can navigate my source files just by clicking on the variables or the method names, I can debug step by step in a graphical environment, and I get refactoring and code completion too. I can also control my servers, databases, log files, and everything from a single tool. No command line, no context change.

Now wait a second, am I telling you that working with Java or C++ can be more fun, more productive, or more comfortable than doing it with Rails? No way! You have come to the right place for taking your love for Rails a step further. You are going to learn how to use a solid, enterprise-proven tool for your Ruby on Rails development.

Do I Need an IDE for Rails Development?

First of all, let's see what an IDE is. The name stands for **Integrated Development Environment**, meaning that you can access most of the ordinary development tasks from a single interface. This is very convenient because you don't need to start several applications, or learn different interfaces when you are working in your projects. Once you know your way around your IDE, even new functionalities look familiar since they all are a part of the same tool.

If you've been programming in static languages like C, C++, or Java you already know there are several mandatory steps in order to get something working. You don't only need to write the code, but also to compile it, link and make it executable, or maybe package it, and if you are using some framework with lots of configuration then you have, in addition, to deal with all those files. And don't get me started

about deploying to your own development server. In these cases, the advantage is so obvious that it's a no-brainer to go for a tool in which the whole process from coding to executing is (almost) transparent. That's why even such primitive tools as TurboPascal or TurboC were so appreciated as they saved you a lot of repetitive key punching.

Now when it comes to dynamic languages, and more specifically to Rails, the development cycle is much shorter. You write the code and unless you are changing some non-reloadable file you can directly test your results in your browser. No compiling, linking, packaging, or deploying. Apparently it would seem that an IDE is not such a good idea here.

Now-a-days IDEs provide much more than a simple task automation. They are really helpers that can auto-complete your code, format it, hint about possible errors even as you are typing the source code, assist in refactoring, and integrate with several other tools. Even if Rails is such a comfortable framework by itself, certainly it could use some of these capabilities.

It has also been said that code assist is not so good with dynamic languages. Since you cannot infer the types statically before execution, it would seem almost impossible to get auto-completion or source navigation by linking on the variables and methods. Well, there is some truth in that. Certainly statically typed languages are much easier to deal with for those matters, and the support for these operations is still not as good in Ruby as it is in, for example, Java or C. However, since the first IDE for Ruby made it to the public, the techniques for dealing with code assist have improved greatly and are now very close to those of other languages.

Working with an IDE is pretty different than doing it with a power editor. Of course a good programmers' editor can make development much more comfortable than using a plaintext editor with no macros or highlighting, but when it comes to more sophisticated tasks, they are limited by the amount of logic that you can embed in them.

Nevertheless, it's not a secret that the whole core team of Rails doesn't use an IDE, but a pretty good power editor, so you really don't need an IDE for Rails development; but if you use one, you are going to find yourself dedicating less time to routine tasks and concentrating on the creative and productive part, which is what Agile Development is about.

About Aptana RadRails

Currently, there are several options to choose from if you are looking for a Rails IDE. One of the reasons I like Aptana RadRails the best is because it's built on top of Eclipse. Eclipse (`http://www.eclipse.org`) is a multi-platform, multi-language, well-tested, enterprise-ready IDE with a lot of plugins for extending its capabilities. It was initially developed by IBM and then released as an Open Source project, being since then a total success.

RadRails was initially started by the middle of 2005 by Matt Kent, Kyle Shank, and Marc Baumbach, who were at the time working for IBM Rational. The RadRails project was built upon the Ruby Development Tools—RDT—plugin, which already supported pure Ruby, but not Rails, development on Eclipse.

After more than a year, part of the core team started their own company, not being able to dedicate enough time to working in the plugin. This was made public in some mailing lists and blogs, causing the community of RadRails users to worry about the future of the IDE. Fortunately the project finally found the support of another company, Aptana, to take over the development of the tool.

Aptana already had a very good Eclipse plugin for JavaScript development, so it had the knowledge and resources to give continuity to the project. Not only that, but Aptana also contacted Chris Williams, the lead developer for over three years of the Ruby Development Tools project, and offered him the opportunity to join Aptana and work full-time on both plugins.

Since Aptana took charge, the plugin has been integrated with the Aptana editors and a number of improvements have been made. Now the plugin is available both in Professional and Community Editions, and it's still an Open Source project, with the lead developer being Christopher Williams. Since the integration, the official name of the project is Aptana RadRails.

To me, this story illustrates an important point. When working with Open Source tools, even when the original team cannot continue the development, it's possible for the community to enjoy the benefits of the tool and to have new releases coming. This is another reason why I prefer RadRails over some other Rails IDE.

If by now you have been browsing Aptana's website (`http://www.aptana.com`), you will have realized Aptana RadRails comes in two different flavors: as a stand-alone IDE or as an Eclipse plugin. The stand-alone edition also installs Eclipse behind the scenes, but some of the Eclipse functionalities like some start-up arguments will not be available if you choose this option. Also, if you are planning to install external plugins, it's probably better to use the standard Eclipse installation and then install Aptana RadRails on top of that.

How Can Aptana RadRails Help Me?

One thing you have to remember is when we are installing RadRails we will not only be able to use the capabilities of Aptana RadRails, which are indeed impressive enough, but also those of Eclipse and the whole ecosystem of plugins available. I have been using Eclipse for working with Java, PHP, C, and for the last two years, almost exclusively for Ruby on Rails. Here is a list of some of the things that I do on a regular basis from my Eclipse plus Aptana RadRails workbench:

- Edit different types of files with syntax highlighting, code assist, code completion, formatting, and error checks. You can edit your Ruby or Rails source code, but also HTML, JavaScript, CSS, YAML, or XML.

- When working on Rails views, the tool is smart enough to know the context in which I am and give me Ruby, HTML, or JavaScript support accordingly. As an example, CSS support not only provides code assist, but also tells me if a given style element is supported only by some browsers.

- Navigate from the code of the view to that of the controller or the model with a keystroke, mark occurrences of any variable with a single click, or go to a class or method definition just by clicking over the source code.

- Use Ruby or Rails command-line tools from a graphic environment. I can run rake tasks, code generators, or tests. I can also install, remove, or update Rails plugins and Ruby gems.

- Manage my database. There are many different plugins to help you work against your database, and many of them are vendor independent, so you can work with different DBMS.

- Start or stop servers and monitor their output. At the moment of writing this book there is support for Webrick and Mongrel servers. JRuby can also be used, but it is not yet integrated and requires some hacking.

- Tail several log files directly from a tabbed window in the workbench.

- Use the Rails console in an integrated window. One of my favorite features of Rails is going to the console to test pieces of code before putting them into my source files. It's very convenient that I can just copy and paste or drag the selection between windows in the workbench for this purpose.

- Browse the different API documentation. Not only Ruby core, Ruby Standard Library, or Rails docs, but also a JavaScript reference, for example.

- Version control: I can work with my repository, create or download projects, modify them, compare my local copy against the version in the repository, and any other functionality your version control software supports in a graphic and consistent manner. I have used it with CVS, Subversion, Perforce, and Visual Source Safe, but the list of available plugins is larger.

- Bug Tracking: You can use different bug trackers and have them integrated with Eclipse, so you can directly get the bug report, modify the code, update it in the repository, and close the bug report from the same workbench. I have been using it exclusively with Trac, but support for Bugzilla and custom trackers is also built-in.
- Write and test regular expressions.
- Keep track of problems and To-Do tasks in my projects.
- Debug my applications graphically. I can set breakpoints just by clicking on a line, then go step by step, or step into or over the code and watching the different values at run time.

Even if the list above is cool enough, there are more things you can do with Aptana RadRails, so let's see what you need in order to install the IDE and start enjoying it at once.

Prerequisites

This is not a book about learning the Ruby language or the Rails framework, but about how to take advantage of Aptana RadRails to help you develop Rails applications. You should already have working knowledge of Rails and you should have your system set up for Rails development, or at least feel comfortable with the steps involved in doing so. Also, in order to install the Eclipse IDE, a Java Virtual Machine (JVM) will be mandatory.

Even if it is not the purpose of this book to provide exhaustive instructions on the Ruby or Rails and JVM installation processes, you can find below some guidelines on how to set up your system in case you are starting from scratch. Be sure you have all the necessary items properly installed in your system before proceeding to Eclipse and RadRails installation.

Java Virtual Machine

As you already know by now, RadRails is built on top of the Eclipse IDE. One of the features of Eclipse is its portability, which is possible by using Java behind the scenes. This means you will need a Java Virtual Machine (JVM) installed in order for Eclipse to work.

Chances are you already have at least a JVM installed in your system, but you have to make sure you are using version 1.5 or higher, since Aptana RadRails will not work with previous releases. Also, some users have reported intermittent crashes, slower performance, and a higher memory footprint when using a JVM other than that of Sun, so in order to avoid potential problems, it's highly recommended to use the official Sun Java Virtual Machine.

However, Eclipse can run on older versions of the JVM, and if for any reason you cannot install a JVM later than 1.4, then you could still use an old version of RadRails. RadRails 0.7.2 also known as **RadRails Classic** was the last release before Aptana took control of the development, and it's still available for download. You should be aware there is no official support for it and there are a lot of new features and bug fixes in the next releases. If you install RadRails Classic, you will not find many of the options that we will be discussing in this book.

A typical Java installation will include the executable 'java' in the path of your system/user, so if you are not sure if it is installed in your system or what version you have you can check it by executing the following line at a command prompt:

```
java –version
```

This should produce an output similar to:

$ java -version
java version "1.5.0_05"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_05-b05)
Java HotSpot(TM) Client VM (build 1.5.0_05-b05, mixed mode)

If you don't have any JVM or your version is prior to 1.5, you should install one before trying to install Eclipse. Since we are not talking about developing Java applications, a minimal set up will do and installing the Java Runtime Environment will suffice our purposes.

Linux and Windows users can get the package directly from Sun at `http://java.sun.com/javase/downloads`. Mac users can get the appropriate software from `http://developer.apple.com/java`.

Ruby and Rails

RadRails uses Ruby and Rails in the background for many of its functionalities. For this reason, when you install RadRails, the plugin will try to find Ruby and Rails in your system. If they are already installed and the plugin finds them, it will self-configure properly.

In case Ruby or Rails are installed, but they are not at the default locations, RadRails will offer you the possibility of entering the location of the executables in your system.

If you have Ruby installed in your system but you don't have Rails, RadRails will assist you in downloading and installing Rails automatically.

If you don't have Ruby or Rails installed, then RadRails will take you to the download location but you will have to finish the process of installing Ruby manually.

You can find below a quick reference in case you need to manually install Ruby or Rails.

Components to Install

The basic components of a typical Rails installation are Ruby (together with RDoc and ri for browsing the API documentation), RubyGems, Rails, and the database adaptor of your choice (Rails includes a non-optimized version of a MySQL adaptor out of the box).

Ruby and RubyGems can be obtained as source code or as a binary package for your system, the second option being much handier since you don't need to mess with the internals of the compilation; but of course it gives you less control over the installation.

For the typical user, a binary package should be good enough, and if you are as lazy as I am, you will find it much more convenient than dealing with a full setup from the source. A bit further in this section you can find some options for installing a binary package depending on your OS.

Whether you decide to use the fast track and go for a binary distribution or walk the path of the glory reserved for the brave by compiling from the source, you will find an excellent starting point at the official download site of the Ruby language (`http://www.ruby-lang.org/en/downloads/`). There you will find the source code and some pointers as how to compile it and where to find up-to-date binary distributions for your environment.

Now let's make a quick tour of the options available for getting up and running with Rails in each of the Operating Systems.

Installing Rails if you already have Ruby and RubyGems

If Ruby is already installed in your system, then there is no need for you to install the whole binary package, unless you prefer so, and you could directly install Rails on top of your current Ruby setup. If Ruby is not installed in your system, you can skip this entry and find the detailed instructions for your particular OS later in this chapter.

If you are running Linux or OS-X, chances are you already have Ruby as a part of your distribution. No version of Windows, at least not yet, comes with Ruby bundled, but of course you will have it already if you have installed it in the past for developing pure Ruby (not Rails) scripts. You can check if Ruby is installed by opening a command prompt and running:

```
ruby –v
```

If Ruby is installed (and in the path), you should be seeing something like

$ ruby -v
ruby 1.8.6 (2007-09-24 patchlevel 111) [i386-linux]

Just take into account that some distributions could be using a fairly old release and you will need at least Ruby 1.8.4 for running Rails.

Apart from Ruby, you will need RubyGems in your system before you can install Rails. RubyGems is a very popular package manager for the Ruby language, making the installation of new packages very easy. In most Ruby installations RubyGems will be already present without any further steps. To check if it's installed in your system, you can open a command prompt and type:

```
gem –v
```

If you get a positive response, then you can directly go and install Rails. In case you don't have it, you can get it from `http://rubyforge.org/projects/rubygems`. The installation package is just a ZIP file. You will find detailed instructions on how to install it inside, but it should be as easy as unzipping the contents of the package and executing from the command prompt:

```
ruby setup.rb
```

Once we have made sure the package manager RubyGems is installed in our system, we just have to use it for installing Rails. To do so, all you have to do is go back again to the command prompt and execute:

```
gem install rails --include-dependencies
```

This will cause RubyGems to connect to a download site, get all the necessary packages, and install them in your system. Please be aware that you will need Internet connectivity in order for this command to work. Besides, if you are behind a proxy, you could experience installation problems. If that's the case, you can try to get detailed information at `http://rubygems.org` or you can just install one of the binary packages for your OS. The binary packages already include RubyGems as a part of the distribution.

If you have completed the Rails installation successfully, then you can skip the next three entries under this section and go directly to *Supported Databases*.

Installing Ruby and Rails on Linux

As we stated before, we will not cover the steps of building from the sources, but those of obtaining a binary distribution.

For the other platforms, there are some binary packages that will install Ruby, Rails, and some useful libraries with a single action. Unfortunately at the moment of writing this book, there is nothing of the like for Linux systems. However, the whole process is pretty easy and will require just a few steps.

For our installation we are going to use a package manager. Package managers come mainly in two flavors in the Linux world. For those distros based on RedHat, the **yum** package manager is used, and for the Debian-based distros, we will use **apt**. Both package managers will connect to the Internet in order to download the necessary files, so make sure you have connectivity before trying to install.

Whatever your package manager is, take into account you will need to install the packages with superuser privileges, so log in as root or use the 'sudo' command when installing.

Installing Ruby and Rails Using yum

For installing Ruby using yum you should open a command prompt and type:

```
yum install ruby ruby-irb ruby-libs ruby-rdoc ruby-mode ruby-ri ruby-docs
```

Please notice the line above should be written in a single command line, with no line breaks. Also notice over time some of the packages can change their names, so maybe when you are trying to install there will be some differences. You can always use 'yum list' or 'yum search' to browse the complete package list.

Now we will use yum to install RubyGems. Execute:

```
yum install rubygems
```

Finally we'll use RubyGems, the Ruby package manager, for installing Rails. Execute:

```
gem install rails --include-dependencies
```

After RubyGems finishes, you are all set and ready to proceed to section *Supported Databases*.

Installing Ruby and Rails Using apt

To install ruby using apt you should open a command prompt and type:

```
apt-get install ruby rubygems irb ri rdoc ruby1.8-dev build-essential
```

Please notice the preceding line should be written in a single command line, with no line breaks. Also notice over time some of the packages can change their names, so maybe when you are trying to install there will be some differences. You can always use 'apt-cache search' to browse the complete package list.

Finally we'll use RubyGems, the Ruby package manager, for installing Rails. Execute:

```
gem install rails --include-dependencies
```

After RubyGems finishes, you are all set and ready to proceed to section *Supported Databases*.

Installing Ruby and Rails on OS X

Since OS X is a Linux-based system, compiling Ruby from the source is one of the options available. However, in this section we are going to focus on the binary distributions.

One of the most popular options when installing Rails over OS X is using Locomotive (`http://locomotive.raaum.org/`). Installation is as easy as can be. You only need to download the package and execute the file `Locomotive.app` inside. Locomotive will install Ruby, RubyGems, Rails, and many useful libraries for working with Ruby. The installation is completely separate from the default Ruby bundled with OS X, so it should not have any impact on your system.

If you prefer to install Ruby and Rails separately or you prefer not to include so many libraries, there are another two popular options. For installing Ruby you can use the One-click Installer for OS X (`http://rubyosx.rubyforge.org/`) or use MacPorts (`http://www.macports.org/`).

If using the One-click installer, you only have to download it from the web page and install it in your system. For MacPorts you must open a command prompt and type:

```
port install ruby rb-rubygems
```

In any case, after the installation you will already have Ruby and RubyGems in your system and, in the case of the One-click installer, some useful additional utilities. Now we only have to install Rails on top of that. To do so, just execute:

```
gem install rails --include-dependencies
```

After RubyGems finishes, you are all set and ready to proceed to section *Supported Databases*.

Installing Ruby and Rails on Windows

Compiling Ruby from the source is certainly possible but terribly hackish on a Windows system, so it's done only by a few users that need to modify the standard distribution in some way.

Fortunately, for the rest of us there are two popular binary distributions that we can use: InstantRails and the One-click Installer for Windows.

The One-click Installer includes Ruby, RubyGems, and some useful libraries. InstantRails bundles the One-click Installer together with Rails, Apache, MySQL, Mongrel, and phpMyAdmin.

If you want to go for the full package, then you can download InstantRails from `http://instantrails.rubyforge.org/wiki/wiki.pl`, unzip the package, and launch the InstantRails executable that will guide you throughout the installation. When you finish, everything will be set up and ready.

If you prefer to install just Ruby, RubyGems, and Rails, you can download the One-click installer from `http://rubyinstaller.rubyforge.org/wiki/wiki.pl?RubyInstaller` and execute it. This will install Ruby and RubyGems in your system. After finishing, we will still have to install Rails on top of that. For installing Rails, go to a command prompt (**Start Menu**, **Run**, and then type 'cmd') and execute:

```
gem install rails --include-dependencies
```

Please notice you will need to have internet connectivity in order for RubyGems to download and install the Rails packages for you. After RubyGems finishes, you are all set and ready to proceed to the next section, *Supported Databases*.

Supported Databases

If you are going to develop Rails applications, you will need both a database and a suitable driver. Depending on which binary package you have installed, MySQL or SQLite3 could have been set up as a part of the process (check the documentation of your packages for details about the included software and libraries).

Out of the box, Rails supports MySQL connectivity through a pure Ruby adaptor. If you are going to work with a database other than MySQL, in most cases you can install the adaptor through RubyGems. Even if you are going to use MySQL, it's advisable that you install the native MySQL one, since it's much faster than the Ruby adaptor bundled with the default Rails package. To do so, you only have to go to the command prompt and execute:

```
gem install mysql
```

The supported databases at the time of writing this book are:

- MySQL
- SQLite
- PostgeSQL
- SQLServer
- Informix
- Oracle
- IBM DB2
- SybaseASA
- Firebird/Interbase

For an up-to-date list of supported adapters and for links and instructions as to how to install them, please refer to `http://wiki.rubyonrails.org/rails/pages/DatabaseDrivers`.

Notice that you need not only the adaptor, but also the database manager itself. It's out of the scope of this book to provide instructions about installing every possible database on every OS, so please check the installation instructions directly at the site of the manufacturer of the database of your choice.

Installing Eclipse

One of the nicer features of the Eclipse IDE is that it's language independent. You can use the same installation of Eclipse for developing in Java, C++, PHP, or Ruby on Rails, for example. The only thing you need is the suitable plugin for your development language.

This means if you already have Eclipse installed for working with a different programming language—Java, most probably–you don't need to install it again for working with Ruby on Rails. The only thing you should be aware of is that RadRails needs version 3.2 or higher of Eclipse, so if you have an older version, you should upgrade. Keep in mind that Eclipse installations are self-contained, so if you want to keep your older version and also install a new one in a separate directory, you can do so without any side effects.

Eclipse installation is pretty straightforward because there isn't really an installer. You only have to download the package for your OS, uncompress it in any directory, and you are ready to go.

You can get the Eclipse package for your platform at `http://www.eclipse.org/downloads`. You can install RadRails on top of any of the listed versions, but if you are going to use Eclipse only for Rails development, then I'd recommend downloading 'Eclipse Classic'. Make sure you choose the right package for your platform (Windows, Linux, or Mac OS X).

You can then uncompress the package in the directory of your choice. This will cause a directory 'eclipse' to be created. Inside it, there'll be a file called `eclipse` (or `eclipse.exe` if you are running Windows). This is the executable to launch. If on Windows, you'll probably want to create a link on your desktop for future use.

As we previously stated, you will need to run Java 1.5 at least if you want to install Aptana RadRails. In some cases, even after you install a newer JVM, eclipse will still use the previous version because the path or the environment is pointing to the old location. If this happens, you can still start Eclipse with the desired JVM by passing an extra argument:

```
eclipse -vm complete_path_to_your_java_executable
```

With that, you are telling eclipse to use the JVM found in the specified directory instead of the system-wide default installation.

One of the concepts you should know about when working with Eclipse is **workspaces**. Eclipse workspaces are a way of grouping together related projects and their configurations. Every time you work with Eclipse, you will be doing so in the context of a workspace. The first time you run Eclipse, it will ask you for a workspace directory. Just choose a new directory to hold the metadata for your projects.
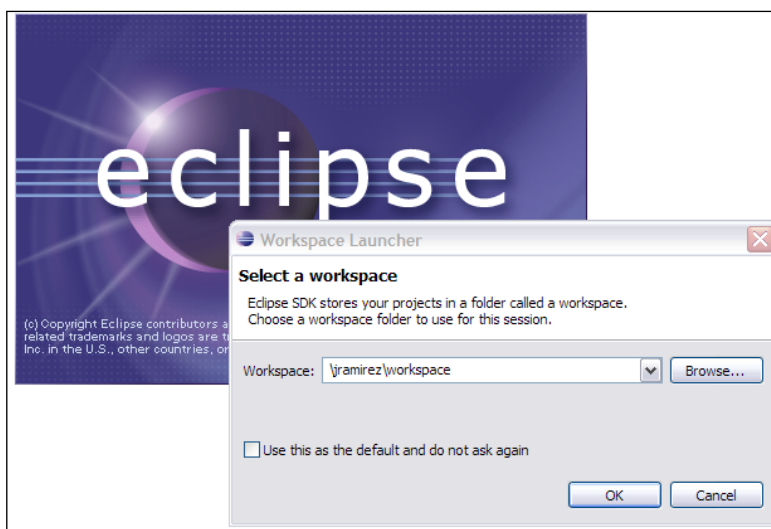
Multiple workspaces can be handy when you are working on many projects at the same time. A typical user will use a single workspace, so you can mark the check box in the **Workspace Launcher** pop-up for Eclipse not to ask about the workspace directory at every startup.

Installing Plugins in Eclipse

Eclipse architecture is designed to be modular and extensible. If you want to add capabilities to your installation of Eclipse, you simply install a plugin. This extensibility is one of the keys for the success of Eclipse and its adoption by a large number of developers and companies.

By making Eclipse pluggable, there is no need for working with different applications for each of your needs, but just for plugging all of them on top of the Eclipse platform. This allows for a more convenient development environment since everything is just a few clicks away, and for easier maintenance. Upgrading is also easier through the Software Update feature of Eclipse, where you can install, uninstall, enable, disable, or upgrade any of the extensions.

For big teams working across different OS, Eclipse simplifies the installation of team-wide tools, since it is multi-platform and so are the Eclipse plugins. This means you can work with the same set of features independently of the target system.

As a result, there are many developers and companies delivering plugins for Eclipse so you can use it for almost any development-related task at any moment of the life-cycle of your project.

When it comes to install plugins on top of Eclipse, you can either use a manual install or the integrated Software Update functionality. The second method is much more convenient, since it's transparent for the user and also provides for automatic upgrades. However, some old or very simple plugins support only manual installation.

Manual installation of a plugin is as easy as the Eclipse installation itself. Basically you only have to download the ZIP file with the code of the plugin and extract it into the 'plugins' directory that you will find listed in your local Eclipse setup. Depending on the plugin, the contents of the ZIP file will be a single JAR or a folder with the name of the plugin and then some files and folders inside. In any case, after restarting your Eclipse the plugin will already be ready for use.

Software Update is even easier to use. You only have to provide the URL for the plugin update manager and Eclipse will guide you through the rest of the process, giving you the option, if available, of choosing optional components, checking plugin dependencies, and allowing for several versions of the same plugin to be installed on your system. We will explain in detail how to use Software Update in the next sections, since Aptana RadRails installation is done through this tool.

In order to work with RadRails, we will need to install two different plugins. First we will install the Aptana for Eclipse plugin, which is a JavaScript/ AJAX-oriented development plugin. It provides support for HTML, CSS, and JavaScript and RadRails uses it for assisting in the editors for the Rails views. Once Aptana is installed, we will finally install Aptana RadRails plugin.

Before we continue, just a quick note for users who already had Eclipse installed. If you have been around RadRails for a while, maybe you have heard about a plugin called RDT (Ruby Development Tools). In older versions of RadRails, that plugin was required in a separate install, so if you installed an old version of RadRails, chances are you have it already.

Since Aptana took charge of RadRails, RDT is bundled together with the RadRails plugin, so you don't need to install it by hand. Moreover, the RDT version bundled with RadRails is not necessarily the most recent version and installing separately could break your RadRails installation, so if you have worked in the past with RDT in your eclipse environment, make sure you uninstall it before getting a new Aptana RadRails.

RadRails Installation

If you still didn't, it's high time to start your Eclipse. The first time you start it you will probably get a welcome screen. If you want to check the general information and help that's OK. Take your time and when you are ready for the installation select the option **Go to Workbench**. If you didn't see any welcome screen but just your typical interface with a menu bar on top, we are fine too. That's what the workbench is.

For some reason, the software update functionality is listed under the **Help** menu in Eclipse. We'll select **Software Updates** and then **Find and Install**.
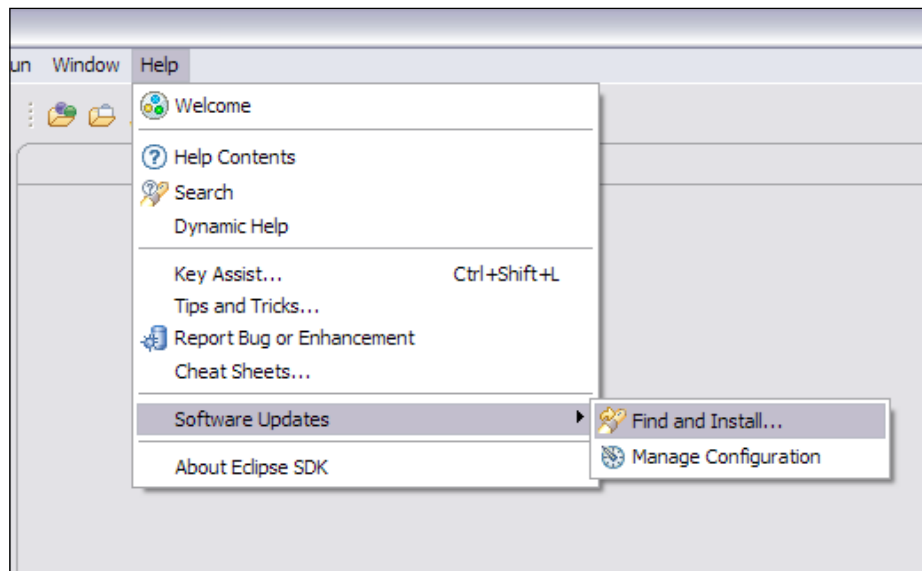
We are installing a new plugin, so we will select **Search for new features to install** in the pop-up that will appear. Next we need to tell Eclipse where to find the plugin we want to install. For that, we will select **New Remote Site** and then in the pop-up we will provide the name and the URL for the plugin. The name is not used internally and it will be just for us to know which plugin we are talking about. My advice is that you use the official name of the plugin, so it will be easier to maintain in the future.

Since we are installing the Aptana plugin, we will type **Aptana** in the name box. Now we need the URL. As a good practice, it's always good to check the official website of the plugin to see if there are any more recent versions. At the moment of writing this book, the stable release can be found at `http://update.aptana.com/install/3.2/`. The pop-up for the remote site should look like this:
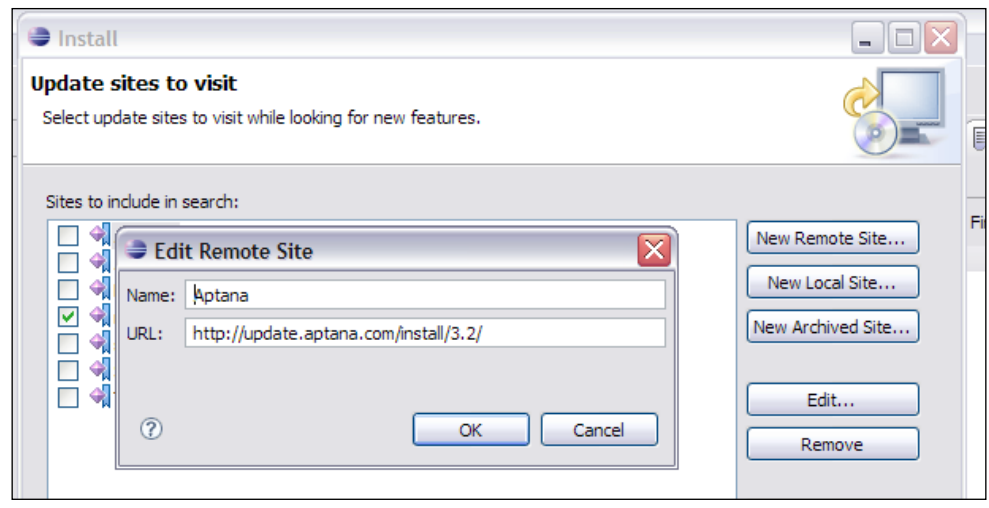
After clicking on **Finish** we will see a new pop-up with all the components included in the plugin. We could select only a subset of them, but we want them all, so we will mark the check box by **Aptana** and hit **Next**.

After accepting the terms in the license agreements, we are presented with yet another dialog. In this case, we are asked about the installation directory for each of the selected components. The default location should be good enough, but if you prefer to install it in a different directory, you can do so by selecting **Change Location**.

At this point, Eclipse has all the information necessary to install our plugin, so it will connect to the defined remote site and start downloading the requested packages. Depending on your Internet connection, this step could take a little while.

After finishing downloading, Eclipse will ask us about installing each of the components. We can just select **Install All** and wait until the completion of the process. When everything is in place, Eclipse will ask about restarting the workbench for the changes to take effect.

Next time Eclipse starts you will see the **Aptana Start Page**. In that screen you have basically some quick links to Aptana-related functionalities. For installing Aptana RadRails, we would have to repeat the steps for the previous plugin, but providing a new name and URL. **Aptana Start Page** will make this process even easier. The central pane of this page has the title **Plugins** and from here you can install or upgrade Aptana plugins. Scrolling down the page you will find the name **Aptana RadRails** and a button to **Install**
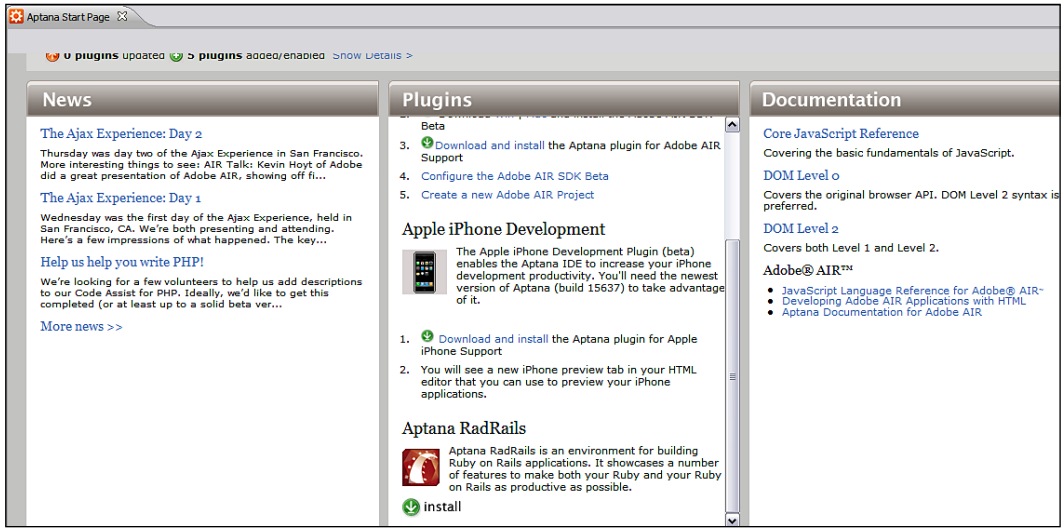
This will take you to the already familiar pop-up for selecting remote sites, but the site information will be already filled in. (Neat, uh?) After accepting the license, telling Eclipse where to install, and waiting for the download and installation of the plugin (exactly as in the previous step) your Eclipse will be ready for Ruby on Rails development.

In the next chapter we will learn how to make the minimum configuration so we can start experiencing the power of Aptana RadRails.

Summary

By now you should already be familiar with the concept of what an IDE is and you will hopefully be expecting to start using it for your Rails development. Using an IDE will help you focus on the important tasks, assisting you on those areas where some automation is possible. Also, you will be able to access several development tools like databases or source control systems from a single interface.

If you already had Ruby and Rails installed in your system, you only had to install a JVM, Eclipse, and Aptana and Aptana RadRails plugins. For those with an existing Eclipse installation, it was just a matter of installing the plugins. In any case, your system should be now up and ready for some configuration and a tour of the Eclipse/RadRails workbench.

2
First Steps

Let's face it! We cannot help ourselves when a new gadget falls into our hands. We want to try every option and take a look at all the features and all at the same time, if possible. Well, I'm sorry, but I'm afraid I will have to abuse of your patience for a little bit longer.

Before we can start programming in Rails with Aptana RadRails, we will have to configure our installation and learn the general concepts of the IDE. However, during this chapter we will also see how to create Rails projects and, after we finish, we'll know everything we need to start coding.

Basic Configuration

In most cases, Aptana RadRails is smart enough so you don't need to change any of the default values unless you want to customize any aspects of the interface. Unfortunately, there are some cases in which RadRails cannot figure out the internals of your installation and some manual configuration will be needed in order for everything to work properly. If that's the case, we will need to tweak a bit of our configuration before developing with RadRails for the first time.

But even if your installation was in the first group and everything was self-configured properly, it's always interesting to know what is the basic configuration RadRails depends on. We'll dedicate this section to explain which are these basic preferences. Later in this book you will find a complete reference for all the Rails-related configuration options.

Eclipse Preferences Dialog

As we already know, Eclipse is the underlying platform on top of which Aptana RadRails is built. As a nice side-effect we get a consistent way for modifying the preferences of each of the plugins running over the platform.

The downside is that the Eclipse configuration or preferences dialog can look intimidating at first sight, specially if you have a large number of plugins installed, because you will find a lot of different sections and options that can be a bit difficult to navigate at first. Fortunately for us, since some versions ago, the Eclipse preferences screen has a built-in search feature that will help us a bit in the process.

As we said before, a typical RadRails installation will work directly after setup, without any further configuration. We are detailing the different configuration steps only for those users who need to change the defaults and as a general reference for everyone else.

Enough talking, let's go and see what the preferences dialog looks like and a general guideline for using it before starting our RadRails configuration. To open this dialog we have to navigate to the Window menu or the Eclipse menu in some versions running on Mac OS X and then select **Preferences**.
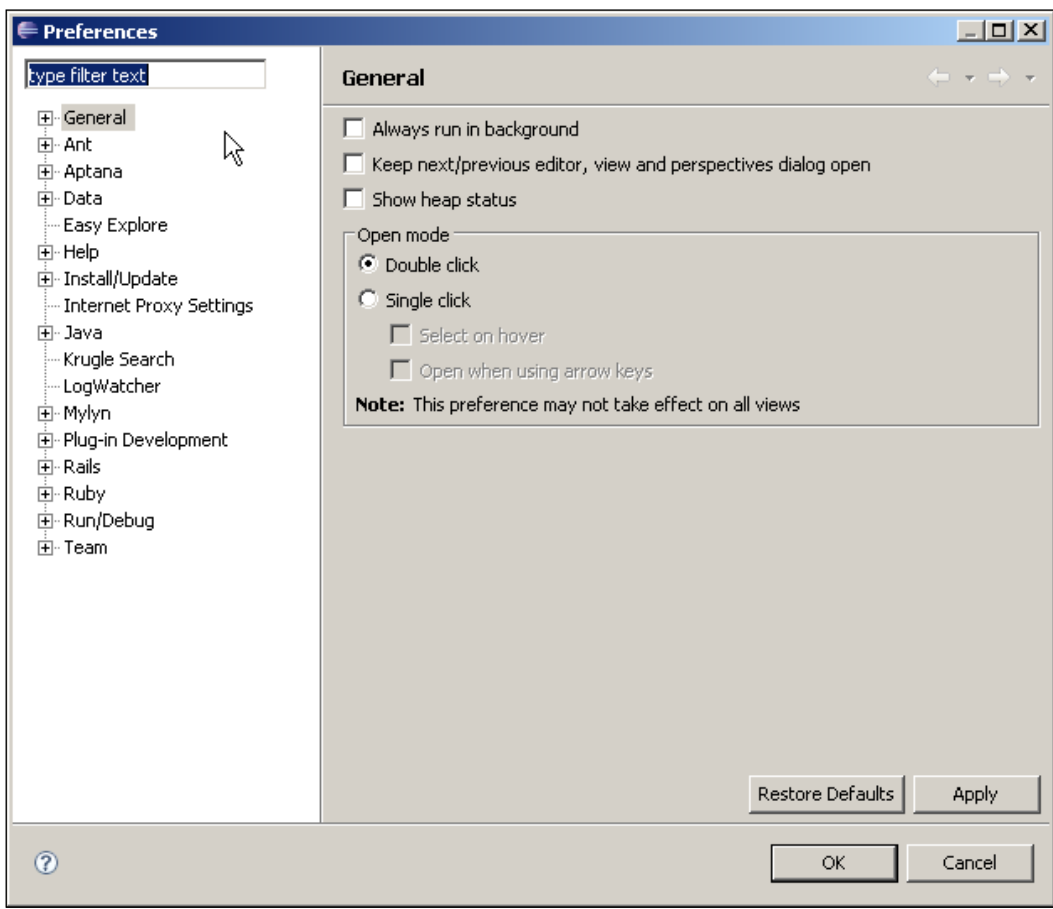
As you can see, the dialog has two parts; to the left you have the navigation tree, and to the right you have the configuration values. The number of entries on the left-side tree will depend on which plugins you have installed. With a minimum installation you should be able to see at least these entries: **General**, **Aptana**, **Data**, **Help**, **Install/Update**, **Internet Proxy Settings**, **Rails**, **Ruby**, and **Run/Debug**.

If you click on the **plus** sign to the left of **General** you will see how the section unfolds presenting yet another sections and subsections tree. Most of these adjustments are just for fine-tuning and you will never need to modify them. Anyway, you can already start to see there can be a lot of different options and unless you are very familiar with the interface it is not always clear in which particular section to find them. Here is where the integrated search comes handy.

Suppose you want to change the encoding in which your files are saved because you need everything to be encoded in, for example, UTF-8. Now you would have two options; either you know where exactly to find this preference or you can just type **encoding** in the search box.
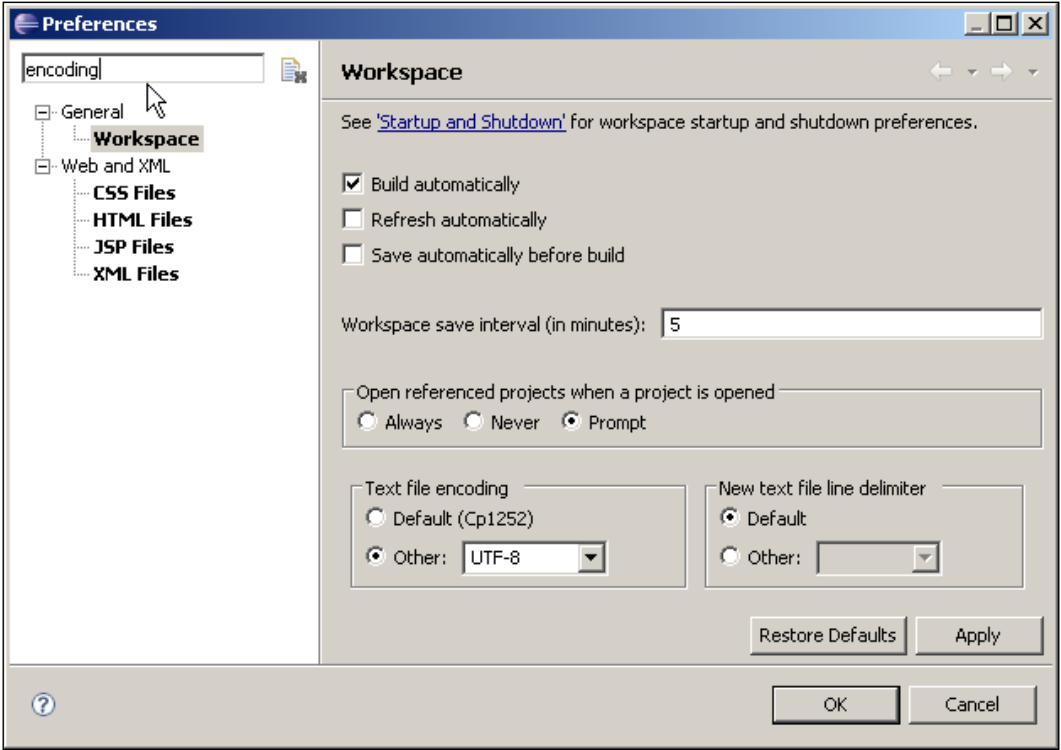
Now the tree to the left will display only the sections in which there are some preferences matching that string, so we can find the one we want easily. If you want to display back the whole set of options, you only have to clear the search box. You can do that with a single click on the icon to the right of the box.

Now that you know how to find your way through the preferences dialog, it's time for taking a look at our RadRails settings.

File Encoding

This is not exactly a RadRails preference, but a general Eclipse one. However it can have a great impact on your development, specially if you are using non-ASCII characters. If you are going to share your code with other developers or deploy to different machines, you have to make sure that all of the instances are using the same encoding for your files.

In Eclipse, you can define the encoding on a per-workspace, per-project, per-folder, or per-file basis. By default, the encoding of the parent object will be used unless you explicitly change it.

Since the most common scenario is to use the same encoding for all your files in all the folders of all your projects, you probably want to set the encoding of your whole workspace and then you don't need to worry about defining the encoding for each of the files, since all of them will inherit from there.

You can find the encoding option by unfolding the **General** section and then selecting **Workspace**. By default, the system encoding will be used, and you can change it by selecting the **Other** radio button and then choosing the option from the drop-down box.
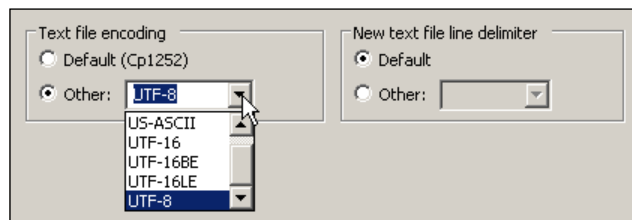

Connecting through a Proxy

As you know, when working with Rails there are many times when you need to connect to external servers, for example for installing or updating gems and plugins, or to synchronize your repository.

If you have direct connectivity then **you don't need to change anything**, but if you are behind a proxy you will have to tell Eclipse about that.

I'm sure as a good Rubyist you are used to not repeating yourself, but unfortunately in this case we will have to set that configuration in two different places. Eclipse has a general preferences section for network connections, but Aptana doesn't use those options and provides a new section instead. It's a good practice to change both preferences sets at the same time, so all your plugins will be consistently configured.

Eclipse's connectivity options can be found under **General | Network Connections** on the left pane of the preferences dialog. Aptana's proxy can be found directly at the root level of the preferences tree under the name **Internet Proxy Settings**. The reason for this dual configuration is that older versions of Eclipse didn't provide connectivity options, so the plugin had to provide its own. Plans are that in the future Aptana will use the connectivity options configured at a global level.

Both sections, the one of Eclipse and the one of Aptana, are alike and they have the same set of options to configure.
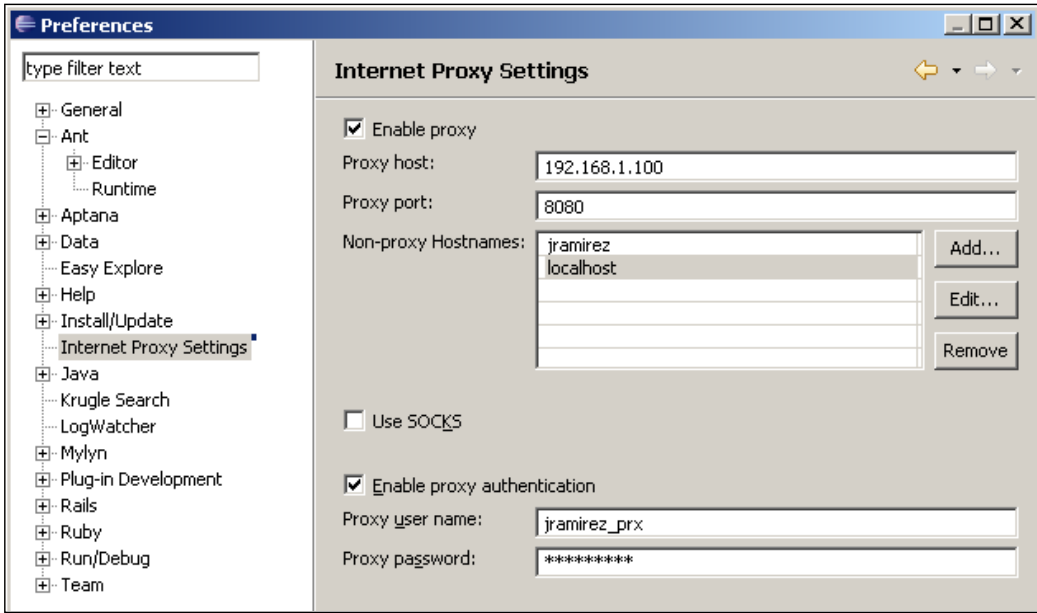
First you will have to check the option to tell Eclipse that you will be using a proxy, then just specify the IP and port of the proxy machine. If your proxy needs authentication, you will have to provide the user and password for it.

By default, all the network connections will be proxied except for the ones to your local machine. If you want to connect to other servers on your side of the proxy, there is no need to proxy these connections too. In that case, you will have to add the names or IPs of these servers to the **Non-proxy Hostnames** list.

Ruby Environment

For the basic configuration of your Ruby environment, you only need to tell Eclipse which is the Ruby interpreter it should be using. In older versions of RadRails, you had to make sure the interpreter was found, but in newer versions the plugin will directly try to find it in your system so you don't have to configure anything. If when installing RadRails you don't get any warnings, then it means it was able to find the Ruby interpreter and self-configure properly.

If RadRails cannot find your Ruby interpreter, it will prompt you to enter its location, or it will take you to the download page if you don't have any.

Only if your interpreter is not found in the system path, or if you have several interpreters installed, or if you want to launch your Ruby interpreter with any special command line options, will you have to configure these preferences.

Open the **Preferences** dialog and navigate to **Ruby | Installled Interpreters**.
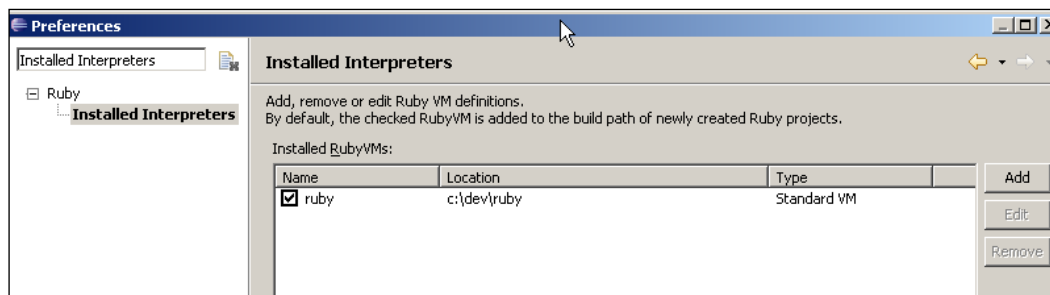
If RadRails was able to find your Ruby interpreter it should display at least one entry like the one in the above screenshot. If there are no entries in this list, or if you have several versions on the same box and the Ruby interpreter of your choice is not listed, you will need to add a new one by clicking on **Add**.
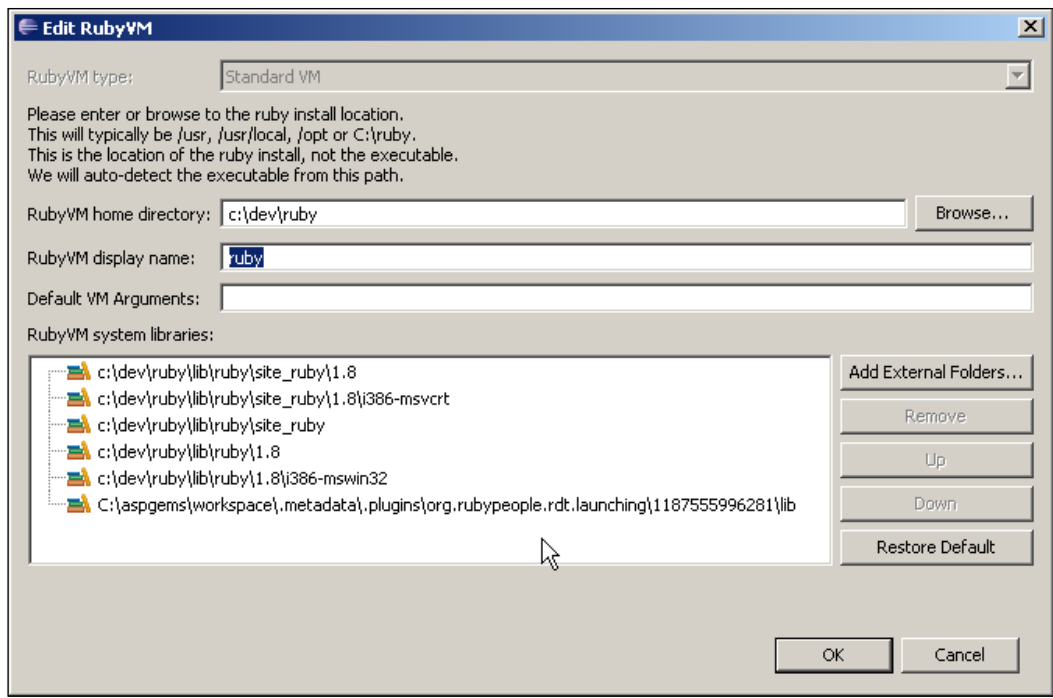
The display name is just for informational purposes. I would recommend to include the version of your Ruby VM as a part of the display name so it's easy to see at a glance which interpreter you are using.

The most important parameter in this dialog is the Ruby VM directory. This is the top-level directory of your Ruby installation. By top-level directory, I mean the one that holds `bin`, `doc`, and `lib` directories inside. Once you select your VM directory, RadRails will automatically display the VM System libraries to include.

Rails Environment

The basic configuration of your Rails environment is just the paths to your Rails and Mongrel scripts. As with the Rails options earlier, RadRails will try to find the proper scripts in your system, giving you the option to enter their paths, or install them automatically, if they cannot be found. The typical user will not need to change this configuration after the initial setup.

If the Rails scripts were not properly configured, it would be impossible to create new Rails projects, execute Rake tasks, or start/stop your Mongrel servers.

These preferences can be manually changed at any time from the **Rails | Configuration** section of the preferences dialog.
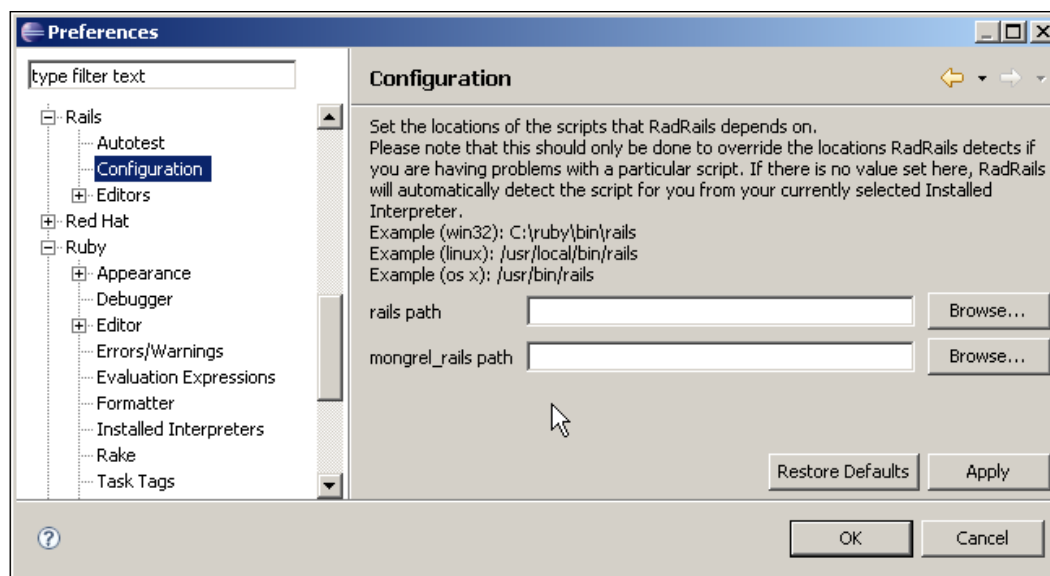
The RadRails team recommends not to change these values manually if the plugin was able to automatically find the Rails installation, because you could end up with your Eclipse pointing to the Rails installation in one directory and the Ruby interpreter in another one, which would lead to unpredictable behavior. Unless you are really sure you want to modify these settings, its advisable to leave them untouched.

After setting your VM and the paths to the different scripts properly, your RadRails will already be operative and you will be able to start creating Rails projects.

Creating a Rails Project

After RadRails installation and configuration, we are ready to start working on a Rails project. There is just one more and obvious step: we need to create our project. To start the project creation process you should open the **File** menu and select **New**. Depending on your particular setup, the first option of your menu may be **Rails Project**. If that's the case, then you can select that option directly and the Rails project wizard will be displayed. If not, you can select the option **Project...** as can be seen in the following screenshot:
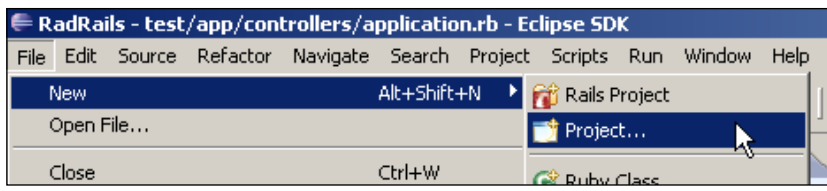
If you had to select **Project...** then you will see a new dialog window with an option **Rails** and then **Rails Project**. This will open the Rails project wizard for you.
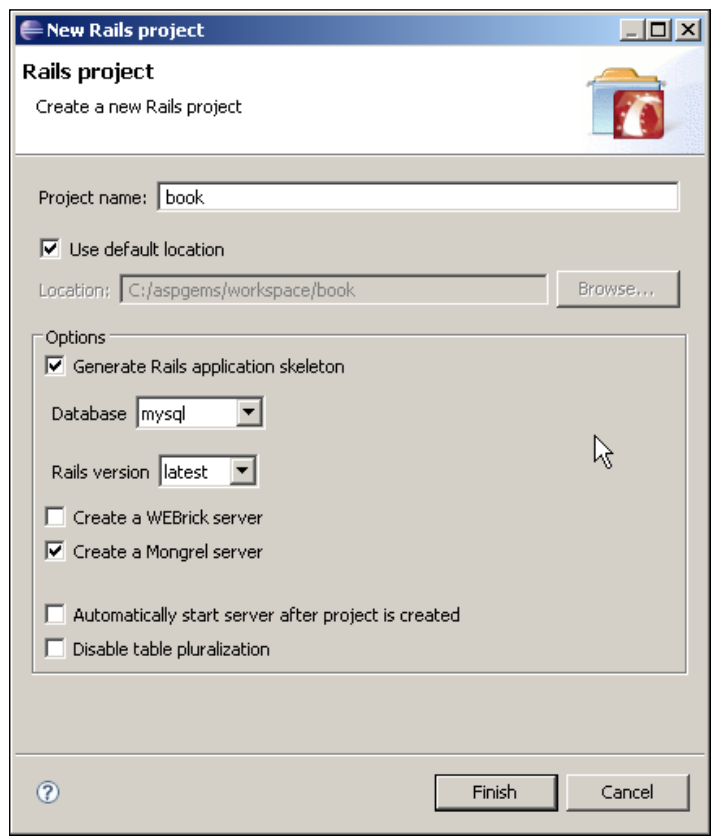
Creating your project is as easy as it should be. The first thing you have to provide is the name for your project. By default, your project will be created inside the directory of your workspace. My personal preference is to create my projects out of the workspace folder. I like to leave the workspace directory exclusively for Eclipse's internal use, and keep my source code elsewhere. Anyway, that's not for any good technical reason, but just because in older versions of Eclipse that was the standard way to proceed.

If you want to use a different directory for your project, you can do so by unchecking the **Use default location** box and then browsing to your chosen location or just typing it into the location box. You must be careful when choosing the location since it's a bit tricky. The location directory you are choosing is the actual directory for your project, and not its parent. Notice that your directory and your project name don't need to match. You could have for example a directory like`/my_projects/first_project/trunk` and use `first_project` as your project name.

By default, RadRails will create a Rails project structure (application skeleton) for you. When creating a new project you should always keep this option on. If you choose to create the skeleton, you have to select which database server type you are going to use and which version of Rails, which defaults to 'latest'. You will be able to select any of the versions installed in your system.

RadRails will also by default add a Rails server, so you can start/stop/debug directly from the Eclipse IDE. If the Mongrel gem is available in your system, RadRails will mark the Mongrel server as a default. If you prefer to use a WEBrick server you can select the corresponding checkbox. If you select the Mongrel option, but you don't have it installed, RadRails will offer the option of automatically installing it.

You can have both types of servers or neither of them, as you prefer. In any case, you can always add/remove servers at any time after your project is created.

Note there is also an option to start the server automatically after creating the project. Since you will usually have to configure the user/password for your database server —unless you are using sqlite, or mysql with a root user and no password— and you will need to restart the server for these changes to take effect, I prefer to uncheck this option and start the server manually when I finish with the configuration.

The last option in this dialog, **Disable table pluralization**, will modify the `environment.rb` file so pluralization is not used in Rails. The most common scenario is to use table pluralization and you should only disable it when working with legacy database schemas.

> By not using pluralization you are ignoring Rails conventions and you shouldn't do it without a good reason.

After clicking on **Finish**, RadRails will call the Rails script we configured before to create your project. Depending on your computer, it could take some seconds until you start seeing RadRails is actually doing something. You will be able to see in the lower part of the screen the console output of this command. Most of the generated files will appear as links at the console, so if you click on them they will open in an Editor.

The result of the creation of the project will be exactly the same as if you were running the Rails script from the command line, with one small difference. In the root of your rails project you will find two files named `.project` and `.loadpath`, which are used internally by the IDE. Those files are not really a part of your Rails project but just for Eclipse to know about its location and configuration.

If you are sharing your project with other developers or deploying to a different machine, you don't need to copy these two files. However, sharing these files could be handy if the whole development team uses RadRails, and you want a quick way for sharing the configuration throughout the different machines. Since I usually work in heterogeneous teams, I tend to ignore these files, but that's something for you to evaluate in your personal case.

# Importing an Existing Project into RadRails

If you've been learning Rails, most probably you will already have some projects created, and it would be nice to have them available in RadRails. All we have to do is import the projects into our workspace. Even if at first sight the natural option would be **Import** under the **File** menu, that's not the procedure to follow. The **Import** option is used for importing projects created in different workspaces or for importing a directory tree into an existing project, but not for importing an existing project directly into a new one.

To import any Rails project into your RadRails workspace, all you have to do is follow the same steps as for a new project, but in this case, you don't want to generate the Rails skeleton since this would overwrite your project contents. Just make sure you uncheck the **Generate Rails Skeleton** option. You will also have to tell RadRails not to use the default location and navigate to your project's directory instead. The first time you are importing projects I'd strongly advise to have a backup copy in order to prevent any loss.

If you want to import your project from a repository using a Software Configuration Management tool, such as Subversion or CVS, then this is not the right way to do it. In that case we will be using the 'Checkout' feature. We'll talk about how to integrate with a Software Configuration Management tool in Chapter 9.

Working with Perspectives and Views

Now we have already configured our environment and created a project, it's time to get familiar with the Eclipse workbench. When you are working with Eclipse you typically have several different areas on your workbench. The most common layout in Eclipse consists of the top Menu or Toolbars Area, the central Editor Area, and several Views distributed in side and bottom panes.
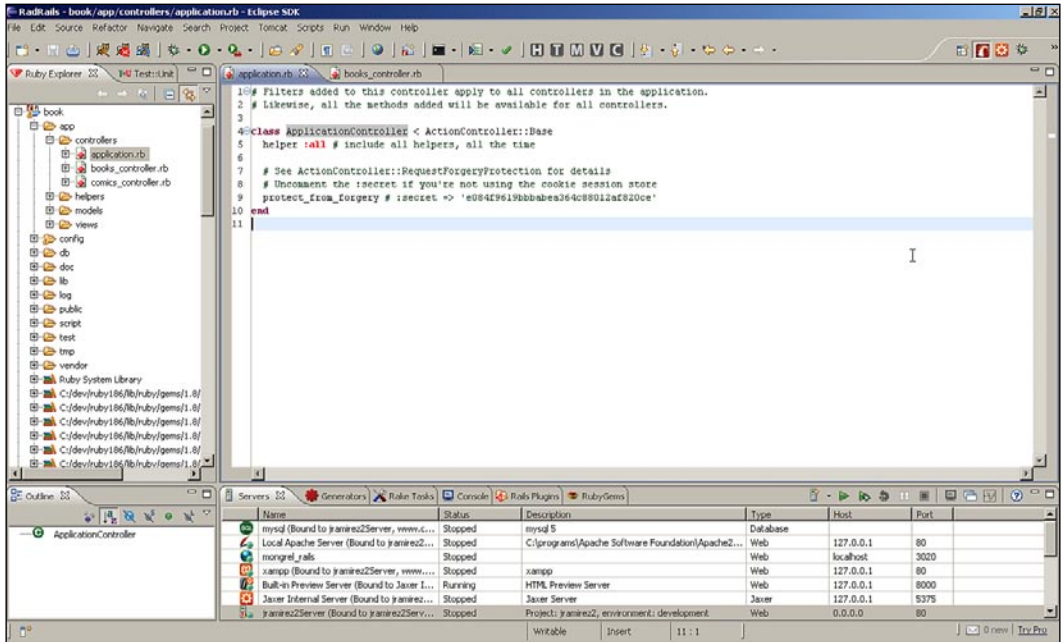
Eclipse Perspectives

During the development of a project, you are not always carrying out the same type of tasks. Sometimes you are programming, sometimes you are interacting with the database, sometimes you are synchronizing your repository, sometimes you are debugging, and so on.

Since all of these different tasks require different sets of tools, it really doesn't make much sense to try to display all the available toolbars or views all the time, but only the ones you need at each moment. Eclipse uses the term **Perspective** to refer to a group of toolbars, views, and features. By having different perspectives and being able to switch between them easily, the IDE can offer a lot of different tools while reducing the complexity.

When you install Eclipse there are some perspectives installed by default, like the Java perspective for example. It is very common for complex Eclipse plugins to install perspectives of their own. When you install RadRails, the following perspectives are available: **RadRails**, **Ruby**, **Ruby Browsing**, **Debug**, **Aptana**, and **Data**. Each of these perspectives is specialized in a different subset of your development needs. You can also customize these perspectives or create your own ones if necessary.

You can see the list of perspectives available and change between them in two different ways. You can find the **Open Perspective** icon on the top-right part of your screen.
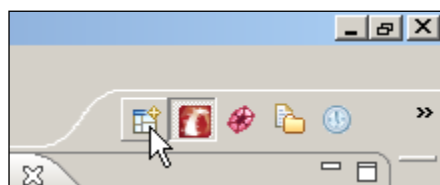
By clicking on that icon you will get a drop-down list of the recently used perspectives. If you want to see all, just select **Other...** You can also see different icons for the related perspectives in the same tab. If you prefer to see the title of the perspectives, you can right-click on top of them and mark **Show Text**. You can find the same **Open perspective** functionality directly under the **Window** menu.

There are also ways for moving with keyboard shortcuts (Eclipse calls them **Key Bindings**) between the different open perspectives, but for now it will be easier if we don't try to be exhaustive about all the possibilities.

> Eclipse IDE is very rich in options. Many of the features can be accessed in different ways (context menus, toolbar icons, shortcuts, and so on). Don't worry if you don't know them all at first. I've been working with Eclipse for over four years and I still get surprised once in a while by discovering some things I didn't know (not to mention that the IDE is very alive and they keep adding new features on a regular basis).

Eclipse Views

As we said earlier, the Eclipse IDE is composed of a main Editor Area, which we'll explain further in the book, and several views distributed among different panes. So, what's a view?

Views are a very important part of the Eclipse IDE and of the RadRails plugin and they are one of the things that make the difference between Eclipse and a programmers' editor. Views can be classified roughly in any of these three groups: Resource Navigation, Information Display, and Support Tools. Views can interact with the Editor Area when appropriate.

Just to make it easier to understand the concept, this is a non-exhaustive list of some of the available views:

- **Ruby Explorer**: This displays a tree of your Rails projects and files, providing information about the structure of your code (classes, variables).
- **Outline**: This displays the structure/hierarchy of the objects in the current editor (modules, classes, methods).
- **Output console**: This displays general output of the activities going on.
- **Search Results**: This shows matches of your ongoing searches.
- **Rake Tasks**: This displays all the available tasks, allowing you to launch them.
- **Rails Plugins**: This lets you install, remove, or update plugins.
- **Ruby Gems**: This lets you install, remove, or update gems.
- **Rails Generators**: This lets you run the Rails code generators.
- **RI**: This displays Ruby documentation.
- **Debug**: This lets you control the execution of an executing Rails application.
- **Variables**: This shows all the variables with values in scope when debugging.

There are more views, but as you can, see each of the above provides interesting capabilities for software development. We will see later in this book the details about all the views important for developing with Rails, but now let's see the general concepts you should know when working with views in Aptana RadRails.

The first thing you should know is how to open a view. This option is found in the **Window** menu, by choosing **Show View**. There are several views to choose from directly. These views are related to the perspective you are using. However, sometimes you want to display a different view, and you can do so by selecting **Other...**. A dialog displaying all the available views, sorted by category, will be shown.
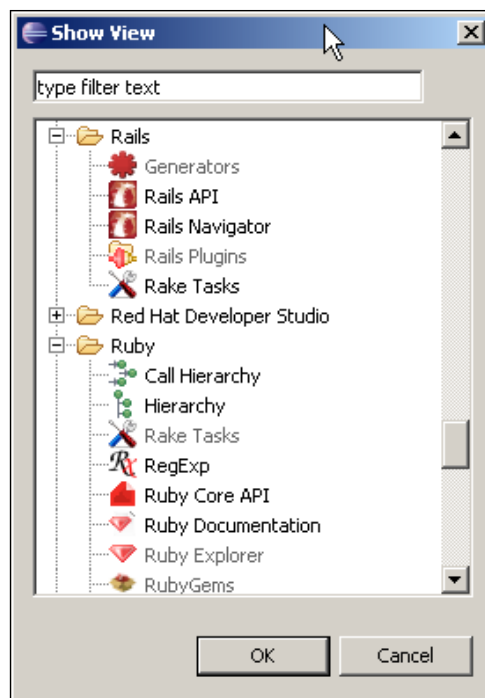
If you select any of the views, it will display. Notice the views already displayed appear faded and you cannot select them. When a view opens, it will dock itself by the pane in which it was opened last time. If it was not used before, it will stack on top or by the side of some of the opened panes. Fortunately, we can arrange the views as we want and group them to have a more comfortable interface.

To move a view to a different position, just click on the title and without releasing the mouse button drag it to the desired location. When moving the view around, you will see a rectangle indicating where it would be docked if you release the button. You will notice some icons appear as you move the view around the screen. When you see an arrow, it means the view will position itself in a new pane occupying the place indicated by the tip of the arrow. If a folder icon is shown, it means the view will be positioned as a new tab in the pane below the cursor. If you want to rearrange the tabs for the different views in a pane, you can just drag-and-drop to move them.

To get comfortable with the environment, just move around some of the views. At first it takes a bit of practice with the mouse, specially when you want to create panes on the corners of the screen, but after a while it feels natural arranging the views like that.

> If you mess up the interface, you can always reset the perspective to the default presentation by choosing **Reset Perspective** under the **Window** menu.

You can maximize or restore any view by double-clicking on the view's title tab. At the top-right corner of every view, there are always a pair of icons representing a minimized and maximized window respectively.
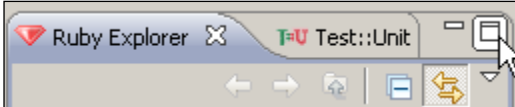
When you maximize a view, the top-right icon will change to represent overlapping windows. Clicking on it will restore the view to the original size and location.

Minimizing is as easy as clicking on the small icon, but there is a catch to this. Eclipse will try to leave as much free screen space available as possible when minimizing views, and at the same time it will give you a way to quickly access your minimized views. In order to do so, when you minimize a view, all the views sharing the same pane get minimized and they will display as icons on a new toolbar by the border of the workbench.
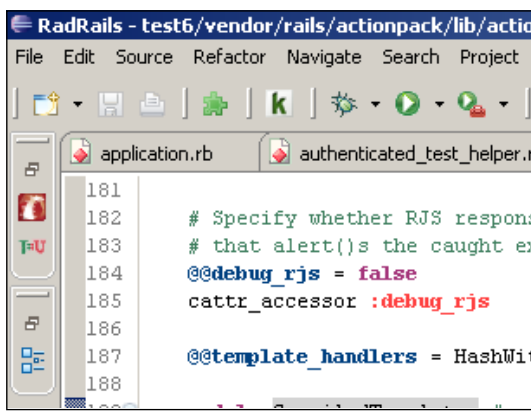
As you can see in the figure above, I have minimized two different view panes and now they display as two sets of icons by the side of the window. You can restore the view group to the original location by clicking on the overlapping windows icon of the corresponding toolbar.

You can also click directly on the icon representing the view you want to display, but in that case it will be opened as a **Fast View**. A fast view is a view like any other, except that it gets automatically minimized when you click on any other window. If you want to minimize a view as a fast view, but you don't want to minimize the whole pane, you can right-click on top of the view title and select **Fast View**.

The minimized view icons will position themselves at the left, right, or bottom of the workbench, depending on where they were originally located. When you minimize a view, there is a small animation in Eclipse to help you see in which border of the window the toolbar will be docked.

The last thing we need to know for now is yet another trick about positioning. Sometimes it's handy to have a non-docked window that can float over any of the workbench areas. You only have to right-click over the view title and select **Detached**. When you want to go back to the usual docking, just repeat the operation over the title of the detached view.

Summary

Before starting working with Aptana RadRails, we had to make sure the default configuration was appropriate and change it where necessary. We reviewed the configuration for our Rails scripts, our Ruby interpreter, the file encoding, and the connectivity from behind a proxy. During the process, we also took a look at what the preferences dialog looks like and how we can search options quickly.

After configuring Eclipse, we created a new Rails project, and we also learned how to import existing projects into our workspace. But before starting to work on those projects, we needed to understand the general concepts behind this IDE.

In Eclipse and RadRails, all the work is basically done in the Editor Area or in different support views and they are grouped together in **Perspectives**. Now that we know how to switch perspectives, and how to open and arrange views, there is nothing to stop us from developing our first application.

3
Your First Application

Here we are! programming in a powerful language specially designed for the Web and using an IDE that promises to help us with many of the mechanical tasks involved in the coding.

If you have been already programming with Rails, you probably know that if we take advantage of scaffolding we can have a simple web application for table maintenance in a matter of minutes (yes/no typo here, it really takes just a few minutes). And we are even talking about the database table creation process.

If we wanted to add validations, a nice design, and some more complexity we would be talking about a few hours. Still pretty impressive, depending from which programming language (or framework) you are coming.

The truth is, creating the wireframe of your application in Rails is quick and easy enough even from the command line, but we'll be learning in this chapter how to do it a bit more comfortably by using RadRails for creating your models, controllers, database migrations, and for starting your server and test your application.

Basic Views

Most of the time when working with our IDE we will be using the Editor Area, of which we'll be talking in the next chapters of this book. Apart from that, two of the views we will be working with more frequently are the Ruby Explorer—the enhanced equivalent of the Rails Navigator, if you were using RadRails Classic—and the Console.

Both of these views are fairly easy to use, but since they will be present at almost every point of the development process, it's interesting to get familiar with them from the beginning.

The Ruby Explorer View

If you have already opened the Rails perspective, then you should be seeing the Ruby Explorer at the left of your workbench. If you still haven't opened the Rails perspective, you should do it now by using any of the techniques we discussed in Chapter 2.
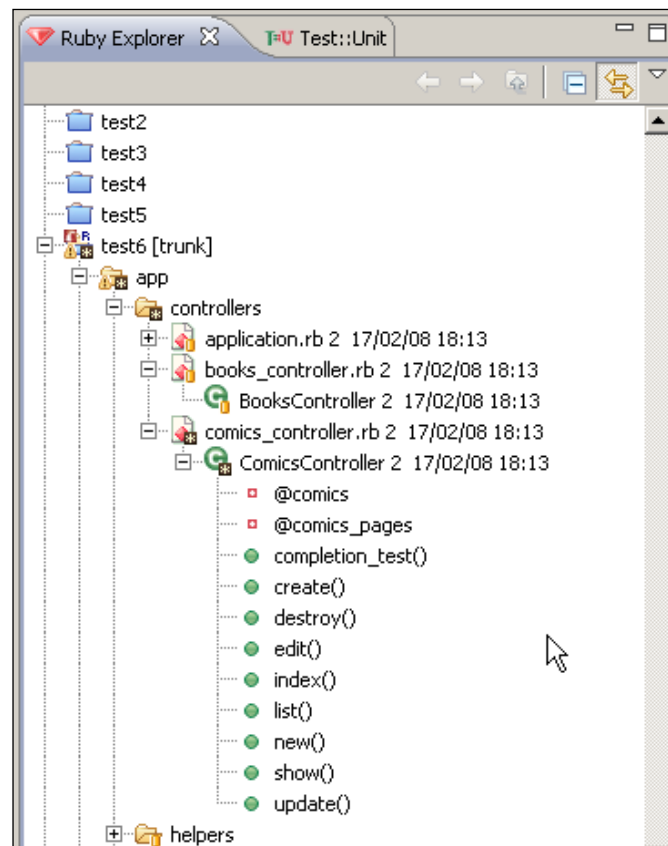
This view looks like a file-tree pane. At the root level, you will find a folder for each of the projects in your workspace. By clicking on the icon to the left of the project name, you will unfold its files and folders. The Ruby files can be expanded too, displaying the modules, classes, variables, and methods defined in the selected file. By clicking on any of these elements you will be taken directly to the line in which it is defined.

Before navigating through the contents of a project, we have to open it. Just right-click on its name and choose **Open Project**.

When opening a project, Eclipse will ask you if you want to open the referenced projects. By default, your projects don't have any references and that's the most common scenario when working with a Rails application.
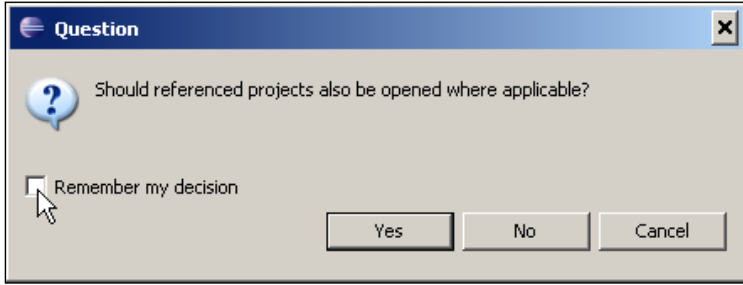
If you want, you can include references to other projects on the workspace so you can open and close them together. To view and change the project references, you can right-click on the project name, then select **Properties**. Notice you can also get here from the **Project** menu by selecting **Properties**. In the properties dialog, you have to select **Project References**. Here you will see a list of all the available projects in the workspace. Just check or uncheck all the projects you want to reference.
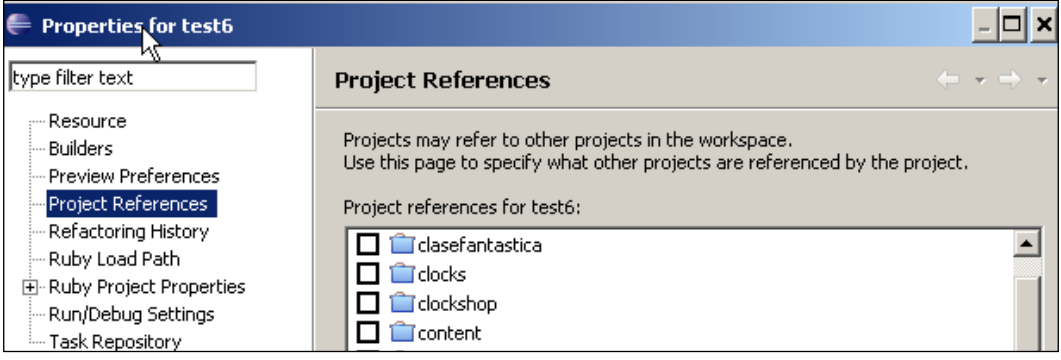
Once your project is open, the mechanism for navigating the contents is pretty straightforward. You can open or close any sub-folders and you can right-click on any item to get a context menu. From this menu you can perform common file operations like creating, renaming, or deleting a file.

We will see more details about creating new files when talking about the Editor Area. There is also a **Properties** option from where you can change the encoding for a particular file, or the file attributes (read only, for example). The **Properties** option is also available at the project level.
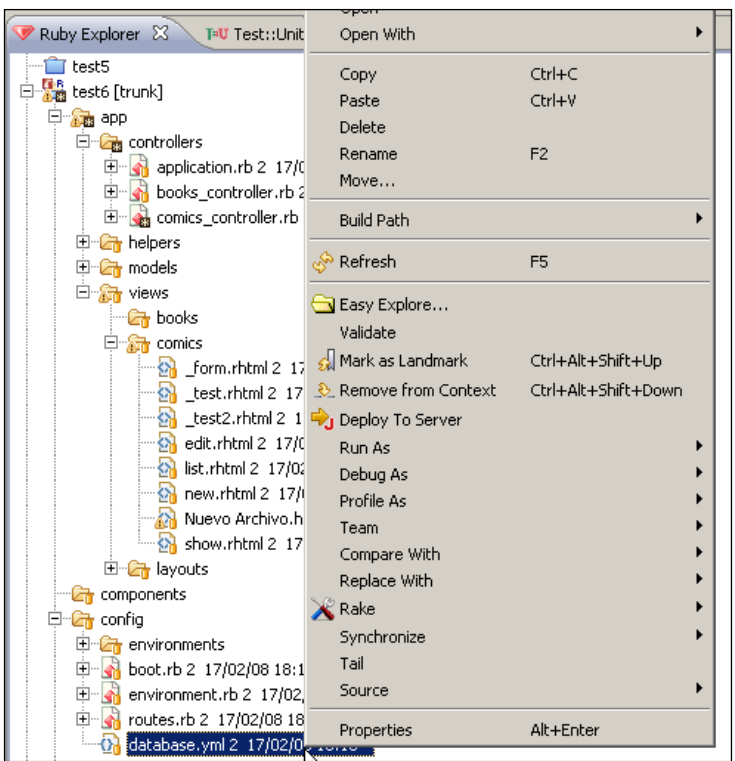
Also in the context menu, you can see there is a **Tail** option. This will work like the tail command in UNIX, displaying the contents of a file as it's changing. This option is especially useful for a quick monitoring of the log files of your application.

You can also find in the context menu two options with the names **Compare With** and **Replace With**. If you select either of them, you will see a new menu in which there is an option named **Local history**. This functionality is really interesting. You can compare your current version against an older version of the same file, or you could replace the contents with a previous one. This can be a life-saver because when using it on a folder the local history will contain copies even of deleted files.

Comparing a file against another copy is a powerful tool, which can also be used when working with repositories or to compare different files between them. Let's try it and see how it works.

Open any of the files in your project tree by double-clicking on the file name. Now go to the Editor Area and add some lines with **Mumbo-Jumbo** text. After you are done, click on the save icon of the toolbar or select **Save** in the **File** menu. Now let's go back to the Ruby Explorer, double-click on the file name and select **Compare With | Local History**.

You will see there are some entries here, one for each time we saved the file. If this was the first time you worked with the file, then there will be only two versions, the original and the one you just saved. Double-click on the oldest local version you have.
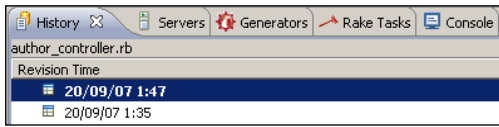
Now a new editor will be opened. The editor is divided into three panes, the top one displaying structural differences, the bottom-left one with the code of the current version, and the bottom-right one with the old version of the code.

At the top pane, you will see the structural differences between the versions being compared. For every added or deleted method or variable—at instance or class level, you will see the name of the element with an icon displaying a **plus** or a **minus** sign. If a method exists in both versions, but its content was changed, the name will be displayed without any additional icons.
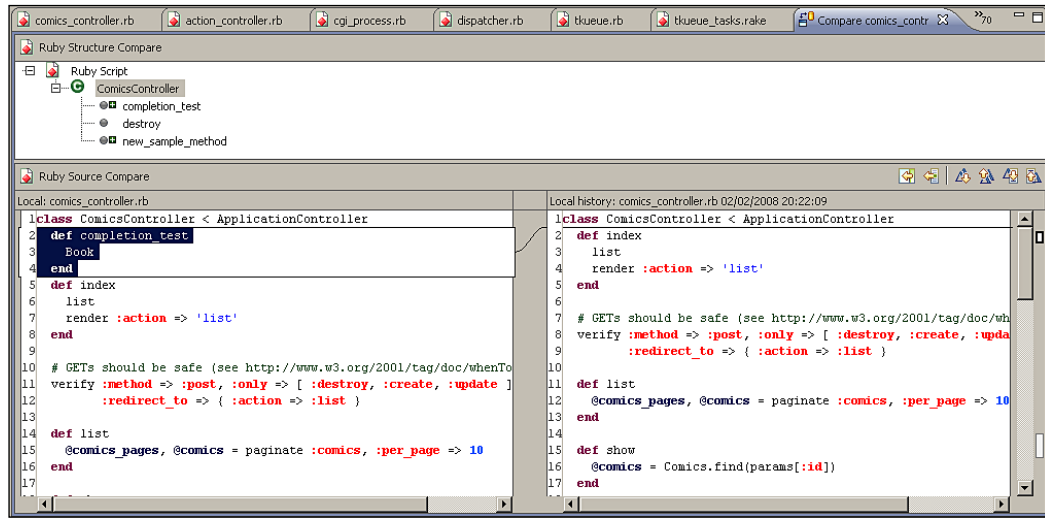
When reviewing the differences/changes you will see the editors at both sides are linked with a line representing the parts that are not equal between the files. When you are on a given change/difference you can select the icon for 'copying current change from right to left' (or the other way round, depending in which of the files the change is), which will override the contents of the left editor with those of the right. You can also just manually edit or copy/paste in your editor as usual.

There is an interesting icon labeled 'Copy all non-conflicting changes from right to left' that will do exactly as it promises. Any changes that can be automatically merged, will be incorporated to your editor. Depending on the differences between the files, the icon could be the contrary 'Copy all non-conflicting changes from left to right'.

When you finish comparing or modifying your current editor, remember to save the contents of the editor in order to keep your changes.

If you just wanted to review the changes without any modifications, you can directly scroll down the editors, use the 'Previous' or 'Next' icons, or use the quick marks by the right margin.
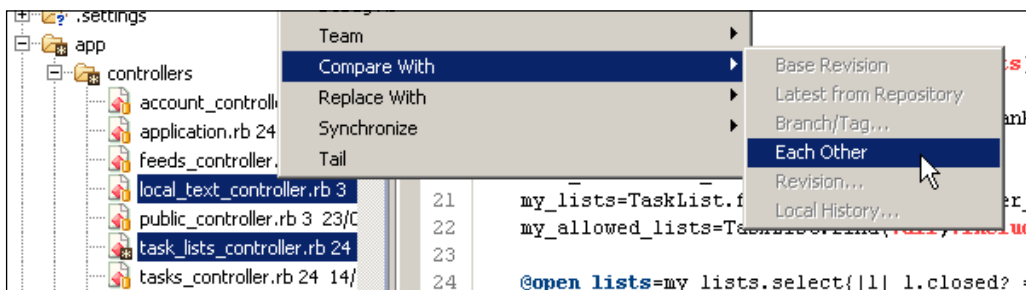
You can also compare two files instead of comparing a file against an older version. Go to the Ruby Explorer and select one of the files, then hold down the control key and select another one. With both files selected, you right-click and select **Compare With** and then **Each Other**. Once opened, the compare editor works exactly the same as when comparing with an old version of the same file.

Ruby Explorer Top Icons

Before we finish with the Ruby Explorer, there are still a few functionalities we want to take a look at.

By default, the Ruby Explorer and the Editor Area are not linked, meaning that when I open a file from the Explorer the editor displays that file but not the other way around. If you select an open editor, it will not be selected for you in the Ruby Explorer. Sometimes, it's handy that both views are not linked, but most times I find it more convenient to have the current file automatically selected in the Ruby Explorer. Just select the icon with the two arrows on the top right of the Explorer view. By clicking on this icon you will toggle the linked views function.
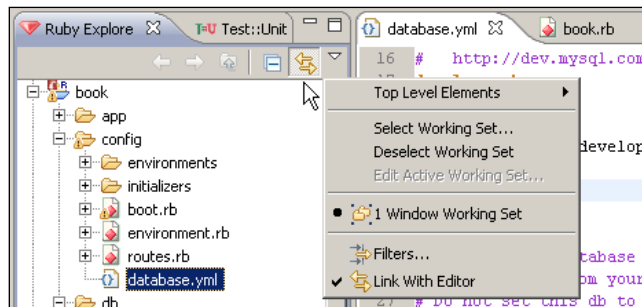
You can also access this option from the menu you get by clicking the small triangle to the right of the icon.

The next icon to the left, with a minus sign, is used to fold all the items. If you have a lot of sub-folders and you have opened lots of them, it comes handy to put some order in your view.

When you have lots of projects in the workspace, even if most of the projects are closed, still they occupy a place in your Explorer view. Often you don't want to see the whole list of projects but just to focus on your current project and hide the others. If you right-click on the name of your project in the Ruby Explorer and select **Go Into**, you will get exactly this functionality. You can also activate it through the **Navigate** menu on the top of the workbench. Notice you can do this on any given folder, and not just at the root project level.

If you go into any project or folder, then the first three icons of the Explorer view toolbar can be used to navigate your tree in a browser-like way. You can go back or forward to the last location, or up a level by using these controls. Unless you have gone into a project or folder, the icons will be deactivated.

The Console View

The console is the standard way for displaying any text output, especially when launching processes or applications such as Rake tasks, servers, or Subversion commands. Some other actions, like selecting **Tail** from the Ruby Explorer, will use an output console too.

When any of these actions is launched, a new console will be displayed in the console view. If the console view is open but it's in a tabbed view and it doesn't have the focus, the title of the tab will be displayed in bold so you can see that there is new output requesting your attention.
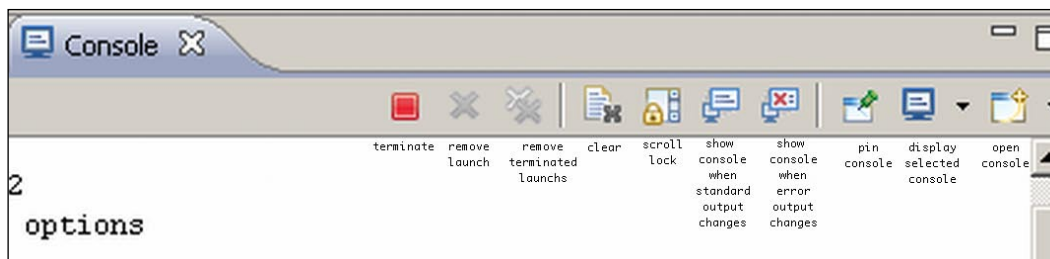
When you launch a command that opens a console, it will not open a new view, but will reuse the existing view tab, hiding the contents of the former console. Nevertheless, the old console remains active in the background. You can select which of the active consoles to display by selecting the icon with a screen (labeled 'Display selected console'). Selecting the icon itself will rotate through the opened consoles, and selecting the small arrow will present a list so you can select which one to display.

By default, when an action is launched or a console receives some output, that console will move to the top hiding the others. You can change this behavior by 'pinning' the current displayed console. Selecting the icon with the pin (see above screenshot) will make your console always be on top unless you explicitly select a new one from the opened consoles list.

The icon **clear** (see above screenshot) represented by a paper sheet and a cross on the corner just cleans the console output. The console stays open and it will display all the new output.

The **scroll lock** icon (see above screenshot) will prevent the console from scrolling when new output is received. For fast changing output is very useful since otherwise, it gets impossible to keep track of what's going on. This icon has a toggle-like functionality. You can switch it on or off with a single click.

If you want, you can have several console tabs opened at the same time, each of them displaying a different console output. To do so, unfold the **open console** menu (see above screenshot), which is the small down-arrow by the top-right of your console view, and then select **New Console view**. That way you can have different outputs displaying side by side instead of having to keep selecting from the drop-down.

Depending on which type of console you are displaying, there will be some additional icons. The captions on top of the icons are self-explanatory, and what you can do with these icons is usually any of these actions: stop the current process or server, close all the terminated consoles, close the current console, and make the console view display automatically whenever it receives new output. This last option will cause the console view to show directly instead of just marking the tab title in bold.

Notice that in some cases the console can prompt you for some input and it will keep waiting until you answer. In that case, you have to type directly in the console, which will be the Standard Input for the process. For example, if you are trying to launch a process that would overwrite some files, the process could ask what to do (yes/no/always).

During the next sections in this chapter, we will be using the console to monitor our generators and to check everything is working properly.

The Generators View

You know that, when you work with Ruby on Rails, there are several generators available, which you can invoke from the command line. By using these generators, you can create controllers, models, migrations, scaffolds, unit tests, or plugins, for example. A generator will create one or several files in the appropriate places of your Rails project, so you can customize them to meet your needs, saving you from all the mechanical and boring process of creating the files with the proper syntax inside.

Doing this from the command line is easy enough, but you do have to remember which flags you can use, change context from your code editor to the command line and back, and so on. By using the Generators view integrated in RadRails, you can comfortably create any of the available generators in a more convenient way.

In the default Rails perspective, you can find the Generators view as a tab in the lower pane of your workbench. If you cannot see the Generators view, you can go to the **Window** menu, select **Show View**, and select **Generators**.
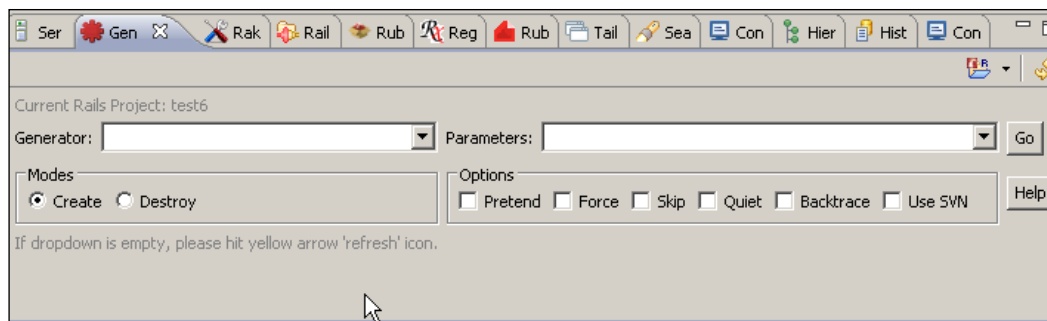
The first thing you have to do when opening the Generators view is make sure there is a Rails project selected in the Ruby Explorer. If you don't have a Rails project selected, then select one now. You can also select the project you want to work with from the 'Select Rails Project' icon in the toolbar of this view.

In the drop-down list by the left of the view, you will see a list of all the existing generators for the current project. As you probably know, in Rails you can install new generators for your projects. To be sure you are displaying the right list of generators for the selected project, you can click on the 'Refresh' icon located at the top-right corner of the view (it's the icon with two arrows). After refreshing, we are sure the generators in the drop-down list are the ones available for the current project.

Once you select any of the generators, there are several things you can do with them. If you are not sure about what a generator does, you can select the **Help** button. This will display a pop-up window with the explanation for this generator. Let's select **controller** in the drop-down list, and then click on **Help** to see how it works.

Now let's go back to the Generators view by clicking on the tab name. Apart from the drop-down list and the **Help** button there are some other options in this view. Let's start by explaining the long drop-down list close to the generators list. Here you can type any parameters you want to supply to the generator. Depending on which type of generator you are launching, the number and meaning of the parameters may vary. Check with the help of the generator if you don't know which are the available parameters. This input element is a drop-down list instead of a simple text box because you can open the list and see the history of the parameters you used in previous executions. (Note that the list will display the recent entries only for this Eclipse execution. When you close the IDE the list will be cleaned.)

When working with the generate script in Rails, you can do one of two things: either creating or removing the files for the selected generator. There is a radio button below the generators list in which you can select which of the two actions to perform.

Finally, when launching a generator, there are some flags you can use to control the behavior of the script. You can find all the flags as checkboxes. The available options are:

- **Pretend**: It just executes the generator displaying the output but without making any real changes in the file system.
- **Force**: If there are any existing files, the generator will overwrite them without asking for your confirmation.
- **Skip**: If there are any existing files, they will be automatically skipped.
- **Quiet**: This will execute the script without displaying any output.

- **Backtrace**: In the case of any errors, this will display debugging information.
- **Use SVN**: Changes in the file system will be reflected on your repository. In order to do so, the 'svn' command must be available in the path.

Now we know our way around the Generators view, we are going to create a very small application by using different generators. Even if it's possible to get everything with a single generator, for the sake of learning the mechanics, we will create the models, migrations, and scaffolded controllers needed for a simple two-table maintenance.

Generating Models and Migrations

Our small application is going to be for the maintenance of two tables: Books and Comics. The first thing we'll need for this application is to create the tables and the models that will represent them from the Ruby layer.

We are going to use these tables as examples, so we are going to keep them unrealistically simple. The Books table will have only a title, author name, and date of publication. The Comics table will have the same fields and an extra field for the illustrator name.

To create the models and the corresponding migrations we will go to the Generators view and select **Model** from the drop-down list. As you probably know, the parameters for generating a model are the name of the model itself and then an optional list of pairs of field names and types separated by a colon (notice there is no space between the name, the colon, and the type), so we will write the following parameters on the parameters field:

```
Book title:string author_name:string publication_date:date
```

After we type in the parameters, we have to make sure the radio button below the generators list is set to **Create** and then we can click on **Go**.

After some seconds we will be presented with the console output. Even when doing routine tasks, be sure you take a look at the displayed output. Sometimes there are small errors, or the application may need some kind of confirmation if there are existing files, and you would get that information as part of the output. Besides, recent versions of RadRails will present the output as hyperlinks to the generated files, so you can click at any file name directly on the console and it will open in the appropriate editor.

After the generation is done, we can go to the Ruby Explorer and look for the **app | models** folder of our project. A file with the name **book.rb** should be there. Also, take a look at the directory **db | migrate**. A new file named **001_create_books. rb** should exist. If you open this file, you will see the commands for creating and deleting the Books table in our database.
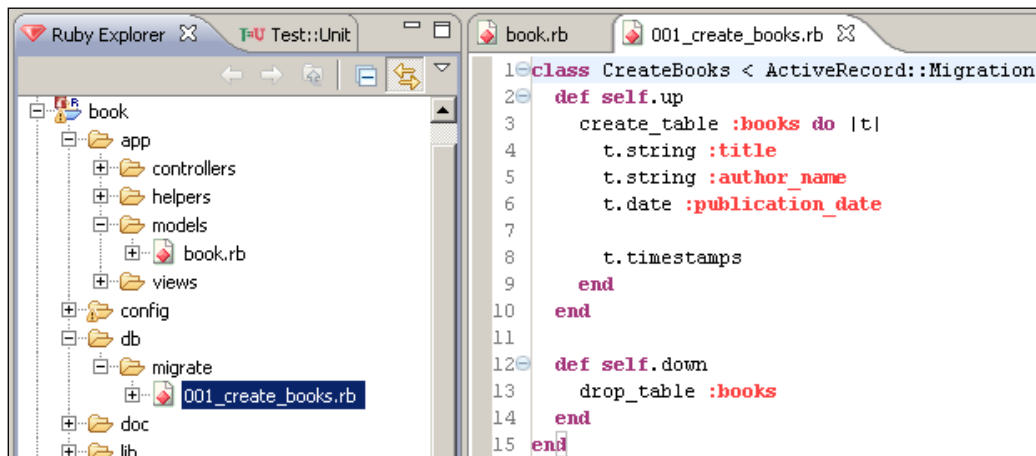
Now let's repeat the operation for the Comics table. First we go to the Generators view and select **Model** in the drop-down list, then we type the following arguments (notice it's a single line):

```
Comic title:string author_name:string illustrator_name:string
  publication_date:date
```

After getting the console output, if you go to the Ruby Explorer you will see the newly created files.

Notice it's also possible to generate a migration file without any model code (for example, when you are adding or removing a column) just by using **Migration** instead of **Model** from the generators list. Also, it's possible to generate a model without a migration by passing the extra parameter `--skip-migration`.

Running Your Migrations

We already have our migrations describing the tables to create, but we still haven't made any changes to the database. In order to actually create the tables, we will have to run our migrations.

The first thing we have to do is configure our database connection. Look in the Ruby Explorer for the file named **database.yml** in the folder **config**. You will have to modify this file and specify your adapter type, host, database name, user, and password.
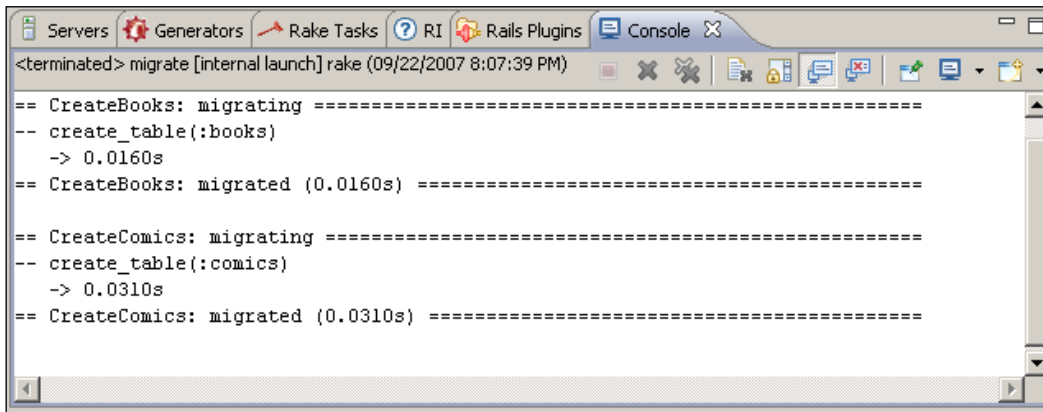
If you are going to use SQLite as your adapter, then you don't need to create the database beforehand since it will be done automatically for you. However, if you are using a different adapter you have to create the database before trying to run the migrations. If you are using Rails 2 and the user configured in your **database. yml** has enough privileges, we can take advantage of the built-in Rake task **db:create** to automatically create your database. If you are using an older version of Rails, or your user doesn't have privileges for creating databases, you will have to create it manually in your DBMS.

To create your database using a Rake task, go to the Ruby Explorer and right-click on the name of your project. In the context-menu, open the **Rake** option and then select **db** and **create**. This will launch the selected Rake task. Look at the console so you can see if there were any errors.

With the database created we can already run our migrations. We can execute the Rake task from the context-menu as we just learned, but there is a specialized Rake Tasks view, which provides some more functionality. Even if we don't need all that functionality at the moment, we can use it for running our migrations. Open this view by selecting the **Rake Tasks** tab. If you cannot see it, then open it from the **Window** menu, by using the **Show View** option.
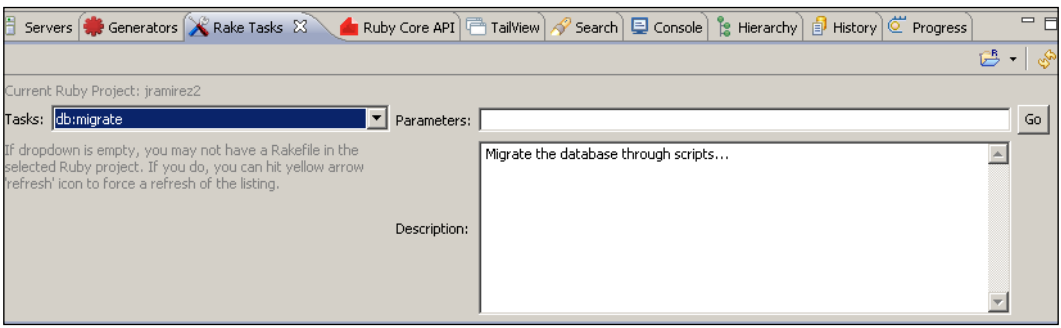
You will see this view looks pretty similar to the Generators view. Make sure your project is currently selected on the Ruby Explorer then hit the **Refresh** symbol of this view. Once the drop-down list is refreshed, you have to select the task with the name **db:migrate** and click on the **Go** button. After some seconds, you should see the results of running our migrations on the Console. If everything went well, now we have our tables created on the database.

Generating Scaffolds

Now the Models and Migrations are created, we will have to create the Controllers and Views so we can start playing around with our application. Generating a controller is as easy as generating a model or a migration. If you go to the Generators view, you can see there is an option named controller in the drop-down list. If we were to use that option, we would get an empty controller, and some related files and folders for testing purposes.

However, Rails provides a more convenient way of creating a controller if you are going to use it for table maintenance. Of course we are talking about generating Scaffolds.

Obviously, you never want to use exactly what the scaffold provides, but you know how it is with programming. If you have to start from scratch, you feel lazy and it takes a while to get up and running; but if you already have some code and you only have to modify it then the things get done much more quickly.

Instead of using the Generators view to generate a stand-alone controller, we can generate a controller containing the source code for all the methods provided by the scaffold. Besides, we will get the views, so we can customize them as we please.

This functionality is provided by the **Scaffold** generator. This generator will create your controller, your views, and if the model doesn't already exist, it will create both the model and the migration file too.

Even if this generator can create the whole set of files, if you prefer to, you still can generate your model first together with the corresponding migration, then run your migration, and then run the Scaffold generator. As long as the name of the Model and that of the Scaffold match, everything will be fine and you will not get any errors.

We will create now the scaffolded controller for the model Comic. Go to the Generators view and select **Scaffold** from the drop-down menu. We will write **Comic --skip-migration** as the argument, and click on Go to launch the generator.

> The '--skip-migration' argument is necessary because we previously created the migration, and the current scaffold for Rails 2 will stop if the migration file exists, without giving the option to ignore or overwrite.
>
> If you are generating a Scaffold without first creating a migration, don't use that argument.

As usual, after some seconds you will see the output in the console telling you about the generated files. Since the Comics model already existed because we created it in a previous step, you will see in the console this file was skipped by the Generator. To see the scaffolded controller, you can go to the Ruby Explorer, and open the file **comics_controller.rb** under the folder **app | controllers**, or you can directly click on the **comics_controller.rb** link in the console. When the file opens, you will see the methods are coded here. If you need to change any of them, you only need to modify the existing source.

Also, if you take a look at the **app | views** folder, you will see a new folder **comics** has been created for you. Inside this folder there are all the necessary views for our small application. Notice a layout named **comics.html.erb** has been generated in the folder **all | views | layouts**.
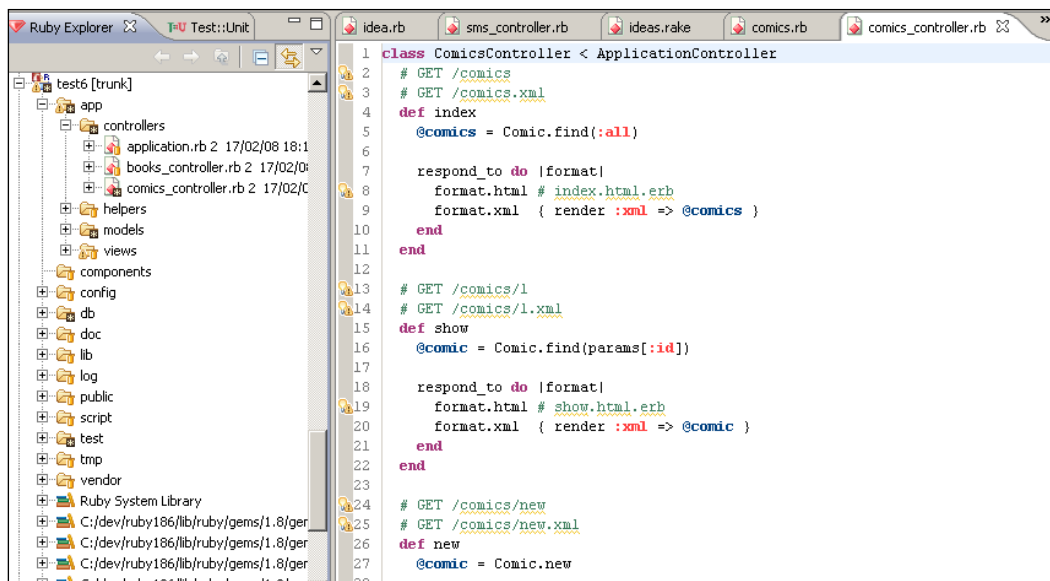
Now let's generate the scaffold for the model Book. Go to the Generators view, select Scaffold from the drop-down menu, write **Book --skip-migration** as the argument, and Click on **Go**. The console will display all the files automatically generated for us.

This is starting to look like a good starting point from which we can customize our application without having to take care of the mechanical details of creating classes, views, and basic methods.

I know what you are thinking; this looks OK, but you want to see it running at once and see how it looks. You know, your wish is my command, so let's start our Rails server and play with our small application.

Starting Your Server

Starting your Rails server from RadRails is really easy, but it may require just a little bit of setup. If you remember, when in Chapter 2 we were creating a new project, there was a checkbox to create a Mongrel server.

Creating a server in RadRails will allow us to start, stop, or restart our servers directly from the IDE and get the output in the integrated console, rather than having to work from a command line.

If you didn't create a server when creating your project, you can do so now. Right-click on the name of your project in the Ruby Explorer, select **New** and then **Rails Server**. We'll go about the server options later in the book, so for now it's OK to accept all the default options.
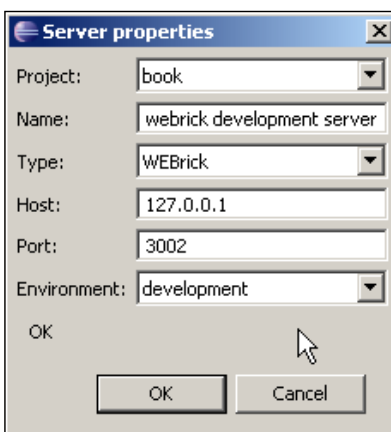
Just a quick note, if you want to work with a Mongrel server, you must have it installed in your system. If you select to create a Mongrel server, but you haven't installed Mongrel, RadRails will ask you if you want to have it automatically installed.

If you had any problems with the automatic installation, you could try to install Mongrel manually as a gem by opening a command prompt and executing:

```
gem install mongrel_rails
```

Note that in some OS you will need either a privileged user or to execute this command via **sudo** (sudo gem install mongrel_rails). Further in this book you will learn how you can install gems directly from the IDE without having to go to the command prompt, but since we are still in our first steps we will have to wait for a while yet.

Now that we already have a Rails Server configured, we only have to start it. We are going to use a new view for that. The name of the view is **Server** and it's one of the default views in the Rails perspective. If you cannot see it, just open it as usual from the **Show View** option of the **Window** menu.
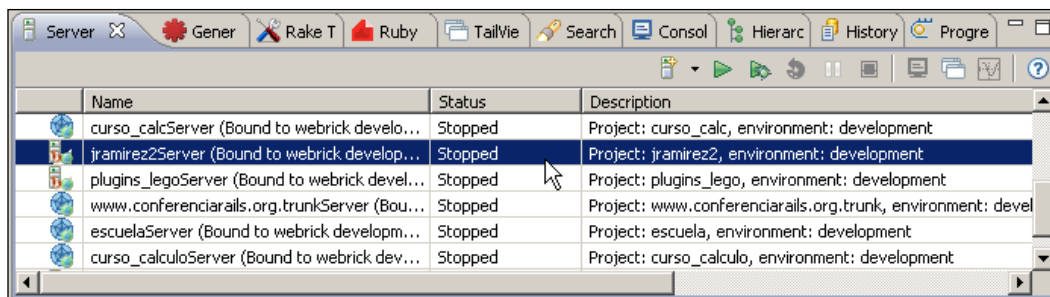

In this view, you will see an entry for each configured server (only one if this is your first project with RadRails). We are going to start our server, so make sure your database is running or start it if it's not. Now, right-click on the entry corresponding to the project we are working with and select **Start**.

After a little bit, you will see the console view displaying the usual server output. Once you see the server is started, we are going to try our application. Directly at the console view toolbar, there is a **Launch Browser** icon. Alternatively, you could go back to the **Server** view, right-click on the name of your server and select **Launch Browser**. In any case, the integrated browser will open and you should be seeing the Rails index page.

First we are going to take a look at our Books table maintenance functionalities. After the host and port of the URL, write **books**, so it will look something like `http://localhost:3002/books` (don't change the port number in the URL, or you will not be able to connect to your server).

In the browser, you will see an almost empty screen with the title **Books Listing** and a link to add a new book. Click on the link and a form for entering the book information will be presented.
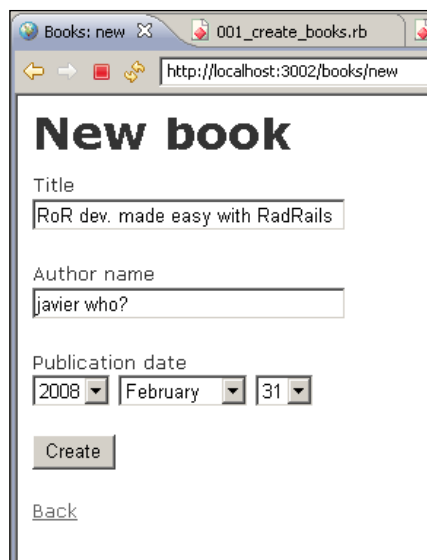
After entering some example data and accepting the form, you'll be redirected back to the listing screen, in which you can already see your brand new book. You can add more books, edit, show, or delete books.

If you change the world 'books' to 'comics' in the URL of the integrated browser, you will see exactly the same but for the Comics table.

Now with this skeleton created, we can start customizing our application, changing the appearance, adding validations, whatever needs to be done, but without all the hassle of the creation of the basic structure.

Monitoring Your Server

You know how convenient is to take a look once in a while at the Rails log file to see which actions are being called, which parameters are being passed, or what SQL is being sent to the database. By using RadRails, we can have one integrated view continuously monitoring the log of our server.

Go to the Ruby Explorer, open the **log** folder and right-click on **development.log**. Now select **Tail** from the context menu. By doing so, you will be display in the Console view the contents of the development log file as they are being written.

If the Console view is not in the foreground, click on the tab named **Console** to bring it to the front. Now go to the integrated browser and just navigate around a bit, creating or editing some records. You will see in real time the information being written to the log of your project.
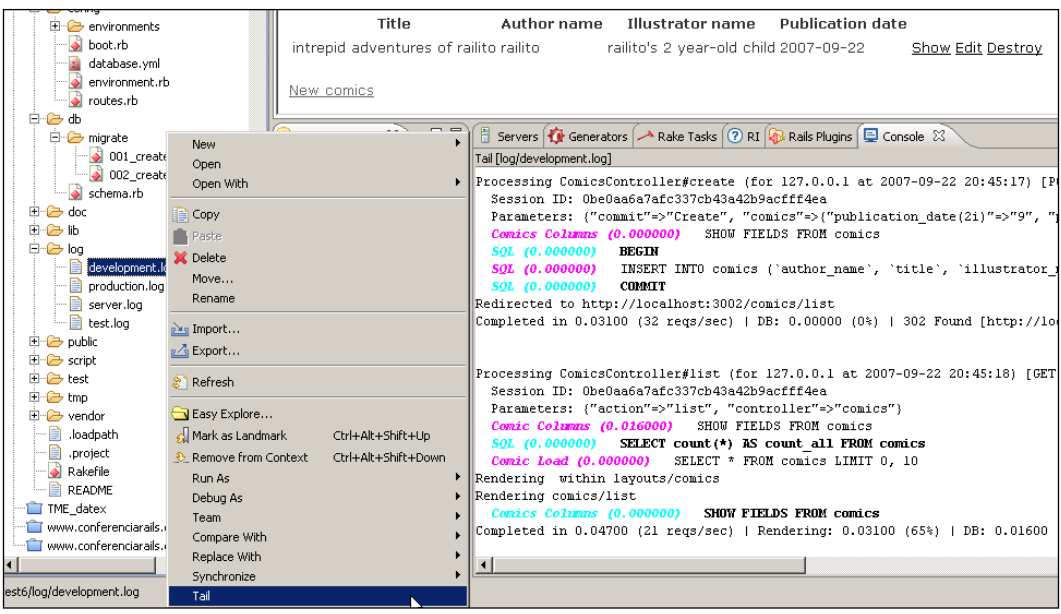
Here the Pin Console icon comes in most handy. Otherwise, the console output from the server and that of the log viewer will keep fighting to become the focus/foremost, since by default every time there is new output in any of the consoles, it will be brought out to the front If you'd rather just stick to seeing one or the other, you ought to 'pin' it down.

This tail feature is ok for a quick inspection of your logs. However, since version 1.0 of RadRails there is a new view, the Tail view, which has some more advanced features at the extra cost of a minimal setup. We will see how to use that view later in this book.

Summary

In this chapter, we have learned how to use two of the views we will be using most of the time when developing with RadRails. By using the Ruby Explorer, we can browse the structure of our projects and make file-related tasks, which include working with the local history or file-comparison. From the Console view, we will have access to the output of every process we launch from our workbench, so we don't need to go to the command line or use a third-party application for monitoring the results.

We have also seen how to use the Generators view in order to create models, migrations, controllers, and scaffolds, and how we can start our Rails server and use an internal browser for quick testing of our applications, making the development process much more comfortable, and letting us focus on what we need to do and not on which tool we are using.

4
Writing Ruby Code

So far, we have been using RadRails only as a convenient tool to make our work a bit more comfortable and forget about using the command line. It's nice not having to remember every single command and option, but RadRails is much more than that.

In this chapter, we will learn how to use tools that will boost our productivity and will help us write our code in a much easier way, letting us concentrate on what we need to do and freeing us from routine tasks.

Even if you can write Ruby code in any plain-text editor, things get considerably easier when your editor lets you navigate between your different classes with a single click, assists you by giving you all the possible methods for any object, outlines the structure of your classes, or provides code templates so you don't have to remember the exact syntax and parameters of every method.

By using RadRails, we will have access to all of these features, and even some more. Once you start taking advantage of the possibilities of this IDE, you will start wondering how could you possibly live before without using it.

A Quick Note about Keyboard Shortcuts

For many of the not-so-frequently-used actions, we will just look them up in the corresponding menu bar or right-click and select from the context menu. However, when you are writing code, often you will find yourself not using the mouse but directly using the keyboard shortcuts (Eclipse calls this feature Key Bindings).

For example, usually when you are typing and you want to find a word in a document, you don't want to take your hands out of the keyboard and open the **Search** menu with the mouse. You just want to click a couple of keys (*Ctrl+F*) and have the dialog in front of you, ready for entering the text to find.

Take into account that in Eclipse many of the shortcuts are context-sensitive, so when you press the key combination it will launch a different option depending on where you have the focus at that moment. For example, if you press *Ctrl+O* when you have the focus on the Editor Area, then a Ruby tool called **Quick Outline** will be presented, but when you do the same action having the focus on the Ruby Explorer, the dialog for **Open File** will appear instead.

When you have been working with the IDE for a while it just feels natural, but at the beginning it might seem a bit annoying that the same shortcuts can render different results. Of course the rationale for this behavior is that having non-context-sensitive shortcuts would reduce the number of operations we can do without using the mouse when editing our code. Still, at the beginning it might feel just a bit odd, so just give yourself some time until you get used to it.

In this chapter some keyboard shortcuts will be mentioned. Note that the key combinations are valid for Windows and Linux users. For Mac users, the key bindings should be the same, but using the *Command* key instead of *Ctrl*. Later in this book, in the chapter about Preferences, you will learn where to see and modify all the key bindings for the IDE.
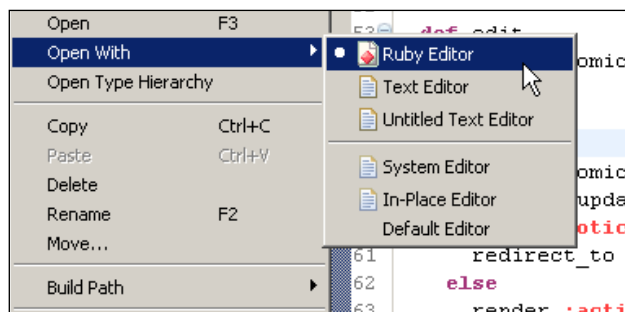
The Ruby Editor

Most of the time we spend developing a Ruby on Rails application, we are writing source code. Thus, it's just logical to expect that a good IDE will provide the necessary tools to make the coding as comfortable as possible.

As you probably remember from the first chapter of this book, when you install Aptana RadRails, one of the key components that comes bundled is the plugin Ruby Development Tools (RDT). RDT provides some really useful views related not to Rails itself but to pure Ruby.

One of these excellent tools is the Ruby Editor, in which we will write all the code for our Models, Controllers, Tests, and Libraries. We will not use this editor for writing our RHTML, ERB, YAML, JS, or CSS files, since Aptana provides more specialized editors for these tasks, as will be seen in the next chapter.

After installing RadRails, the Ruby Editor will open by default when you open any file with a `.rb` extension from the Eclipse IDE. Files with the extensions `.rbw`, `.cgi`, and `.rake` will also use this editor, which will be used too for some special file names like **Rakefile**. If for any reason your source file has a different extension, you can still open it with the Ruby Editor by right-clicking on the file name in the Ruby Explorer and selecting **Open With** and then **Ruby Editor**.

In Chapter 8 you will see how you can create associations between any extensions and the different specialized editors available in Eclipse.

In the previous chapter, we generated a file named `comics_controller.rb`. We can now open this file to start getting familiar with the Ruby Editor.

Syntax Highlighting

The first thing you will notice is the syntax highlighting. The editor will colorize the coding elements using different colors and font styles, so you can follow the source code in a more comfortable manner.

In our controller you can see how the comments are presented in green, the keywords in purple, the instance variables in dark blue, the symbols in red, and the string literals in light blue. Other language elements such as global variables or regular expressions get their own highlighting too, even if we cannot see it in our controller since we don't have any right now.

If you don't like the default color schema, or if you are having difficulties in reading it because of your screen or for any other reason, it is possible to change it and adapt it to your needs.

You only have to go to the **Window** menu, then select **Preferences**. In the preferences dialog, open the **Ruby** option, then **Editor**, and finally **Syntax Coloring**.

At the bottom of the **Syntax Coloring** pane, you can see a sample of the current color schema. Above that, there is a box titled **Element**. If you click on the plus sign by the left of the word **Ruby**, all the possible language elements will appear.
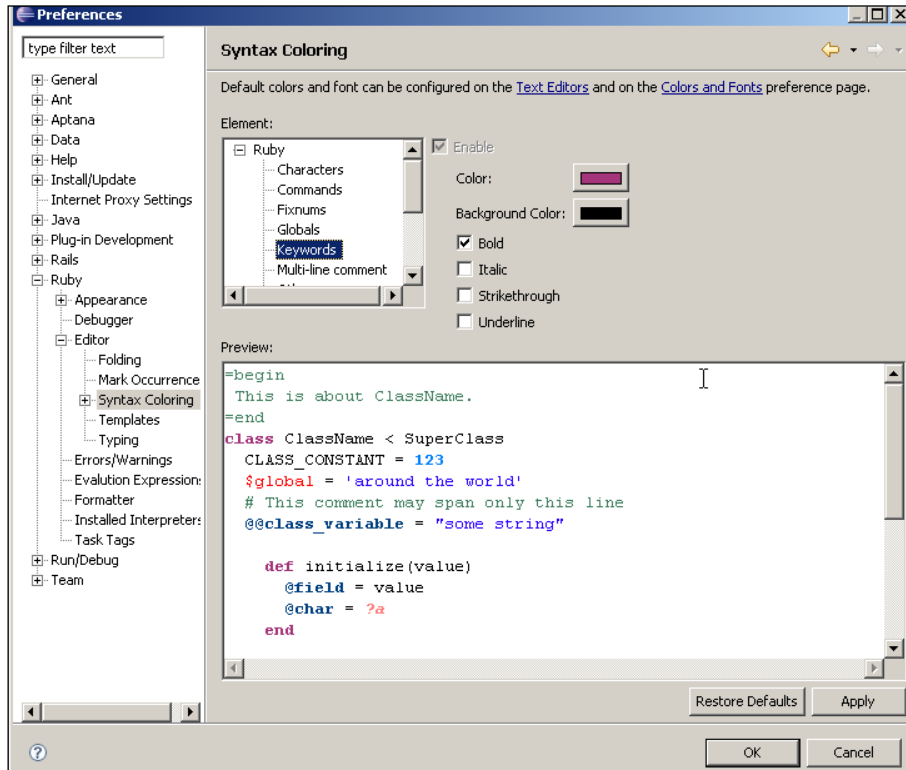
Now you can select any of the elements, so you can see and change the current configuration. Just for the sake of trying, you can select the label **keywords**. You will see the current configuration is bold with a foreground color of purple. Just play a bit and change it to Italic or Underline to see the effect. You will get an idea.

If you change the current syntax highlighting too much and want to get back to the starting point, you can always select **Restore Defaults**.

Outlining the Structure of Your Ruby Code

When you are writing a program in any language, it always helps to have an easy way to outline your project, browse the structure of your classes, and see the hierarchy and the contents without all the implementation details.

This is specially useful in a language like Ruby and a framework like Rails, which encourage highly structured code and small methods whenever possible to achieve the DRY (Don't Repeat Yourself) principle. Since, typically, your methods will be composed of just a few lines, you will probably end up with a lot of small methods and maybe also with a bunch of instance or class variables to control the status of your objects or share information between the different methods.

Apart from the inner structure of your objects, another important point about the structure of your application is its class hierarchy. Being able to easily browse and navigate the hierarchy of your objects makes it easier to follow the track of the different inherited methods and properties. Moreover, by easing the representation of your hierarchy, you will feel encouraged to design your application in a structured way.

In addition to the features we already went over when talking about the Ruby Explorer in Chapter 3, Eclipse and RadRails offer different views and tools for outlining your project's contents and make your coding faster and more comfortable.

Quick Outline

First of all, let's open the code of our **comics_controller.rb** file under the folder **app | controllers** in our project.

If you right-click over any part of the editor view, you will see an option called **Quick Outline**. As usual, close to the name of the option you have the keyboard shortcut for invoking this option (*Ctrl+O*).
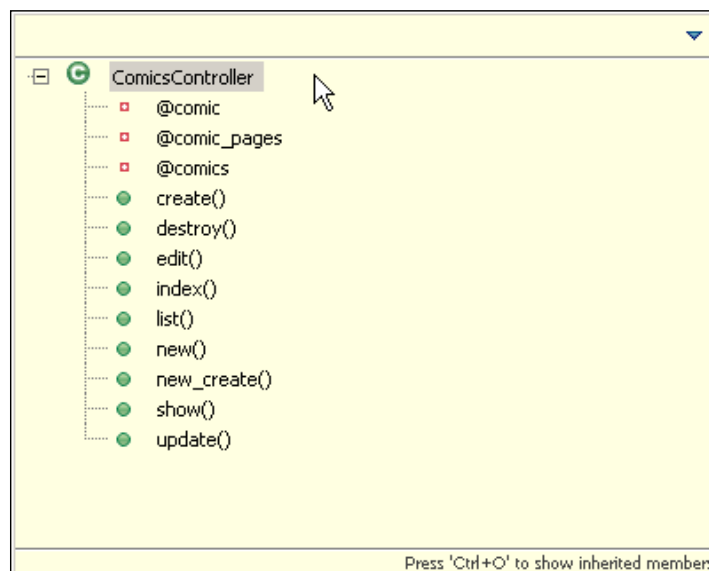
When the **Quick Outline** opens, you will see an outline of all the modules and classes in your current source file presented as a tree. By clicking on any of the elements of the tree you will go directly to that part of your source code in your editor. This comes in handy for quickly moving between different methods or for going directly to any variable declaration.

In our case, when we outline the code of our comics controller, we will see a first level representing the class itself and then different entries for the methods and the instance variables used in this class. Below the methods, we will see the local variables they use. When you click on any of these elements, the outline will close and your cursor will be positioned at the declaration of the element you selected.

The order in which elements appear is the same in which they are defined in the source code. If you prefer to have alphabetical sorting, then you can select the down arrow at the top-right corner of the **Quick Outline** dialog and select the **Sort** option.

In our short example, we can see a sample of what a quick outline can look like, but depending on which source code we are working with we can see more element types represented in this tree.

When we are working with a source code in which explicit `require` or `load` statements are present,, there will be an entry name **require/load declarations** on top of your tree. Under this option you will find the names of all the requested files. Even if the require or load statements appear scattered through your source code, they will be presented grouped together in the outline.
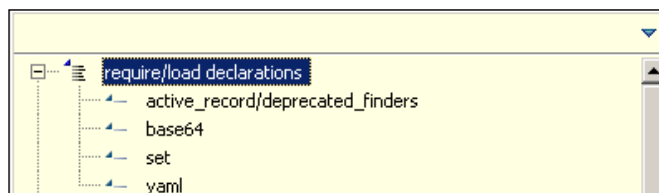

You will also find a tree branch for each of the Classes or Modules defined at the top level of your source file. Under each class or module you will find another branch for each class or module nested inside. At every branch you will find the outline for the defined constants, the class/module, and instance variables, and the different methods. Besides, should any methods define local variables inside, a new branch will be presented with the outline of the variables.

Each of the different elements appears with a different icon in the tree, so you can differentiate them in an easy way. Also, when outlining the methods, different icons will be used depending on their visibility (public, protected, or private). The list of possible elements and the meaning of their icons is as follows:

- **Require and Load statements**: These always appear at the top with a small blue triangle by the name of the target file.
- **Global Variables**: These appear by the top of the outline, below the require/load branch. They are represented by a miniature icon resembling the Earth.
- **Modules**: These are represented as a blue circle with a white **M** inside. A module can be folded or unfolded to show or hide the details inside it when outlining.
- **Classes**: These are represented as a green circle with a white **C** inside. As with modules, they can be folded or unfolded.
- **Constants**: These appear in the branch corresponding to the module or class defining the constant. They are represented by a large solid lime-green square.
- **Instance Variables**: These are represented by a red hollow square. They appear below the branch corresponding to their class or module.
- **Class Variables**: These are represented by a red hollow square with a small **c**. They appear below the branch corresponding to their class or module.
- **Local Variables**: These are represented by a gray solid circle with a white **L** inside. They appear when you unfold the details of the method in which they are defined.
- **Public Methods**: These are represented by a small solid green circle. If it is a class (singleton) method it will also have a small **s**.
- **Protected Methods**: These are represented by a small solid yellow diamond. If it is a class (singleton) method it will also have a small **s**.
- **Private Methods**: These are represented by a small solid red square. If it is a class (singleton) method it will also have a small **s**.
- **Constructors (initialize methods)**: They are represented by a small solid green circle with a small **c**.
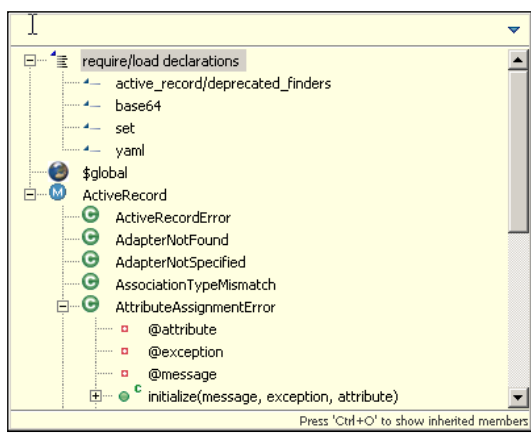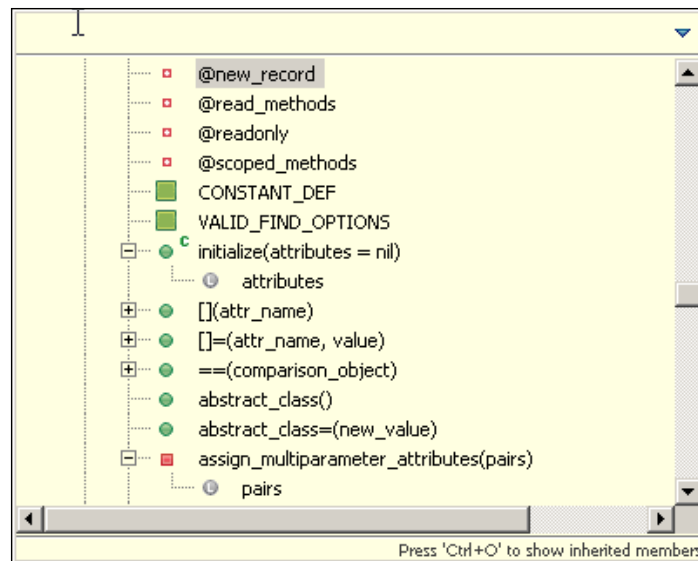
So far, everything we have seen about the quick outline is just a convenient way of navigating your code in a very similar way to with the Ruby Explorer, with only the advantage that we can invoke the outline with a keystroke without having to move from editing our code.

Besides that functionality, there is a very interesting feature of the Quick Outline, which is not present in the Ruby Explorer. By the bottom-right corner of the Quick Outline window, you can see a label displaying **Press 'Ctrl+O' to show inherited members**. If you do so, you will see now we are presented with many more methods and variables than before.

Now the list will include the entries available from our class, not only the ones explicitly declared in this source code, but also the entries for each parent class in the class hierarchy and for every included module.

In order to know where each of the elements comes from, a literal will display by the right of each definition indicating the class or module in which the definition resides. By clicking on any of these entries, you can go directly to the defining source.

Since the amount of information can be a bit overwhelming when using this feature, when you know the name of what you are looking for you can just start typing on the upper bar of the Quick Outline view. As you type, the entries will filter to display only those matching your search string.
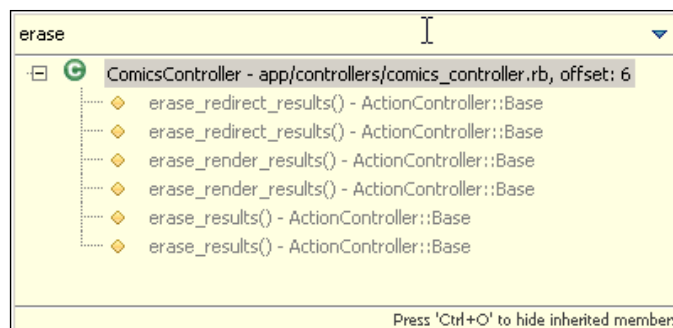
There is still one more filtering feature when quick outlining. Sometimes you want to filter out a whole set of entries by name and display only the entries not in that group. You can do so by clicking the top-right arrow in the **Quick Outline** window and then selecting **Filters**. In the text box you can write any text pattern you want to filter out. You can use the wildcards ***** and **?** to represent any string or any single character. You can introduce several patterns using a comma as the pattern separator.

Just be aware the filter will persist between different calls to Quick Outline, so if you don't delete it or deactivate it by unchecking the box in the filter dialog, next time you outline any class, your filters will be applied.

Type Hierarchy

Often, you are interested not so much in the structure of the code itself but in the relationships between your different classes: which are ancestors or descendants of the current one, and which members it is inheriting from where.

RadRails provides a convenient view for outlining your Type Hierarchy. You only have to select the name of a class in the Ruby Explorer, or position yourself inside the code of any class in the editor area, and then right-click and select **Open Type Hierarchy**.

A new view will open with an outline of the parent and child classes for the selected one. By the side of this tree, you will see the list of member methods for the current class. The symbols to represent them are the same as you can find when quick outlining.
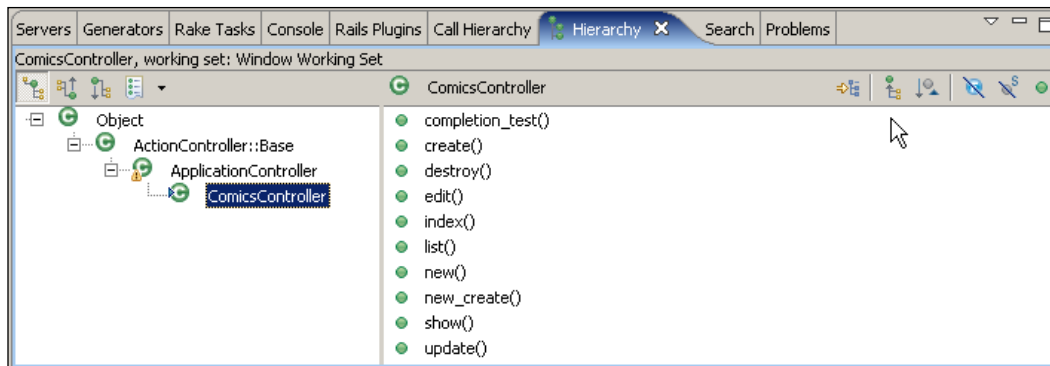
Of course, if you click on any of the classes or methods, the editor area will display the selected element accordingly.

By default, the Hierarchy will be rendered in both ways, for ancestor and descendant classes. You can easily change the scope of the outline. At the top-left corner of this view you will see three icons representing a tree. In the first one there is just a tree without an arrow, in the second there is an up-arrow and in the third one there is a down-arrow.

If you hover the mouse cursor over the icons the literals will read **Show the Type Hierarchy**, **Show the SuperType Hierarchy**, and **Show the SubType Hierarchy**. By selecting them, you will see the default tree with all the classes in the Hierarchy, a bottom-up, or a top-down view of the classes as from the current class.

It's interesting to note that when using the **SuperType Hierarchy** you will be able to see the included modules as multiple inheritance. When selecting this kind of display, the focused class will be shown as the root of the tree and the leaves of the tree represent the parent class and all the included modules.

You can also change the scope of the outline by selecting the context menu of this view in the top-right arrow and selecting the corresponding option.

In this same context menu, there are a couple of interesting options. If you open it and go to **Layout** you will be presented with different options for displaying the tree and the detail of the methods.

You can choose a horizontal layout, in which you will have two panes side by side, a vertical layout in which you will have two panes displaying one on top of the other, or an automatic layout, in which depending how deep is your tree, Eclipse will decide whether to display it horizontally (the default) or vertically. If you are only interested in the Hierarchy and you don't care about the methods, then you can select the last layout, named **Hierarchy View Only**, and only the pane with the tree will be shown.

Another interesting option in this view's context-menu is **Show Qualified Type Names**. By selecting it, you will get the path of the file in which the different classes are defined right by the name of each class. This can come handy when you have a lot of classes with the same name in different branches of your application.

Notice that there is a group of icons by the top-right corner of this view. You can use them in a similar way to when outlining for hiding or displaying details about the class contents. For example, you can hide the methods defined at the class level or those that are not public.

There is just one detail left before we finish with this view. If you want to change the current class regarding which the hierarchy is presented, you can right-click on the name of the class in the hierarchy tree and select **Focus on** (YourClassName).

If you want to focus on a totally different class, which is not represented in the current hierarchy, you can right-click on any class name and select **Focus on...**. This will display a dialog in which you can type the name of the class you are looking for. As you are typing, the classes with matching names will be filtered. When you see the class you are looking for, you can just select it and it will display in the hierarchy view.

General Outline View

With the Ruby Explorer, the Quick Outline, and the Hierarchy View, we have all our outlining necessities well covered. However, when working in a Rails application, I sometimes prefer to use the Ruby Explorer as a file browser, without opening the structure of the types. When I start opening the details, specially when I have a lot of methods, I find myself scrolling a lot to change from one file to another.

Of course I could still use the Quick Outline, but sometimes I want to type in a file or scroll it down as I see the outline, and with Quick Outline I cannot do that.

There is yet another view for Outlining in RadRails. It doesn't do anything you cannot do with the Ruby Explorer or the Quick Outline, but it has one interesting feature. You can keep it always open and dock it in a corner of your workbench, and it will display the outline of the class currently open in the Editor Area.

Since some versions ago, this view is opened by default at the Rails perspective, but if for any reason it's not shown in your workbench, you can go to the **Window** menu, then select **Show View** and then **Other...**. In the opening pop-up you have to open the **General** category and then select **Outline**.

The Outline view represents the already familiar icons for the different variables and methods inside our class. You have a simple icon group for sorting the contents or filtering out some elements, and you can also filter by a name pattern from the context menu of this view (open it by clicking on the down arrow by the top-right corner of this view).
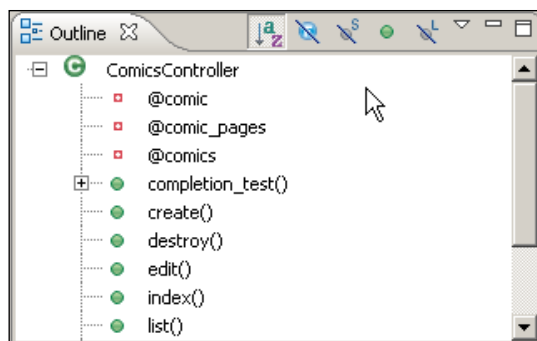
My favorite configuration when developing is the Ruby Explorer on top, and then the Outline view right below on a small square. That way I always have the list of files on top, and the structure for my current editor below. Make sure you go to the context menu of the view and select the option **Link With Editor**. This way, whenever you open a new file on the editor area, you will automatically get the outline and you can quickly change from one method to another.
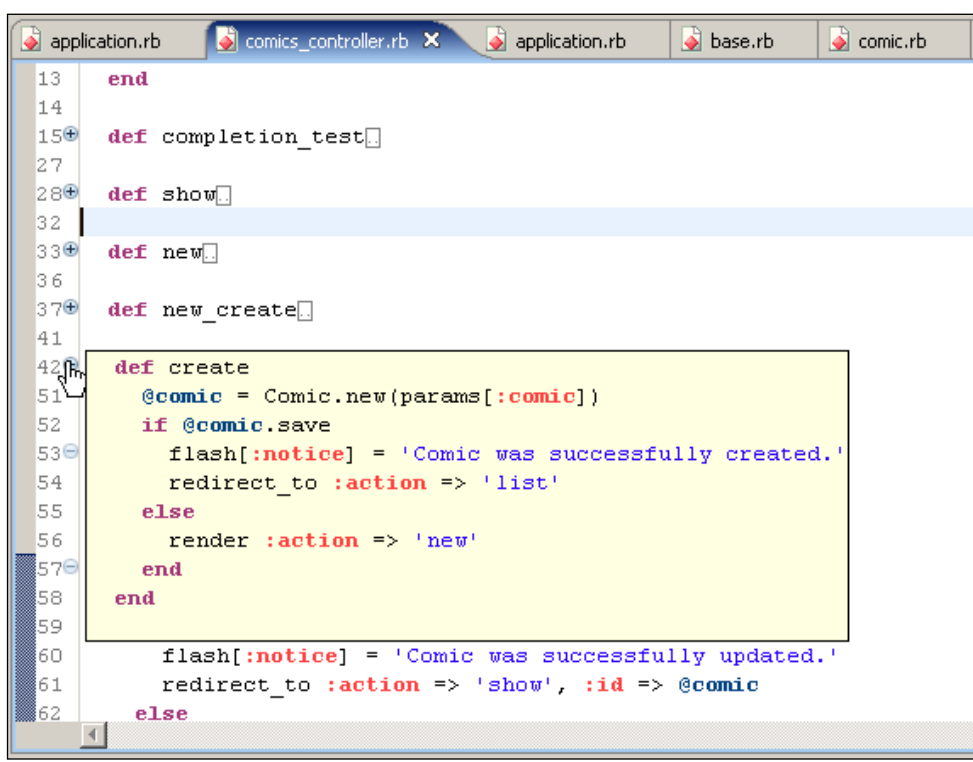
Code Folding

Often when we are writing code for one class, we need to compare the source in two different methods, cut and paste between them, or just look at them at the same time because they need to interact. This, of course, is a fairly easy task. But sometimes, especially if the code blocks we need to work with are not close to one another in our code, we can find that there are other methods or even comments in the middle and we need to scroll very frequently.

In this kind of situation, we can take advantage of the Ruby code folding features. The Ruby Editor automatically detects collapsible blocks of code and allows us to 'fold the code' so only the first line of the block will remain visible. By using this feature, we can hide the parts of the code that are getting in the way and we can focus on the rest of our source.

The Ruby Editor allows us to collapse modules, classes, methods, and RDoc comments. Whenever you open a Ruby source, the Editor will parse it and identify the possible folding points in your code. At each of these points, you will see a small circle with an enclosed minus sign by the left ruler of the editor.

If you open the Comics Controller in the Editor Area, you will see we have these folding handlers by the left of the class definition and by the name of each of the methods. When you click on the folding symbol, only the first line remains visible. Now the circle by the ruler will display a plus sign instead of a minus one. Besides, by the end of this remaining line, you will see a small square with two periods. These are the indicators that this line is the first of a collapsed block.

If you want to see the whole code again, you only have to click back on the folding handler for this block. However, if you only want to take a look at the code but you don't want to expand it, you can just place the cursor over the folding symbol and wait for an instant. After a brief moment, you will see the hidden code, but without having to open it.

Code collapsing or expanding can be done individually block by block or you can do it to a group of blocks with a single click. If you right-click with your mouse over the rule area at the left of the editor, you can see an option named **Folding**. As you can see, you can expand or collapse all the blocks with a single click.

You can also choose to collapse all the RDoc comments or all the members. In this context, a member would be any method that is defined within the scope or a class or module. Folding all the members comes in very handy when you are browsing well documented Ruby code and you only want to know what the code does and what's the signature of each method. It would be an equivalent of reading the RDoc documentation for that source file directly in your editor.

On the other hand, collapsing all the comments comes handy when you are working with well documented code that you already know. In this case, the large comments at the beginning of your methods will collapse to a single line, so you have more useful space for the code itself.

In this folding menu, there is still another option named **Reset Initial Structure**. Apparently this option is equivalent to **Expand All**, which is true by default. However, one of the configuration options for the Ruby Editor is how the collapsible code is presented when you open a new editor. You can choose to automatically fold all the comment blocks, for example. In this case, **Expand All** and **Reset Initial Structure** will render different results.

We'll go over all the configuration options related to RadRails in a later chapter of this book.

Code Formatting

One of the keys for the success of Ruby as a language is the readability and expressivity of the source code. These features get strengthened when your source code is clean and well organized.

Unlike some other languages, Ruby doesn't take into account how the source code is written as long as the syntax is correct. This means you can write your code without proper indentation, mixing arbitrary numbers of spaces, and so on. Even if the Ruby Interpreter will be able to process your code, the Human Interpreter working with your code will have a hard time trying to make sense of badly formatted code.

Unfortunately, even when we try to keep the code tidy, there are many occasions when we don't manage to produce code as good as we would like. This can be specially true when we are pasting code from different places, or when we are patching code with temporary solutions that end up being permanent.

Whenever our code format gets out of control, or if we are inheriting poorly written code from an external source, we can use the RadRails built-in code formatter.

Code formatting will take any source code and will apply the correct indentation rules to make it pretty. By default, the Ruby formatting rules will be: using spaces instead of tabs and using two spaces for every indentation level. These are the common conventions when submitting source code to places like the Rails Development Repository or `Rubyforge.org`.

If for any reason you don't like these rules, you can use your own. We'll see how to do that later in this book when talking about the RadRails configuration options.

In order to see how formatting works, let's open the source code for the Comics Controller and mess around with it by removing or adding spaces to make it look confusing.

Now we have two options. Either we can select the lines we don't like and apply the formatting, or we can just not select anything and apply the formatting so the whole source code will be formatted.

The code formatting option can be accessed from two different places. You can go to the **Source** menu and then select **Format** or you can directly right-click on the editor area and select **Source | Format**. In both cases, RadRails will directly format either your selection or your whole source. As usual, you can also use the keyboard shortcut displayed by the **Format** menu option (*Ctrl+Shift+F*).

```
def create
          @comic = Comic.new(params[:
      if @comic.save
 flash[:notice] = 'Comic was success
  redirect_to :action => 'list'
                        else
render :action => 'new'
end
          end
```

```
def create
    @comic = Comic.new(params[:comic])
    if @comic.save
      flash[:notice] = 'Comic was succe
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end
```

Code formatting is undo-able, so if for any reason you don't like the format of the code, you can always get the original formatting back by selecting **Undo** in the **Edit** menu. Usually there is no reason for undoing the formatting, but in strange cases where your source code is using the heredoc format for strings (that is, `<<-EOF`) or if you are using multi-line regular expressions, the formatter could fail and not be able to indent properly.

Indenting Code Blocks

Apart from the automatic formatting you can always change the indentation of a block manually. Suppose you have a piece of code and you are going to surround it with a new `if` or a `begin`/`end` statement. You only have to select the whole block, and click the *Tab* key. If you want to unindent it, then you have to use the key combination *Shift+Tab*. You can also perform these operations by selecting the target block and right-clicking on it. The options **Shift Right** and **Shift Left** will allow you to change the indentation level.

Commenting Code Blocks

A very common operation specially when testing or debugging our code is commenting a whole block of statements. As you know, Ruby as a language doesn't provide any syntax for commenting out a block. You could always use the `=begin`/`=end` syntax for commenting, but those comments have a special meaning and are only intended for documentation purposes, so it wouldn't be right to use them.

The lack of a block comment syntax in Ruby makes commenting a block an unnecessarily uncomfortable task. Fortunately, once again RadRails comes to the rescue.

Whenever you need to comment out a whole block of code, you only have to select it and right-click on it. Now from the **Source** menu choose the **Toggle Comment** option. This will comment out your code with a single click. When you want to get the code back to normal, you only have to select it and repeat the operation. This functionality can be accessed also from the **Source** menu at the top of your workbench.

Code Completion

Code completion is another of the RadRails built-in capabilities that you cannot get in a plain editor or from the command line. By using code completion, RadRails will try to help you code as you type.

Let's try a basic example. Let's open our already famous Comics Controller and manually add a method named `completion_test`. Just write in your editor area:

```
def completion_test
```

After you hit *Enter*, you will see how RadRails automatically inserts a correctly indented `end` statement for closing the block. This will happen for every `if`, `for`, `while`, `case`, or `begin` statement, for example, as well as in any general case where a block of code needs an `end` (classes, modules).

Also, whenever you type a special character like a bracket, square bracket, quote, double-quote, RadRails will automatically insert the corresponding closing character.

This is starting to get interesting, but what would you say if I tell you that when you just start typing the name of a class, module, or variable, RadRails will present you with all the possible classes in scope that match the given name? Or type the name of any class, model, or object for RadRails to display a list of that object's methods, constants, and variables? Well, that's exactly what code completion can do for us.
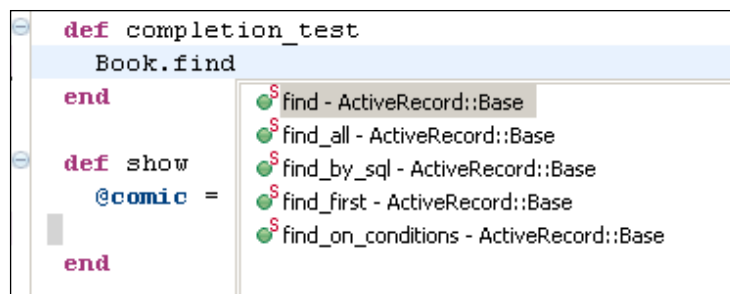
Notice that given the dynamic nature of the Ruby language, sometimes it's just impossible for RadRails to infer the type of a given variable. In those cases, it will present a generic list including all the globally accessible variables, constants, and methods. Take into account that, for this same reason, dynamically created methods will not be shown either.

To invoke code completion you only have to press the key combination *Ctrl+Space* —*Command+Space* for Mac users—any time when working in the editor area. If you invoke code completion while in the middle of a class or module name, it will present a list of the matching classes available. Let's try it out! We can go to the **completion_test** method we just created and type the characters **Bo** (notice the capital B) and then press *Ctrl+Space*.
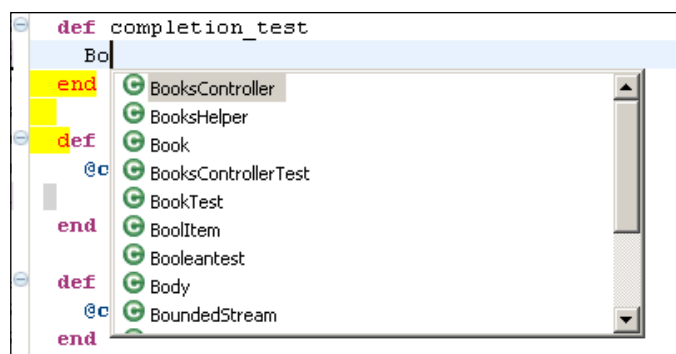
As you can see, a list of matching classes is displayed. One of these classes is the **Book** class we created in a previous chapter. Just select it and press *Enter*. Now let's type a period and the word **find**, so by now your line should read **Book.find**, and press *Ctrl+Space* again.

Now, RadRails presents the list of possible methods starting with the word **find** for the current object type. You can keep typing when the list is open and the candidates will filter themselves according to the text you type. The arguments for every method will be displayed too, as well as the documentation extracted from the RDoc for that method.

When there is a single candidate starting with the characters you typed, RadRails will not present a list for choosing but will directly insert the corresponding match in the editor area.

After selecting the method you are looking for, it will be inserted in the editor. If it has any parameters, they will be inserted too. You can use the *Tab* key for moving across the different arguments in a convenient way.

Code completion will work in exactly the same way for variables and constants' names too.

When completing code with the *Ctrl+Space* combination, sometimes you will see in the pop-up a list of options with a small icon representing a code sheet. These options are not really for code completion but for code templates, which we'll learn how to use right away.
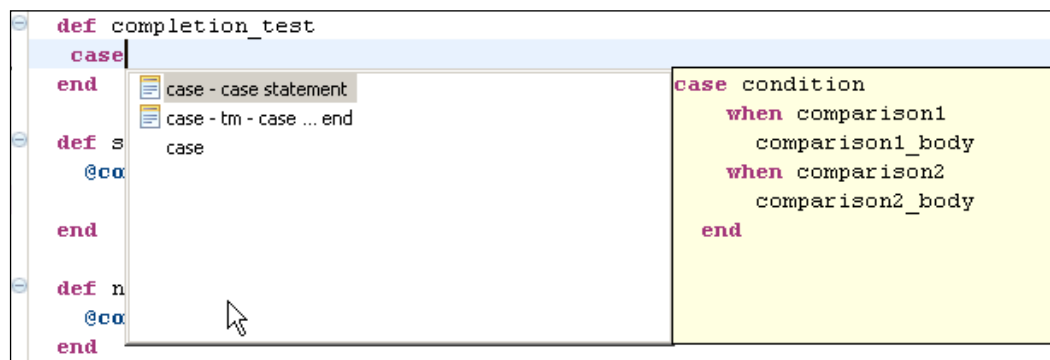
Code Templates

Code templates are another great feature RadRails provides. They are fragments of generic code that will be automatically inserted in the editor area in the same fashion as for code completion.

Each code template has a name, and by typing that name and then pressing *Ctrl+Space* the code snippet will be inserted. Moreover, the templates can define special 'replacement regions' that will be navigated by hitting the *Tab* key for the user to directly insert the dynamic part of the snippet. These regions work in the same way as we saw for method arguments, but they are not restricted to parameters, but can apply to any part of the code.

It will be easier to see with a simple example. Go to the **completion_test** method we defined in the previous section and type the word **case**. Now hit *Ctrl+Space*. A window with different options will be displayed.

If you move the cursor over the different options, you will see in a new window the code template that would get inserted. If you were getting many candidates in this pop-up, and you wanted to display only the code templates and not variables or method suggestions, you could click at the bottom of the pop-up, on the **Click to show Template Proposals** link. That way, you would be presented only with templates to choose from. For this example, just select the first candidate in the list.

A block of code with the structure for a **case** statement will be inserted. Moreover, the cursor will be placed in the first replacement region, which in this case is the one corresponding to the condition. By using the *Tab* key you can move to the other replacement regions defined. To move back, you can use *Shift+Tab*.

There are code templates for a large number of common statements: working with migrations, defining relationships in your models, validating fields, controlling the application flow, defining classes, iterating over collections, and rendering from your controllers. You can take a look at the complete list of defined templates by going to the **Window** menu, then selecting **Preferences**, and then **Ruby | Editor | Templates**.
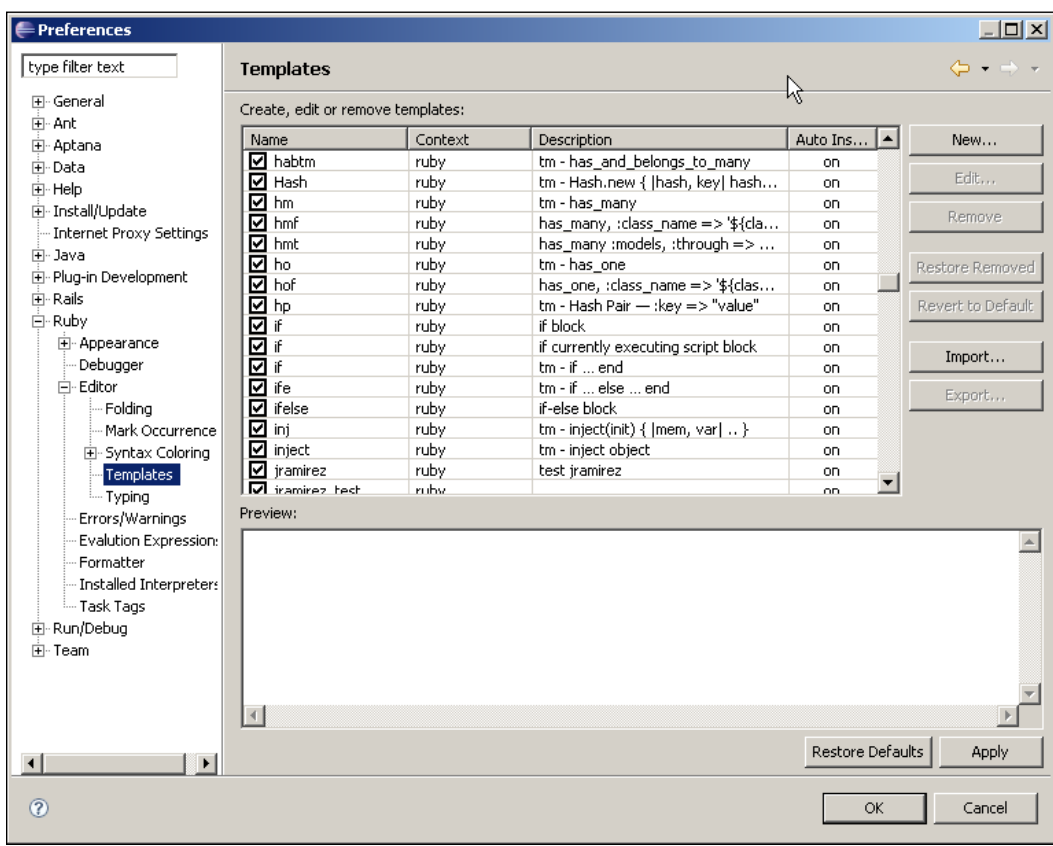
If you move through the list by selecting the different lines, you will see the template that will get inserted when using the given name. Don't get overwhelmed by the number of available templates, since you will probably use only a subset of them. The trick is to browse the template list and figure out which are most useful for you, and then increase the number of templates you use part by part.

If you have been working with the TextMate editor in the Mac OS, then you are lucky because all the supported templates in TextMate are also available in RadRails under the same key combinations.

Defining Your Own Code Templates

Even if the number of available templates in RadRails is impressive, there are always some operations that we repeat again and again in our code and it would be nice to be able to incorporate them as templates in our code.

Fortunately, the Eclipse platform provides an easy way to define your own templates so you can speed up your development.

A template in Eclipse is only a piece of code that will be directly inserted underneath the position of the cursor in the editor area. The only 'magic' going on when using a template is the cycling over the replacement sections.

When you are creating a new template, if you want to create a replacement section you only have to enclose the name of the section like this:

```
${name_of_the_replacement_section}
```

You can define as many sections as you need for your template. Imagine we want to construct a template for searching any ActiveRecord model by two different fields. We could define a template like this:

```
${ModelName}.find_all_by_${first_field}_and_${second_field}
```

When using this template, RadRails would allow us to introduce the `ModelName`, the `first_field`, and the `second_field`.

The section names are really variables for RadRails, so you cannot use spaces or any other special characters you wouldn't use when defining a variable. If you use the same variable name in different places, when you type a value in the first replacement section with that name it will be copied in the rest of the occurrences.

There are some reserved variable names with a special meaning for Eclipse:

* `${cursor}`: This variable is not used as a replacement value, but for positioning the cursor. When the code template gets inserted, the cursor will be automatically positioned in the place defined by this variable.
* `${time}`, `${date}`, and `${year}`: System time, date, and year respectively. Useful for comments.
* `${user}`: Name of the logged in user.
* `${dollar}`: Inserts a dollar sign.
* `${word_selection}`: Will be replaced by the selected word in the editor.
* `${line_selection}`: Will be replaced by the selected lines in the editor.
* `${class}`: Will insert the name of the surrounding class or module.

- ${classfqn}: Will be replaced by the fully qualified name of the surrounding type.
- ${method}: Will be replaced by the name of the surrounding method.
- ${methodfqn}: Will be replaced by the fully qualified name of the method.
- ${file}: Will be replaced by the current file base name.
- ${path}: Will be replaced by the relative path to current file from the project.

Suppose we want to create a template to surround a code block with `begin` and `end` statements. We also want the cursor to be positioned at the beginning of the block. We could write a template like:

```
begin
  ${cursor}
  ${line_selection}
end
```

To create the template, we need to open the **Preferences** dialog and go to **Ruby | Editor | Templates**. By selecting **New** we will be able to create a code template. We only have to write a name, an optional description, and the code of our template. Checking the box that says **Automatically insert** means that when hitting *Ctrl+Space* if there is only one template corresponding to the typed name, it will be directly inserted into the editor area. If you uncheck this option, even when there is only one candidate, a selection pop-up will appear.
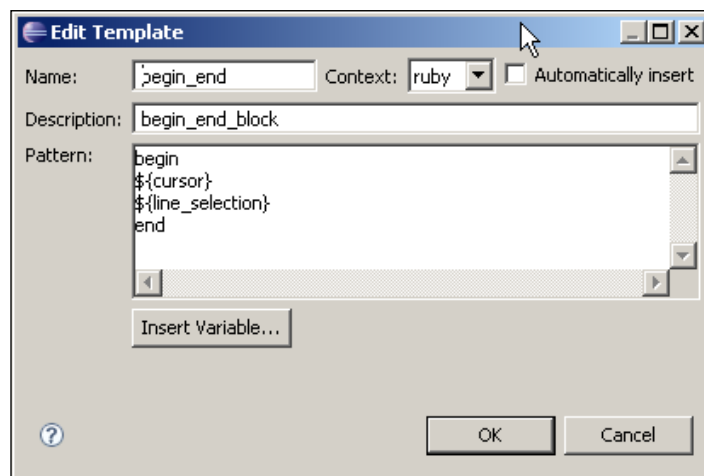
To insert the special variables, you can just type them in or you can select them from a list that will appear when you click the **Insert Variable...** button.

That's everything you need to know for creating your own templates or editing existing ones. For inspiration I would recommend you to take a look at some of the defined templates and see how they work.

Navigating Your Code

Apart from the outlining of the code and the code completion we have seen so far, RadRails provides more helpful tools when we are developing Ruby code. One of the most interesting features is the ability to navigate the source code in a comfortable way.

General Source Navigation Tools

Eclipse provides a couple of interesting options when we are working with a large number of source files, as is the case with many Rails projects because of their structure.

If you open the **Navigate** menu you will see the last three options are three arrows. The first one, named **Last Edit Location** will take you to the last position in which you entered any change to a source file.

The other two options **Back** and **Forward** work in a similar way to the back and forward buttons on an Internet browser, allowing you to move back and forth among the files you have been browsing.

You can also find these three options as icons placed in the toolbar at the top of your workbench.

Matching Brackets

Another basic tool when working with our source code is the ability to find the different matching brackets.

When working with RadRails, you can match the following elements in your Ruby source code: Class Definitions, Module Definitions, Method Definitions, Round Brackets, Square Brackets, Curly Brackets, and Block Statements.

As you are moving the text cursor along the code, whenever you pass over one of these matchable elements, you will see a small grayed square displaying over the matching element. RadRails will try its best to mark the matching element, but your code must be correct or otherwise it might be unable to find the corresponding bracket. In cases of incorrect nesting or missing opening or closing brackets, it will most likely fail to match the pairs.

If you not only want to see the match, but to move directly between the opening and closing element, you can go to the **Navigate** menu, then select **Go To** and finally **Matching Bracket**. Notice that you can also use a keyboard shortcut to launch this option.

Declarations of Classes, Modules, Methods, and Variables

In almost any piece of code, there are a lot of references to classes, variables, or methods that are defined elsewhere. The first and most basic help that RadRails provides is identifying where these definitions come from.

Open again the source code for our Comics Controller and go to the **show** method. Now hover your mouse cursor over the **Comic** class name. After a brief moment, a tool-tip with the name of the file in which this variable is defined will be displayed. You can hover over the **find** method name too. You will see now the tool-tip shows the name of the parent class defining the method as well as the method's signature.

This is already interesting enough since you can directly know where the different methods are defined and you could go and open those files to browse them. Fortunately, RadRails can also ease that task. If you hold down the *Ctrl* key and then click on top of a class or module name, a method, or a variable, RadRails will automatically open the file in which the declaration resides and will place the cursor exactly at the point of the declaration.

You can try it by holding the *Ctrl* key and clicking on the **find** method name. Now RadRails will open the source code for **ActiveRecord::Base**.

Notice that you can also use the shortcut *F3* for moving to the declaration and even if usually there is no difference between shortcuts and mouse actions, in this case using the keyboard instead of the mouse will save a bit of CPU time.

If you do the same for a variable, it will take you to the first line in which the variable is used in your code. You can also access this functionality by right-clicking on top of the element you want to navigate to and then selecting **Open Declaration**.

If RadRails cannot determine exactly which is the declaring class because of the dynamic nature of the language, a dialog will be shown with all the possible candidates so you can choose the one to which you want to navigate.
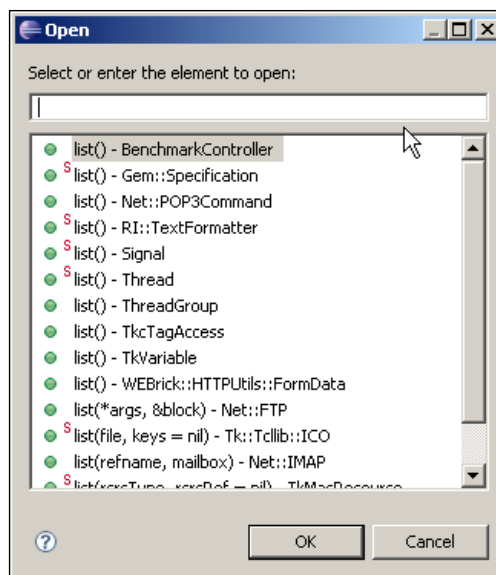
Navigating Your MVC Code

Rails follows the Model-View-Controller Pattern. Moreover, testing is a strongly encouraged practice when developing a Rails application. This separation of responsibilities in Rails means that the source code for a logical entity will be scattered through multiple files: a file for the model, a file for the controller, a file for the helper methods, several files for the views, and one more for the functional test.

Since the structure of a Rails project is always the same, it's fairly easy to know where each of these files is, but it's still a bit tedious having to navigate the Ruby Explorer in order to open them.

RadRails provides shortcuts for moving between all these related files in a simple way. If you open the **Navigate** menu you will see the options for quickly accessing these files. The options are **Switch to Helper**, **Switch to Test**, **Switch to Model**, **Switch to View**, and **Switch to Controller**. You can also find icons for navigating these files on the toolbar at the top of your workbench.

Navigating to Helpers, Tests, Models, or Controllers requires just a single click. When it comes to Views, things are just a bit different. Since for a single Entity you can potentially have several views, whenever you try to switch to a view, you will get a dialog asking you to which action view you want to navigate. A combo with the available options is displayed so you can select it directly.
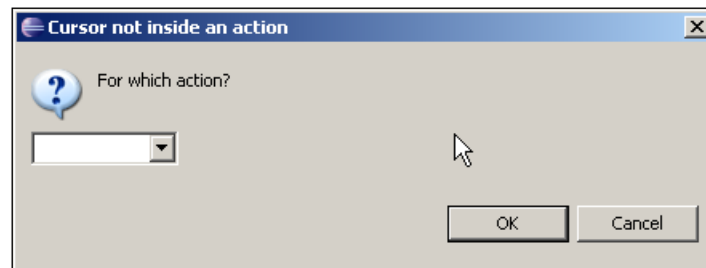
If you are navigating to a view from within an action in your controller, RadRails will try to find a view with the same name as the current action. If one is found, it will directly open that source file without asking.

Opening Types and Resources

Often you want to open a class or a file, but you don't remember exactly where it is located in the Ruby Explorer. Or maybe you know it but you want a quick way to open it directly.

In these cases, you can go to the **Navigate** menu and select **Open Type** or **Open Resource** depending on what you want to look for and how. If you want to look for a class or module and you know its name, then you would choose **Open Type**. If you want to look for a file, then you would choose **Open Resource**.

Even if the 'Type' and 'Resource' dialogs are slightly different, the way for searching in both of them is almost identical.

At the top of the pop-up you will see a text box. In this box you can enter the name of the Type or File you are looking for. As you are typing, the matching Types or Resources will be displayed for you to choose.

If you are not sure about the exact name, you can use the wildcards ***** and **?**, representing any string and any single character respectively.

The first difference between the two windows, apart from the object of search, is that in the Type dialog the path of the containing class is displayed by the side of the name, and in the Resource dialog the path for the file is displayed in the lower pane of the window.

The second difference is that in the **Open Type** dialog you can do CamelCase searches for types. Thus, if you write **AC::F** you will get matches for **ActionController::Filters** or **ActionController::Flash**.
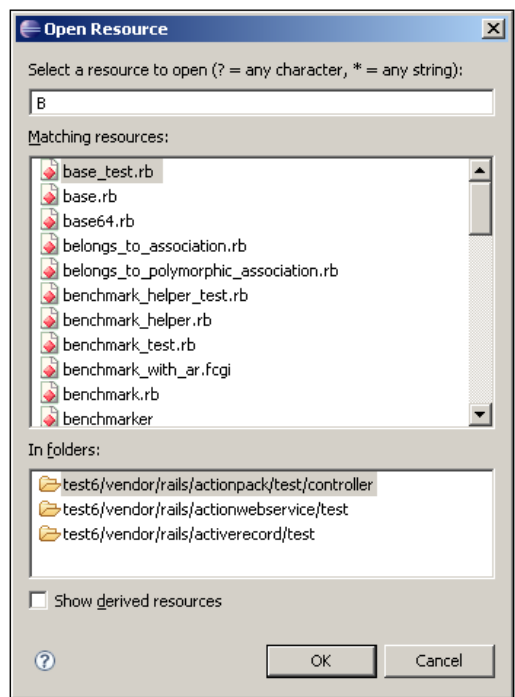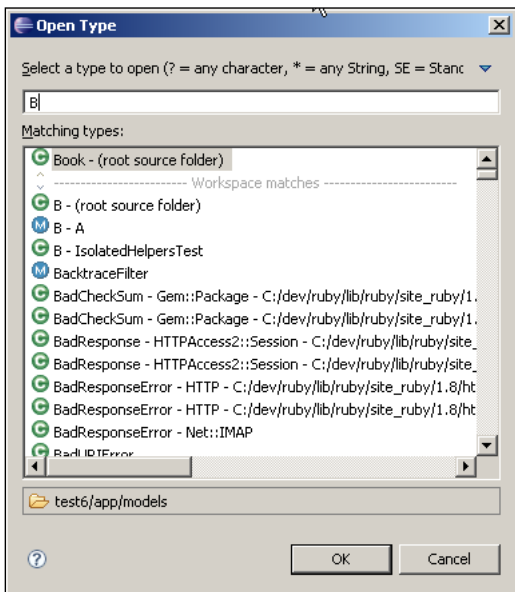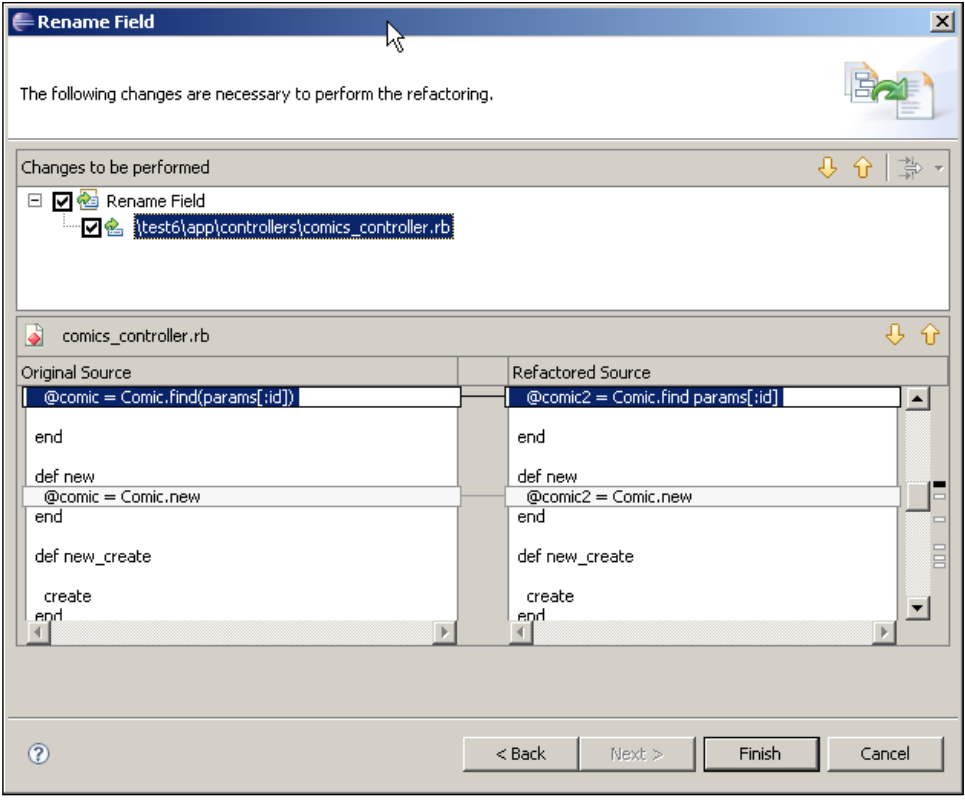
Refactoring

Refactoring is the term for making reference to changes made to our existing code in order to make it more generic or simply to making changes with a wide scope (renaming a method and all the references to the method, for example).

RadRails offers a very complete set of refactoring tools. Even if each of them covers a different functionality, the general way of using them is the same for all the refactoring operations. First we will have to select one of the available refactoring operations, then we will be presented with a preview to see which changes are going to be made to the code, and only if we accept the preview will the code be modified.

Since the dialog for previewing the changes is always the same, let's see an example to know what everything means before going through the details of the different refactoring options.

The preview dialog displays a file list on top with all the affected files, a left pane displaying the current contents before refactoring, and a right pane with the preview of the contents after refactoring.

When you select any of the files in the top pane of the window, the contents of that file are presented below in the 'before' and 'after' panes.

The differences between the two versions are displayed in the same way as when comparing two different files. You can use the up and down arrows to navigate to the next or previous change, or you can directly click on the rectangles by the side of the right pane representing the fragments of the code in which modifications will be introduced.

If you accept the changes, all the selected files will be changed accordingly. If you want to apply the changes only to some of the files, you can uncheck the boxes by the left of the names of the files to preserve.

Let's see the most interesting refactoring operations available in RadRails.

Generate Accessors

You can access this option from the **Source** menu or by right-clicking on the editor area and then selecting **Source**.

Generating accessors will create methods for getting and/or setting the values of your instance variables.
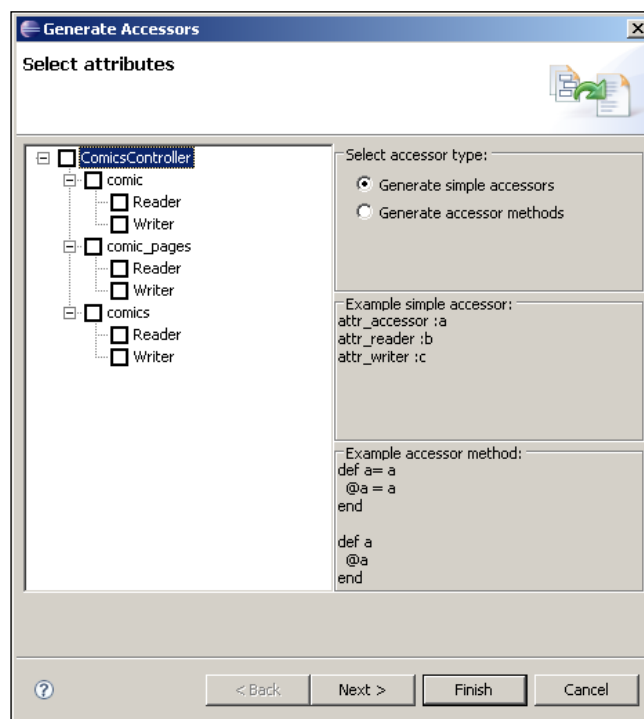
When you select this option, you will see a dialog with all the available instance variables for your current class. You can choose whether to create read or write methods and also which style to use: the simple one introduced by Rails in which you only declare the accessors in a single line, or the traditional Ruby-style one explicitly defining a method for each operation.

Generate Constructors

This option is available both from the **Source** menu and from the editor area's context menu under the option **Source**.

Generating a constructor will create an **initialize** method for your class using any of the fields (instance variables) you select. The only option available for this operation is deciding which fields will be used as a part of the method's signature.

Convert Local Variable to Field

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to one variable, the text cursor must be over the target variable when choosing this option.

Converting a local variable to a field will take an existing local variable and promote it to an instance or class variable. In the options dialog we can select, if we want, the variable defined at the class level and if we want to explicitly initialize it in the constructor of the class.

Encapsulate Field

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to one field (instance variable), the text cursor must be over the target field when choosing this option.

Encapsulating a field is just another name for creating the accessors for the field. In this case, we will have the option for making the accessors `public`, `protected`, or `private`.

Extract Method

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to a fragment of code, we will have to select the target of the refactoring when choosing this option.

Extracting a method will take a piece of code inside a method, create a new method from it, and replace the original fragment in the parent method with a call to the new one. If the extracted fragment was using any local variables, they will be passed as parameters to the new method. Also, if RadRails detects the fragment is setting a local variable, it will be used as the return value.

When extracting a method, we will have to provide at least a name for the new method. If the method was using variables and will need parameters, then we can arrange the name and order of them. We can also specify the visibility of the new method. You can choose between `public`, `private`, `protected`, or just leave the same visibility as the parent method.
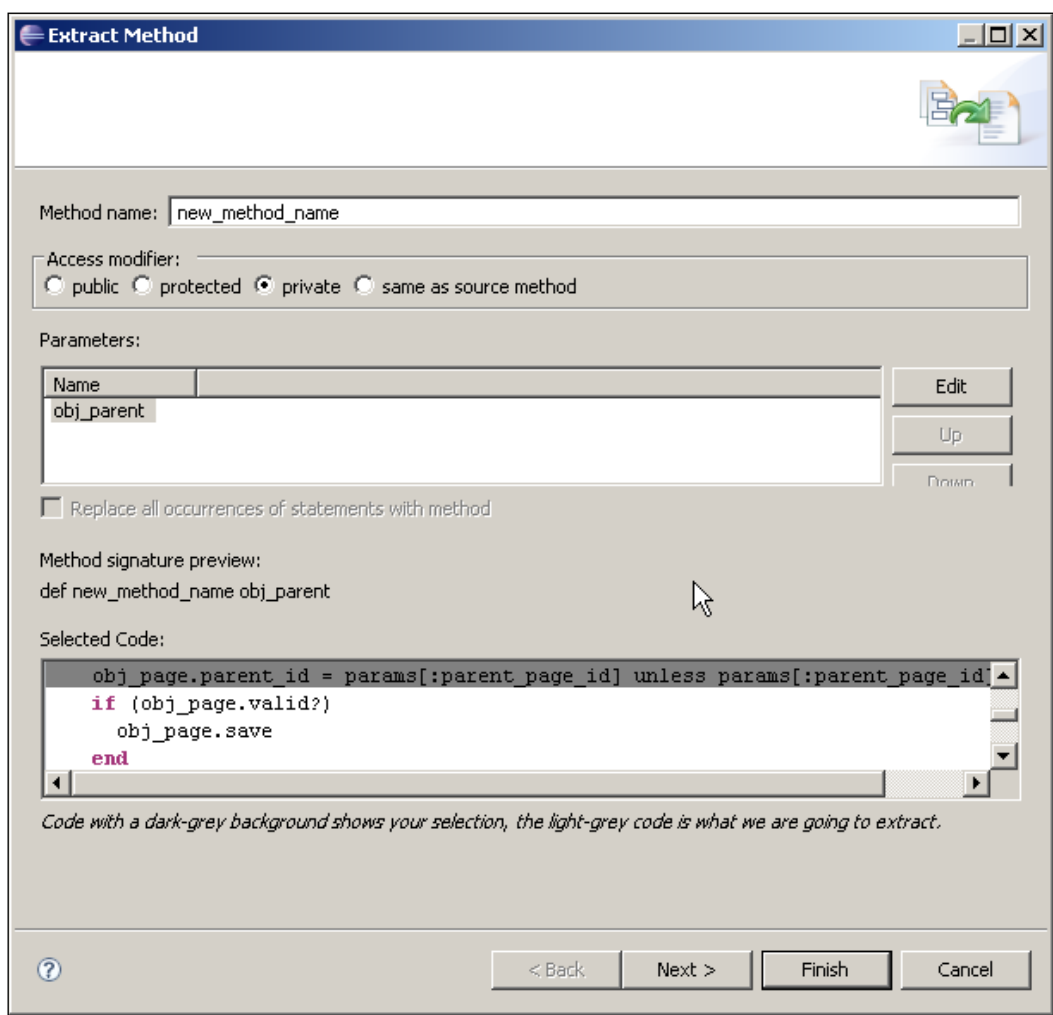
This refactoring tool comes in handy when you realize your methods are starting to be larger than they should and you want to make your classes a bit thinner.

Extract Constant

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to a String or Numeric literal, the text cursor in the Editor Area must be positioned on the literal before invoking this option.

Extracting a literal as a Constant will only ask for a name for the new constant. Then it will define the new constant assigning it the literal value, and it will replace the occurrences of the literal in the old code with the name of the constant.

Inline Method

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to a method definition, you will have to place the text cursor on the method name before invoking this option.

Inlining a method will replace the call to a method with the actual contents of the method itself. When we inline a method we are given the option to delete the original method. If we remove it, then this operation is exactly the opposite of extracting a method from another.

Usually inlining is done when we have very small methods with a single line of code.

Rename

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to a Class, Module, Field, Variable, or Method definition, you will have to place the text cursor on the corresponding name before invoking this option.

Renaming will only ask you for a new name and will preview the changes to the defining file and all the found references. Note that since Ruby is a dynamic language it can be difficult to track down all the references and you can end up with only a partial rename.

Split Local Variable

This option is available both from the **Refactor** menu and from the editor area's context menu under the option **Refactor**. Since the refactoring applies to a local variable, you will have to place the text cursor on the variable name before invoking this option.

Due to Ruby's dynamic nature, sometimes we abuse the local variables and we use them as temporary storage for keeping different types of values. This can potentially lead to highly coupled code and to a source difficult to read and maintain.

We can use RadRails for splitting a local variable. By splitting a variable, RadRails will create as many variables as assignments we are doing to the same variable. Let's imagine we have a source code like:

```
tmp_var='text_string'
puts tmp_var
tmp_var=10
puts tmp_var
```

In this case, after refactoring and splitting the variable we would obtain two separate variables: one for the first assignment and the first put, and a different one for the second.

When defining how to split a variable, RadRails will present each of the occurrences of the variable with a different background color, and it will also display a text box of each color for us to provide the names for the new variables.
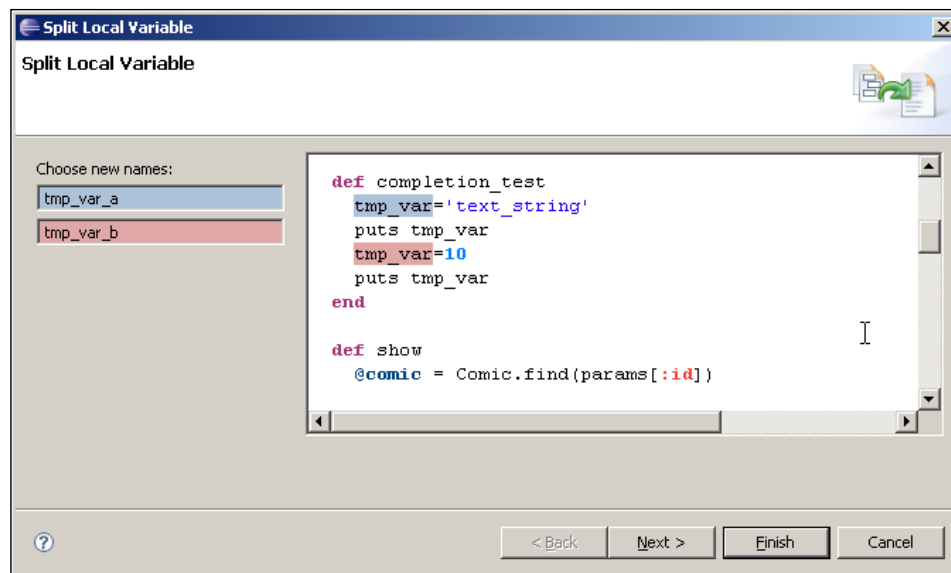
After previewing and accepting the changes, we will have a much more readable and maintainable code.

Searching in Ruby Projects

Searching for occurrences of a word in our projects is a very common task when programming and it is one feature of every single text editor, no matter how small. Apart from the general-purpose search features of Eclipse, RadRails provides interesting features related to searching our Ruby projects.

We will learn how to search text within the current file, how to search text across different files, and how to search for text within the Ruby context.

Searching within the Current File

The simplest search feature of Eclipse is the typical **Find** dialog. You can bring it up either from the **Edit** menu, in the option **Find** or with the familiar keyboard shortcut *Ctrl+F*.
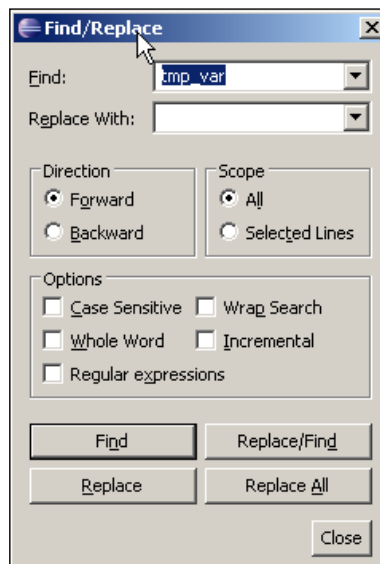
As you see, there is nothing too special about the **Find** dialog. You can search forward and backward, wrap your search so it will start from the beginning after reaching the end of the file, make the search case-sensitive or search only inside the current selection. Of course, you can also replace the found string with a new one.

There are only two features that require a brief explanation: Regular Expressions and Incremental Find.

Regular Expressions work by using a pattern to look for. As you probably know, you can compose fairly complex patterns by using regular expressions and the valid patterns can be slightly different from one programming language to another.

If you want to know which is the valid set of patterns for searching Eclipse, you can just select the checkbox for **Regular Expression** and then when you start typing on the **Find** box you will notice a small lightbulb close to it. If you hover over this symbol it will invite you to click *Ctrl+Space* for auto-completion of your pattern. When you press that key combination, a drop-down with all the valid patterns will be displayed. If you hover over the different patterns, you will see a little help explaining their meaning.

The incremental checkbox works by selecting the occurrences in your file as you type. If you mark the incremental check and then start typing, you will see how the matching words are marked in your file. As you enter more characters, the search will be more restrictive.

If you prefer, you can use incremental find at any time without having to open the **Find** dialog. The keyboard shortcut *Ctrl+J* will activate incremental find. If you prefer to search backwards incrementally, you can use *Ctrl+Shift+J*. Once you enter incremental search, you can exit and go back to normal edition mode by pressing *Esc*. Because of the special nature of Regular Expressions, incremental search can only be used with ordinary strings, without using Regular Expressions.

There is another built-in feature in RadRails that is not really a search but is closely related. Whenever you select a variable in your code, RadRails will automatically mark with a dark background all the occurrences of the same variable in your code. RadRails will also mark occurrences in the file when selecting a class or method name.

If you feel uncomfortable with this behavior, it can be turned off from the **Preferences** dialog, but it can be really handy when programming. It is specially useful in dynamic languages like Ruby, because you can visually detect misspelled variables or methods, which otherwise would fail on execution.

Searching across Multiple Files

Often you will need to find in which file a given string appears, or the occurrences of your search string throughout your whole project or even your workspace. In these cases, you cannot use the **Find** dialog, but once again Eclipse will help us by offering a specialized dialog.

You can access the **Search** functionality either from the **Search** menu and then selecting the **Search** option or by using the shortcut *Ctrl+H*. This will bring up the **Search** dialog.
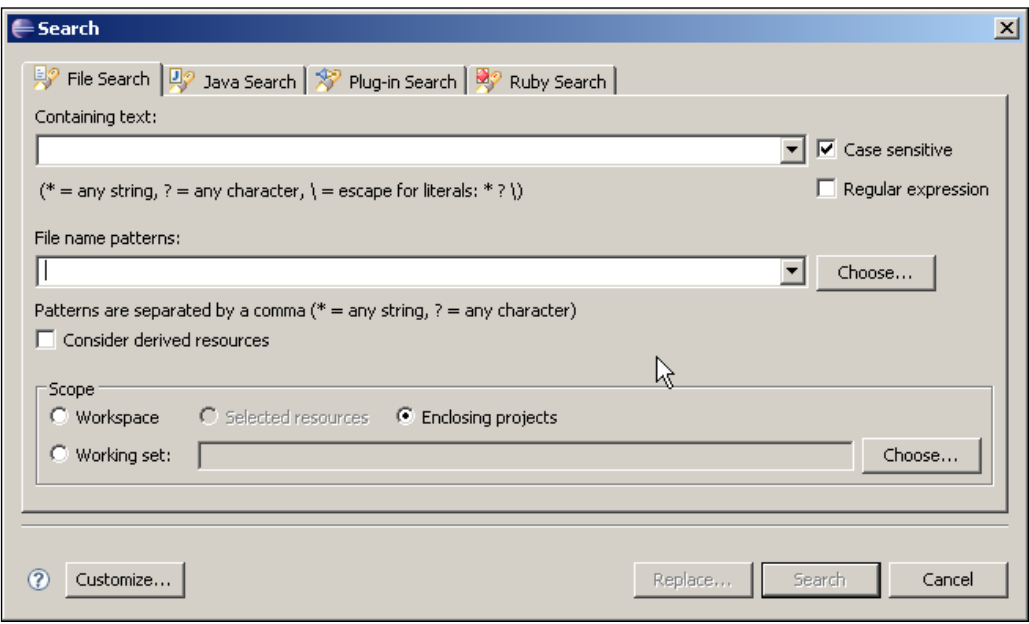
This dialog is also pretty easy to use. First you have the text box for the string to search for and as in the **Find** dialog you can make a case-sensitive search and use regular expressions.

If you want to restrict the type of files to include in the search, you can enter any file name patterns in the second text box. By limiting the files to search you will get faster results, specially in large projects. If you want to use several patterns you can separate them by a comma. For example, for searching only in files with `.rhtml` or `.rjs` extensions you would use the following pattern:

```
*.rhtml, *.rjs
```

You can also select the extensions of the files to search by using the **Choose** button. This is specially useful when you want to search all the extensions except a few ones. You can check all the known extensions with a single click and then just uncheck the ones you want to exclude.

The checkbox labeled **Consider derived resources** will not have any effects when working with Rails projects. Derived resources are files that are created as a result of actions such as compiling in languages like Java. Since in Ruby we don't generate derived resources, using this check will be pointless.

Finally, you can indicate a scope for your search. If you choose **Workspace**, it will search files across all the open projects in the whole workspace. If you choose **Enclosing Projects**, it will search across files within the current project in the Ruby Explorer, or if you select different projects, in all the selected. If you choose **Selected resources** it will search within the selected files and directories.

Depending on the current selection in your Ruby Explorer, some of the scopes will display grayed-out and will only be available when they make sense for the current selection.

There is another scope called **Working Set**. You can define your own working sets by using the **Choose** button. A working set is just a group of folders. You use working sets when you frequently want to search only in some folders.

By creating a working set you define a scope of folders to search and you can reuse it easily for future searches. When creating a working set search for working in Ruby/Rails projects you must choose the type **Resources** from the available options.

After entering the search string, the file pattern, and the search scope, when you press the **Search** button, Eclipse will start looking for the entered text in the target files.

As it finds any matches, it will display the file path and the number of matches in a new view at the bottom of your workbench. Just double-click on any of the found files to open it. The cursor will be placed by the first occurrence of the searched for string in that file.

If you prefer to navigate occurrence by occurrence instead of file by file, you can use the Up and Down Arrows (Find Next/Previous Match) in the toolbar of the Search Results view.

Ruby Search

RadRails offers a special search dialog for running searches related to Ruby elements. You can access this functionality either from the **Search** menu by selecting **Ruby**, or from the ordinary search dialog by selecting the **Ruby Search** tab.
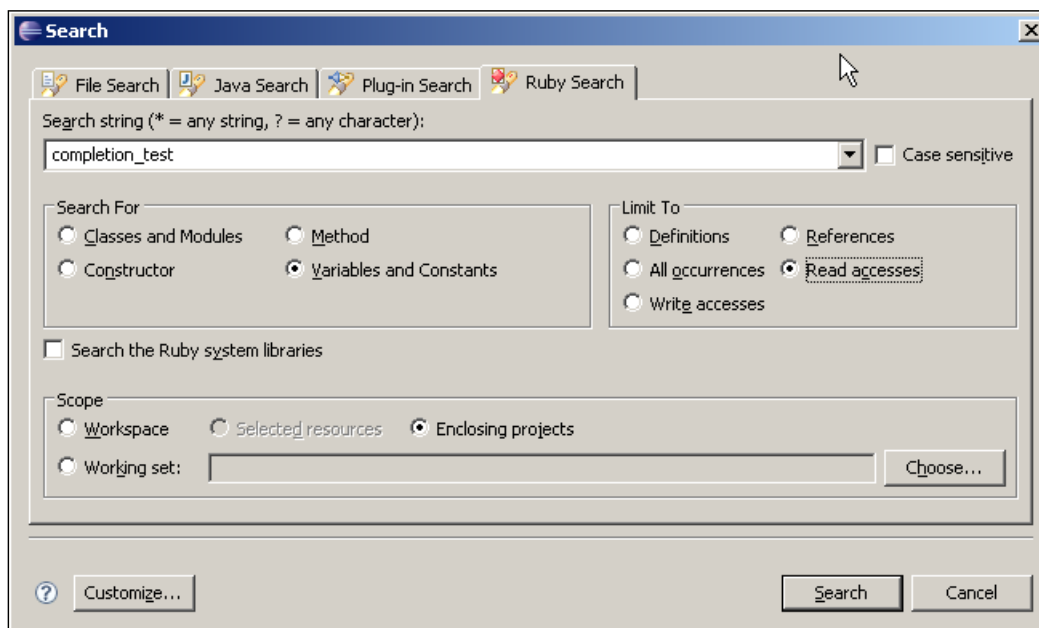
The text box and the scope for Ruby Search are identical to those we just saw in the previous section. The difference from the ordinary search is that when using the **Ruby Search** you must restrict your search to one of the following elements: Classes and Modules, Methods, Constructors, or Variables.

For any of these elements, you can instruct RadRails to search only in the definitions, in the references (anywhere but the definitions), or in both of them (by selecting **All Occurrences**).

When searching for variables, you can further restrict the search and look only for **Read access** or **Write access**. When searching for methods, you can use complex expressions like **ActiveRecord::Base.connection**.

Should you need it, you could also tell RadRails to search the **Ruby system libraries** by selecting the checkbox in this dialog.

A nice feature of the **Ruby Search** is that the results will be presented in a class tree way, rather than displaying only the file name, which would be the case when using the general search. Thus, you can get a better understanding of where the occurrences of the search string were found.

Call Hierarchy

The Call Hierarchy view is another feature closely related to searching for references to a given method. To display this view, we must select a method in the Ruby Explorer, the Outline view, or the source code itself, right-click, and select **Call Hierarchy**. You can also find this option in the **Navigate** menu under **Open Call Hierarchy**.

When you open this view, you will see two panes side by side. In the left pane, there is a tree in which the root is the name of the selected method. In this tree, you will find an entry for every method referencing the one we selected. If you click on the name of the calling method, you will see all the lines in which the method is called in the right pane of the view.
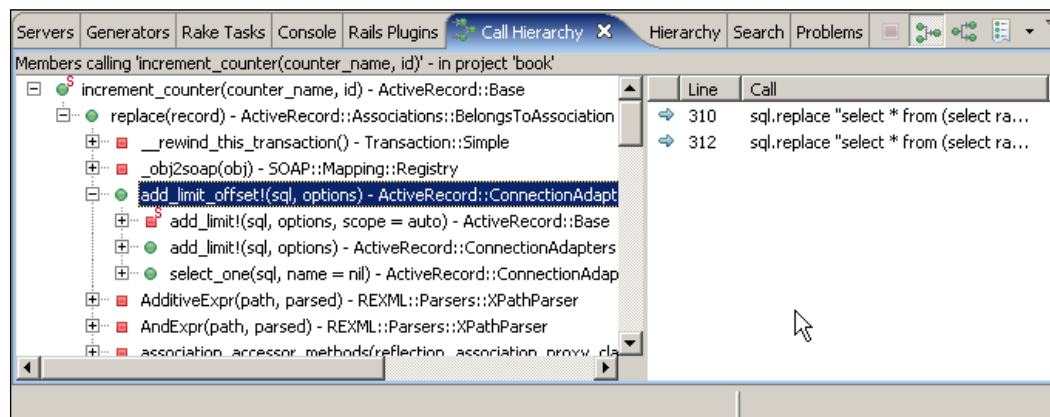
You will notice some of the referencing methods have a plus sign close to them. If you open any of these entries, you will find the methods referencing it. This way you can examine the whole call hierarchy, starting from your selected method and tracing it back to the very first method at the top of the invocation.

If you prefer, you can see the call hierarchy from the other side. You can see a tree in which the selected method is the origin and the different entries will represent the methods called by it. When opening the branches corresponding to these methods, you will recursively see the methods they call.

You can change the type of hierarchy to display by using the two tree-like icons at the top-right corner of this view. The first one is the **Caller Hierarchy** in which you can see the methods calling the selected one. The second one is the **Callee Hierarchy** in which you will see the list of methods that get called within the body of this method.

By using the context menu of this view, you can change the layout of the panes, the search scope, and apply filters. From these filters you can restrict how many levels (call depth) to display as a maximum in the hierarchy.

Please notice that even if this view is perfect for tracing the execution path of a given method, it can render inaccurate information in some cases, due to the dynamic features of the Ruby language.

Summary

In this chapter we learned how to use Eclipse and RadRails to assist us when writing code. Now we know how we can edit our code, browse its structure, format it, and navigate between the different classes and files.

Moreover, we saw how we can take advantage of code completion and templating, so we can program much faster and reduce the possibility of committing errors.

We can also use the refactoring tools of RadRails to modify our code in a comfortable way without having to track all the places where we must update the references to the changed code.

Finally we learned how to search for any text within our project and how to make the search more intelligent by restricting it to the Ruby elements we are interested in.

By now, you can already see how working with an IDE can really improve the speed and quality of your development. Once you master all the techniques we went over in this chapter, you will be able to focus on the creative process of coding, as you can delegate a big part of the mechanical process to RadRails.

5
Coding Rails Views

You already know that there are different kinds of views in Rails. When you are working with XML, you can use Builder templates (previously known as RXML) when working with AJAX, you can use RJS templates, and when you want to display any other content (most likely, but not only, HTML) you can use RHTML or ERB files.

Builder and RJS templates are composed of pure Ruby code, so when you open them in RadRails they will directly open with the Ruby Editor and you will be able to use all the features we discussed in Chapter 4.

When working with ERB or RHTML templates, you write mixed code. You typically write HTML and then introduce the dynamic part of your template between `<%` and `%>` or `<%=` and `%>`. Not only do you mix HTML and Ruby code, but you can also include JavaScript and less frequently, inline CSS code too.

RadRails provides a smart built-in editor for working with ERB/RHTML templates. This editor will try to assist you by giving you context-sensitive help depending if you are in the Ruby, HTML, CSS, or JavaScript part of a template.

By using RadRails, you will get help not only when working with your Rails files, but also when dealing with the static part of your application. Since RadRails runs on top of the Aptana Studio, you will get all the power of the built-in HTML, CSS, and JavaScript editors.

ERB/RHTML Templates

In a typical Rails application most of your views will fall under this category. You simply write a file with a `.rthml` extension (or `.erb` if working with Rails 2) and then you can create dynamic content by mixing in Rails code throughout the static parts of the view.

As you already know, it's good practice to keep your views as clean as possible, so you should take as much of your Rails code as you can out of your views to make them simple. Actually, Rails is a framework built around the Model-View-Controller pattern, so one of the premises for a well architected application is to separate presentation from logic. There are a number of good reasons for this, like for example sharing your views with the design team, or being able to reuse the same business logic to produce output in different formats.

In Rails, there are basically two ways of making your views cleaner: moving logic to the helpers and separating your views into partials. By moving logic to helpers, you get to reuse those functionalities you extract, and you replace all the code you write in your helper with a simple call from your view. And by separating your views into partials, you get simpler views without all the details and you can also reuse your partials in different parts of your application.

Views Navigation

In RadRails, navigating from the code of your view to the associated helper is a very easy task. You can directly click on the yellow **H** icon on your toolbar (it reads **Switch to helper** when you hover on it) or you can go to the **Navigate** menu and select the **Switch to helper** option.
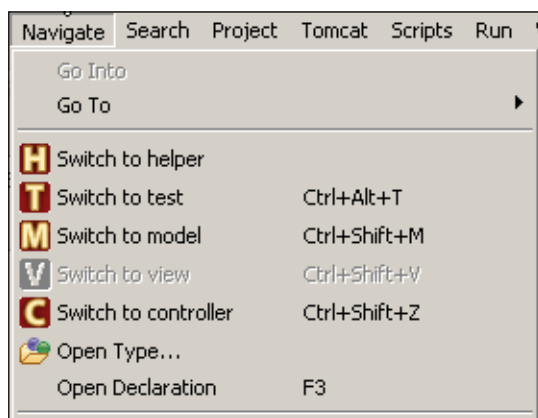
You can try that out by opening the view **list.html.erb** under the directory **app | views | comics**. If you use any of the methods above to navigate to the helper, you will be taken to the ComicsHelper, which is just an empty module.

What if you want to go back to your view? Well, you could always use the 'Back' and 'Forward' arrow icons of the default Eclipse toolbar (also available as options under the **Navigate** menu), but those work like the back and forward buttons of a web browser, allowing you to move between the last seen files. If you didn't open the view recently, this will not be available from the back/forward buttons.

As you noticed when using the **H** icon for switching to the helper, there are other similar icons too. With them you can navigate between the Helper, Unit Test, Model, View, and Controller.

When you want to navigate to the helper, unit test, model, or controller, there is no ambiguity because there is only one of those files associated to a given controller/ model. But due to the structure of Rails, there are many views associated. This means if you try to switch to your view from a Helper, for example, RadRails will not know what view you want to switch to.

Anyway, RadRails knows to which controller your helper is related, so it will present you with a list of all the actions in that controller. By selecting your action, you will be taken to the corresponding view. If the selected action doesn't have an associated view, RadRails will prompt you to create a new one.
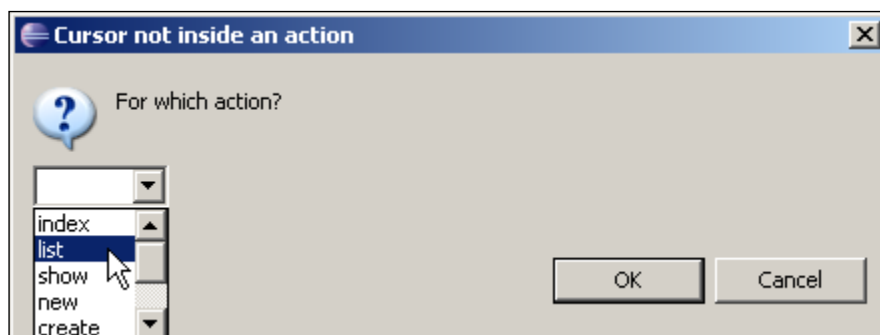
For example, you can switch to a view from the **V** icon or from the **Navigate** menu and, when the dialog opens, select the **list** action from the drop-down list.

As we saw, from your view you can switch to a test, model, or controller too. For switching back to the view, the behavior will be the same as with the helper. For the controllers it will be just a tad different. If your cursor is situated within the code of an action, you will be taken directly to the corresponding view. Only when the cursor is not within an action method you will be prompted with the action selection dialog.

View Templates

In Chapter 4, we saw how you can take advantage of code templates when writing Ruby code. RadRails offers the same kind of functionality when working with our views, but since the type of code you use in views has not much in common with that you use in your models or controllers, it provides a separate set of templates.

Besides, since when working in views we are mixing HTML with Ruby code snippets, the templates come in two flavors: those adding the special characters `<%` `%>` and those writing HTML code directly to your view.

To see the templates in action, we can go to the **list.html.erb** file and then in any blank part of the code you can write `<%` to start a Ruby code snippet. At this point, you will be presented with all the available templates. You can also directly hit the key combination *Ctrl+Space* (remember, if on a MAC, you have to use the *Command* key instead of *Ctrl*) to get a list of available templates without starting a code snippet, but in that case you will be presented also with pure HTML suggestions apart from the available templates list.
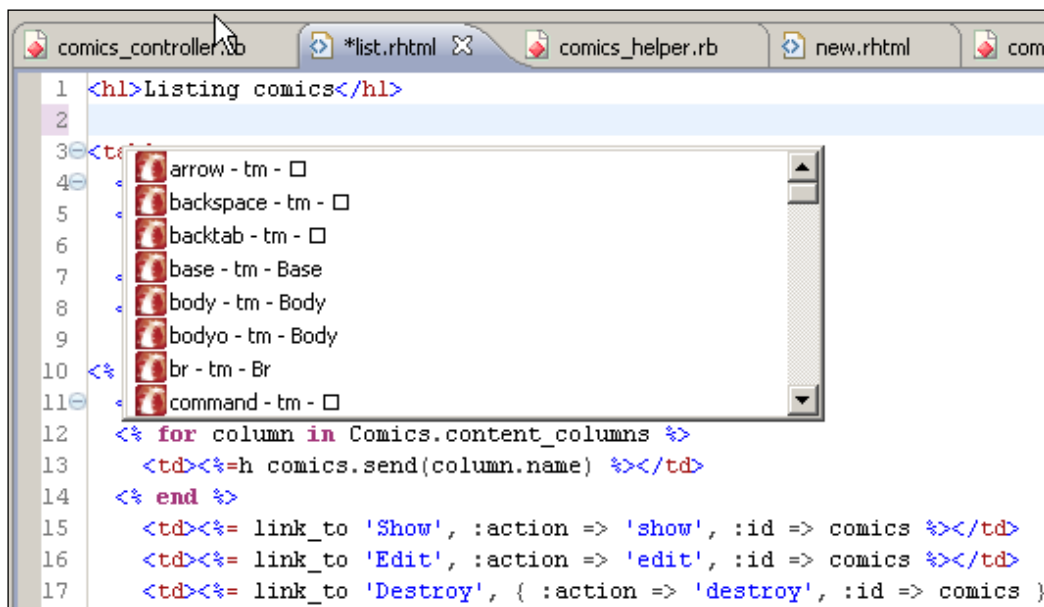
As you can see, many of these templates just represent HTML snippets or even HTML escaped entities, and then there are others providing actual Ruby code.

If you want to check the complete list of available templates and take a look at what they do, you can go to the **Window** menu and open the **Preferences** dialog. From here, navigate to **Aptana | Editors | ERB/RHTML | RHTML Templates**. This will display all the templates together with the associated code for each of them.
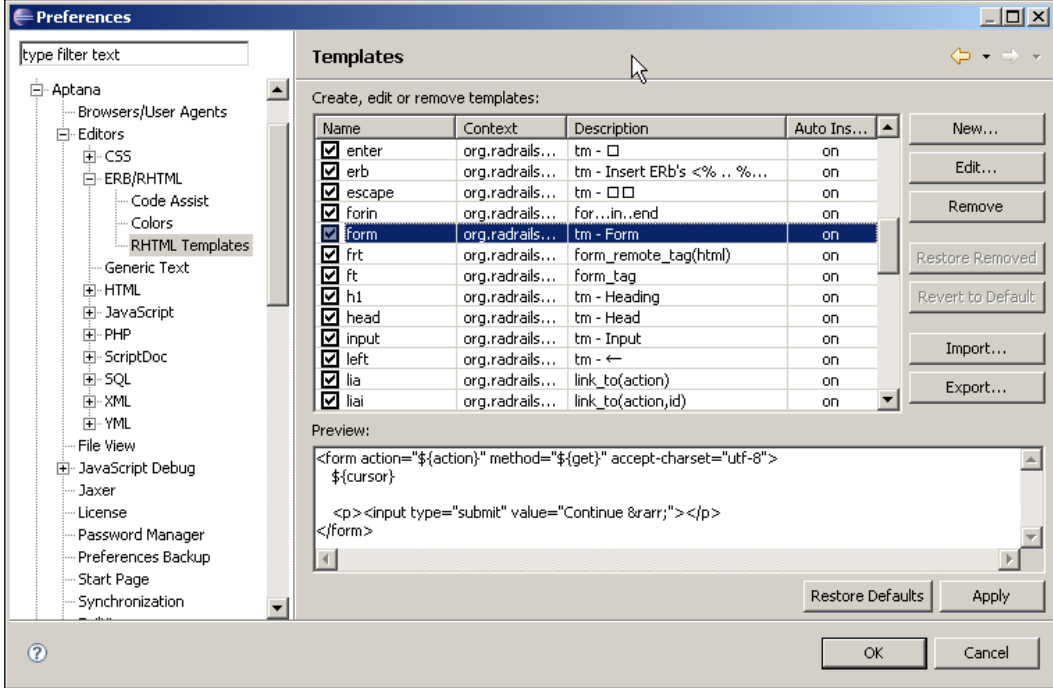
If you want to modify any of the provided templates or if you want to create your own, you only have to follow the instructions we gave in Chapter 4 for creating Ruby templates. The only difference is the templates you create under this option will be available only when working in the code of your views, and the templates created from the Ruby Editor option will be available at any other time.

HTML Code Assist

In a Rails view, we don't only write Ruby code but, more frequently, we write HTML. Thus, it would seem just logical that RadRails provided some kind of code assist for such an important part of our views too.

Since RadRails RHTML editor works on top of the Aptana built-in editors, we can use all the features they provide, which by the way are pretty impressive.

In the same file we were editing, try typing a **<** symbol and then hitting *Ctrl+Space* (in the default configuration, the help will display directly, even if you don't use *Ctrl+Space*). You should see a list of all the available HTML elements.
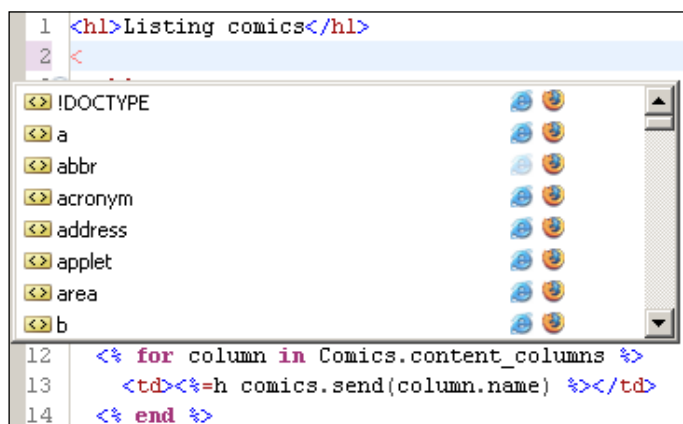
When you choose one of these elements, RadRails will automatically insert the matching pair to make sure you don't forget to close it. However, if you delete the closing tag, the next time you write **</** in your code, the editor will automatically suggest the appropriate closing tag.

Once you are inside a tag, you can hit *Ctrl+Space* again to get a list of the available attributes for that tag.

Another nice feature we get when working with views is the discovering of matching tags. If you move the cursor to one opening or closing tag, RadRails will mark with a rectangle both the opening and closing tags so you can quickly see the contents of your element.

Just take into account HTML code assist can only work properly when your HTML code is well formed. If you are nesting tags in a wrong order or if you are closing more tags than you opened, then the editor might be unable to provide suggestions for you, or the suggestions might be inaccurate.

Refactoring into Partials

As we saw in Chapter 4, refactoring can be a big help when working with the code we have already written. We usually refactor when we want to reuse our code in different places or just to clean the code and make it more readable and maintainable.

Since we shouldn't be creating any methods in our views, it doesn't make sense to have available all the options we can use when refactoring pure Ruby code, but it doesn't mean there is nothing to refactor in our views.

One of the most typical refactoring task for a view is separate a big file into different partials. We can do this by using the **Extract Partial** feature of RadRails. First you have to select the lines you want to extract to a new partial and with the selection on, you can right-click and select the option **Extract Partial** in the context menu. This option is also available under the **Edit** menu (notice it will be available at the Edit menu only when editing a view).
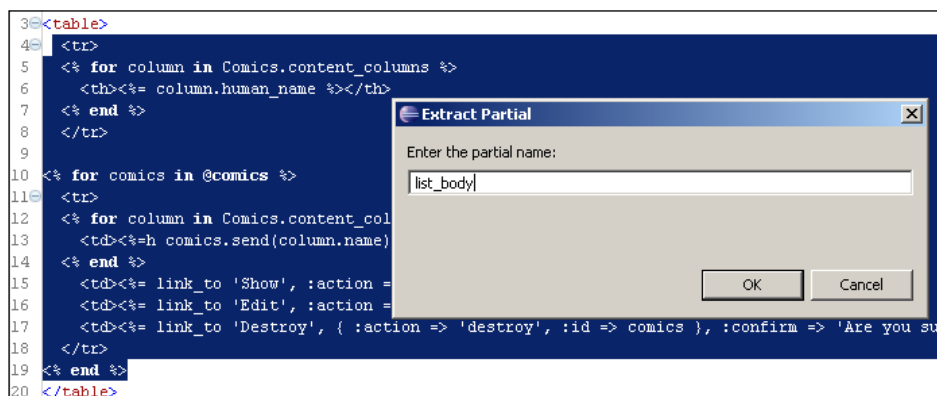
A dialog asking for the name of the partial will be presented. You can write the name with or without the preceding '_' character. Please note that if you write the name of an existing partial, RadRails will not refactor the code, but will inform you that there is already a partial using that name.

Once the extraction is done, you will notice two things. First, the selection in your original view will have been replaced with a 'render :partial' statement. Second, there should be a new partial file in the same folder as your original view with the name you provided and the contents of your selection.

There is one point to be aware of. Unlike what happened when refactoring methods in Ruby code, at the moment of writing this book, RadRails will extract your selection to a new partial without taking into consideration any variables you are using.

What this means for you is that if you are using any local variables in your partial that were defined in the calling view, you will need to add them manually to the "locals" Hash in the render statement.

Outline

If I tell you about the outline of a view, maybe you will think it doesn't make sense. Since we are not going to have methods, there would be nothing to outline. That would be true if we think of outlining the Ruby code, but since we already stated our views should be as clean as possible regarding the Ruby code inside, the outlining we have has to do with the structure of the HTML.

If you are using the default layout for the Rails perspective, whenever you open a Rails view you will see the layout of the HTML structure in the **Outline** view. If you cannot see the **Outline** view, you can make it visible from the **Window** menu, under **Show View** and then in the **General** category.
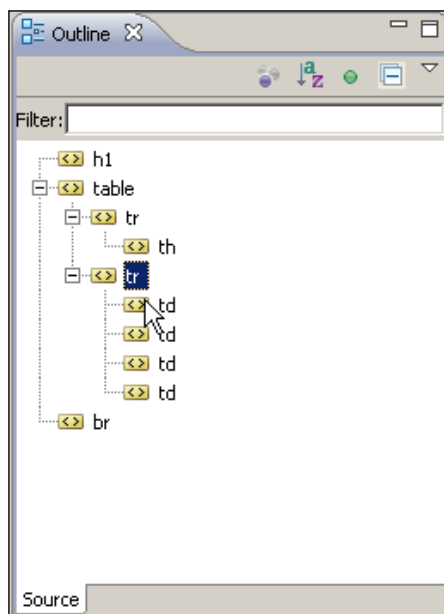
With the **list.html.erb** file open, you will see the outline representing the header of the document, the table and then a `<br/>` element. If you click on the table element, you will see the detail of each row inside.

Notice the editor will provide code folding/unfolding icons matching those elements in your code that are also expandable in the outline view. In our case, we can fold the table and the `<tr>` elements.

This kind of outlining is specially useful when we are integrating our views with the HTML a designer provided or if we are working with big files. Even if the indentation is not right (which very often is the case when in a RHTML view) the outline will allow us to navigate our code easily.

For those cases when the code is not properly formatted, we can make Eclipse format our source code automatically. You can right-click on the editor and select **Format**. This option is also available under the **Edit** menu.

Editing HTML Files

It's not so usual to edit plain HTML files when working in a Rails application, but sometimes you just want to provide a static page for displaying server errors or maybe for displaying during a controlled downtime of your server for maintenance purposes.

When editing HTML files, you will have available the same functionality regarding code assisting, formatting, and outlining as we just saw for your RHTML views. There are only some small differences.

The first difference is that since we are talking about plain HTML files, there will not be templates available. You can still type **<** to get the list of the available HTML elements and then use *Ctrl+Space* for the valid attributes, but there are no templates with snippets of code inside.

The other difference we find when editing a plain HTML file, is that since we are talking about pure HTML, Eclipse will be able to validate our code. To do so, we only have to right-click on the editor or on the file name and select **Validate**.

Once validation is done, the possible errors and warnings will be marked with **red** or **yellow** icons by the left margin of the editor. If we hover over these icons, we will get the explanation of the validation problems found.

Editing JavaScript Files

Even if the Rails helpers and RJS views make dealing with JavaScript an easy task, there are many times when we have to write our own JavaScript files, most typically in the `application.js` file under the **Public | JavaScripts** directory.

Aptana provides a very convenient help when working with our JavaScript files. To try out this editor, we can, for example open the `dragdrop.js` file, which is always available under **Public | Javascripts**.

When you open it, the first thing you notice is the **Outline** view. Variables will be represented with a small green circle, objects will be represented with a bracket, and functions will be represented with a green circle with the letter **F** inside. If there are variables defined inside the functions, they will be represented in the outline too.
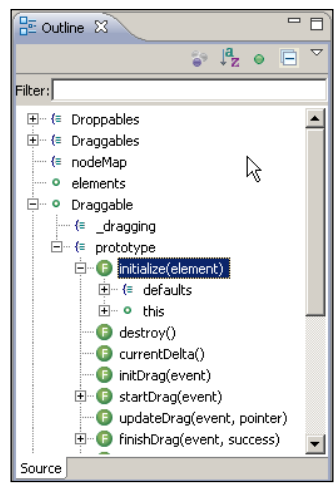
As usual, we can navigate the code directly by selecting the different elements in the outline and the cursor will be positioned directly at the point in which the selected element is defined in the editor. You can also use code folding from the left margin of the editor view. Also from the editor you can click on any variable or function name and you will go directly to the declaration.

You have to note that JavaScript is a dynamic language, so it's not always clear where the declaration of a function or variable is. In those cases, the editor will not be able to navigate to the definition. This happens specially if we are writing very complex JavaScript, like for example the source of the Prototype library itself. For the kind of JavaScript you'd usually write in your `application.js` file, RadRails should be able to find your declarations without any problems.

Apart from navigating to your declarations, there is some code assist available for JavaScript too. If you hit *Ctrl+Space* you will be presented with a list of all the JavaScript keywords and functions, as well as with the variables you defined in your JavaScript.
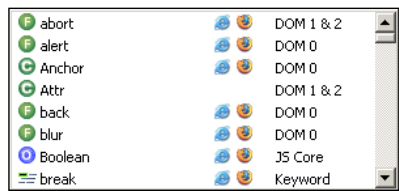
By default, the editor will be able to analyze the current file and all the JavaScript base files that are created initially in a Rails project, so if you are using objects or functions defined in a separate JavaScript file, the editor will not be able to open the declaration.

Aptana provides the means to include more files to search for our definitions when using code assist. To use this functionality we have to open one of the Aptana views that are not displayed by default in the RadRails perspective. Go to **Window | Show View**, and then select **Other**. From the **Aptana Standard Views** category, select the **References** view.
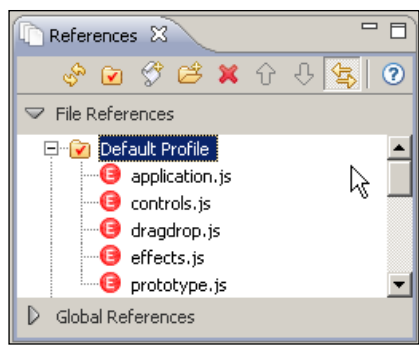
Once the **References** view is open, you will see that the default set of JavaScript files used by Rails are already included. That's the reason why you can get code assist for functions defined in the `prototype.js` library, such as **$** or **$$** when working in any of your JavaScript files.
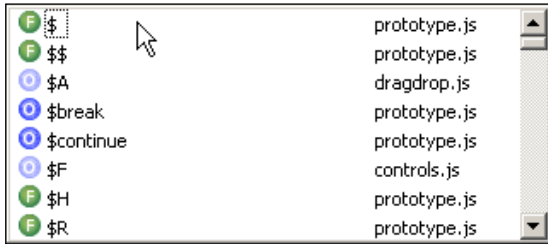
You can see in the figure above, when including more than one JavaScript file, code assist will indicate which file contains the definitions of the displayed elements. If you want to add a JavaScript file to the ones being examined for code assist suggestions, all you have to do is drag your JavaScript file from the Ruby Explorer to the **Code Assist Profile** view.

Finally, when working with JavaScript, Aptana will provide code assist and auto-completion for your variables if the type of the variable is clear for the editor.

For example, if we define a JavaScript variable and we assign a string literal to it, then we can type the name of the variable later in the code and, after we write a dot, the editor will display the functions available for the Objects of type String in the JavaScript API.
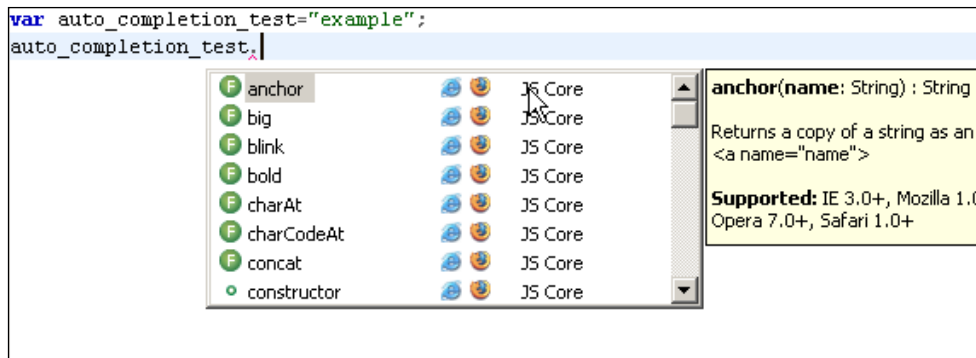
There is one point we didn't mention before. If you are working in an RHTML view, you know you can write JavaScript code inside by using a `<script>` block. When you are inside of those blocks, RadRails will provide the same content-assist features as in stand-alone JavaScript files. However, I recommend you that try to keep your JavaScript separated from your views whenever possible, since it will make them cleaner and more maintainable and easier to reuse.

Editing CSS Files

There is no modern web application without at least one CSS file, and more often than not we have several of them. Aptana provides a specialized editor that will help us write CSS in an easy way.

In order to try this, we can go to the **Public | Scaffold** directory of our application and open the `scaffold.css` file we generated in Chapter 3, when using the Scaffold Generator.

Again, you will notice there is an **Outline** view displaying the structure of our stylesheet. The different elements are represented with an open bracket, and the properties defined inside are represented with a green circle labeled with the property name and the assigned values.
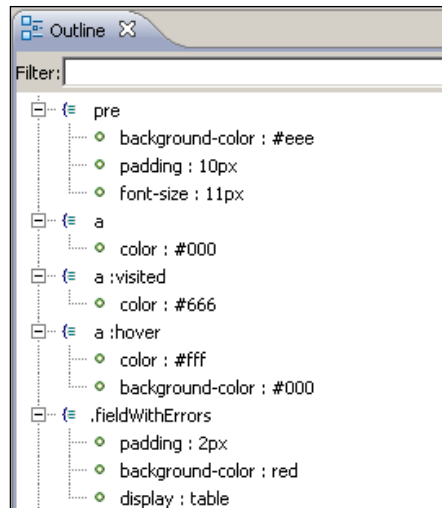
As with the other outlines, you can navigate your code directly from the tree in this view. You can see code folding is also available from the left margin of your editor.

To get code assist in this editor, you have to type a letter and then *Ctrl+Space*. Try typing **d** and then *Ctrl+Space*.
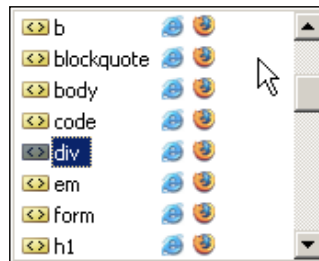
At this point, you will see a list of all the available elements accepting CSS styles. The cursor will be set in the first element starting with the letter you typed.

Once inside an element definition block, you can repeat the operation of writing a letter (or the beginning of a word) and then hitting *Ctrl+Space* and you will be presented with all the available properties.

When you are filling the values for a given property, Aptana will check if the values make sense. For example, you cannot write `font-family: 15px` or `font-size: verdana`. The editor will raise an error if you try to do so.

There is still another interesting feature when using code assist in your CSS. Actually the same feature is also present for the HTML and JavaScript code assist, but I find it the most useful when coding CSS. When you start typing and get a code assist pop-up, you can see some icons representing the IE or Firefox browser by the right of the suggestions.

This icons indicate that this property is compatible with such browsers. If you scroll up and down the list of suggestions with your cursors, you will see a tool-tip explaining in detail in which versions of the browsers the property is available.
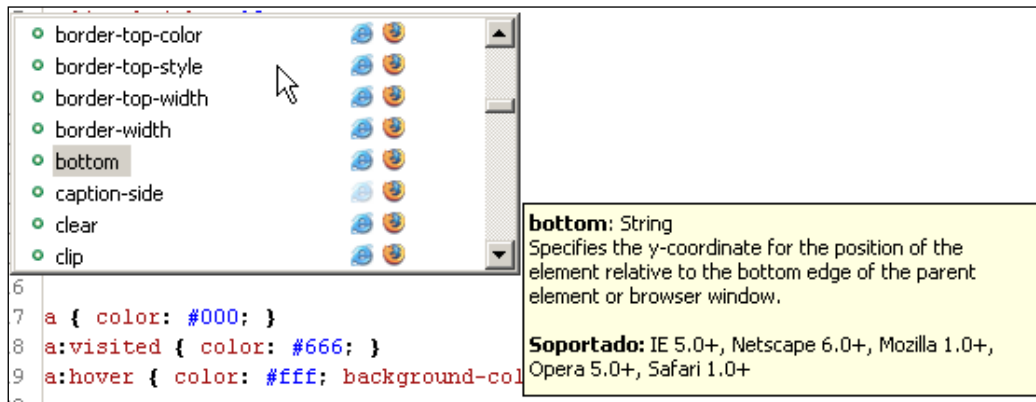
If you want to display the quick reference icons for browsers other than IE or Firefox you can do so by going to the **Window** menu, opening the **Preferences** dialog, and Navigating to **Aptana** | **Browsers | User Agents**. Code assist can provide browser-sensitive help for IE, Firefox, Opera, Netscape, and Safari.

As it happened with JavaScript, if you write your styles directly in your RHTML views using the `<style>` tag, you will get all the code assist features directly in your views, but I would strongly recommend you not to define inline styles in your application if you can help it.

Recent versions of Aptana incorporate a pretty handy feature. At the bottom of the editor, you will notice a tab displaying **Firefox Preview**. Depending on your OS, more tabs might be shown (such as for IE Preview). If you click on the tab, you will see a preview of a predefined template applying the selected CSS. To get back to the source file of your CSS, just click on the **Source** tab.
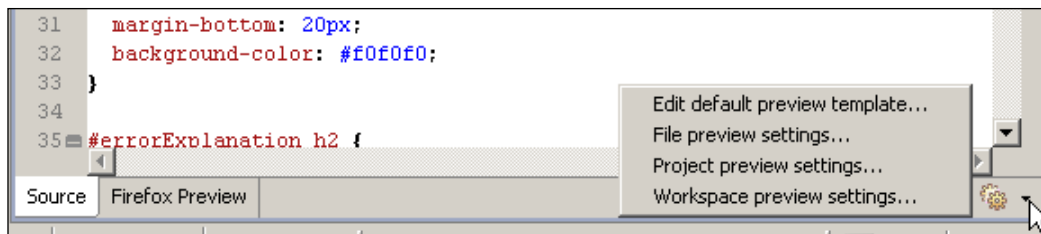
The preview template uses pretty simple HTML with generic elements like `h1`, `h2`, and `div`. If you want to customize the template being used for the preview to make it look more similar to one of the pages in your project, you can do so by opening the menu by the right-bottom corner of this editor and selecting **Edit default preview template...**.

Notice when you preview a CSS, the preview is done directly over the file, not using your Rails server. This means if you are using any images in your CSS, they should use relative paths. Otherwise, they will not be displayed in the preview because there is no way of setting the document root for absolute paths. In a typical Rails layout, the images in the CSS should use a path like `url(../images/image_file_name)`.

Summary

In this chapter, we saw how to use Aptana RadRails for writing all the necessary code for the views of our application. RadRails not only offers help for writing the RHTML templates, but also provides assistance for coding HTML, CSS, and JavaScript files thanks to the underlying Aptana plugin.

By using the code assist features, we will be able to write code more quickly and more easily than if we were using a plain editor. Besides, the built-in browser-sensitive code assist will help us deal with the most popular browsers in the market, which is always a big help when working on the client-side of our application.

6
Debugging Your Application

Some people say that in Rails you don't need a debugger since you already have tests. Nevertheless, there has always been a built-in debugger in Rails, and the core team has dedicated some effort to get a better debugging experience in version 2.0 of the framework by using the fantastic Ruby-debug gem developed by Kent Sibilev. Moreover, one of the points made by David Heinemeier Hansson in the announcement of Rails 2.0 was the possibility of using the debugger even in your tests. So it would seem it makes sense to have both a good testing framework and a good debugger in Rails.

The state of the art in debugging Ruby code a while ago was pretty primitive, so I can understand Rails supporters tried to play down the importance of a good debugger, since there was not a good tool that could stand comparison with those of other languages.

Even today, working directly with the built-in Rails debugger feels a bit odd, since you have to instrument your code by adding breakpoint statements before you can use it or set breakpoints from the command line referencing source line numbers directly and then you have to walk through your code with one-word commands with cryptic one or two-letter abbreviations.

Ruby-debug is a really fast and powerful tool, but it feels too much like the good old times of `dbx`, or even `debug` and `edlin` under DOS, much fun, but a slow process. And we shouldn't forget one of the main reasons we are using Rails is productivity. Of course I recommend you to learn how to use ruby-debug, the same as with the rest of the command-line tools, but most of the time I prefer to use the graphical front end and save myself a lot of time.

I personally cannot think of developing a real-life modern application in Rails or any other environment, without a good debugger. During my first Rails projects, in which such a tool was unavailable, I spent countless hours and countless log entries tracing the execution of my applications. Sometimes it really felt like going back to development in BASIC.

Fortunately, those times are past. The command-line debug tools are already a huge step forward, but the real power comes when those debuggers get integrated into an IDE so you can walk through your code, inspect, and modify variables in execution time, and set or remove breakpoints with a single click, basically making possible a debugging experience as smooth as you would expect in any other programming language.

I'm pretty sure you have guessed by now that RadRails provides a good Ruby debugger. In former versions of the IDE, the debugger was focused on pure Ruby, which made it difficult, but possible, after some hackish configuration, to debug a Rails application. Since a few versions ago, RadRails has provided an excellent debugger, which allows you to debug your Rails applications out of the box.

After I started using this debugger in my everyday development, I had almost forgotten about the logger object and the log file, and I found myself solving problems more quickly and comfortably than before, especially, when I was working with a piece of code with which I was not familiar, such as a plug-in, Rails internal code, or code written by my co-workers.

Here is my two cents' worth: keep writing your tests, but don't underestimate the value of debugging. It will make you more productive and it will help you grasp the internals of the modules you are using.

Getting Started with Debugging

Debugging in RadRails has evolved much since previous versions, so if you had some prior experience with RadRails and found it kind of buggy, give it a second chance because now it's really worthwhile.

The process of debugging is so easy now that you actually don't need to do anything special. You only have to launch your server or your stand-alone script in debug mode and use it right away.

Debugger Configuration

Every Ruby installation will have the `debug.rb` library available, allowing you to debug a Ruby script, set breakpoints, and examine the value of your variables. Unfortunately, this library has some issues with performance basically due to the fact of creating a lot of binding objects that are almost immediately collected by the garbage collector, resulting in really slow execution for complex scripts (such as the Rails framework itself).

If you want to get the most out of the RadRails debugger, it's recommended to install the ruby-debug-ide gem, which depends on the much faster ruby-debug-base gem. Without this gem, the debugger will still work, but much more slowly since it will be using the standard `debug.rb` library.

You could install the ruby-debug-ide gem directly from RadRails, but we will not be learning how to do so until next chapter, so for now we can install directly from the command line. Open a command prompt and type:

```
gem install ruby-debug-ide
```

Please note that if you are under a linux-based system, you will need to execute as a privileged user or use the **sudo** command.

After checking your installation, connecting to the gem server, and installing it, it should display a message similar to **Successfully installed ruby-debug-ide-#VERSION**. If you don't have the ruby-debug-base gem installed, you will be prompted to install it as a part of the installation process.

RadRails will try to auto-configure itself for using ruby-debug if it is present in the system, but sometimes it will not be able to detect the installation, so it doesn't hurt to check the preferences and make ourselves sure we will be using ruby-debug. Go to the **Window** menu, then open **Preferences** and look for the **Debugger** option under the **Ruby** category.
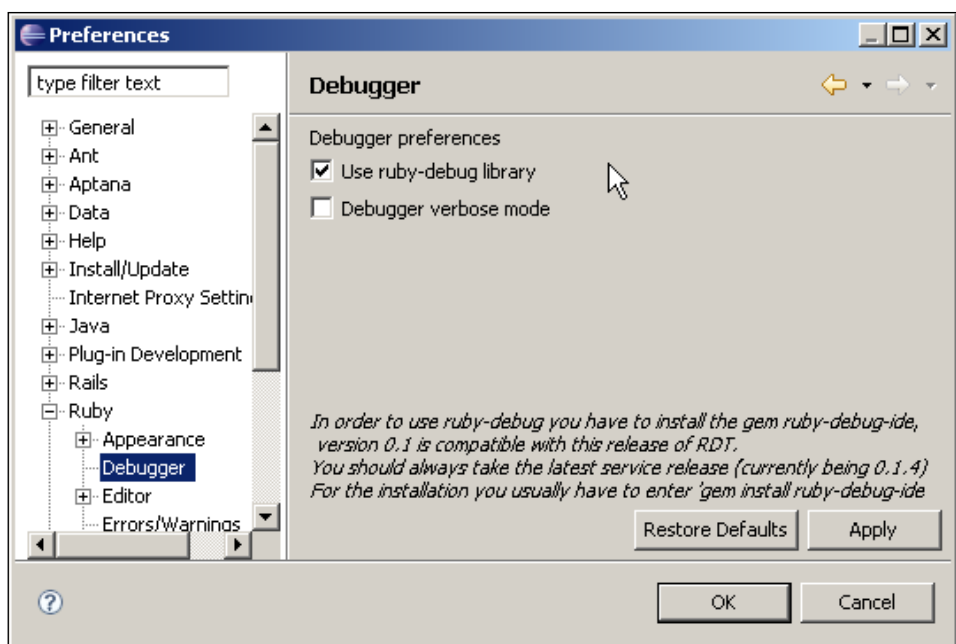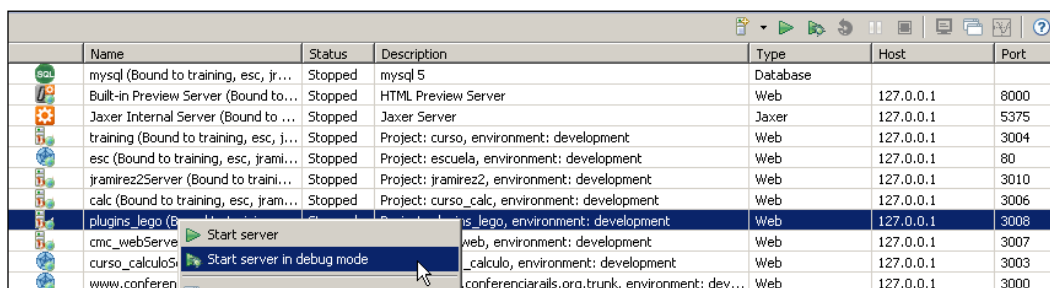
Chances are it will be checked already, but if not we have to check **Use ruby-debug library** so RadRails will use it when debugging. That's it; now you are all set and ready for debugging your application.

Starting Your Server

Debugging is just one step away. We only need to start our server in debug mode before we can start using the debugger.

As usual, we have to navigate to the **Server** view to start our server. If you didn't remove it, you should have a server already configured for your project. If you still don't have it, just create one by right-clicking on the content area of this view and selecting **Add**.

Make sure your server is stopped before trying to launch it in debug mode. The column **Status** of the server view should read **Stopped**. If it doesn't, right-click on the server name and select **Stop Server**. Now we can start it for debugging. You can right-click directly on the server name and select **Start server in debug mode**. If you prefer, you can use the debug icon on the toolbar of this view.

Once the server starts, the **Status** column should read **Running** and your application is ready for a debugging session. As you can see, starting your server for debugging is as easy as starting it for a normal session. Don't misunderstand this, though. A server that started with the debugger on will be slower than an ordinary start, so use it only when you are going to have a debugging session and start as usual otherwise.

Debugging a Ruby Script

This book focuses on using RadRails for working with Rails applications, but of course you can use RadRails for debugging a stand-alone Ruby script too.

The first thing we have to do is locate the script in the Ruby Explorer view. Now just right-click on the file name and from the context menu select **Debug As**. A new menu will open with several options (they may vary depending on your Eclipse installation). From this menu select **Ruby Application**. Your script will be launched with the debugger on.

The **Debug As** option is also available from the **Run** menu of your workbench. In this case you have to select the script in the Ruby Explorer by left clicking on it before launching the debugger.

In this menu, there is another useful option labeled **Debug History**. From this option you can relaunch the scripts you were debugging recently.
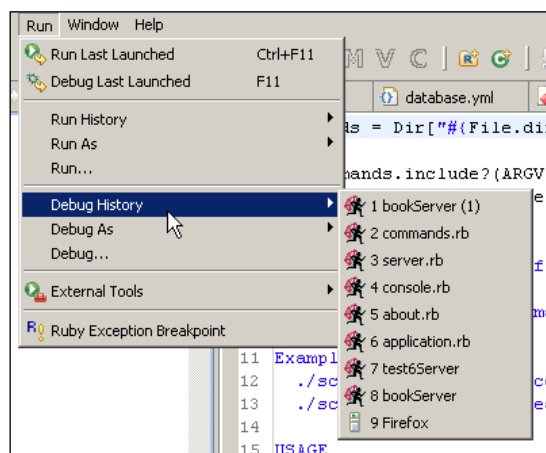
Notice that from the History menu you can also relaunch a server for debugging by selecting the name of the server. After selecting the target server in the menu, you should see the server output in the console view and the status changed to **Running** in the Servers view.

Using Breakpoints

So far we have learned how to start a debugging session, but unless we do something more we will not see any differences from a normal execution (apart from the performance being slightly slower).
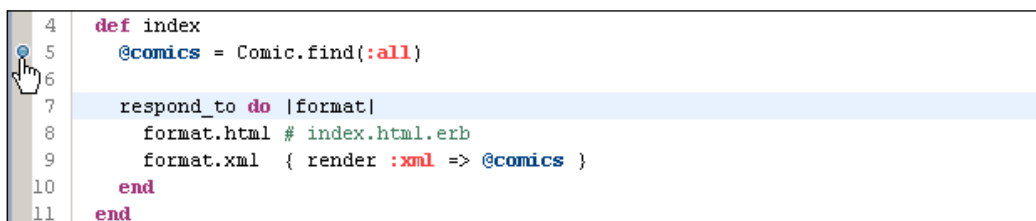
The first thing we need to do when we want to debug an application is set the points at which we want to examine the status of our program. We will use the term **Breakpoint** to refer to each of these points. Whenever the debugger hits a breakpoint, execution will suspend so we can examine the variables or just step through the code and see the execution flow.

Defining a breakpoint in RadRails is a very simple task. All we have to do is double-click on the left margin of the editor. You can double-click either over the line number or in the left gray area. The breakpoint will be set either way. You will see a blue circle indicating there is a breakpoint set at the current line.

If you want to remove a breakpoint, you only have to double-click on top of it. If you don't feel comfortable with the double-click, you can also right-click on the left margin and then select **Add breakpoint** or **Remove breakpoint** to get the same results.

In previous versions of RadRails the editor wouldn't allow you to set breakpoints in RHTML/ERB views, but now you can do it in the same way as with Ruby files, so you shouldn't see any differences.

To see the debugger in action, let's define a breakpoint in our **comics_controller. rb** file under **app | controllers**. We want the debugger to stop when we try to list the existing comics, so let's set the breakpoint at the first line we have in the method **index** by double- clicking on the left margin by that line.

```
 4    def index
 5      @comics = Comic.find(:all)
 6
 7      respond_to do |format|
 8        format.html # index.html.erb
 9        format.xml  { render :xml => @comics }
10      end
11    end
```

Now we need to access the **index** action from a browser to see how the debugger hits the breakpoint. Before opening the browser make sure you have started the server in debug mode as we explained in the previous section. Now we can test the debugger. We could open our action in any browser, but for this sample we will use the integrated browser in RadRails. Go to the **Servers** view, right-click on the name of your server and select **Launch Browser**. In the URL box of your browser just add the 'comics' controller's name. It should look something like `http://localhost:3000/comics`.

Note that the port number could be different for you depending on your server's configuration.

As you know, when we call a controller without an action name, rails will automatically call the **index** action for that controller, so if our server was started in debug mode and the breakpoint was properly set at the **index** action, RadRails will prompt you to open the **Debug** perspective.
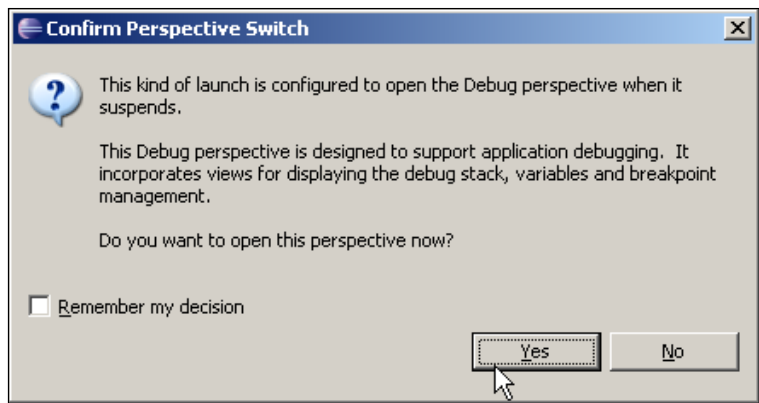
Select **Yes** to enter the debug perspective, which will provide specialized views for our debugging session. If you want RadRails to switch automatically to this perspective whenever the debugger hits a breakpoint, check the **Remember my decision** box.

The first thing you will notice in this perspective is that there are some new views available. Don't worry if you don't know what they are for, since that's exactly what we are going to learn in this chapter.
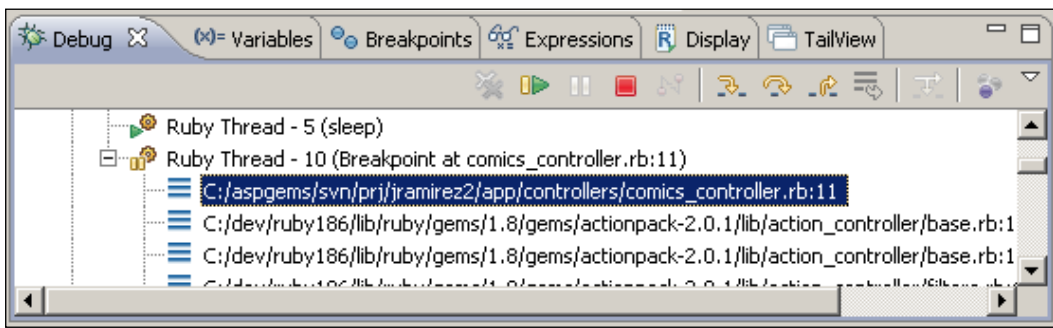
Before starting the explanation for different views, let's focus on our editor. You will see the line where we set the breakpoint is highlighted with a blue background. This means the execution is paused at this line. The debugger will always stop before executing the highlighted line.

At this point there are several things we could do. We could examine the values of any variables, we could evaluate a Ruby expression in the current context, or we could execute the rest of the request step by step. This time let's just let the execution continue as usual. To do so, go to the **Run** menu and select **Resume** (or use the keyboard shortcut *F8*).

The execution will resume and since we don't have any other breakpoints the action will finish and the result of the list will be displayed at the browser tab.

The Breakpoints View

Once we have set the breakpoints for our application, it can be difficult to keep track of which breakpoints we have and in which files and lines of the code we placed them. It would be convenient to have some way of displaying and managing the list of breakpoints. Well, it's our lucky day because that's exactly what Eclipse/RadRails provide under the **Breakpoints** view.

This view should be visible in the default debug perspective, but if for any reason this view is not displaying in your workbench, you can go to the **Window** menu and select **Show View**. The **Breakpoints** view will be available under the **Debug** category.

At this moment your **Breakpoints** view will display a single line representing our breakpoint at the list action of the comics controller. The information you see in this view is the name of the file and the line number where the breakpoint is set. If you double-click on the breakpoint information, the editor will display the exact line it references. You can get the same result by right-clicking and selecting **Go to File** in the context menu or by using the 'Go to file for breakpoint' icon on this view's toolbar.
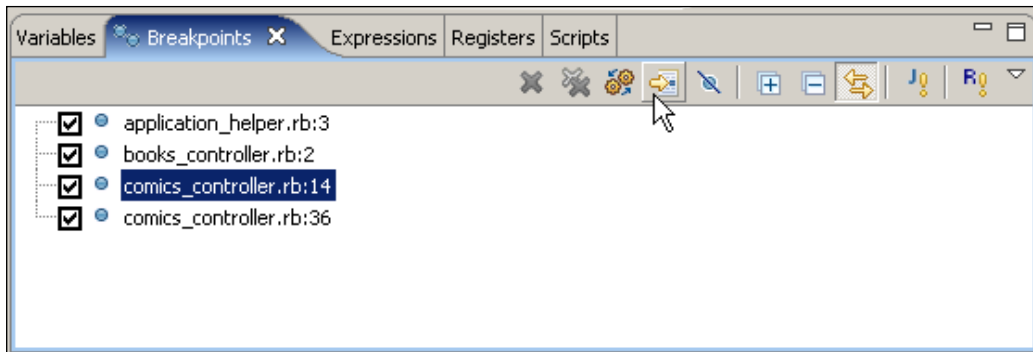
There are some other interesting things you can do with your breakpoints from this view. You can see there is a checkbox by every breakpoint. The check means the breakpoint is enabled. If you want to temporarily disable a breakpoint you can just uncheck its box. When the debugger hits the breakpoint, it will skip it if it is disabled, but you can always enable it again later. You can also toggle the enabled/disabled status by right-clicking and selecting **enable** or **disable** from the context menu.

If you want to permanently remove a breakpoint, you can select it and use the *Delete* key or you can use the **Remove** option from the context menu. There is also an icon on this view's toolbar for removing the selected breakpoint and another one for removing all the breakpoints at once.

When you want to define just a couple of breakpoints, the default view is just perfect, but if you are debugging a complex application and you have a lot of breakpoints, it might be advisable to group them in some way. In the menu for this view (the one you get by clicking on the small triangle by the right of the view's toolbar) there is a **Group By** option so you can choose how to display your breakpoints. By default they are ungrouped, but you can group them by file, type, or project, for example.
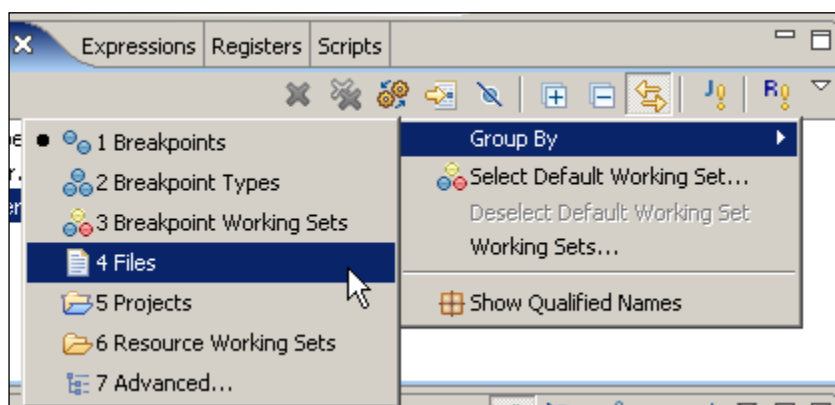
When you use a grouping criterion, you will notice the breakpoints appear in a tree-like fashion, with a first level for the group and then the list of breakpoints it contains. You can use the **Expand all** or **Collapse all** icons on the view's toolbar to show or hide the details of the groups.

Setting Generic Breakpoints for Exceptions

A very useful feature of RadRails debugger is the ability to set a breakpoint that is not associated to a given line but to a Ruby exception. In this way you can get the debugger to pause the execution whenever that exception is launched at any time.

To define a Ruby exception breakpoint, you have to use the 'R!' icon in the breakpoints view toolbar. You will be presented with a dialog to select the exception for which you want to set the breakpoint.
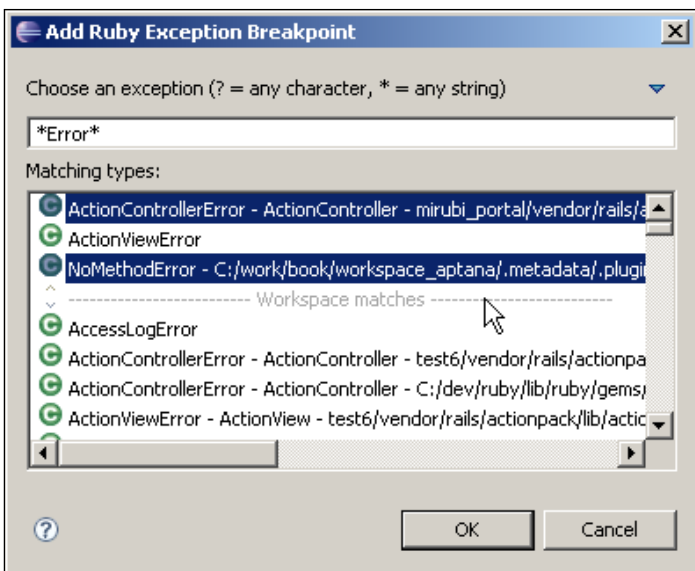
You can just scroll down to the exception you want to intercept or you can type a pattern in the text box to filter out the exceptions with matching names. If you want to add a breakpoint for several exceptions at once, you can select them as you hold the *Ctrl* key or the equivalent in your OS, so you can make a multiple selection. In this case, you will see that a different breakpoint is set at the Breakpoints view for each of the exceptions.

In this list, you can actually select any class and not only exceptions, but if you select a class that is not an exception, RadRails will inform you about that with a message at the bottom of the dialog.

Apart from the process for defining them, Ruby exception breakpoints behave exactly the same as any ordinary breakpoint. If you want to separate the source code breakpoints from the Ruby exception ones, you can do so by grouping the breakpoints by type. In this case you will get two groups, one for the source code breakpoints, and one for the exceptions.

Exporting and Importing Breakpoints

In some cases you might want to export the current definition of your breakpoints for future use or maybe for working on different computers. From the content area of the Breakpoints view you can right-click to open the context menu and select **Export Breakpoints...**.
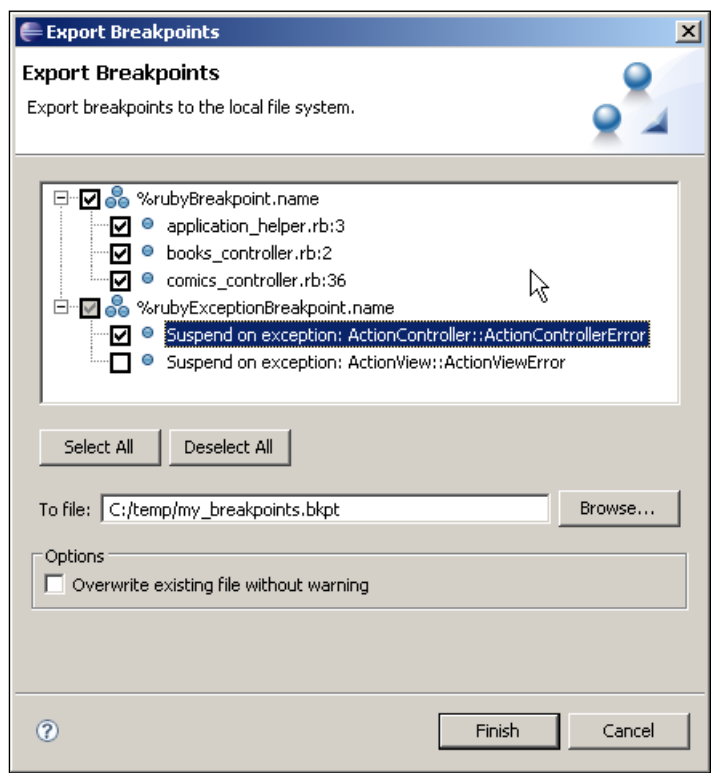
In this dialog you can select which of the breakpoints you want to export (use the **Select All** button for a full export) and where to store the export file. Even though the extension of the file is bkpt, this file is a plain XML file with the definition of your breakpoints.

To import a bkpt file, use the context menu and select **Import Breakpoints...**. Just locate the bkpt file and Eclipse will import the information to your Breakpoints view.

The Debug View

So far we have been learning how to use breakpoints so we can pause execution at any point we want, but debugging is much more than just stopping the application flow. In this section we will see how to use the **Debug** view both for inspecting the history of method calls and to execute our application step by step.

The Debug View and the Stack Frame

In Computer Science, the Call Stack often shortened as 'the stack' is a dynamic structure containing information about the active subroutines of a running program. This stack is divided into Stack Frames, which is the name we use for referring to a call or to a subroutine that has not terminated yet with a return.

In Ruby, a Stack Frame represents each call to a method that is made during the execution of our scripts, together with the status of the variables at that point. By examining the history of the Stack Frames, we can see how we got to the point we are at, and we can inspect the statements that were recently executed to better understand the execution flow.

To see the stack frames in action make sure you have started your server in debug mode and then let's open again the URL for our comic list at `http://localhost:3000/comics`. You can use the built-in browser or just any browser in your system.

When the debugger reaches the breakpoint at the list method of our controller, the execution will pause as we saw before. So far, this is the same process we did in the previous section, but we will focus now on a different view.

The **Debug** view should be available at the default Debug perspective. If your workbench is not displaying this view for any reason, you can open it from the **Show View** option of the **Window** menu.
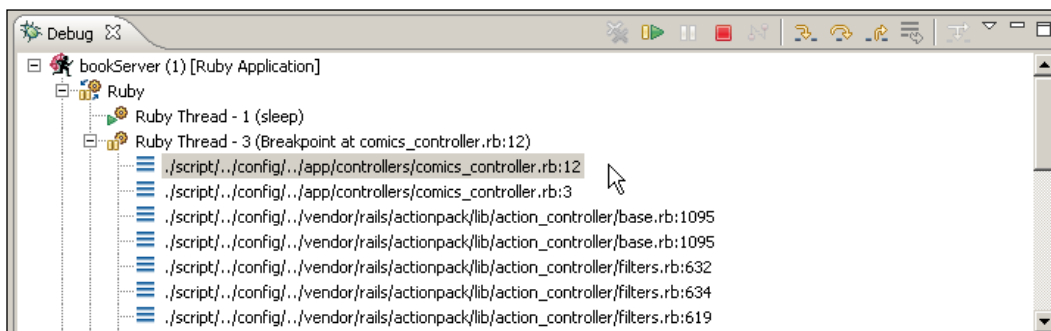
In this view you will notice a line highlighted with a gray background. That's the indicator of the current line where the execution is paused. This line appears grouped with others under a section with the title **Ruby Thread** and then the name of the file in which this thread is stopped right now. In our case it will be the **comics_controller.rb** file.

You might be wondering what this thing is about having different threads if Rails is a non-threaded framework. Well, even though in Rails you cannot use threads, it doesn't mean the server in which your application runs is not using threads internally. Fortunately we don't have to worry about that. The server will execute every request you send in a thread of its own. You can just ignore the rest of the threads and concentrate on the one in which your request are being processed.

Under this thread, you can see the already famous Stack Frames. The line on top is where our breakpoint is, and each of the lines below represents a method call. If you click on any of the Stack Frames, the Ruby Editor will open the referenced file at that method call line. In this way you can traverse back through the flow of your execution and you can even examine the internal Rails code that originated the call.

Sometimes when debugging you may want to go to a given point of the stack but since the stack frame can be long you don't want to scroll it down to find where it is. You can use the built-in **Find** feature of the **Debug** view. If you right-click and select **Find** or use the **Find** option of the **Edit** menu, or if you use the shortcut *Ctrl+F*, a find dialog will appear.
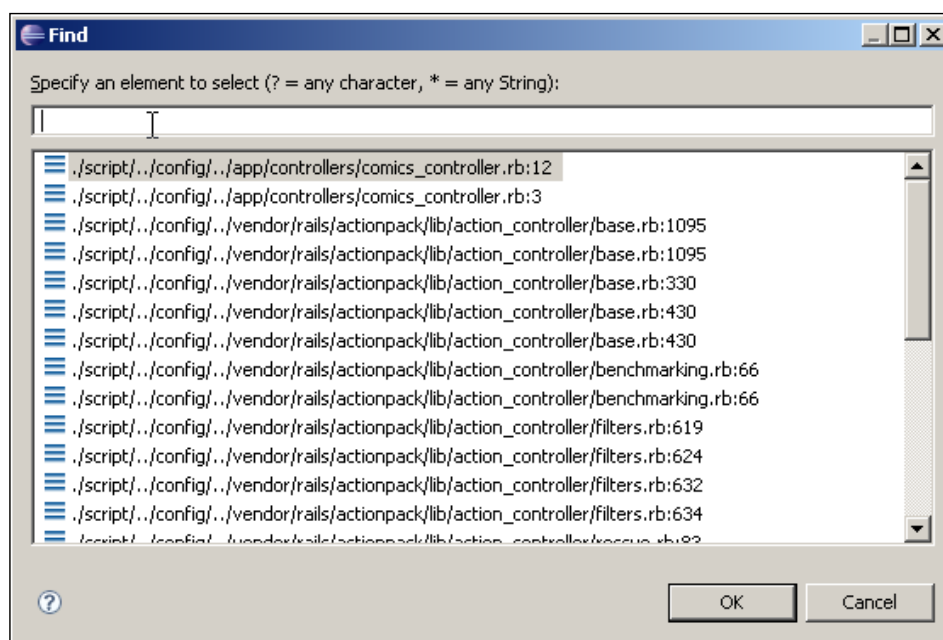
By now you should already be familiar with this kind of dialog in RadRails. Just type the pattern of the line you want to find and the lower pane of the window will filter only those elements matching your pattern.

Stepping through Your Application

So far we know how to set breakpoints to pause our program's execution and how to move back in the method call history. Now we are going to learn how to move forward in your applications execution in a controlled manner.

When you want to go over the details of a piece of code to detect a possible bug, it's not very efficient to define a breakpoint at every line of the code. For these cases we can use the stepping capabilities of the debugger.
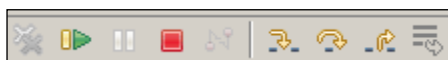
You can use the step through controls from the **Debug** view toolbar, from the **Run** menu, or when you are more comfortable with debugging with their keyboard shortcuts. As usual, you can see the shortcut by the right of each option in the **Run** menu.

As you can see, this toolbar is divided in two areas. At the left you can see three icons, which look like any ordinary multimedia player and which are pretty intuitive to use. The one with the green triangle, called **Resume**, will continue a paused execution and will not stop until the next breakpoint is reached or until the program finishes.

The next control is called **Suspend** and it's represented by two parallel bars. As you probably guessed, this will suspend program execution as if it had hit a breakpoint. It's not a very frequently used control when debugging a Rails application because request cycles are typically pretty short and you don't even have the time for manually pausing them, but if you are debugging a long-running process you could find it useful.

Finally we find the **Terminate** control represented by a stop icon. By clicking this icon you will stop the running process. This would be equivalent to stopping your Rails server from the **Servers** view.

The controls **Resume**, **Suspend**, and **Terminate** are available from the **Run** menu, but only **Resume** has a keyboard shortcut (*F8*).

These three functions are the very basics of controlling the execution flow, but usually when debugging you will want more flexibility for moving through your code. You can get this by using the three controls on the right side of the **Debug** view toolbar. These controls are represented with yellow arrows indicating the execution flow and their names are **Step Into**, **Step Over**, and **Step Return**.

To illustrate the usage of these controls we are going to define a very simple method and set new breakpoints. First remove the breakpoint we already have. If you set more than one for your tests, remove them all. You can easily remove all the breakpoints at once with the **Remove All Breakpoints** icon on the **Breakpoints** view or from the **Run** menu.

Now we will create a method named **get_comics** with a single line for getting all the comics from the database and we will change the first line of the **index** method to use our new method. You can see the final result in the figure below.

```ruby
 3    def get_comics
 4      Comic.find(:all)
 5    end
 6
 7    def index
 8      @comics=get_comics
 9
10      respond_to do |format|
11        format.html # index.html.erb
12        format.xml  { render :xml => @comics }
13      end
14    end
```

Now, let's set a breakpoint at the first line of the **index** method. Hence, the debugger will stop whenever we call the index action of the 'comics' controller and before executing the **get_comics** method. Go to your browser and reload the page `http://localhost:3000/comics`.

When the debugger stops, there are several options for us to choose from: **Resume**, **Step Into**, **Step Over**, or **Step Return**. Depending on which part of our application we are interested in debugging, we will be choosing one or another.

As we already know, choosing **Resume** would continue execution until the next breakpoint was hit, but let's see what each of the other options would do.

If we tell the debugger to **Step Into**, it will continue execution, enter the **get_comics** method and then stop at the first line of this method. **Step Into** basically executes the call to the referenced method in the current line and then stops after the call before executing any of the lines inside that method.

We can try it either by using the **Step Into** icon on the **Debug** view toolbar, by using the corresponding option at the **Run** menu, or directly with the shortcut *F5*. You will see the debugger entering the **get_comics** method and pausing at the first line. At this point we can click on **Resume** to let execution continue as usual.

If you choose to **Step Into** on a line where several method calls are referenced, the method in which the debugger will stop will be the one with higher precedence either by being the leftmost method or by parenthesis use. In any case, the method into which you are stepping will always be the method that would be called first in a normal execution.

Consider a line such as:

```
Comic.find(:first).title.upcase
```

In this line we are using three method calls, `find`, `title`, and `upcase`. If you choose to **Step Into** in a line like this, you would be taken to the first line of the **find** method.

Now let's see what the **Step Over** functionality does. Go to your browser again and reload the page so we will get the debugger suspended at the first line of our **index** action. Stepping over this line means to execute all the code associated with that line and to continue execution until the next line in the current method, where the debugger will pause again. This means if in the current line there are any calls to other methods they will be all executed but we will be stepping over them without pausing to see what they are doing.

If you use the **Step Over** control (the shortcut being *F6*) you will see the debugger stops at the next line, the one with the **respond_to** method call. Of course the call to the **get_comics** method will be made, but we just don't go through the details of it. Just resume execution so we have the debugger ready to try the last stepping method.

Finally, we can choose to **Step Return** when the program's execution is suspended. This will cause all the code of the current method to be executed and the debugger will stop exactly at the next line after this method's return, which will be the line after this method was referenced at the calling method.

To see how it works, reload the same page again in your browser for the debugger to stop at our breakpoint. Now if you click the **Step Return** control (the shortcut being *F7*) you will see the execution resumes and suspends again at a line inside the **perform_action** method of `ActionController::Base`, which is the method Rails uses to invoke our actions. Don't forget to use **Resume** to let the request cycle finish normally.

Maybe by now you are wondering if you can modify a line of code in real time when the execution is suspended by the debugger. Well, you can modify the code, but it will not have an effect until you resume execution and enter the request cycle again. The way Rails works in development environment is that the code for the classes is loaded the first time the class is used in the request cycle, so you will have to force a new request in order for your changes to be effective.

With these stepping capabilities, you can inspect the execution flow of your application, so you can understand where and how the different methods are called. This is already pretty interesting, although often you don't want only to see how the code is executing, but also the values of the variables currently in scope.

In the next section, we will learn how you can inspect the status of your application once you have reached the point you want by using the stepping-through controls.

Variables and Expressions

A very useful feature is being able to inspect and even modify the variables and constants in your code as it's executing. There are two views that will help us inspect the status of our variables when debugging: the **Variables** view and the **Expressions** view.

The Variables View

As you have surely guessed, this view will allow us to display and modify the values of the variables in the current scope.

To see how it works, open the comics page in your browser and wait for the debugger to stop at the defined breakpoint. Now open the **Variables** view. It should be available as a tab in the default Debug perspective, but if you cannot see it, you can open it from the **Show View** option of the **Window** menu under the **Debug** category.
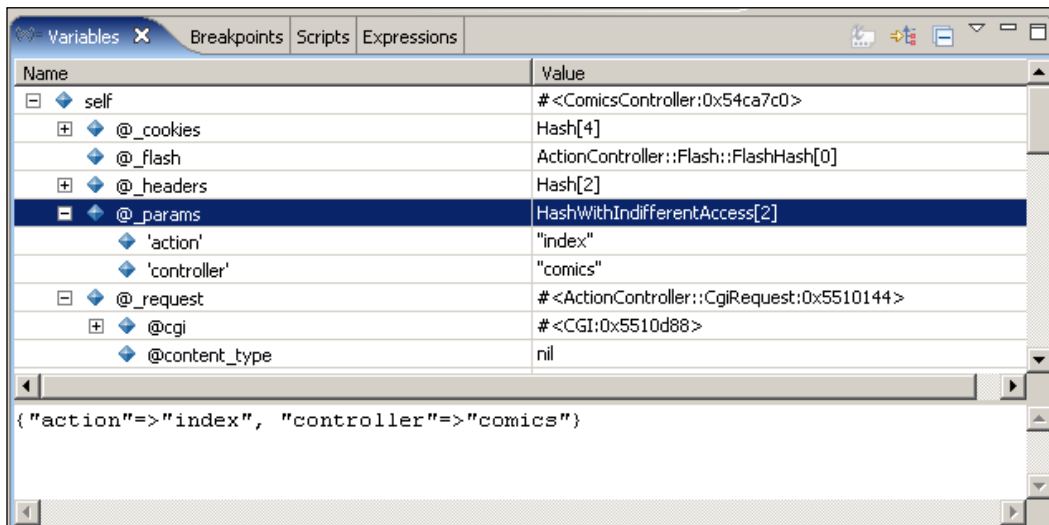
This view displays all the variables currently in scope. You will see a single entry named **self** which is the reference to the object containing the current controller. If we had any local variables defined in our code and they were available in the current scope, they would appear directly at the same level as **self**.

You can see we have two columns at this view. The first one will display the name of the variable and the second will display its value. If the variable is a simple one, the value will display directly, but in the case of more complex objects, a summary will appear (usually the name of the class and the object ID or the number of elements in the case of object containers such as Hashes or Arrays).

You can unfold the different elements for example, the **self** variable and then the **@_params** one to see the details inside.

As you can see, the default layout uses two columns and a lower pane. If you click on any variable, the lower pane will display the contents of the variable. This can be especially useful when inspecting the values of large strings or of object containers with other elements inside.

You can change the layout of this view from the **layout** option of the menu available at the right of this view's toolbar. You can choose to have a vertical or horizontal layout, to hide the lower pane (by selecting **Variables View Only**) or even not to use columns for displaying the values. In this case the value for each variable will appear right after its name.

If you are not using columns, you can toggle on and off the functionality of displaying the type name of the variables. To toggle the type names you can use the first of the icons in this view's toolbar, labeled **Show Type Names**. This icon will appear faded out if you are using a multicolumn display.

On the menu of this view, you can also toggle the display of class variables. By default it is disabled, but you can turn it on by checking **Show Class Variables**.

You have probably noticed that there can be a lot of variables in scope at a given moment. If you want to see the value of a single variable, you can use the built-in search functionality of this view. By right-clicking on the variable list or by choosing **Find** in the **Edit** menu (or directly with *Ctrl+F*) you will get a **Find** dialog, which works as usual. If you type the desired pattern, the dialog will filter only the matching variables. If you double-click on one of these variables, the dialog will close but this variable will be selected for you in the **Variables** view.

We can not only display the values of the variables, but we can also modify them. If you right-click on top of any variable in the **Variables** view and select **Change Value**, a new window will appear for you to enter the desired value.
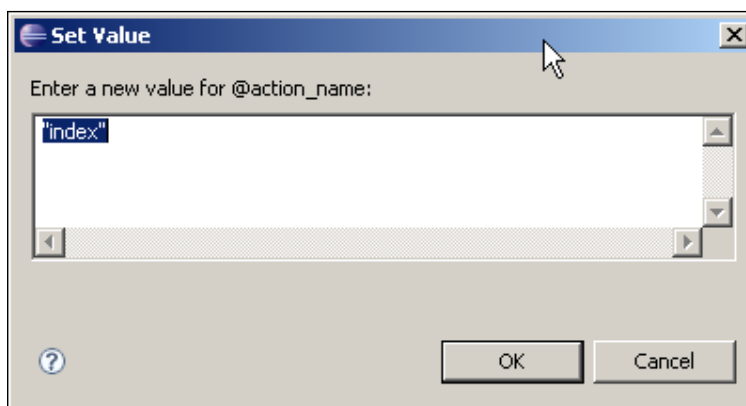
If you prefer it, you can also change a variable's value directly from the lower pane in which the value is displayed. Just type the new value, then right-click and select **Assign Value**. Unless you are working with the lower pane hidden, you might find this method a bit more convenient since there is no pop-up window for modifying the value. In any case, the results will be the same.

In latest versions of RadRails everything is working fine, but in older versions there was a minor annoyance when changing the value of a variable from the **Variables** view. You changed the value, but the old one would still appear. The changes actually took effect immediately, but they would not display at the **Variables** view until you let the debugger move at least one step in your code. If you are experiencing this behavior, it means it's high time to get an updated version of RadRails.

The Expressions View

The **Expressions** view is similar in many ways to the **Variables** view. You can change the layout, choose to display the types or only the names, and use a lower pane or not.

The difference between the Variables and the Expressions view is that we don't display the value of a variable in the code, but the result of evaluating an expression in the current context.

If you have used Eclipse with other languages than Ruby, you might have used the Expressions view for setting **watch expressions** that get evaluated automatically every time a breakpoint is hit. As of today, the Ruby expressions you set in this view will not get re-evaluated automatically, but you have to do it manually (as we will learn just below). There are plans for including the watch expression functionality in future versions of RadRails, so maybe by the time you are reading this book this feature is already implemented, and in that case there could be some differences with the contents of this section.

You can select any expression in your code, like the condition in an `if` statement, and see the result of the evaluation. If you select a variable name, you will see the value as in the **Variables** view but you will not be able to modify it.

To inspect an expression in your code, make the debugger hit a breakpoint, select the expression to evaluate, and then right-click and select **Inspect**. Note that this is the **Inspect** option directly, and not the one with three trailing dots. The selected expression and the result of the evaluation will appear at the **Expressions** view.

Apart from the expressions in your code, there are certain frequently used expressions that are available as **Quick Inspect** expressions. To see the expressions we can evaluate this way, make the debugger stop at any breakpoint and then at any line of the editor right-click and select **Inspect...**. Note that this is the **Inspect...** option with three trailing dots, not the one with just the word **Inspect**. A context menu will appear with several options.
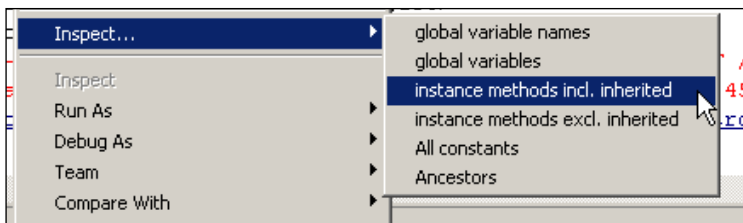
By selecting any of the available expressions you will see the results in the **Expressions** view.

If you want to add more expressions to inspect, you can do so by going to the **Preferences** option of the **Window** menu and then to **Ruby** and **Evaluation Expressions**.
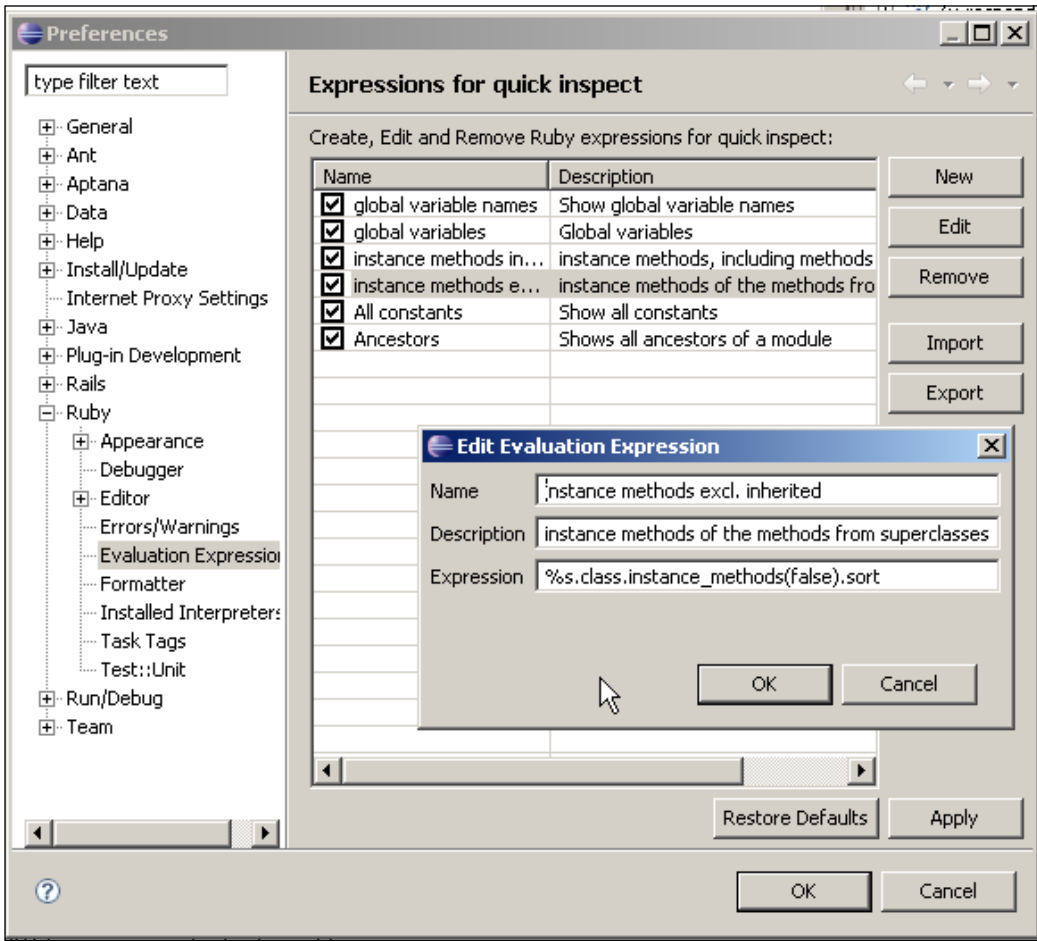
If you select one of the expressions and click on **Edit** you can see the source code of the expression. For example, the code for the **All Constants** expression is:

```
Module.constants.sort
```

and the code for the **Instance methods including inherited** expression is:

```
%s.class.instance_methods(true).sort
```

If you compare these two expressions, you will find a difference between them: in the second expression we are using the wildcard `%s`. There is a good reason for this. In the first expression, the one for the `constants`, the result will be always the same independently of the selection. Since the expression doesn't need a context, it doesn't use any special mechanism.

The second expression, on the other hand, will display instance variables. Instance variables are associated with an object, so the expression needs to know the context of the object in which to execute the expression. The wildcard `%s` represents the current selection in the editor. If there is no selection, then **self** will be used instead.

You can add any expressions for **Quick Inspect** by using the **New** button of this dialog. If you want to share your expressions with other users or between multiple installations, you can export them to XML and then import them back with the **Export** and **Import** buttons.

The Display View

So far we have seen how to inspect the values of variables and expressions and how to modify the values of variables at run time. This, in combination with the stepping capabilities of the debugger, gives us much control over what's going on in our code. And yet, there are occasions when you want to execute arbitrary code or just change the value of a constant in the context of the execution. This can be done by using the **Display** view.

The **Display** view is not available in the default Debug Perspective, so we will have to open it manually. Go to the **Window** menu, then select **Show View** and choose the view called **Display**. If there is more than one view with this name, select the one in which the icon contains an **R** for Ruby. The one with a **J** stands for Java and will be useless in a Ruby debug session.

What this view does is basically execute any code as if we were in the Rails console or in IRB, but within the current context of the line in which the debugger is paused. The values of any variables will be available when executing and any changes you make will have immediate effect in your debugging session. As it happens with the Rails Console, this view is one of my favorites when debugging. And it has the added value of being pretty straightforward to use.

The first thing we need to do is suspend the execution at any breakpoint. Since the **Display** view will execute any code within the current debugging context, we need first to have the debugger stopped at some point or else the **Display** view will be disabled.

Now we can execute any code in the same way as in the Rails console. Write the code you want to run, and press *Enter*. Alternatively, you can select it, and click on the icon in this view's toolbar labeled **Evaluate the selected text**. The selected code will be executed and the return value will be displayed.
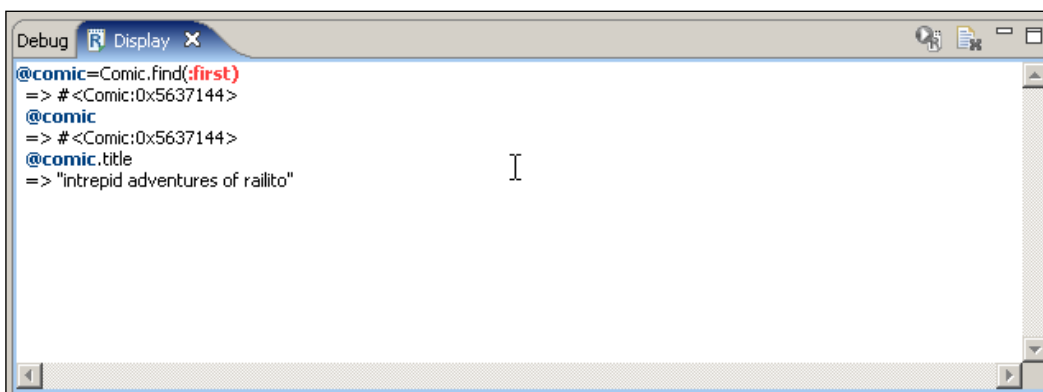
If after a while you want to clean the **Display** view, you can use the **Clear Console** icon of the toolbar.

This view was created very recently in RadRails and is still undergoing some changes to make it even easier to use and more similar to working directly with the Rails console (except that execution will be made in the context of the current breakpoint). By the time you are reading this book, some changes in this direction might already have been made.

Useful Tools for Debugging

Apart from the debugger itself, Eclipse and RadRails provide some complementary tools which can be useful when examining our application for errors. We will explain in this section how you can link errors to source code directly, and how to tail your log files.

Linking Errors and Source Code from the Browser

There is a nice feature you can use that is not strictly for debugging but can help you when finding an error. If you are using the built-in browser in RadRails, every time you get an error in the browser with a stack trace, the lines of the stack will be linked with your source code, so if you click directly on any of them, RadRails will display the exact line of that source file in the Editor.

Tailing the Log Files

Even though debugging is great for finding errors and you will not need to use the log as much as before, there are still many times when you want to use log files for debugging purposes. For example, you might want to read the log to see the parameters your actions are getting, to know which actions are being executed, or to examine the SQL statements that are being produced by Active Record.

There is a basic **Tail** view available in Eclipse. From the Ruby Explorer you can right-click on any log file and then select **Tail**. Now every time there is a new line written in that log file, you will get the output in the **Console** view.

In RadRails there is a more convenient view, called **Tail** view, which provides some interesting features. We will learn all the details in next chapter, but for now you can already use it by going to the **Servers** view, right-clicking on the name of your server, and selecting **Open Log**. You can also select the **Open Log** icon located in the toolbar of the **Servers** view.
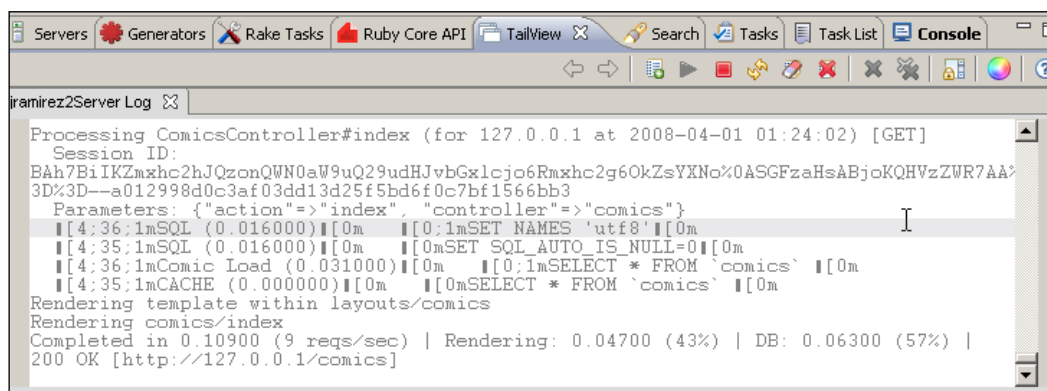
Summary

In this chapter we have learned how to use RadRails' built-in debugger to interact with our code at run time. We know how to start a server or a stand-alone script in debug mode, how to set breakpoints so the execution will suspend at the selected points, and how to intercept any Ruby exceptions.

Once the execution is suspended, we know how to interact with variables, expressions, and even execute arbitrary code in the **Display** view. The stepping capabilities of the debugger make it easy to walk through your code step by step and understand the execution flow.

After finishing this chapter you will find yourself spending less time in fixing your errors and having more time available for focusing on the business logic of your application and on the creative process of the development.

Sure you could fix your errors without the debugger too, but it would be much slower and it wouldn't be half as much fun.

7
RadRails Views

By now you should be comfortable with the general interface of Eclipse and RadRails. You know already how to create Rails projects, write and debug Ruby code and views, and work with HTML, JavaScript, and CSS files. We could say most of our programming needs are fulfilled with that.

When developing a Rails project, there are more things to do than the source code itself. We have to start, stop, and monitor our servers, generate code templates, run our test suites, install plugins and gems, generate documentation, keep control of to-do items, or run Rake tasks for different purposes—database migrations, for example.

RadRails provides different views for supporting these tasks that are a part of the development but not of the coding itself. And, of course, it does it so we can control everything from within the IDE without having to go back to the command-line interface.

We already had a glimpse of some of these features when using the Generators, Rake, or Servers views briefly when we needed them in previous chapters. Now you will learn how to take full advantage of all the RadRails views, to help you take care of routine processes and just focus on getting things done.

Opening the RadRails Views

Some of the views that we will go through in this chapter are available as part of the Rails default perspective, which means you don't need to do anything special to open them; they will appear as tabbed views in a pane at the bottom of your workbench. Just look for the tab name of the view you want to see and click on it to make it visible.

However, there are some views that are not opened by default, or maybe you closed them at some point accidentally, or maybe you changed to the Debug perspective and you want to display some of the RadRails views there. When you need to open a view whose tab is not displaying, you can go to the **Window** menu, and select the **Show View** option.
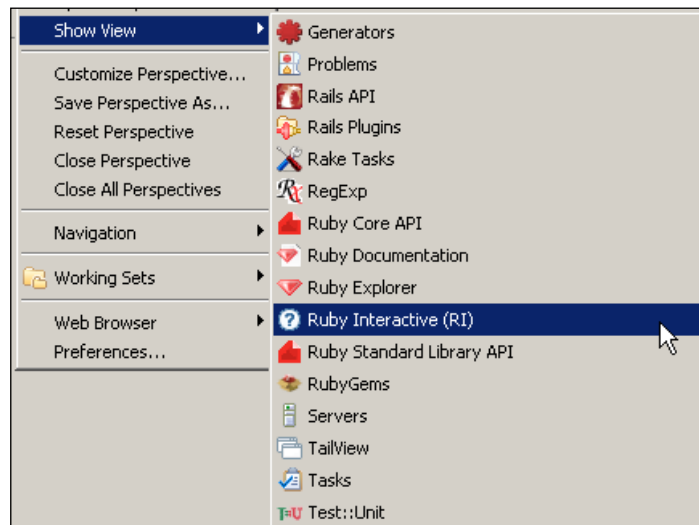
If you are in the Rails perspective, all the available views will be displayed in that menu, as you can see in the screenshot above. When opening this menu from a different perspective, you will not see the RadRails views here, but you can select **Other...** as we did in previous chapters. If this is the case, in the **Show View** dialog, most of the views will appear under the **Ruby** category, except for the **Generators**, **Rails API**, and **Rake Tasks** views, which are located under **Rails**.

Documentation Views

As happens with any modern programming language, Ruby has an extensive API. There are lots of libraries and classes and even with Ruby being an intuitive language with a neat consistent API, often we need to read the documentation.

As you probably know, Ruby provides a standard documentation format called RDoc, which uses the comments in the source code to generate documentation. We can access this RDoc documentation in different ways, mainly in HTML format through a browser or by using the command-line tool RI. This produces a plain-text output directly at the command shell, in a similar way to the `man` command in a UNIX system.

RadRails doesn't add any new functionalities to the built-in documentation, but provides some convenient views so we can explore it without losing the context of our project's source.

Ruby Interactive (RI) View

This view provides a fast and comfortable way of browsing the local documentation in the same way as you would use RI from the command line.
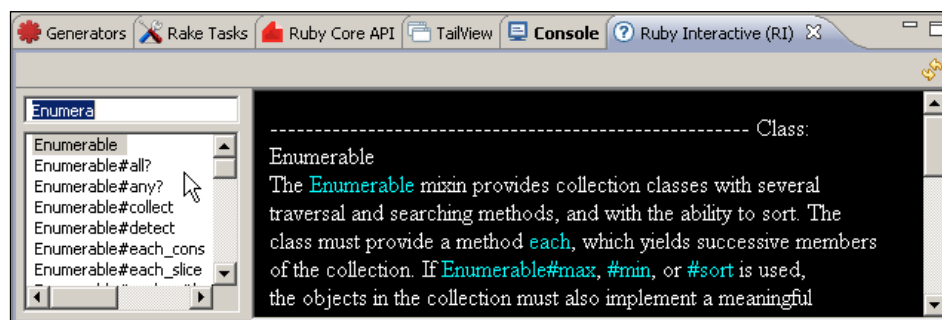
You can look either for a class or a method name. Just start typing at the input box at the top left corner of the view and the list below will display the matching entries. That's a nice improvement over the command line interface, since you can see the results as you type instead of having to run a complete search every time.

If you know the name of both the class and the method you are looking for, then you can write them using the hash (pound) sign as a separator. For example, to get the documentation for the **sum** method of the class **Enumerable** you would write **Enumerable#sum**.

The documentation will display in the right pane, with a convenient highlighting of the referenced methods and classes. Even if the search results of RI don't look very attractive compared to the output of the HTML-based documentation views, RI has the advantage of searching locally on your computer, so you can use it even when working off-line.

Ruby Core, Ruby Standard Library, and Rails API

There are three more views related to documentation in RadRails: Ruby Core API, Ruby Standard Library API, and Rails API. Unlike the RI view, these ones look for the information over the Internet, so you will not be able to use them unless you are on-line.

On the other hand, the information is displayed in a more attractive way than with RI, and it provides links to the source code of the consulted methods, so if the documentation is not enough, you can always take a look at the inner details of the implementation.

The **Ruby Core API** view displays the documentation of the classes included in Ruby's core. These are the classes you can directly use without a previous `require` statement. The documentation rendered is that at `http://www.ruby-doc.org/core/`.
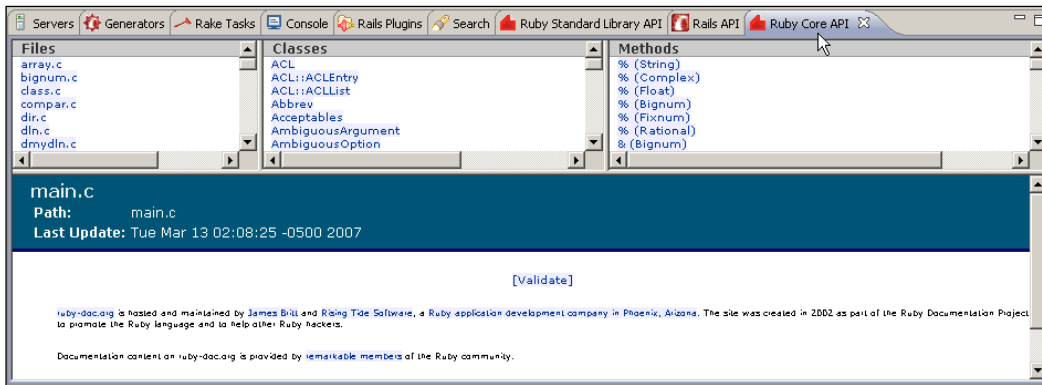
You are probably familiar with this type of layout, since it's the default RDoc output. The upper pane displays the navigation links, and the lower pane shows the detail of the documentation. The navigation is divided into three frames. The one to the left shows the files in which the source code is, the one in the middle shows the Classes and Modules, and in the third one you can find all the methods in the API.

The **Ruby Standard Library API** is composed of all the classes and modules that are not a part of Ruby's core, but are typically distributed as a part of the Ruby installation. You can directly use these classes after a `require` statement in your code. The Ruby Standard Library API View displays the information from `http://www.ruby-doc.org/stdlib`.
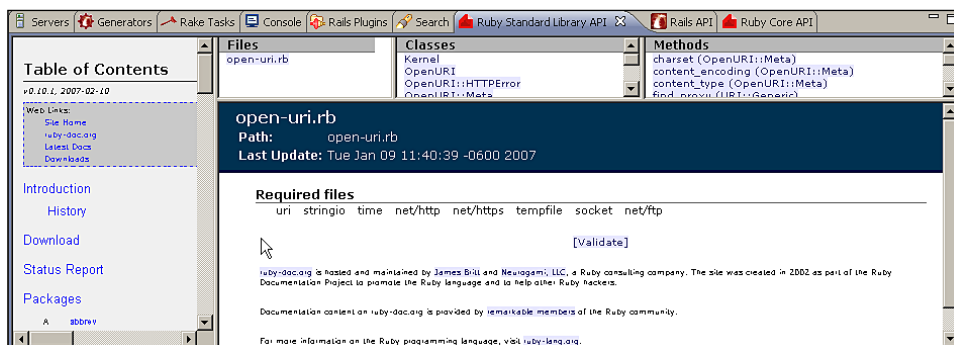
In this case, the navigation is the same as in Ruby Core, but with an additional area to the left, in which you can see all the available packages (the ones you would require for using the classes within your code). When you select a package link, you will see the files, classes, and methods for that single package.

The last of the documentation views displays information about the Rails API. It includes the documentation of ActiveRecord, the ActionPack, ActiveSupport, and the rest of the Rails components. The information is obtained from `http://api.rubyonrails.org`.
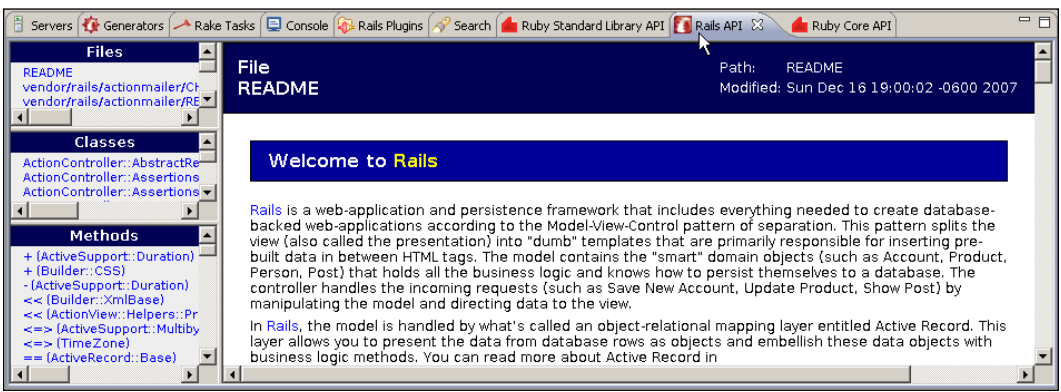
In this case the layout is slightly different because the information about the files, classes, and methods is displayed to the left instead at the top of the view. Apart from that, the behavior is identical to that of the **Ruby Core API** view.

Since some of the API descriptions are fairly long, it can be convenient to maximize the documentation views when you are using them. Remember you can maximize any of the views by double-clicking its tab or by using the **maximize** icon on the view's toolbar. Double-clicking again will restore the view to the original size and position.

Servers View

We went briefly over the Servers view when creating our first application and also when talking about debugging. This is a fairly simple view, but it's also a very useful one. Basically, you can start and stop your Rails servers (WEBrick, Mongrel, or LightTPD), launch the built-in browser, or start a debugging session.
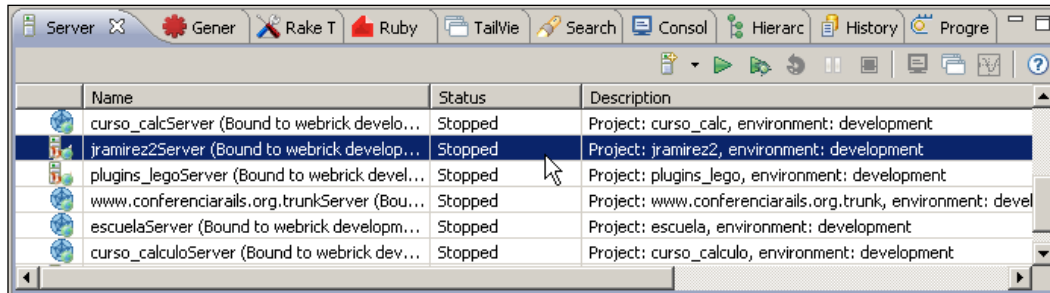
The **Servers** view displays all the available servers for the current workspace, and not only for the current project. This view provides a 'project' column, so you can know to which project your server is associated at a glance.

The very first time that you open RadRails after a fresh installation this view will be empty. By default, when you create a new Rails project, RadRails will select the option to create a Mongrel server for it. Unless you manually uncheck this option at the **New Rails Project** dialog, whenever you create a RadRails project you will see a new server in this view.

If you chose not to create any server from the **New Rails Project** dialog, nothing would appear in this view, and you wouldn't have a way for starting your server from the IDE, having to switch to the command line for that operation.

In that case, you can add a new server for your project from this view. The first icon in this view's toolbar is the one for adding a new server. You can click directly on the icon, or open the drop-down with the small arrow by its side, and then select **Rails Server**. There are also options for adding a new server both in the context menu of the project in the **Ruby Explorer** and from the **New** option under the **File** menu. No matter how you choose to add your server, a pop-up will appear prompting you to fill in the server properties.
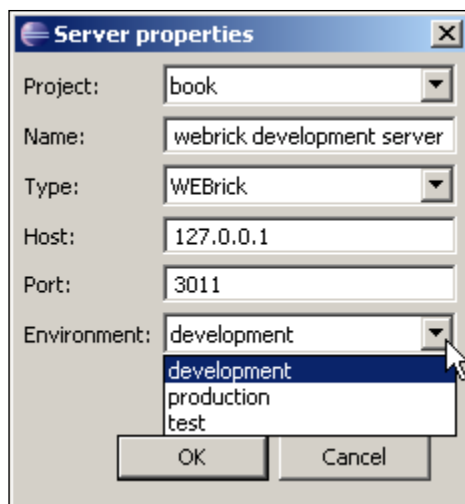
For the **Project** name you have to select a project from the drop-down box. Note that, only open projects will display in the list, so make sure your project is open before trying to add a new server for it. The name of your server will be the one showing in the **Servers** view, so it's in your interest to provide a meaningful name, especially if you have many projects in the same workspace.

The type of project must be one of the three available options: **WEBrick**, **Mongrel**, or **LightTPD**. The only arguments you can provide to the server are the IP address or **Host** name, the the server port number, and the Rails environment. RadRails will default the port number to the first port starting from 3000 on which you don't have any other server configured. If you want, you can change this value and configure two different servers to use the same port.

The new server will be listed in the **Servers** view. After adding a server, you can at once start it, start it in debugging mode, or stop it. You can perform these operations directly from the view's toolbar or from the context menu by right-clicking on the project name. When you start a server, its output will be displayed at the **Console** view, providing the same information you would get when starting from the command line.

If you prefer to have access directly to your server logs, you can open the context menu and select the **Open Log** option. This will display your server's log in the **Tail** view. Later in this chapter you can find a section dedicated to this view.

If you need to restart the server, you could just stop and start it or you can use the convenient restart option. If you want to change the properties of your server (port, environment) you can use the **Edit** option or just double-click on the project's name and enter the new settings. Observe that you cannot edit the server's properties while it's running, so you would need to stop it first.

As we have seen already in previous chapters, when your server is started you can easily open the built-in browser pointing to the home page of your project from the **Launch Browser** option of the context menu.

If you are a user of the Professional version, the **Server** view will also display an option for launching under profile mode to identify the bottlenecks in your application. Profiling is not available under the RadRails Community version, so we will not cover it in this book. You can find more information in the online documentation of Aptana.

Starting a Server with Additional Arguments

Usually, the only options you need to use for starting your Rails server are the port and environment. However, there are some cases in which you want to provide extra arguments like the mime-types definition file or the timeout for Mongrel, for example. In these cases, you cannot use the built-in Servers view, but it doesn't mean you cannot use Eclipse for managing your server anyway.

If you need to pass extra parameters to the 'script/server' command, you can use the Rails Shell View that we will be explaining later in this chapter.

If you want to launch your server with extra parameters and without using 'script/server', for example when using mongrel cluster, you can still do it from Eclipse by configuring it as an external tool. We will be learning more about this option later in this chapter.

Managing Non-Rails Servers from the Servers View

Since version 1.0, the Servers view has support for Apache, MySQL, and generic web servers. This means you can start or stop your servers and have access to the logs directly from the Servers view.

Please note that your MySQL or Apache will need to be installed and properly configured before trying to start your servers from this view. Installation of MySQL and Apache are out of the scope of this book, so refer to the documentation of those tools if you have any questions about how to set them up.

To add a MySQL or Apache web server to your servers list, use the **Add Server** icon from the **Servers** view toolbar, and select MySQL or Apache instead of the Rails Server option. A dialog will display asking for information about your server.
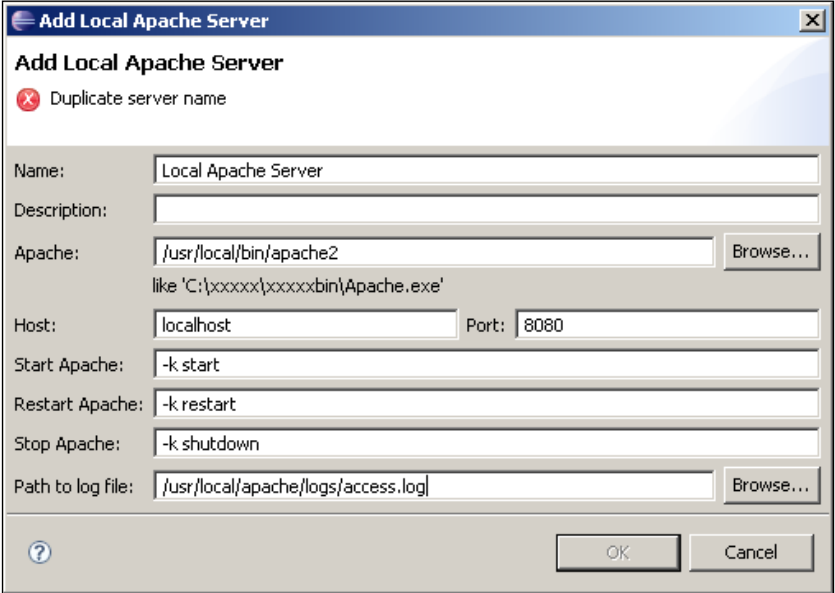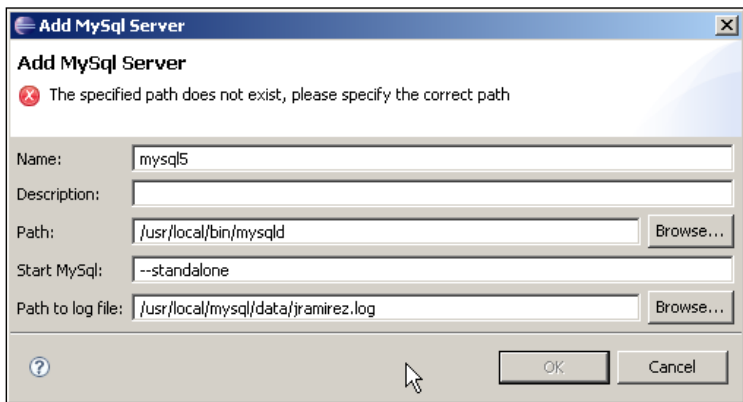
You need to provide information about where the executable is located, and where the log file is. Depending on your server, there will be some extra options you can configure. These options are pre-filled with default values that should work fine for the typical user.

After you click **OK**, a new entry will be listed in your **Servers** view, and you will be able to start, stop, and browse the logs in the same way as with any Rails server.

Launching External Tools from Eclipse

Eclipse has an option for starting any external process you want, displaying the output of the process in the Console view. By configuring our processes as external tools, we can launch them, stop them, and monitor their output from the IDE without having to go to the command line, and without having to retype the same arguments every time.

External tools can be a good way of controlling servers such as **memcached**, **nginx**, **sphinx**, or **mongrel cluster**, but they can be used for any process you would start from a command line.

As an example, we will explain how you could use External Tools for launching a Mongrel server with some extra parameters. It is not really necessary to use an external process for this, since the Rails Shell view will let us pass any extra options that we want to a Mongrel server. This example is just provided for educational purposes, having the advantage that you don't need to install any extra servers to try how to use the External Tools option.

To create a new External Tool, you have to open the **Run** menu and then select the 'External Tools' submenu and then again the **External Tools** option. This will bring up a dialog for managing your external tools. From here you can add any new processes or change the settings for the existing ones.

In a fresh Eclipse configuration you shouldn't have any external tools, so the first step will be creating one. We want to launch a process (program), so select the icon **Program** in the left pane of the dialog, and either right-click on it to bring up the context menu, or select directly the **New Launch Configuration** icon at this dialog's toolbar.

Now we have to enter the information about the process we want to launch. First you have to provide a meaningful name for this External Tool. In our example we can use **mongrel with timeout**. Next, we have to enter the location of the program to launch. When we start a Mongrel server, what we are really doing is launching a Ruby script, so we will have to select the location of our Ruby interpreter here.
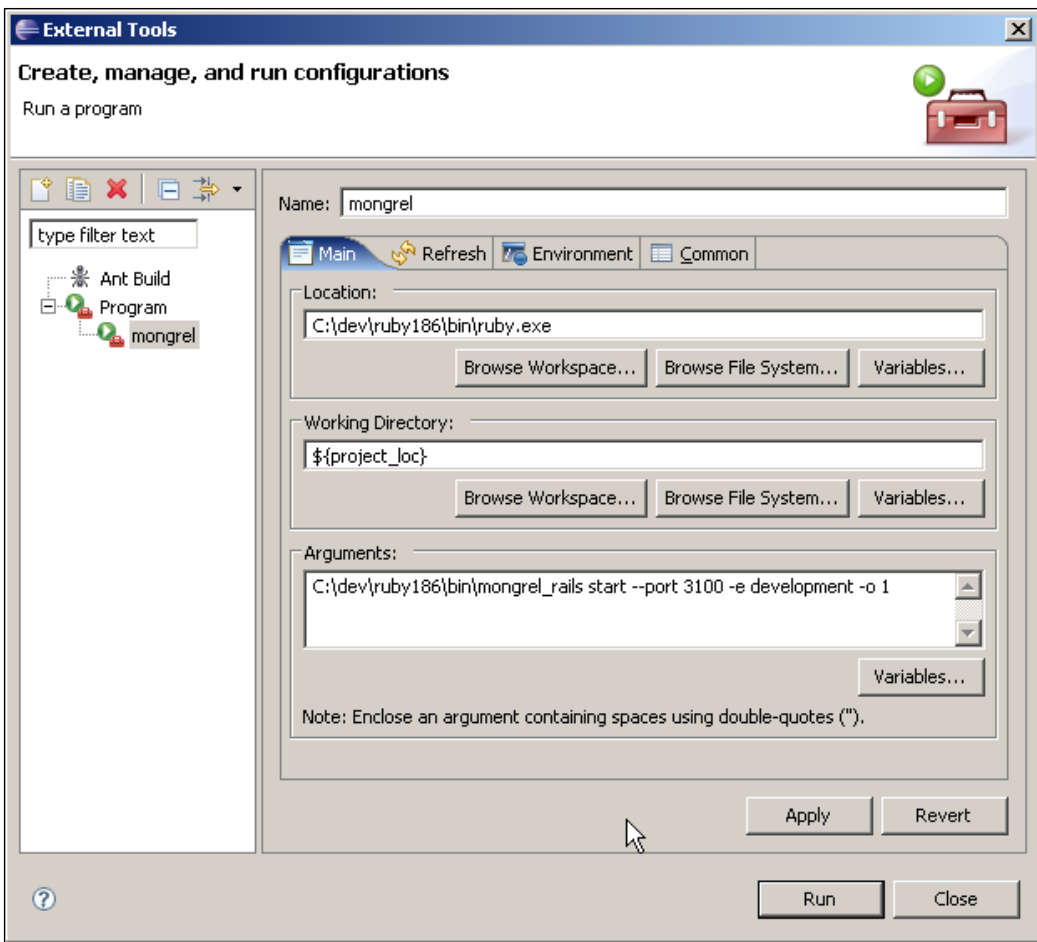
For the working directory, you can select the path of any of your projects or, even better, you can use the **Variables...** button. Eclipse provides some useful variables you can use. In our case, we will select the **project_loc** variable, which is the absolute path to the currently selected project. By using a variable instead of the path of a given project, you will be able to launch the same External Tool for different projects without having to write a different configuration for each of them.

Finally, we have to provide the necessary arguments for the process we are going to launch. In the case of Mongrel we will first need to specify the full path to the mongrel_rails Ruby script (which will typically be in your Ruby bin directory), and then the selected command line for starting the server. In my example I will be using the following arguments:

\dev\ruby186\bin\mongrel_rails start --port 3100 -o 1

We are telling mongrel to start a server at `port 3100` with a timeout of 1 second. By using the Servers View you would not be able to set the timeout (which anyway is OK for practically every development scenario).

After setting these properties, you can select the **Apply** button and then **Run**, at the bottom of the dialog box. Mongrel will start, presenting the output in the **Console** view as if you had started it from the **Servers** view. If there are any errors, you can check them in that view and correct the configuration accordingly. If you want to stop your process, use the **stop** button in the **Console** view.

Once you have configured the process, you can directly launch it from the **Run** menu under the **External Tools** option. The name of your process will appear as one more option of this menu. You can also use the **External Tools** icon at the workbench toolbar.

For configuring Mongrel, we only need to set the options at the main tab of the External Tools dialog, but if you are going to use the External Tools dialog for configuring other processes, you might find interesting the options tabs we didn't use in our example.

These tabs have options for logging the output to a file, hiding the console output, setting the character encoding for your process, setting environment variables, or even refreshing your workspace after the process finishes (in case it might create files or modify the existing files).

Rails Console

Coming from a background of languages like C++ or Java, one of the things I first loved about Ruby/Rails was the IRB-based console in which I could execute code and have immediate results. After two years working with Rails, that's still one of my favorite features of this development environment, allowing me to develop much faster by being able to try things on the fly in a trial-and-error manner.

Fortunately RadRails provides an integrated Rails console so we can launch it without going to the command line. To open the Rails console, there is an icon representing a system shell on the workbench toolbar. That icon will open by default the console in the development environment. Should you want to start in production or test environments, you can click on the arrow by that icon and select the appropriate environment.
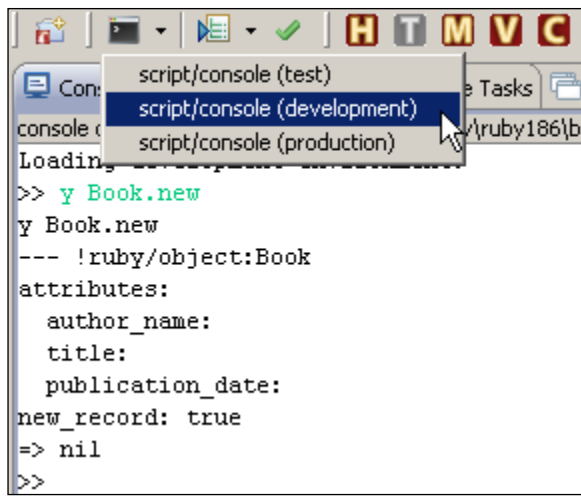
The Rails console works exactly as from the command line with one single limitation. RadRails internally uses the Console view of Eclipse for displaying the Rails console, and the Eclipse console uses the arrow keys for moving the cursor up and down the output area, so you cannot move through the console's command history with the up and down arrow keys.

At the time of writing this chapter, there are plans to make the cursors work as expected in this view, so it's likely they will behave like that by the time you are reading it. If not, you can get similar results; you can select the previous commands and just copy and paste.

Rails Plugins View

You know how it is with software development these days. For almost anything you want to do, there is someone who has programmed it already (well, at least to some extent). One of the tasks when starting a new project is choosing which components to incorporate. In Rails, one of the most common ways of adding software components to a project is by using plugins.

Plugins are a perfect way both for adding functionality from external sources to our projects and for encapsulating our own libraries either for internal reuse or for making them freely available.

RadRails provides a convenient way of installing/removing plugins to your project. The **Rails Plugins** view is displayed by default in the Rails perspective, so you don't need to explicitly open it from the **Window** menu.
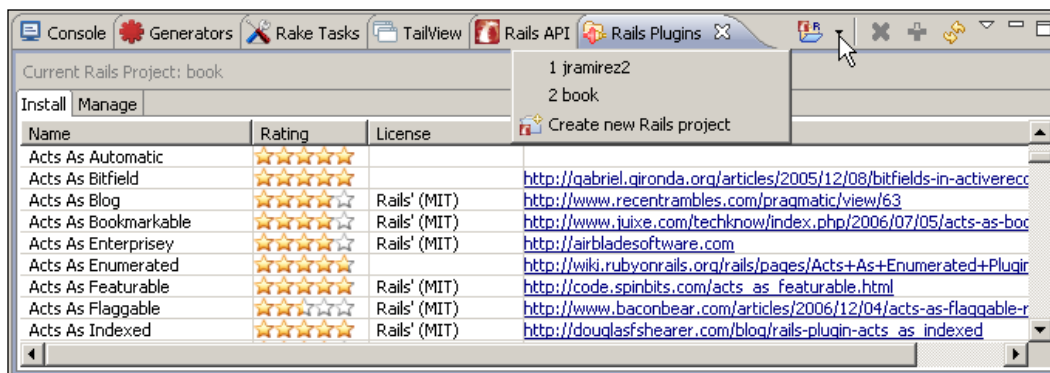
When you are installing or removing a plugin, you will be doing it over an existing Rails project, so there must be a way of telling RadRails to which project you will be applying the changes. By default, the currently selected project in the Ruby Explorer will be the target of the changes. To be sure, you can see a label above the list displaying **Current Rails Project:** with the name of your project. If you want to change the project, you can either select a new one in the Ruby Explorer, or you can use the **Select Rails Project** icon in this view's toolbar, as displayed in the above screenshot. This icon is a nice addition incorporated in Rails 1.0 and will be present in all the RadRails views in which the changes are applied locally to a single project.

If after selecting your project you cannot see a list of plugin names in this view, you can use the **refresh** icon in the toolbar. When you refresh the list, RadRails will connect to a public plugin repository to compose the list, so allow some seconds until the list is ready.

At the time of writing this book, there was no way of telling RadRails to add new plugin sources or to manually install a plugin from an unlisted location. For any of these operations you will have to go to a command line and perform them manually, or use the convenient Rails Shell view, explained later in the chapter.

After the plugin list is loaded, you will see the name of the plugin and when available the rating, license, and home URL. You can use the column headings to change the sort order of the list. If there is a home URL, you can click on it and the web page for the plugin will be loaded in the internal browser.

If after examining this information you want to install the plugin, you can use the **Install** icon in this view's toolbar, or you can directly select **Install** from the context menu.

When you install a plugin from the command line, you can choose two options related to the subversion repository management: use externals or perform a check-out via subversion. If you want to apply any of these options, they can be toggled from the pull-down menu (the big down arrow in the upper right of the view).

Using externals will tell subversion to download the code locally, but every time you check for updates the original plugin repository will be checked. By using checkout, you will force the plugin install script to install via subversion even if the repository is accessible directly via HTTP. Take into account that you will need to have subversion installed in your system if you want to use any of these options.

So far we have been working in the list displayed when opening this view, but if you look again you will see there are two tabs available in the view: **install** and **manage**. Under **manage** you will see the list of plugins installed in your project. If you want to remove any of these plugins, just select it and use the **remove** icon in the view's toolbar.

RubyGems View

Ruby gems are another way of encapsulating software components to reuse them in different projects. Actually, Rails itself is packaged as a set of several gems (ActiveRecord, ActionPack, ActionMailer, ActiveResource, ActiveSupport, and Rails).

Unlike plugins, gems can be used in a Rails project or in a stand-alone Ruby script. Besides, the gem package manager allows easy installation and management of the installed gems.

The main disadvantages of gems over plugins are that installing gems typically requires access as an administrator of the machine, and that they are shared by all the rails projects in that machine. On the other hand, since Rails plugins are installed individually in each project, deploying to a different server is easier since everything is auto-contained. Also, if you want to modify the code of the plugin, you can do so without interfering with any other projects.

In a real-life Rails application, part of the functionality will be provided by Rails plugins and part of the functionality will be provided by Ruby gems. This is specially the case for components that require binary libraries, such as database drivers, for example.
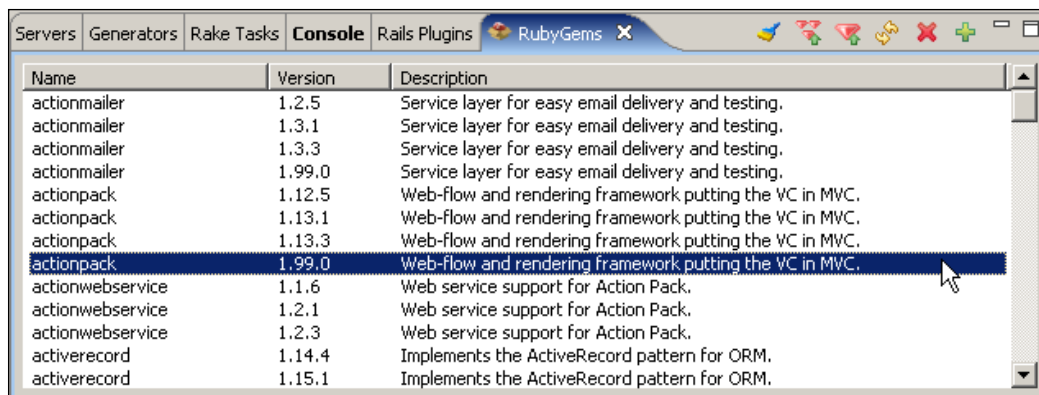
The first thing you will see when opening the **RubyGems** view is the list of gems installed in your system, their versions, and a short description. Don't worry if you see in this list several versions of the same gem; that's perfectly normal. In the same machine you can have different versions of a gem, like for example Rails 1.2.3 and Rails 2.0.1. If you cannot see the list of installed gems, you can hit the **refresh** icon in this view, also available from the context menu, to force RadRails to update the view.

We will now explain the different icons in this view's toolbar from left to right. The first icon, represented with a brush, is the **Cleanup Gems** option. If you select it, you will remove the older versions of the gems in your local installation. Only the most recent version of every gem will be kept. Generally speaking, it's not a bad idea to clean up your gems once in a while, but before doing it make sure you don't have any application depending on an old version of any gems.

The next icon is the **Update All Gems** one. As you can imagine, when you select this option, the gem package manager will check if there are any new versions of any of your gems, installing them if appropriate. This method is safe since it will not remove any older versions. It will also try to update **rubygems** itself, unless you are running on JRuby, since at this moment doing so tends to break JRuby's **rubygems** installation.

The next icon, labeled **Update Gem**, will also check for new versions but only for the currently selected gem. If you don't select any gem in the view, this icon will appear disabled.

The next option is **refresh**, and we already know it's used for reloading the gem list information from the local installation. The next option is **Remove** and it's no surprise that using it will remove the currently selected gem for the selected version. As with the **Update Gem** option, if no gem is selected this icon will be disabled.
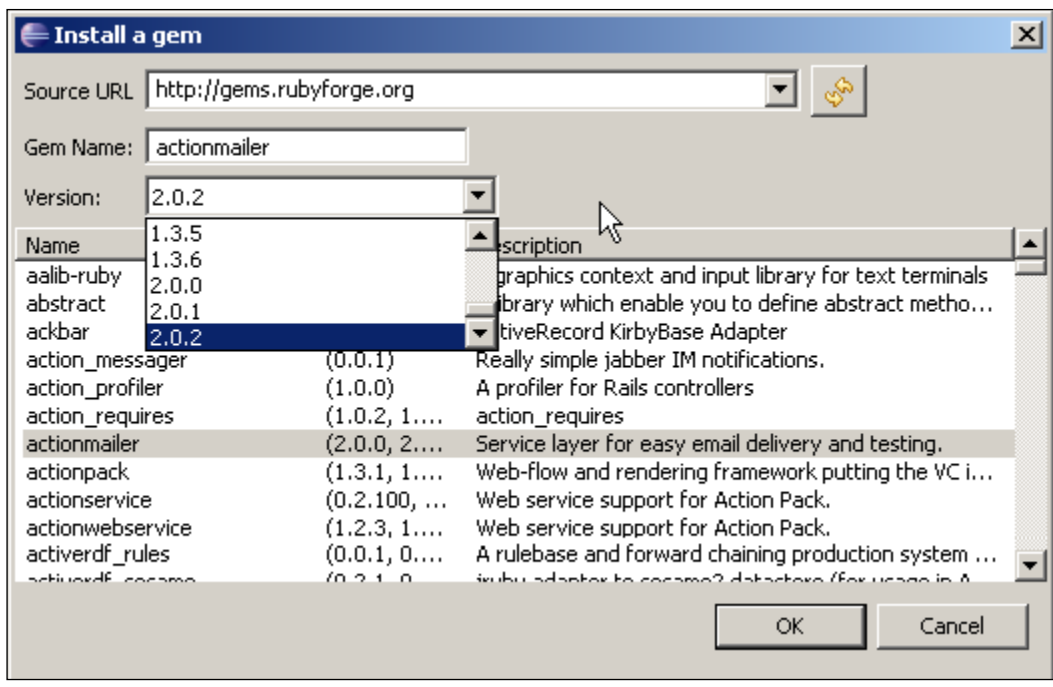
The last of the options is **Install**. When you select this icon, a new dialog will pop up. The first field is the **Source URL** for the gem you want to install. Gems are usually hosted at `http://rubyforge.org`, so you won't probably need to change the default location. If for any reason your gem source URL is a different one, you can enter it in the box and hit the **Refresh** button.

Refreshing the list of gems can take some seconds, so be patient until RadRails gets all the information needed. After a while, you will see the list of gems available for installing. You can filter the results by typing in the **Gem Name** text box. If you type, for example, **acts_as** you will see the list of available gems starting with those letters.

Even if you type the exact name of the gem, you still have to select it from the gem list so RadRails can refresh the drop-down list with the version numbers. Only after selecting the name and version number of the gem, can you proceed to install the gem. If you try to install without selecting a version number, you will receive an error message and nothing will be installed.

Apart from the output at the Console view, you can that check the gem was properly installed by checking the RubyGems view, where the installed gem should now be listed.

Rake Tasks

As happens with many other development environments, in a Rails application it is usual to launch different processes for manipulating files (cleaning log files, for example), initializing data, and so on. Usually these processes have dependencies between them and it's useful to have a tool to deal with these issues in a comfortable way.

In languages like C or JAVA this is done with popular tools such as `make`, `ant` or `maven`. In Ruby, we have the `rake` utility, which from Ruby 1.9 and in some older distributions too is packaged together with the standard Ruby installation. If `Rake` is not installed in your system, you can install it directly like any other gem.

Rake can launch different processes while keeping track of the dependencies between them. In Rake, every process you want to launch is called a 'task'. One advantage of Rake is that, unlike other platforms in which you need to learn a new language, the code you use for writing your tasks is pure Ruby code.

It's no surprise then that Rails provides a smooth integration with Rake tasks and that in a common Rails application, many management operations are achieved via such tasks. Out of the box, Rails provides tasks for cleaning up files—logs, cache or temp files—, for interacting with the database—creating the schema, moving between migrations, rolling back, etc.—for launching our test suite, or for generating documentation and managing Rails versions.

Apart from the built-in tasks, it's common to create different tasks in our projects for initializing data, running scheduled jobs, getting data from external systems, interacting with processes such as indexers or cache systems, or any other operations we need to launch automatically in the background of our application.

If you just want to run a Rake task without any parameters, you can directly right-click on the name of your project in the Ruby Explorer and select **Rake** from the context menu. A menu displaying the available tasks will open for you to select the one you want to launch.

If you want to pass any parameters to your tasks, or if you want to see their descriptions, RadRails provides a specialized view. The **Rake Tasks** view is available by default in the Rails perspective, so you don't need to do anything special to open it, just select its tab to display it.
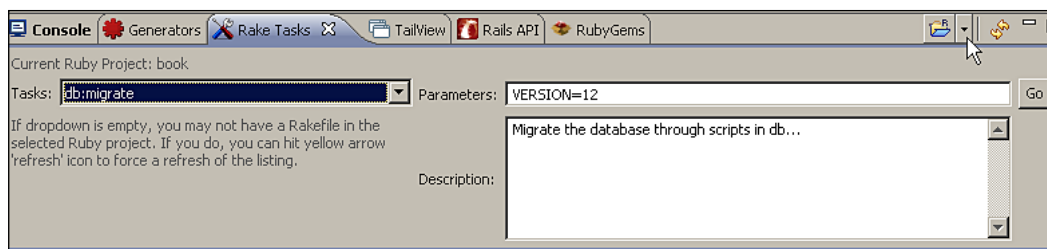
The drop-down list of this view will show the available tasks for the currently selected project in the **Ruby Explorer** view. If you change the selection to a different project, the task list should reload. If it doesn't, you can use the **refresh** icon in the toolbar. Also, as in the **Plugins** view, you can use the **Select Ruby Project** icon instead of selecting your project from the **Ruby Explorer**.

When you select any of the tasks in the list, a short description will appear in the big text box at the right of this view. If you want to launch that task without any parameters, just click on the **Go** button. If you want to pass any parameters, you can use the long text box at the top of the view. For example, in the screenshot above, I'm passing the parameter **VERSION=12**.

The most common use scenario is launching a task with or without parameters, but sometimes you want to launch a task and pass extra parameters to the `rake` command itself. For example, suppose you want to pass the argument `describe` to `rake`, so it will output in the console the full description of the tasks instead of executing them, or maybe the `trace` option for debugging purposes. In those cases, you must use not this view but the Rails Shell view, explained later in this chapter.

Generators View

There is no question about Rails being a framework oriented to programmer's productivity. You can get results much faster than with other tools of the trade. The reasons for this productivity boost are many.

In the first place we have the underlying language Ruby, which allows us to get the same things done by writing less code. Not only is the language important, but the Rails framework itself has been designed with a strong understanding of which things are important for most web applications, making these tasks very easy to develop and leaving the least frequently used features out of the framework.

Finally, another interesting productivity mechanism in Rails comes from the code generators. By using generators you can create code templates for many common operations. Out of the box, Rails provides generators for controllers, models, migrations, plugin skeletons, scaffolds, and resources.

You can generate the basic code with these generators and then adapt it to meet your needs. Even if the generated code is fairly basic and it requires modifications, it's a very fast way of setting up your application. As we saw when creating our example application, by using generators you can get a full maintenance application in a few minutes. And that already includes the (empty) methods for testing your models and controllers.

If you don't like the default generators, you can create your own totally customized generator for your coding requirements. The good thing about generators is that you only need to write them once and then, by packaging them as plugins, you can use them in any other project.

RadRails supports the use of code generators through the **Generators** view, which is available directly in the default Rails perspective.
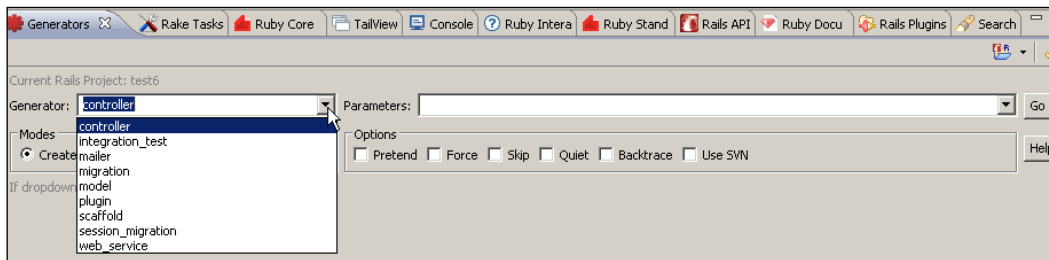
By now this kind of layout should be familiar already. First make sure you have selected the project in which you want to run the generator, and hit the **refresh** icon if the generators list doesn't get updated.

Now you can select the generator you want to execute from the drop-down list or you can just type the name in. If you want to pass any parameters to the generator, you can do so in the long text bar at the top of the view. Notice that this is a drop-down list, and if you unfold it you will see the history of recently used parameters, which can be handy sometimes.

The script for running generators accepts some arguments from the command line. In this view there are some checkboxes so when you select them those arguments will be passed to the generator. In this way you can just pretend to run the generator so you can see the output without actually changing anything, skip existing files without prompting you, or use subversion to persist the changes, for example.

Rails Shell View

The Rails Shell view is a good example of why I prefer RadRails over other existing IDEs. Someone asked in the Aptana forums if RadRails could incorporate a nice feature he saw in a different IDE. Chris Williams, the lead developer of RadRails, asked him to enter the desired features into the **Aptana Issues Tracker**, where you can report any bugs or ask for any functionalities you would like to see in RadRails. Some months later, in RadRails 1.0, the Rails Shell view was already available.

From the Rails Shell view, you can have the best of two worlds: the power of the command line and the ease of use of a development IDE. You can basically run rails-related commands from a command shell with content-assist.

The commands you can execute from this view are: `rails`, `gem`, `rake`, `script/about`, `script/console`, `script/destroy`, `script/generate`, `script/plugin`, `script/runner`, and `script/server`.

If you are not familiar with using these tools from the command line, maybe you will not use the Rails Shell view very often. But if you are familiar with these commands, you might find yourself more comfortable with the Rails Shell than with the GUI views. As an additional advantage, from the Rails Shell view you can pass any extra parameters to these commands. You could, for example, manually install a plugin or a gem not listed in the **Plugins** or **RubyGems** views.

This view doesn't have a tab of its own, but it's accessed as a part of the **Console** view. Go to the **Console** view and open the drop-down list by the **Open Console** icon. From the list, select the **Open a Rails Shell** option.
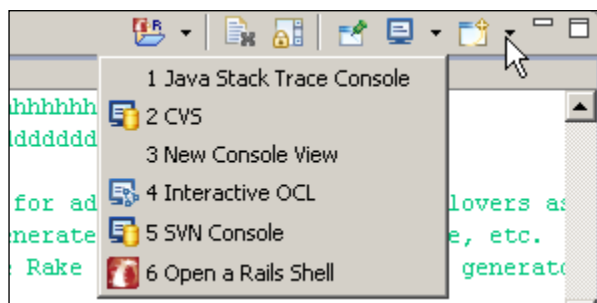
By default, the Shell will be active for the currently selected project in the Ruby Explorer. As usual you will see the current project in a label at the top of the view, and you can use the **Change Active Project** icon to select a new project. Alternatively, you can directly type **switch** or **cd** in the **Rails Shell** view and a list of the available projects will appear for you to select one. Note that only the open Rails projects will be available.

If you click *Ctrl* + *Space* (or *Command* + *Space* in Mac), the list of available commands will appear.
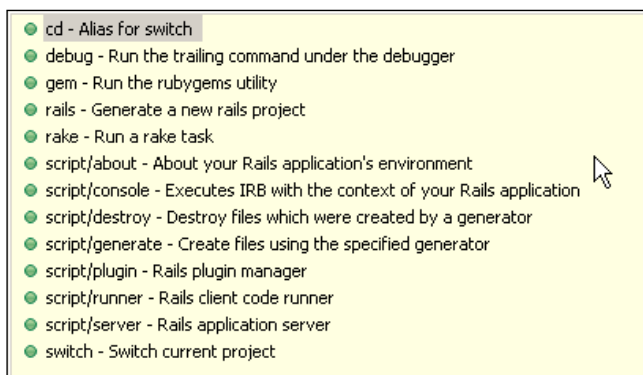
Now you can either select the command directly from the list or start typing so the list will filter the matching commands. Every time you type a command, a new content-assist window will display with the suggested options for that command. For example, if you type **rake** and then *Space*, you will see a list of the available tasks for your project. Be careful **not to hit** *Enter* after typing **rake** without any arguments. By default, the `rake` command without any arguments will try to run the tests for your project.
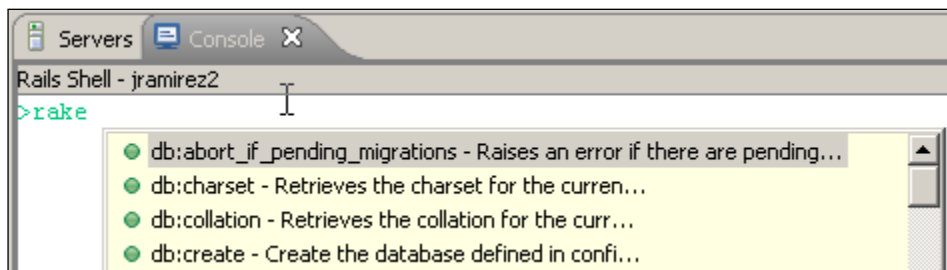
It's worth noting that some commands accept more options or flags than the ones appearing as suggestions. Even if they don't display in the list, if you pass those extra flags they will have the expected effect. You could for example execute a Rake task with trace (debug) information by typing:

```
rake –trace stats
```

You can use this view also for starting a rails server via the `script/server` command. If you want to start the server in debug mode, you can use the command:

```
debug script/server
```

In the professional version of RadRails, there is also a **profile** option for starting your server under profile mode. If you try to run the **profile** mode in the Community edition you will get an error message.

Finally, in the Rails Shell view it's possible to use the keyboard arrows to move through the command history.

RegExp View

It's not in every project that you need to write complex **Regular Expressions**, but when you have to do it, it's always nice to have a way for testing and refining them. In Ruby you can always open a console and evaluate your expression against a given string, but with complex expressions it can take a while to see why the string is not matching properly.

RadRails incorporates the **RegExp** view, which will help us test regular expressions and execute them step by step to see where the pattern is not matching your data.
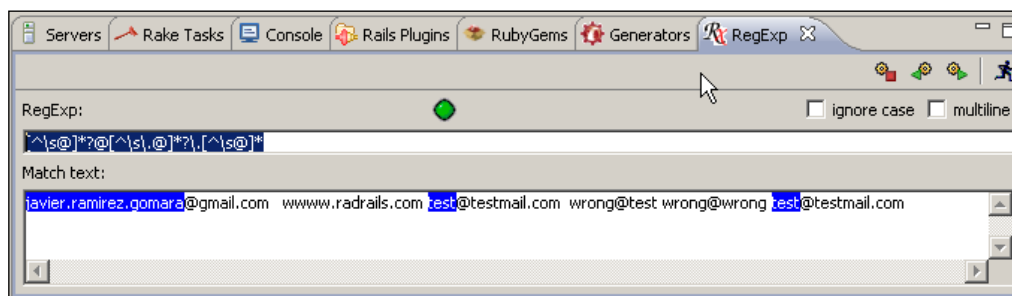
The **Regular Expression** box is for entering the regular expression, and the **Text to match against** box below is for entering the data to match it against. Once you write your expression, you can use the icons on this view's toolbar to evaluate it.

The icon most to the right is labeled **Validate RegExp**. When you select this icon, the circle at the top of the view will change colour to Green or Red. Red means either your expression was wrong, or it was right but no matches were found. Green means the expression was matched correctly against the text and there were some results.

To actually see the matches of the expression, you can use the **forward**, **backward**, and **reset** icons. When you click on **forward**, RadRails will break down your regular expression into its subpatterns and will display how every part is matching with your text. Every time you click on **forward** you will see the matches for the next part of your expression. This is a very interesting way of testing your patterns, since often a part of the expression is right but it doesn't match your text because of a subpattern. With this tool you can see how the different parts are being evaluated.

For example, in the preceding screenshot I wrote a simplistic expression for finding email addresses in a text. The expression is not complete, but for this example's purposes it will do. Basically I'm telling RadRails to find strings composed of several characters that are not spaces or '@', then the '@' symbol, then again some characters except blanks, dots or the at symbol, and finally a dot and again some characters. The pattern for this could be:

```
[^\s@]*?@[^\s\.@]*?\.[^\s@]*
```

And as the Match Text I'm using a combination of both valid email address and non-matching text:

```
javier.ramirez.gomara@gmail.com    wwww.radrails.com test@testmail.com
wrong@test wrong@wrong test@testmail.com
```

If you execute the expression by using the **forward** icon, you will see how the first part of the expression will match the first part of the three legal email addresses in that text, then the next part will match the three at symbols, and so on.

If you prefer to see the whole matches instead of the partial results, you can use a trick. Surround the whole expression with parentheses and then you can use either the **Validate RegExp** or the **forward** icon. In any case you will get the full matches for your expression. In our example, it would look like this:

```
([^\s@]*?@[^\s\.@]*?\.[^\s@]*)
```

When I'm writing regular expressions, I like to go step by step until I have it right and then use the parentheses to make sure the whole strings are being matched properly.

Finally, the two checkboxes you can see in this view are used for telling your expression to be case-insensitive or to match expressions even if the strings are divided by a line break.

Problems View

When writing your code, if you have a syntax error you quickly fix it because Eclipse will warn you, and because if you don't, then your script will not run. If we are not talking about errors but warnings, even if Eclipse warns us, we usually are much more tolerant and we tend not to pay enough attention, thinking we can always fix that later. By the time, and if, we want to clean up our code, we will most probably have lost track of where the warnings were happening.

By using Eclipse **Problems** view, we can see a list of all the places in our code where we have unresolved warnings. The list of warnings is configurable, as we will see in the next chapter, and can include warnings about deprecations, empty blocks, variable names, common possible errors, unused variables, and so on.
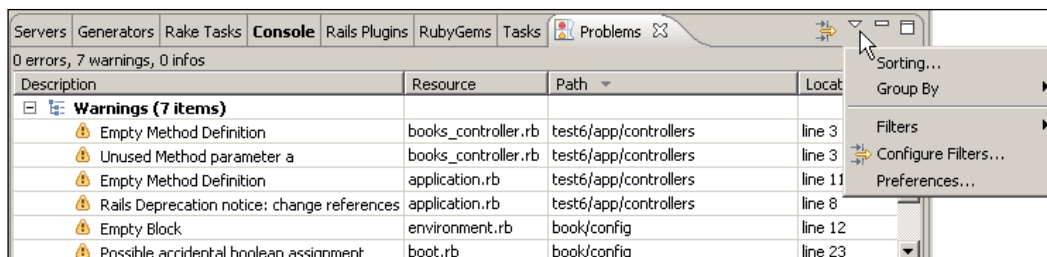
If you open the **Problems** view, you will see a list of the current warnings for your project. If you cannot see any warnings, open one of your controllers and just define an empty method that receives some variable you will never use. A definition like this would do the trick:

```
def empty_method(a)
end
```

After you save the file, you will see two warnings in your Problems view, one about having an empty method definition, and another one about an unused variable. If you double-click on a line of this view, the corresponding file will be opened in the editor and the cursor will be placed at the line containing the warning.

As you can see, the list gives you information about in which file the problem is happening, the name of the container file, the relative path in the workspace, and the line number. You can sort the list on any of these columns by clicking on the column name. Thus, you can group together errors with the same description, in the same project, or starting with the same path.
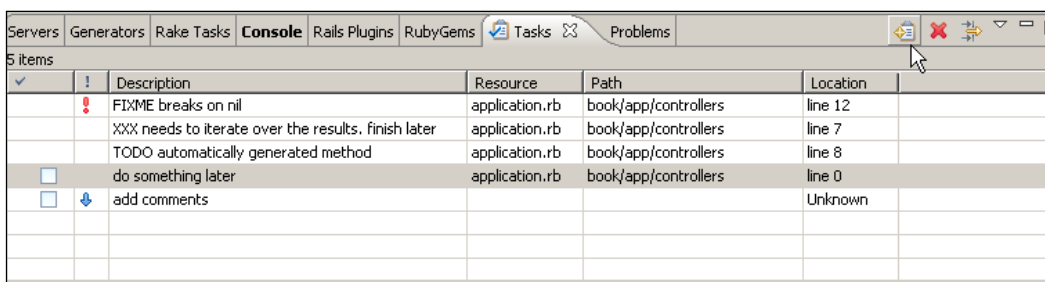
There are some extra options in this view. If you click on the view's menu icon (as shown in the above screenshot) you will see these additional options. If you want to sort the lines by more than one column, by description and then by path, for example you can use the sorting option. The **Group By** option of this menu will not have any effect here as it makes sense only for developing Java projects with Eclipse. Use the **Preferences** option if you want to limit the total number of warnings rendered.

Finally, the **Configure Filters** option, also available as an icon on this view's toolbar, will allow you to filter warnings only for the current project, the current selection, or the whole workspace (the Window Working Set). By default you will be presented with warnings for all the opened projects in your workspace.

Tasks View

When you are working on a project, you sometimes leave incomplete methods or things to fix later. As you know, it's a very good practice to write a comment in the code so you will not forget you have to finish or fix that code in the future. A common coding convention is writing a comment with the word **TODO** (or **XXX**) for incomplete functionality and with the word **FIXME** to mark a piece of code that must be fixed.

RadRails supports the use of these coding conventions, and offers the **Tasks** view in which you can see the list of annotations in your code and in which you can also set to-do items manually.

The menu of this view is very similar to that of the **Problems** view, the main difference being that in this case you can filter out results by using the priority column too.

First of all, we are going to set some annotations in our code, so we will see some entries in this view. Open the file `books_controller.rb` and at any place in the code write the following comments:

```
#XXX needs to iterate over the results. finish later
#TODO automatically generated method
#FIXME breaks on nil
```

After saving the controller, you should see three entries in the **Problems** view. Each line displays the type of annotation as well as the associated comment. You will notice that at the left of the **FIXME** annotation there is a **red** exclamation mark. This is because the priority is automatically set to **High** on **FIXME** and **normal** on **TODO** and **XXX** annotations.

Just a quick note here. Since Rails 2.0, there are some Rake tasks available for searching for annotations in your code. The recognized annotations in these tasks are **FIXME**, **TODO**, and **OPTIMIZE** (also supported by RadRails). Annotations with **XXX** syntax are a common convention amongst software developers, but since they are not supported by the Rails tasks it could be advisable not to use them. Of course nothing will break if you do, but by using only the annotations Rails understands, you can be sure people using a different IDE (or even no IDE at all) can take advantage of your code annotations.

You can add your own annotations or change the priority of the existing ones from the **Window** | **Preferences** dialog. We will see how to do this in the next chapter.

Apart from using this view for code annotations, you can manually add to-do items. You can right-click on the content area of this view and select **Add Task** or you can use the **Add Task** icon of the toolbar. A dialog will appear for you to enter the description of the task, the priority you want to assign, and a checkbox indicating whether it is completed or not. If you create a task with High priority, it will display the red exclamation icon, and if you assign low priority, it will display a blue down-arrow.

Once a task is set manually, you can click on the task description or in its priority in the Tasks view and edit the information directly on the list. You can also check the task as completed at any time you want.

When you add a task manually it is not associated to any resource. If for any reason you want to add a task and you want to assign it to a source code file, you can open the file you want in the editor and then select the **Edit** menu and then the **Add Task** option. You will see the same dialog as before, but the **Resource**, **Folder**, and **Line** fields will be filled. You can also add an associated task to a resource by right-clicking on the left margin of the editor and selecting **Add Task** from the context menu.

Test::Unit View

There is always an excuse not to write your tests: the deadline is close and you don't have the time, the requirements are not clear so it's difficult to write the test code, generating data for testing is not always easy. When working with Rails, some of those excuses lose strength, since the framework facilitates the preparation of the test database, the generation of the testing suites, and the execution of the tests themselves.

Because of that, the proportion of developers writing applications in Rails who systematically write tests is larger than in other development environments. Actually there are many developers who write the tests even before than the code itself. Of course this practice is not particular to Rails, but the framework makes easy to adopt it.

Since testing is such a relevant part of the Rails philosophy, it's only natural that RadRails provides a specialized view to help us test our application. This view is called **Test::Unit** view and it appears as a tab by the Ruby Navigator in the default Rails perspective. Whenever a test suite is run from within RadRails, the **Test::Unit** view will display the results of the execution, allowing you to examine the details.

There are several ways of running your tests from RadRails. The easiest way is to navigate to a unit test file (under the **test/unit** directory of your project), right-click on the name of the unit test file you want to execute, select **Run As**, and then **Test::Unit Test**. Before running your test, make sure your test environment is configured properly (database configuration, fixtures, and unit test code).
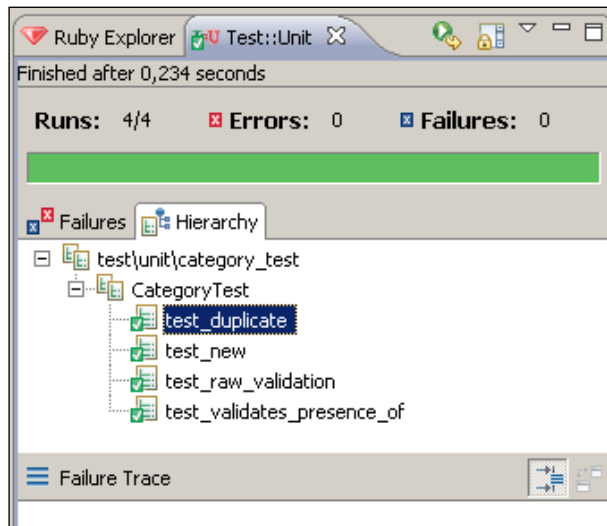
After the test is run, the **Test::Unit** view will present the results. Near the top of the view you will see a bar either in green or red color. Green means all the tests passed correctly, and red means there were failures or errors. A failure is reported when one of the tests didn't pass, and an error is reported if an exception was launched when trying to run a test.

Above the bar, you can see how many tests were run, how many errors you got, and how many failures. Below the bar there are two tabs. In the first one you will find only the list of tests with errors or failures, and the second tab labeled **Hierarchy** will display the list of all the executed tests, both successful and not. If you double-click on any of the items, the corresponding file with the definition of the test will be opened in the editor area.

When you select an item with errors or failures, the **Failure Trace** pane of the **Test::Unit** view will display the details of the failure, or the Stack Trace in the case of an exception.

In the toolbar of this view there is an icon for relaunching the last performed test, another one for locking the scroll (in case launching big test suites producing a large output), and the menu for this view, in which you can change the layout.

So far we have launched a single unit test suite, but as you know in Rails you can also use functional and integration tests. For running all the unit tests in your project, or for running the integration or functional tests, you have to use an icon located on the toolbar of the Workbench.
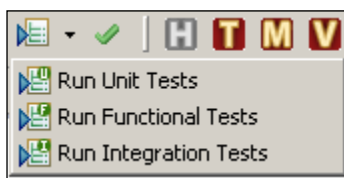
If you hit the icon labeled **Run All Tests** all the Unit, Functional, and Integration tests for your project will be run. If you want to launch only one test type, click on the small arrow by that icon so you will get a submenu allowing you to run the different test suites. No matter in which way you launch your tests, the output will be displayed at the **Test::Unit** view in the same way as we saw before.

You can also choose to execute your tests automatically. In this case, the tests will be run either every time you save a file or at a given interval of time defined by you. If you want to run your tests automatically you have to configure RadRails for that. Go to the **Window** menu, select **Preferences...** and then navigate to the **Rails | Autotest** dialog.
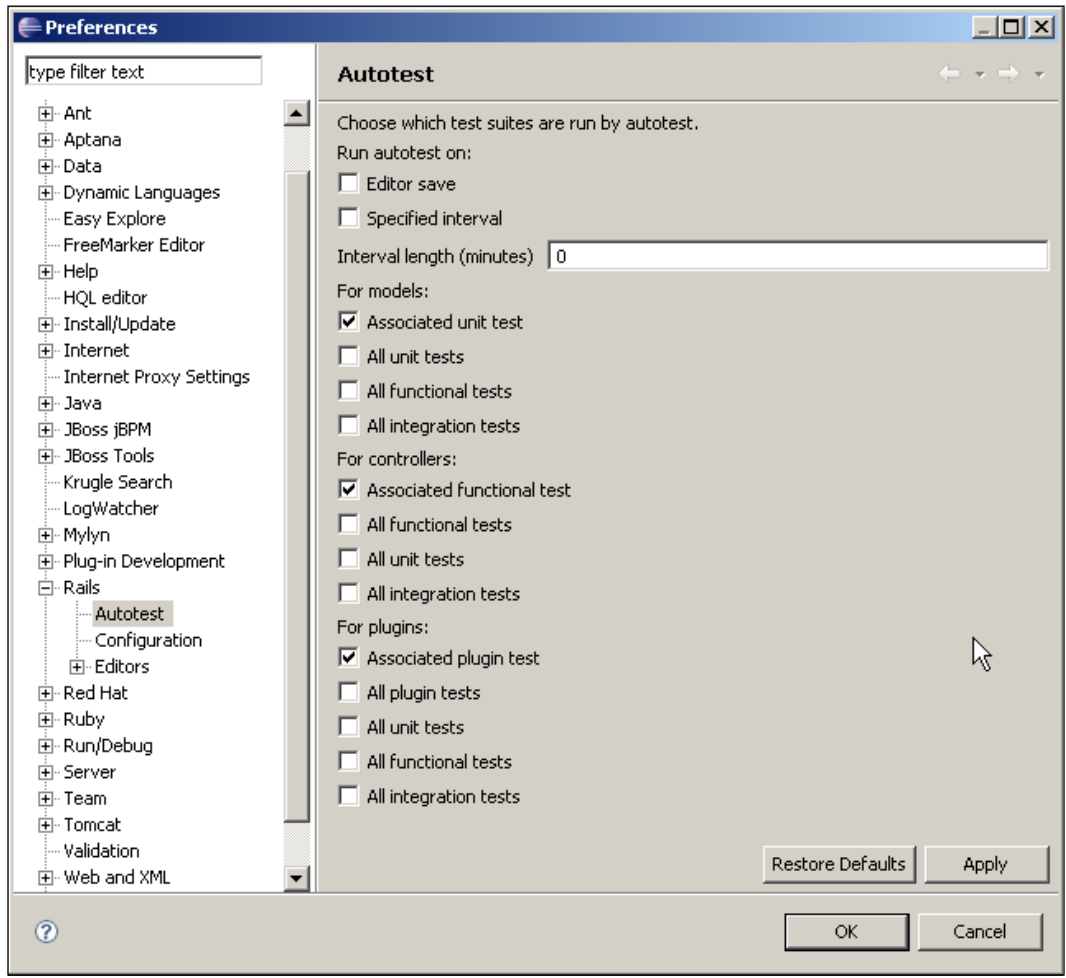
The first thing you have to tell **Autotest** is how it will be launched: after saving a file in the editor or at a regular interval. The two options are mutually compatible, so you could mark both of them if you want.

After choosing when the **Autotest** will be run, you have to instruct RadRails about which tests to launch. By default it will try to launch only the associated unit tests for the model, controller, or plugin file you are saving. Modifying any other files will not launch Autotest, which is one of the reasons why you could possibly want to launch the test suite at regular intervals.

You can tell **Autotest** to run not only the associated unit tests, but also all the unit tests for your project, all your integration tests, or all your functional ones. Once you set the options as you think is better for you, just click on **Apply** and then on **OK** to close the preferences dialog.

If you chose to **Autotest** after saving the editor's contents, you can try opening a model or controller, making a small change (maybe a blank character or a comment) and saving. At the bottom of your eclipse workbench you should see the status line informing you about the Autotest progress. When the test finishes, you can see the results as usual in the **Test::Unit** view.

There is an additional feature of Autotest. The icon on the toolbar workbench close to the **Run All Tests** one labeled **Manually Run Autotest Suite** will remain in green if all the tests were passed, or will change to a white cross over a gray background in the case of errors or failures.

In order to get your attention, for some seconds this icon will display an animation of a small yellow blinking cross. In the figure below you can see the four possible images for this icon.

When editing the contents of a model, controller, or plugin you can use this icon no matter whether it's displaying in green or gray and force running of the associated tests (as configured at the Autotest preferences) even without saving the file or waiting for the established interval.

Summary

This chapter explained how you can use RadRails for dealing with many of the development tasks you would otherwise have to run from the command line in a much more convenient way.

Except for exceptional occasions when you need to pass uncommon arguments to the command-line tools, you can manage all your Rails development processes from within the IDE by using the built-in views. If you find yourself going frequently to the command line to launch some processes, we also learned how you can call external tools from Eclipse, so you can have everything just a click away.

By using RadRails views, you can manage documentation, servers, the Rails console, plugins and gems, Rake tasks, code generators, code annotations, warnings, to-do tasks, regular expressions, and test suites.

8
Configuration Reference

We already know everything there is to know about using Eclipse with Aptana RadRails for developing Ruby on Rails applications. You know how to use the workbench, the different code Editors, and the specialized views.

As with any development tool, there are plenty of configurable details you can customize to make your experience more comfortable. Being a general-purpose IDE, Eclipse has a large number of preferences not generally used when developing Rails projects. So we will focus only on the most interesting options for our Ruby on Rails projects.

As we saw in Chapter 2, the configuration is managed from the **Preferences** dialog, which is the last option in the **Window** menu. This dialog is divided into two parts: a tree with the different categories or sections you can configure, and the preferences contained in each of these sections.

Since Eclipse is a very extensive IDE and you can add new plugins for your specific needs, the preferences tree can be different in different installations. Also, in different versions of Eclipse some preferences can be located under different sections. The preferences described in this chapter are valid for Eclipse 3.3. If you are using Eclipse 3.2, most of the options will not vary but in some cases you might find small differences.

To help you navigate the preferences and quickly find what you are looking for, you can use the search box at the top of the tree pane. This becomes especially handy when you know—more or less—the name of the preference you want to set, but you are not sure under which section it is located.
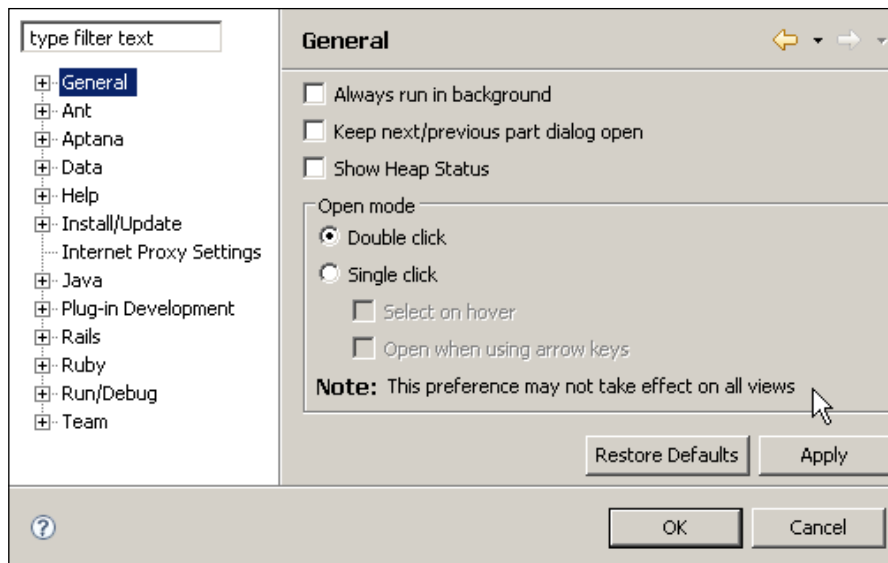
A few of the preferences will require you to restart Eclipse, or at least the workbench, in order to apply the changes. The preferences that need a restart are usually the ones about global changes to the appearance of the workbench, like changing the position of the tabs in the editors or using a different theme for Eclipse. In that case, a window will pop up informing you of this situation. You can either restart Eclipse directly from this pop-up, or just ignore it, keep working as usual, and restart it at a later time. However, unless you don't want to restart immediately because you want to keep adjusting more preferences, it's advisable to restart at this moment to avoid potential interface problems.

General

Under this section, we will find options that don't apply directly to RadRails, but to the configuration of the Eclipse workbench, such as the fonts and colors to use, the file extensions to recognize, or the keyboard shortcuts.

Appearance

When you click on **Appearance** you will see a pane with some general presentation details, like the position or the shape of the editor and view tabs.

The most important configuration under **Appearance** is the **Colors and Fonts** option. You can change the font style, size, color, and decoration, as well as the color for the background in the different areas of the workbench.
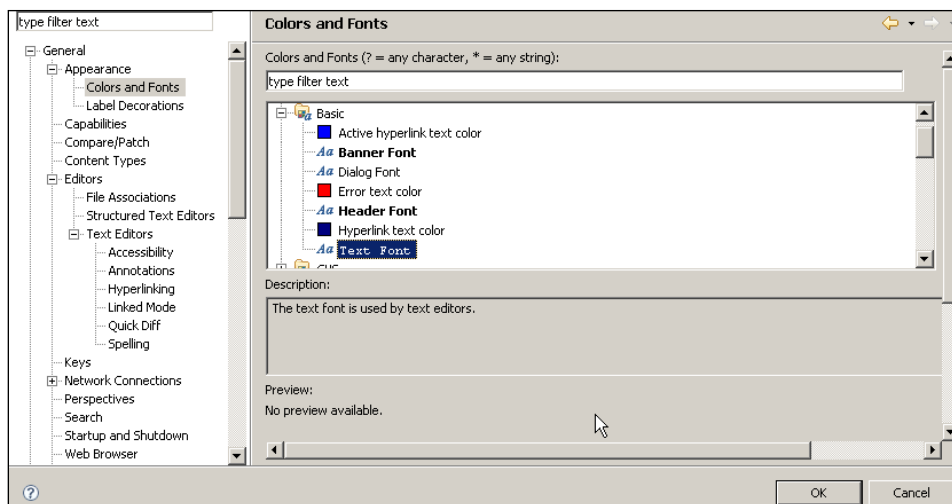
This configuration panel has a main area in which you can select the element you want to modify. As you can see, there are different groups of elements, represented as folders. As the name suggest, there are two kinds of elements to change: colors and fonts. Colors are represented with a solid square of the currently selected color, and fonts are represented with two letters and a title using the selected font.

One of the typical changes you might want to make is setting a different size or font family for the text in the editor view. You can do so by selecting **Text Font** under the **Basic** group and either double-clicking on the label, or use the **Change...** button. The classic font dialog will appear. In the case of changing a color, a color-selection dialog will pop up.

If you want to reset a single element to the default configuration, you can select it and use the **Reset** button by the right. If you went too far in your changes and you want to revert all the colors and fonts to the default configuration, you can use the **Restore Defaults** button at the bottom of the preferences dialog.

There is a description box in this panel, where you will see information about where the currently selected element is used in the workbench. For some of the elements—mainly for the ones under the **View and Editor Folders** group—there is also a preview available for you to see where the changes will apply. You can see the description and preview areas in the above screenshot.

Editors

Under the **Editors** section we can configure several options for working with the editor view. The main panel is pretty simple, and allows you to disable the tabbed interface for this view or set a maximum number of files to open.

As we have seen in the course of this book, Eclipse and RadRails provide different specialized editors, depending on which type of file we are going to modify. It's not the same editing a YAML file as it is editing an HTML or a pure Ruby script. The **File Associations** panel displays which extensions can be opened with which editors, and which is the default editor for every type.

You can add new extensions and associate them to the editor of your choice, or you can modify the currently configured associations, although this is rarely necessary, since RadRails will by default configure all the needed associations for Ruby, Rails, and Web development.

When setting an editor to handle a given extension, you can choose from the list of currently available editors in Eclipse, or you could select an external tool. In that case, when you open the file from Eclipse, the external editor will be invoked instead.

The **Structured Text Editors** preferences won't apply when developing with RadRails since they take effect only for the default non-specialized editors. Aptana provides custom editors that, as we'll see later, have their own preferences in a section of their own.

Next in the **Editors** section we can find the **Text Editors** preferences. From this panel we can tweak a bit more the general behavior of the editor view.
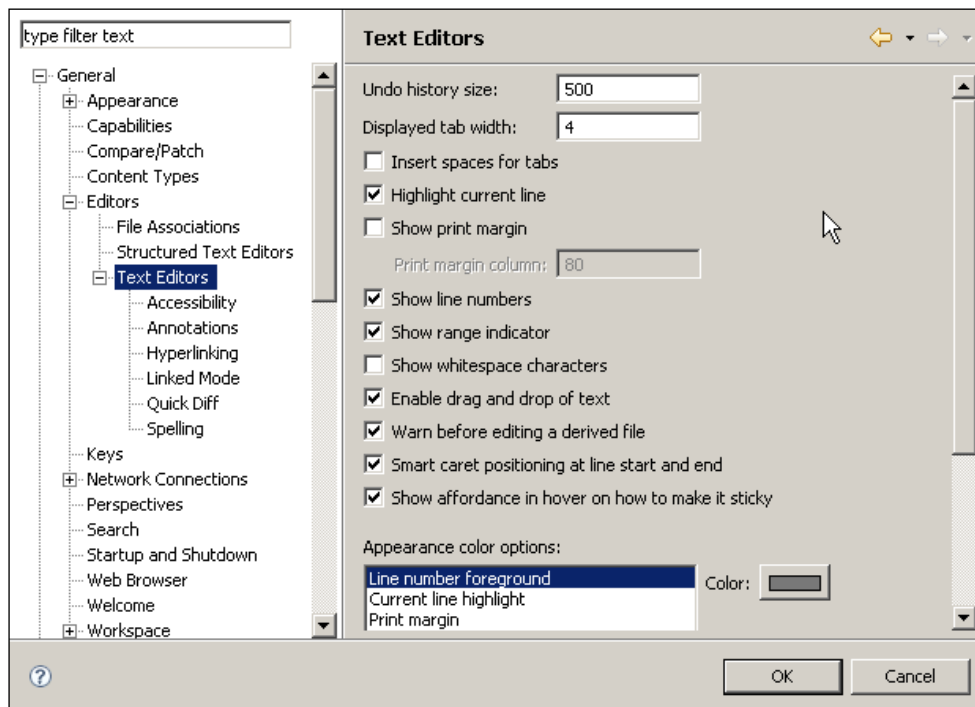
The first option will change the size of the history for the **Undo** command, although this is typically not necessary. Next, you can set the number of spaces to use in the editor for representing a TAB character. There is an option here for using spaces instead of TABs, but it has effect only on non-specialized editors, so changes here will do nothing when editing Ruby source files. We will see later (under **Ruby | Editor** and **Ruby | Formatter**) how to choose between using TABs or spaces for indenting your code.

By default, Eclipse highlights the current line you are editing. Should you want to disable this option, just uncheck the appropriate box. You can do the same for toggling on/off line numbers.

The **Range Indicator** is a little visual help Eclipse provides when we select a class, module or method name from the **Ruby Explorer** or any other outline view. When you select any of these elements, a **Range Indicator** will be displayed in the left margin of the editor view. This indicator is represented by a shadow to the left of the lines enclosed by the selected element.

Finally, from this panel we can set some specific colorization for text editors that is not available under the **Colors and Fonts** section.

Annotations

From **Annotations** you can change the colors and the way in which the different annotations will be displayed. If you select an annotation to be displayed in the **Vertical ruler** the corresponding icon will be displayed by the left margin (as happens in the lines where you have errors or warnings).

The **Overview ruler** means the right margin of the editor. By selecting 'highlighted' in the 'Text as' selection list, the text for the annotation will be rendered with a different background color. If you select 'squiggles' in that list, the text will be underlined with a wavy line.

Linked Mode

In Chapters 4 and 5 we learned how to use code and view templates in our editors. You only had to type the name of the template and use *Ctrl+Space* to insert the template in the editor area.

Most of those templates presented replacement sections, in which we could enter the values we wanted and that we could navigate with the *Tab* key. Eclipse uses the term **Linked Mode** to refer to this state in which you are working in the editing area but moving across the available links provided by the template.

When working with other languages than Ruby, Eclipse will enter **Linked Mode** also for some refactoring or **Quick Fix**, but in Ruby you will find this behavior only with the code templates.

In the **Linked Mode** preferences you can choose the colors to use for the different sections (called **Ranges** in this dialog). In the picture above you can see the effect of changing these colors when using a template.

Quick Diff

The next preferences we will go over are the **Quick Diff** options. By using this feature, we can see at a glance the changes we made to the current file with respect to the latest saved version, or to the latest version in the source repository (if we are using one).
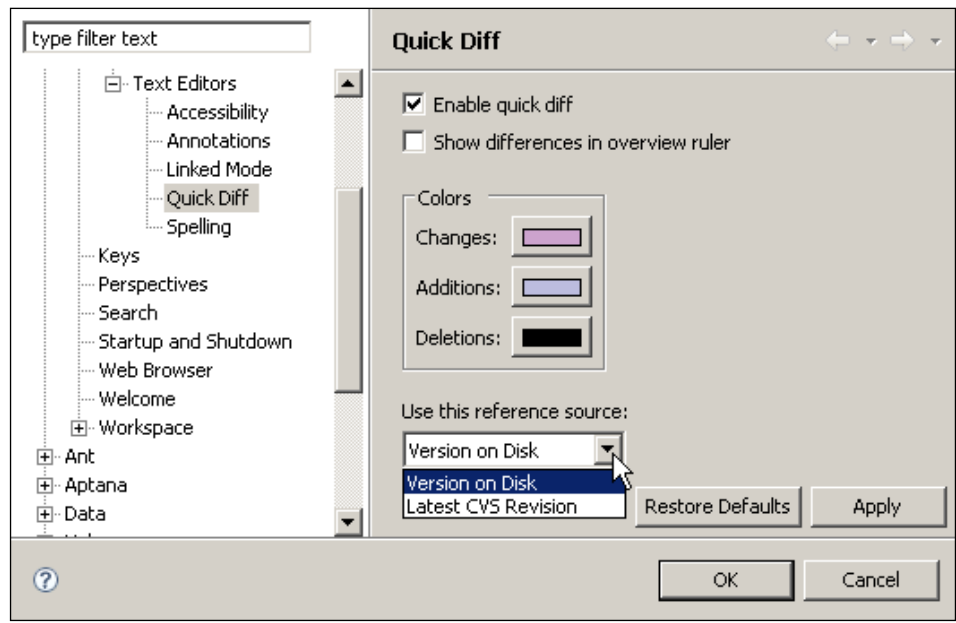
When **Quick Diff** is enabled, if you edit any file on the editor you will see a change in the colorization of the left margin as you are typing. By default a modification is marked by shadowing the margin in a magenta tone, an addition in pale blue, and a deleted line in black. If you want these changes to be displayed also by the right margin, select the **Show differences in overview ruler** option.

By selecting it from the combo in this panel, you can make **Quick Diff** mark the changes relative to the latest saved version or to the remote copy on your source repository. The list of available repository types will depend on which plugins for source control you have installed.

You can also toggle the **Quick Diff** feature directly from the editor view, by right-clicking on the left margin and selecting the **Show Quick Diff** option.

Spelling

Before finishing with the **Editors** settings, there is another one in which you could be interested. You have probably realized when you are writing your code (most noticeably within your comments) sometimes you get warnings about the spelling of the words.

Under the **Spelling** preferences in the **Text Editors** section you can configure how you want Eclipse to spell-check the contents of your Ruby editor. By default Eclipse will try to check your documents using a built-in dictionary with the syntax of Ruby. In some rare cases, Eclipse will not show this default behavior, and then it will warn you about spelling errors within the keywords of the language. In that case, you have to select the **Ruby Spelling Engine** at the **Spelling** preferences dialog.

The **Ruby Spelling Engine** will detect only Ruby keywords, so you will be getting a lot of warnings about the spelling of your comments. You can add a generic dictionary for the language of your choice, so you can get meaningful spell checking in your comments too. In future, Eclipse will bundle some dictionaries, but as of version 3.3, there are none available when you install it.

Fortunately, there are a lot of dictionary files available for other spell checkers that you can use here. Eclipse will understand any plain file with a list of allowed words. If you google a bit, you will find many files you can use. For the English language, there is a popular web page with different compilations of such dictionary files: the 'wordlist' sourceforge project at `http://wordlist.sourceforge.net`.

Should you want to turn spell checking off, you can do so by marking off the 'Enable spell checking' box. That will disable both the spell checker and the warnings.

Keys

Software developers are a special kind of users. They spend long hours working mainly with their development IDE, and in many cases they like to be able to use as many shortcuts as possible, not being as easily scared of a few keystrokes as a typical computer user. So a tool designed for software developers should be very keyboard friendly.

As we have seen in this book, many of the commands Eclipse and RadRails offer are available directly as a shortcut. Using the keyboard or the mouse depends on how often you use those commands and on your personal taste, but even the most compulsive mouse-users can benefit from the keyboard shortcuts or, as Eclipse calls them, key bindings. You can see and modify the currently defined bindings through the **Keys** option under **General** preferences.

Eclipse is a pretty extensive IDE, with a lot of commands, and the keyboard has a limited number of keys, so it would seem impossible to define key bindings for many commands. And if we think about extending Eclipse through plugins for adding new functionality, as Aptana does with RadRails, then it seems even more difficult.

Eclipse uses two different mechanisms in order to allow as many bindings as possible: contexts and key sequences. When developing with Eclipse, the context is defined by the area that has the focus. By using contexts, Eclipse allows us to assign the same key binding to different commands. For example, when editing a Ruby file, the shortcut *Ctrl+Shift+M* will open the associated model, but when editing a Java file, the same combination would add an 'import' statement.

The contexts in Eclipse are hierarchical, with the **In Dialog and Windows** context the most generic. Below that we can find **In Windows**, and then **Editing Text**, and then **Ruby Editor**, for example. If you install new plugins, they can define their own contexts too. If a key binding is not defined for the current context, Eclipse will go up the context hierarchy to see if it's defined on an upper level. Thus, different plugins can use the same key bindings without interfering with each other.

The second way in which Eclipse allows us to define a large number of shortcuts is by using **Key Sequences**. Typically, the keyboard shortcuts we use in any application are defined as a keystroke. A keystroke is the pressing of any 'non-special' key in combination, optionally, with one or more modifier keys. The modifier keys are *Ctrl*, *Alt*/Option, *Shift* and, only for the Mac, the *Command* key.

A key sequence is composed of one or more keystrokes. In Eclipse, a key binding is the association of a command to a key sequence. Most used commands are mapped to single keystrokes but, in some special cases, key sequences make sense, most usually when you have a number of related commands.

Run Ant Build	Alt+Shift+X, Q
Run Eclipse Application	Alt+Shift+X, E
Run JUnit Plug-in Test	Alt+Shift+X, P
Run JUnit Test	Alt+Shift+X, T
Run Java Applet	Alt+Shift+X, A
Run Java Application	Alt+Shift+X, J
Run OSGi Framework	Alt+Shift+X, O
Run Test::Unit Test	Alt+Shift+X, U

Every time you press a keystroke that marks the beginning of a key sequence, you will be presented with the possible keystrokes for completing the sequence. For example, you can hit the keystroke *Alt+Shift+X* and you will see a small pop-up at the bottom-right corner of the workbench. In our case, hitting *U* would run the Unit Test for the current class.

The number of keystrokes in a sequence is not limited in Eclipse, but common sense—combined with a limited number of fingers—says that if you define your own shortcuts, you should try to make them as short as possible.

Now you know everything that you need for understanding the **Keys** preferences dialog. This dialog can be used as a reference for finding which is the shortcut for a given command, for adding a shortcut to an unmapped command, for changing the current bindings (if you prefer a different combination), or to completely remove a binding.
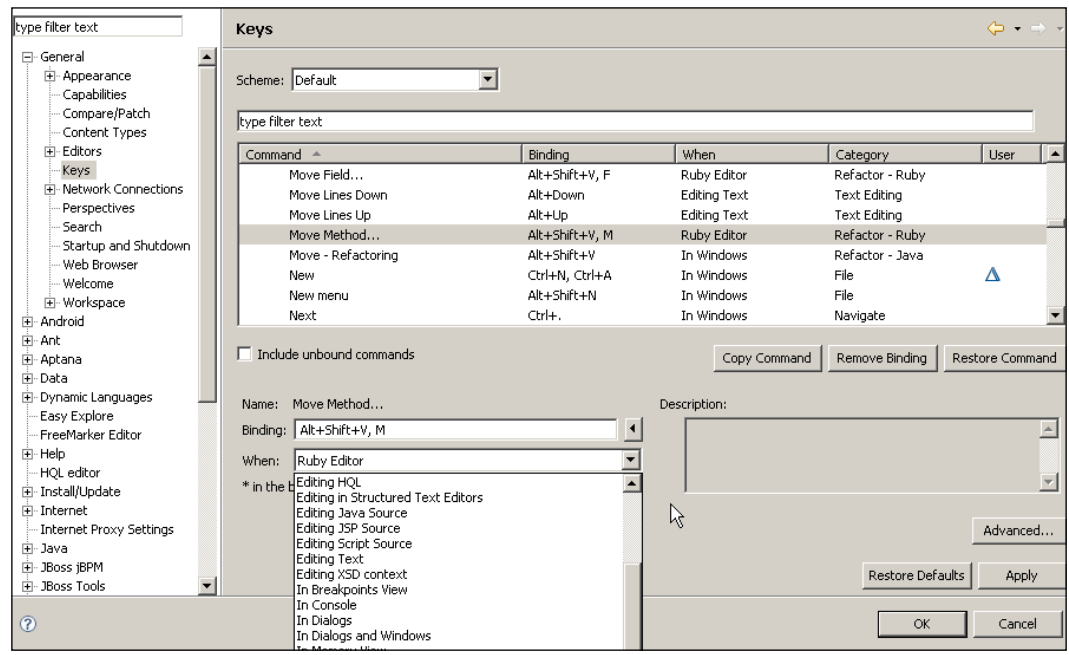
The text box at the top of the list acts as a filter. If you want to search for a command, just type in some text to filter the results. If you prefer, you can use the column headers for sorting the list.

The first column is the name of the command associated to the key binding, which is displayed in the next column. Then you can see the context under the **When** column, and a category, which is not configurable and refers to how the commands are internally grouped in Eclipse. The last column, named **User**, will display an icon if the binding was created or modified by a user.

If you see a star in the **Binding** column, it means there is a conflict between two sequences assigned to different commands in the same context.

For modifying or removing a binding, the first thing you have to do is select it on the list. Then you can use the **Binding** and **When** text fields to select the key sequence and context for launching the command. If you want to restore the original binding for a command you can use the **Restore Command** button, and if you want to restore all the default bindings for all the commands, you can use the **Restore Defaults** one.

Even if the binding list seems pretty large, there are many commands without a key binding by default. In previous versions of Eclipse you couldn't do anything about those commands, but if you are using Eclipse 3.3 or higher you can assign bindings to any available commands.

If you check the box **Include unbound commands** right below the list, you will see the whole list of commands, even those without an associated key sequence. You can select any of these commands for creating your own bindings.

Workspace

The last of the general preferences that can be interesting when developing with RadRails is located in the **Workspace** section. In this panel you can select some general options, such as the default encoding for your files. You can set the encoding for any project or file by right-clicking on its name on the Ruby Explorer, but if you assign the default encoding here, it will not be necessary to set it individually.

Aptana

As you know, Aptana RadRails uses the underlying Aptana Studio for the addition of ERB/RHTML, CSS, JavaScript, HTML, YML, and XML files. Aptana Studio provides more features apart from those editors, but they are not relevant for developing with RadRails. As with the **General** preferences, we will focus now on the preferences under the **Aptana** category that can affect our development experience when using RadRails.

Browsers/User Agents

In Chapter 5, we saw how we could take advantage of Aptana for editing style sheets, HTML, and JavaScript files. One interesting feature was the possibility of displaying small icons representing the compatible browsers when using code assist.

The list of browsers to represent with these icons is available at the **Browsers/User Agents** preferences. Just mark on the list the editors for which you want to have these icons displayed.

Editors

In this section we can configure the appearance and behavior of CSS, ERB/RHTML, HTML, JavaScript, XML, and YML editors. The preferences for the different editor types are very similar between them, with only small differences because of the different syntax between the languages.

Please notice, the preferences in this section will not apply for editing pure Ruby code (such as models, controllers, helpers, tasks, tests, and general Ruby classes and modules). We will see how to control those preferences later, under the **Ruby** section.

Code Assist

Depending on the type of editor you are configuring, you will find here preferences for **Auto-Activation** and **Auto-Insertion**.
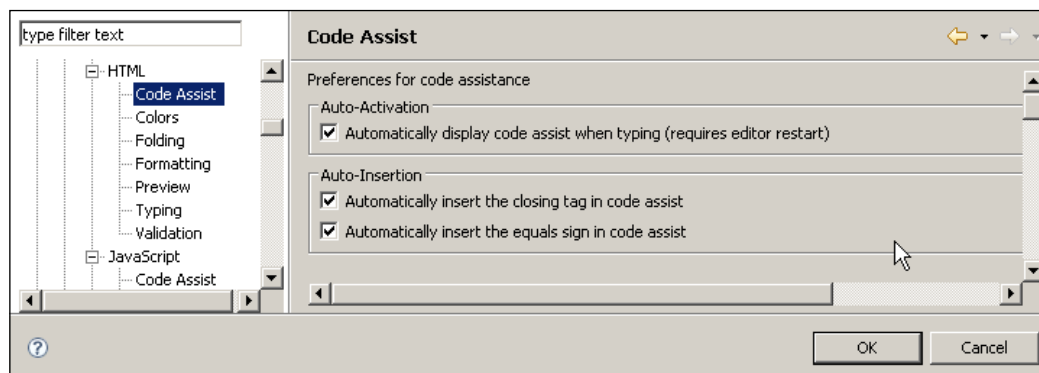
Auto-Activation means that code assist will evaluate your code as you type and will try to propose assistance on the fly. If you are experiencing performance problems because of this (most probably on the ERB/RHTML editors), or if you prefer not to have assistance until you manually invoke it by using *Ctrl+Space*, you can disable it.

Auto-Insertion means the editor will automatically insert the closing of the elements you open. For example, in the CSS editor you can configure auto-insertion of semicolons in your statements, and in the HTML editor you can have it insert your closing tags.

Colors

Previously we set the colors for the general editors, and from these preferences we can set them individually for each of the specialized ones. The color preferences dialog is divided in two parts: **Editor Options** and **Tokens**.

The Editor Options are disabled by default, meaning that the general preferences will be applied. If you want to override them, just check the box and select the colors you prefer.

Tokens are the different unique types of text in a programming language. For example, you have tokens for string literals, URLs, identifiers, keywords, or start and end tags. If you want to change the color settings for any of the tokens, just select it and click on the color box to the right of the token name. You cannot change the font family or size for the token, but you can set the font type to Bold, Italic, or Underlined if you want.
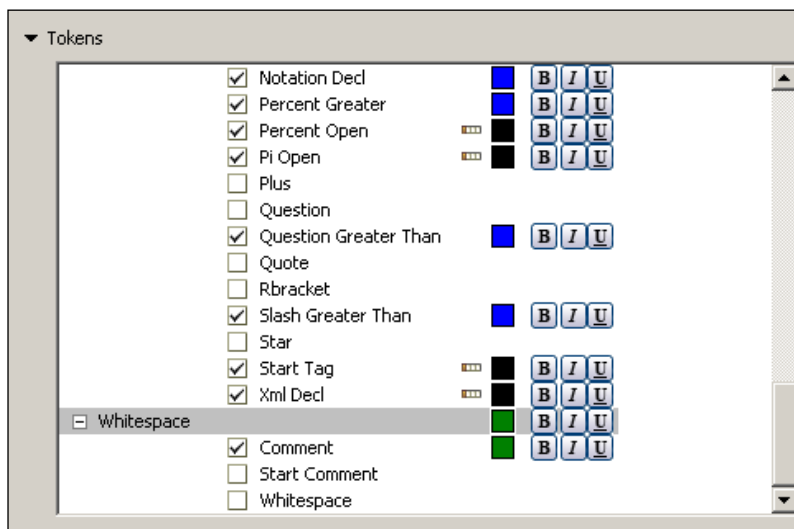
Folding

You already know we can fold parts of the code such as block comments, JavaScript function definitions, HTML tags, and so on. Should you want to, the folding preferences allow you to disable the folding functionality.

In some of the editors, you can also instruct Eclipse to automatically fold some of the sections initially, like block comments or function definitions. In the HTML folding options you can also configure which tags are foldable. By default, only `<html>`, `<head>`, `<body>`, `<script>`, `<style>`, and `<div>` will be available.

Formatting

Sometimes, when writing code, we don't follow the conventions about indentation, spacing, placement of closing braces, or new lines. In these cases, we can use the **Format** option for Eclipse to re-format the code by using these conventions. This option is available from the **Edit** or **Source** menus—depending on which editor we are working with—and the rules for the formatting can be defined in the **Formatting** sections of the different editors.

Depending on the particular editor, there can be some extra options, but the way of setting them is always the same. When you click on the **Formatting** preferences you will see a text box displaying an example of well-formatted code according to the current rules.

If you want to change this configuration, you have to use the **Edit** button. You will see a dialog similar to the one in the figure below (the JavaScript Formatting dialog)
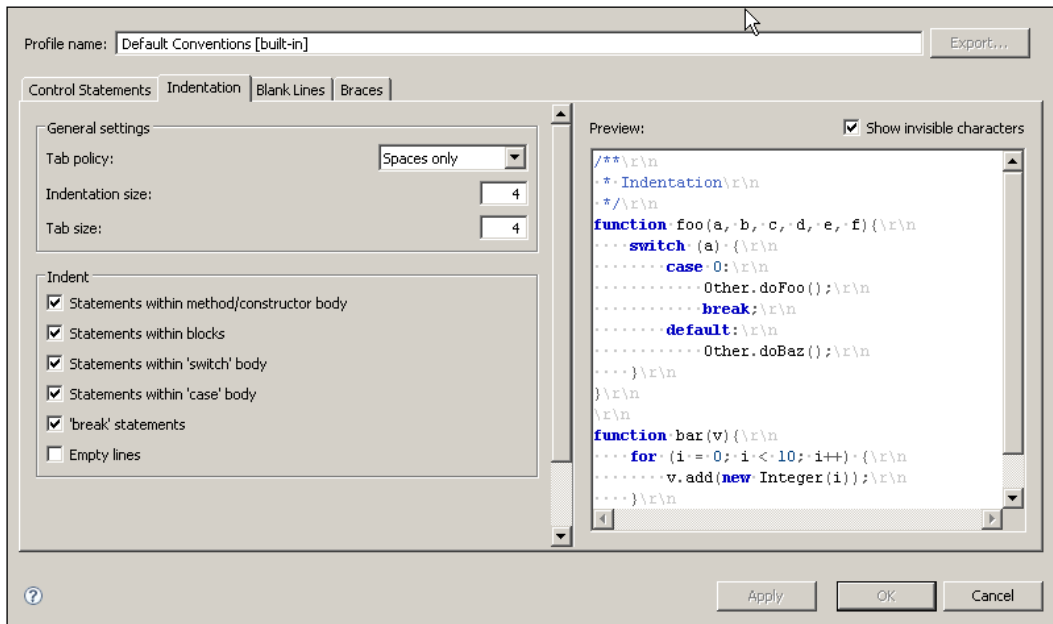
You will see different tabs with the options and the preview pane for displaying the changes. Under **Indentation** you can set the TAB policy. If you want to use spaces instead of TABs (this is the convention for Ruby code), you can select it from the drop-down box. You can also set the number of spaces to use.

If the editor you are configuring provides more indentation options, they will be displayed here. For example, in the JavaScript editor you can set different indentation policies depending on the parent statement of the indenting block.

The **Blank Lines** tab allows you to define where to place extra blank lines in the code and if the formatter should preserve any extra lines without content. You could use this, for example, to set a separation of a couple of lines between method definitions.

In the **Braces** tab you can set your preferences for the placement of braces in the source code. You can choose to have the braces in the same line, at the beginning of the next line, or indented at the next line.

In some of the editors' preferences you can find an extra **Control Statements** tab in which you can further refine the behavior of the formatter. For example, in the JavaScript formatter you can decide whether to present simple `if` statements in a single line instead of using two.

A convenient way for testing the effects of the changes on the formatter is by selecting the appropriate option and watching the changes in the preview pane. If you want to revert to the original settings, you can use the **Restore Defaults** button.

Typing

These preferences control how the editor behaves when typing. Unlike the **Formatting** preferences, that take place only after using the Format command, the Typing options will have direct effect, as you write the code. For example, if you enter an opening bracket in the Ruby editor, Eclipse will automatically insert the matching closing bracket. In this panel we can configure these operations.

As happened with the formatting, each editor will have a slightly different set of preferences to configure, but they all are very similar.

The first option in this dialog is for auto-inserting matching elements, such as braces, quotes, brackets, or end tags. Then you can choose, where available, if new lines should be auto-indented according to the previous line.

In the JavaScript, HTML, and XML editors, you can also choose if single or double quotes will be used when defining attributes.
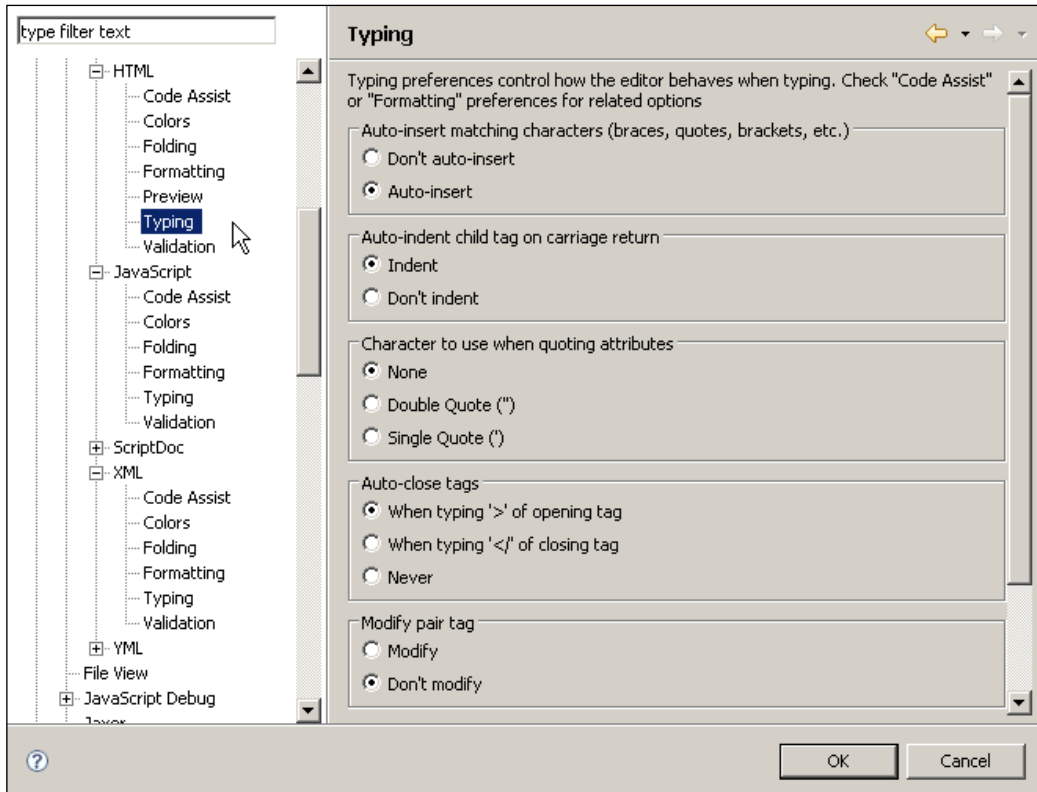
RHTML Templates

This option is only available for the ERB/RHTML Editor. From here, you can add or modify the code templates available when editing Rails views. Please refer to Chapter 5, *Coding Rails Views*, under the section *View Templates*, where we have already explained how to use this feature.

Start Page

Every time you start Eclipse, you will see the Aptana Start Page, on which you can find the latest news about Aptana as well as information about new updates. You can control in these preferences when to display this Start Page.

Rails

There are only three sections under **Rails** preferences, and we have seen them already in other parts of this book. **Autotest** allows us to configure if and how RadRails will automatically launch the test suites after saving your files. Please refer to Chapter 7 for the explanation of these preferences.

As we saw in Chapter 2, in the **Configuration** section you can set the paths to your Rails, Rake, and Mongrel scripts. However, in most cases these paths will be left blank, since RadRails can find the necessary scripts automatically. You should set them manually only if you are experiencing problems with your installation.

In previous versions of RadRails, before the project was taken over by Aptana, the RHTML editor was configurable from the **Rails** preferences. After the integration of RadRails and Aptana, many features were added to the RHTML editor and its configuration was moved to the **Aptana** section. The **Editors** option here is just a link to the configuration preferences under **Aptana** and it's been announced that it will be completely removed in future versions of RadRails.

Ruby

Most of the options under this section affect to the way we work with the Ruby editor and the Outline and Ruby Explorer views.

We will not go over all the preferences here, since we have seen already some of the functionalities in previous chapters of this book, like the **Debugger** and **Quick Inspect** preferences in Chapter 6, the templates in Chapter 4, the Installed Interpreters in Chapter 2, or the Editor Folding, Typing, and Formatting options previously in this chapter.

Appearance

Here you can configure how the information about the members and methods of your classes will be displayed in the Ruby Explorer and the Outline view. From the main dialog you can choose to display or hide the parameter names of the methods in these views.

If you open the **Members Sort Order** option, you will be able to modify the order in which the different elements will appear.
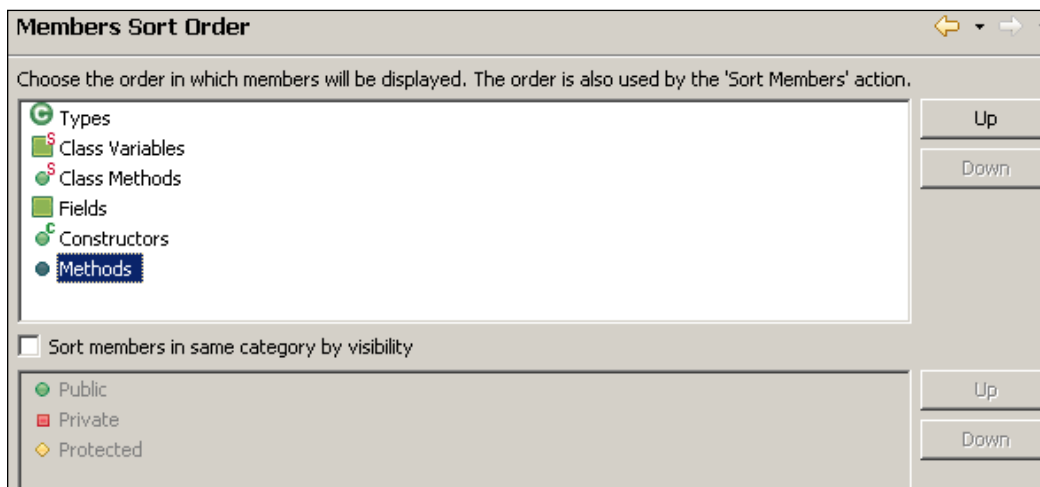
If you want to move a type of member up or down in the display order, just select it and use the 'up' and 'down' controls to the right. You can do this configuration as you have an expanded class on the Ruby Explorer. That way, if you click on **Apply** you can check the changes immediately. If you want to revert to the default configuration, just use the **Restore Defaults** button.

Editor

The main dialog of the editor allows you to change some general settings, like the highlighting of matching brackets—which will work also with do...end blocks—or disabling the reporting of problems as you type. You probably will not want to change this unless you are experiencing performance problems.

You have probably noticed, when you are editing a Ruby source file, that if you select the name of a method or a variable RadRails will highlight with a different background all the occurrences of that word in the file. By default, RadRails will do this with all the Types, Local Variables, Constants, and methods. If you want to disable the highlighting for some of these elements, you will find the corresponding checkboxes in the **Mark Occurrences** section.

Syntax Coloring

Apart from the generic coloring defined at the general **Color and Fonts** preferences, here you can modify the coloring applied to Ruby code and expressions.

To modify the presentation of any Ruby element just select its name in the scrollable list and set the desired foreground and background colors by using the buttons to the right. You can also modify the font properties, but not the font family or size.

As usual in this kind of dialog, you will see the changes in the **Preview** pane below and you will be able to restore the default coloring settings with the **Restore Defaults** button.
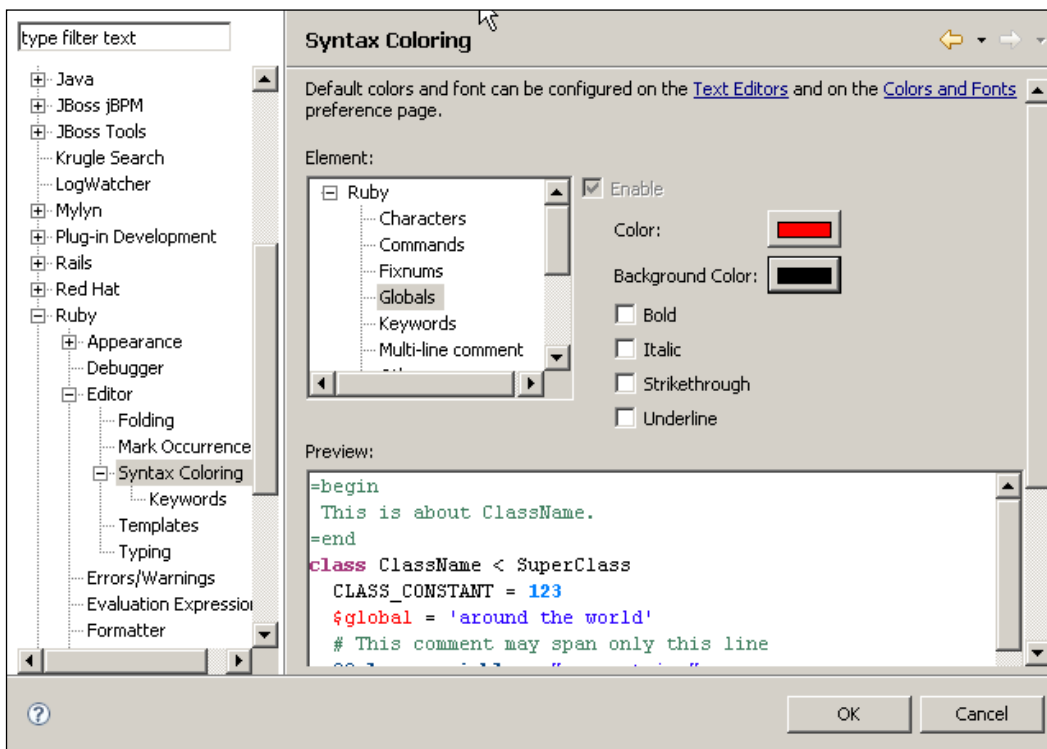
There is a nice additional feature here. You will see one of the elements listed as configurable is **Keywords**. If you select this element and modify the colors, you will be changing the presentation of Ruby keywords such as `class`, `def`, `if`, and so on.

Sometimes, you would like to treat some terms as if they were keywords; for example because you have created some kind of extension and you want to be able to differentiate your additional commands at a glance.

If you select the **Keywords** option located under **Syntax Coloring** you will be presented with a list (initially empty) of user-defined keywords. Here you can add any terms you want to be colorized as if they were proper Ruby keywords.

Errors/Warnings

As we saw in Chapter 7, RadRails will automatically display information about warnings in our code. This information appears by default both in the left margin of the editor and in the Problems view. What we still don't know is what is a warning for RadRails.

In the **Errors/Warnings** preferences we will be able to tell RadRails if something is a warning, an error, or if it should just be directly ignored.
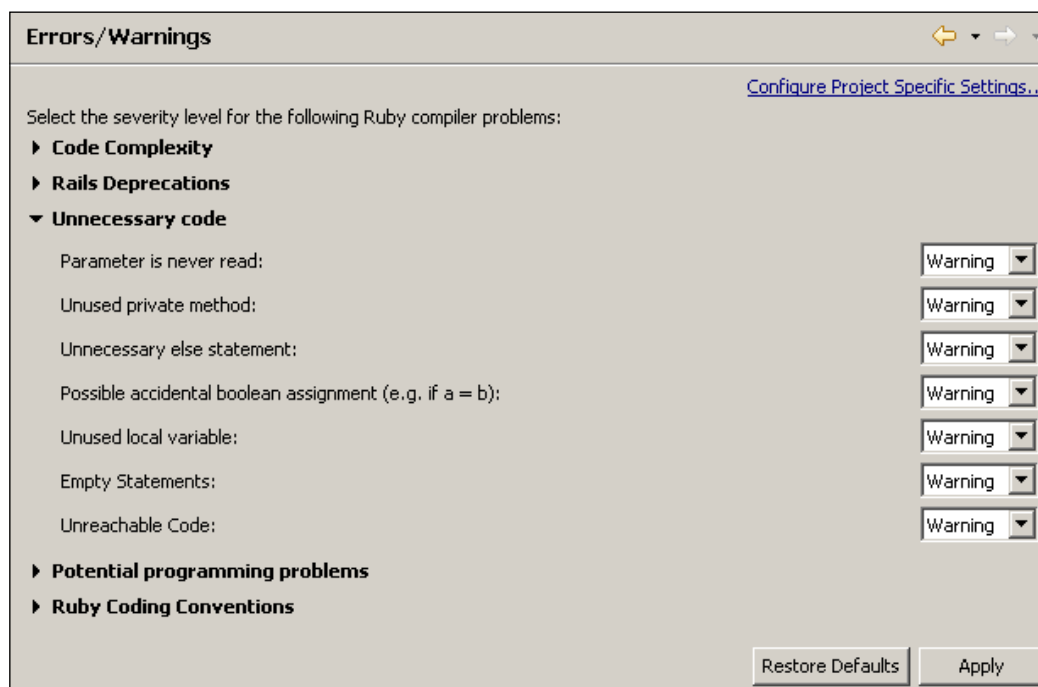
RadRails will automatically check for many different problems in our code. Since there is a large number of potential problems, the presentation of this dialog is divided in different sections to make it easier to navigate.

You can just click on the name of any section for it to open or close. Note that if you open more than one section at the same time you will have to use the scroll bar to the right for displaying the whole set of available parameters.

For each of the individual properties we can choose if we want to receive a warning, an error message, or just ignore it so RadRails will be silent about it. Some of the potential problems under the **Code Complexity** section take an additional number to specify things like the maximum number of parameters to use in a method definition.

As with the rest of the preferences, if you want to revert to the original set of preferences you can use the **Restore Defaults** button.

Task Tags

In the previous chapter we saw how to use special annotations in our comments so we could later see this information in the Tasks view. RadRails recognizes the FIXME, TODO, OPTIMIZE, and XXX annotations, each of them with a given priority.
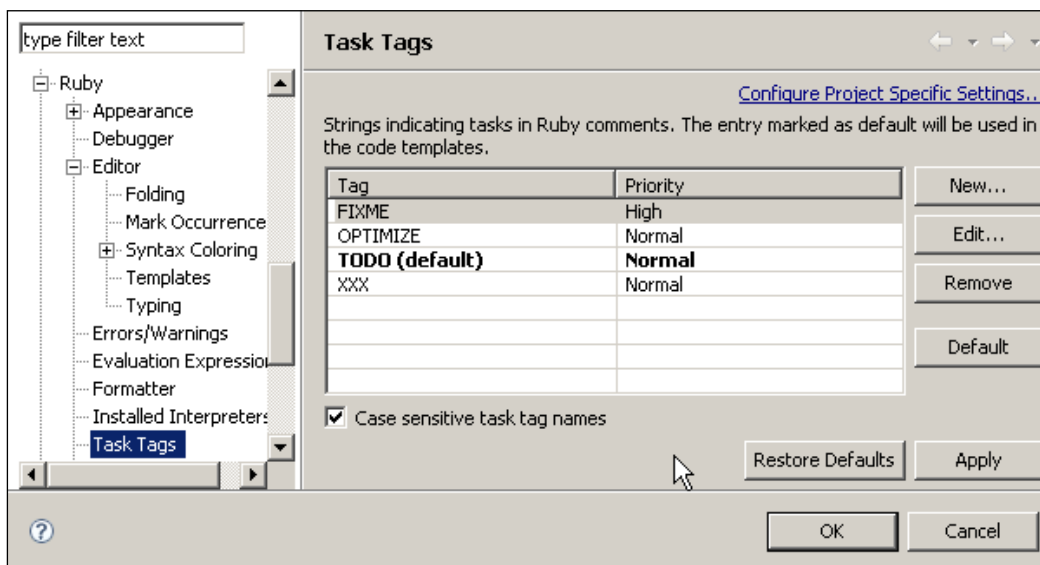
If you want to add your own annotations for displaying on the Tasks view, or if you want to change the priority for the defined annotations, you have to use the **Task Tags** preferences.

This dialog is really simple to use. You can select any of the defined tags and click on **Edit...** to change the current tag or the priority, or you can use the **New...** and **Remove** controls to create or delete an annotation type.

The checkbox at the bottom of the list controls whether the tags are case sensitive or not. The convention is they are case sensitive and uppercase.

Summary

This chapter explained how to modify the Eclipse and RadRails preferences to customize our development environment and make ourselves more comfortable with it.

However, most users will not need to change any of the preferences, since the defaults follow the coding conventions and are appropriate for a vast majority of developers. You will probably not modify your preferences until you are an experienced user and want to make minor adjustments to your installation.

And remember you can always use the **Restore Defaults** button to revert any changes you made and get back to the original configuration.

9
Other Useful Plugins

You already know how to use Aptana RadRails for developing your projects in a more productive way. But so far we have focused on Rails-related tasks only. When developing a project, there is more to it than the coding part and the pure Ruby/Rails operations we have learned to manage with RadRails.

In almost any project, you will need to work with a database. Actually, if you are not using a database, then working with Rails is probably an overkill. Even if migrations are great, and from the Rails console you can access your database contents, on many occasions you will want to connect directly to your database. This will be the case if you need to check the structure of your schema, or launch complex queries.

Also in many projects you will want to work with a code repository so you can check out a project, make changes, and then commit them back.

Of course you know you can use the command-line interface or a graphic front end for dealing with your database or your version control system, but it would be much more convenient if you could just use everything in an integrated manner from Eclipse.

Using such a powerful and popular IDE as Eclipse has many advantages, one of them being that for almost any development need you might have, there is a plugin supporting that feature. Actually, in most cases you will have several plugins to choose from. You only have to install them, set the preferences accordingly, and start managing all your development tasks from a single program.

Eclipse plugins are typically located at free hosting websites such as `http://sourceforge.net` or `http://code.google.com`, but in many cases the plugins are placed in other locations. Fortunately there are some pages with updated listings of plugins, categorized and rated by their users. Two of the most popular sites are `http://www.eclipseplugincentral.com` and `http://www.eclipse-plugins.info`, with more than 1500 different plugins referenced.

Apart from managing databases or version control systems in the Eclipse plugins ecosystem ,you can find plenty of other useful plugins like those for working with bug-trackers such as **Trac** and **Bugzilla**, for dealing with XML and other file formats, or for working with almost any programming language/framework of your choice, so you can develop all your projects, in different languages, from a single IDE.

In this chapter, we will discuss two plugins that will come in handy for virtually any Ruby on Rails developer: one for managing your database connections and one for working with code repositories. We will not be exhaustive about all the options and details of these plugins, but after finishing this chapter you will know enough to use them in your Rails projects.

Database Management

As of today, RadRails has two built-in views for interacting with your database: the **Data Navigator** and the **Query** view. These views use the configuration of your **database.yml** file to connect to the database and run the queries. If all you need is displaying the name of your tables, or the name and type of your columns, or running simple queries, these two views come in handy. But if you want to take a look at the details of your database, to run more complex queries, or to modify the stored information, you will need to use a more specialized plugin.

Eclipse doesn't provide any tools for database connectivity out-of-the-box—or not yet at least—but this being such a central part of almost any development, there exist many plugins to help you manage your databases. Some of the most popular are SQL Explorer (`http://www.sqlexplorer.org`), QuantumDB (`http://quantum.sourceforge.net`), the Eclipse Datatools Project (`http://www.eclipse.org/datatools`), and DBViewer (`http://www.ne.jp/asahi/zigen/home/plugin/dbviewer/about_en.html`).

All of these plugins are developed in Java and they connect to the target database through a JDBC driver. If you don't know what that is, don't worry, JDBC stands for Java Database Connector and it's a standard for making portable database drivers. All you will need to do is to download the appropriate driver for your database and you are ready to go.

Each of the named plugins has its sweet spot and its defects. During the years, I've been using all of them in the development of several projects and, as of today, the plugin with the best balance of capabilities, ease of installation, and usability is DBViewer. It features hassle-free connectivity to any database with a JDBC driver, navigation and manipulation of your schema's structure, SQL editor with code formatting, code templates and code assist, export of search results, and quick editing of the database contents.

While writing this book, I was delighted to learn that the Aptana team also likes this database plugin the best. They plan to bundle a slightly modified version of this plugin as a part of the release of Aptana Studio 1.2, so you will not need to install it separately. After that release, the **Data Navigator** and the **Query** view will be removed.

Not everything can be perfect though. Of all the plugins mentioned above, DBViewer is arguably the one with the poorest documentation (at least in English, since there is a user's manual available only in Japanese) but it's nevertheless very intuitive to use, so that's not a serious problem. Chris Williams, the lead developer of Aptana RadRails, kindly let me know they will provide at least some minimal English docs once the plugin gets integrated.

Please note that we are talking about a tool for developers and not database administrators, so there is no built-in support for creating databases, managing privileges or performing maintenance, or tuning tasks. You can however, get some of these functionalities—to some degree—by using the integrated SQL editor.

In this section, we assume you have working knowledge of SQL and you know how to write SQL statements. We will focus only on explaining the integration of DBViewer with Eclipse so you can manage your database connections, without explaining the different concepts or the SQL syntax, which would fall out of the scope of the book.

Installing DBViewer

In the first chapter of this book, we have already explained how to install a plugin in Eclipse. In the case of DBViewer, you can either download the latest version from the plugin website and manually copy it to your plugins directory, or you can use Eclipse's Update Manager to automatically get it installed. This is my recommended option, since the process is transparent, and it allows updating easily to newer versions. Of course, after the release of Aptana Studio 1.2, you will not need to install it separately.

If you are reading this book before that release, just go to the **Help** menu, and select **Software Updates | Find and Install**. In that select **Search for new features for install** and then choose the **New Remote Site**, as we did for installing the Aptana Studio and RadRails plugins in Chapter 1.

The update site for DBViewer is `http://www.ne.jp/asahi/zigen/home/plugin/dbviewer`. Just fill in the dialog as in the next picture and then proceed with the installation. After accepting all the steps of the set-up process (terms of acceptance, plugin download, and installation), Eclipse will ask you to restart your workbench. When Eclipse starts again, the DBViewer plugin will be available.
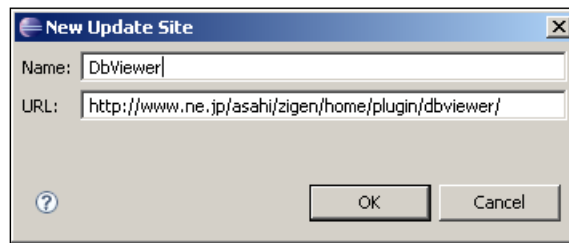
In order to check that the plugin was properly installed, go to the **Window** menu and select **Open Perspective | Other**. Depending on the number of plugins installed you will have different perspectives available. One of them should be named **DBViewer**. Please note that this perspective will be renamed to **DBExplorer** after the integration of this plugin into Aptana Studio.
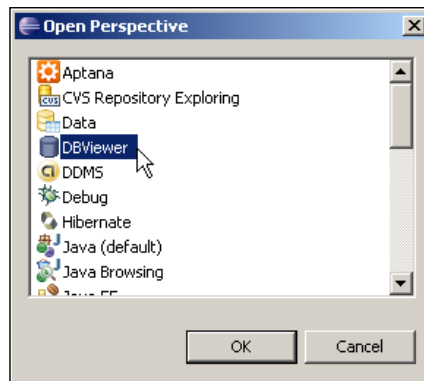
When you open it, the default **DBViewer** perspective will display three new views: **DB Tree**, **SQL Execute**, and **SQL History**.

Creating New Connections

The first thing we need to do is create a connection to our database. You will need a suitable JDBC driver for the database of your choice. Since MySQL and SQLite are the two most popular options when working with Rails projects, we will explain the details of how to configure these connections, but extrapolating them to any other driver is a trivial issue, since there are only minor differences. Once again, after the

integration of DBView with Aptana, MySQL and SQLite will be available by default. Still, the following steps illustrate how you can configure a connection against any database server, provided a suitable JDBC driver exists.

If you want to use MySQL for your connection, download the official JDBC connector from `http://www.mysql.com/products/connector/j/`. The driver doesn't need any installation; just unpack the downloaded file in the directory of your choice. The driver itself is a file with a `.jar` extension and it will be named something like `mysql-connector-java-5.0.7-bin.jar`

If you prefer to use SQLite, there is not an official JDBC driver, but there is a pretty good one at `http://www.zentus.com/sqlitejdbc/`. There are different versions of this plugin: a portable pure-Java solution and some pre-compiled binary versions depending on your OS. Since we will be using the driver only, working interactively from Eclipse, and database performance is not an issue here, the pure-Java version of the driver will be perfect. Just download it, and unpack it to any directory. The driver is a file with a `.jar` extension and it will be named something like `sqlitejdbc-v037-nested.jar`

As we just saw, you can always download your JDBC driver from the Internet, but in the case of SQLite3, MySQL, PostgreSQL, Oracle, and IBM DB2, those drivers will already be available in your system. As a part of the installation process, RadRails copies those files into the `plugins` directory of your Eclipse installation.

The `plugins` directory is typically located right below your Eclipse directory, but since you can configure where you want to install your plugins, it could be elsewhere too. If you want to use the JDBC drivers already installed, just locate the `plugins` directory and then look for a filename like `*mysql*.jar` or `*sqlite*.jar`.

Now you have the appropriate drivers in your system, we can already create a connection. Please note that we cannot use DBViewer to create the database and assign the necessary privileges, so the database must exist before trying to create the connection. Go to the **DB Tree** view and right-click on the **DBViewerPlugin** label. From the context menu select the **Add** option. A dialog will open for entering your connection details.

In the **DataBase Define name** text box enter the name you want to assign to this connection. This will be the name displayed in the **DB Tree** view, so it's advisable to enter something meaningful.

Now we have to select the JDBC driver for our connection. Click on the **Add File** button and navigate to the file of the driver you want to use. Remember the file we want is the one with a `.jar` extension.
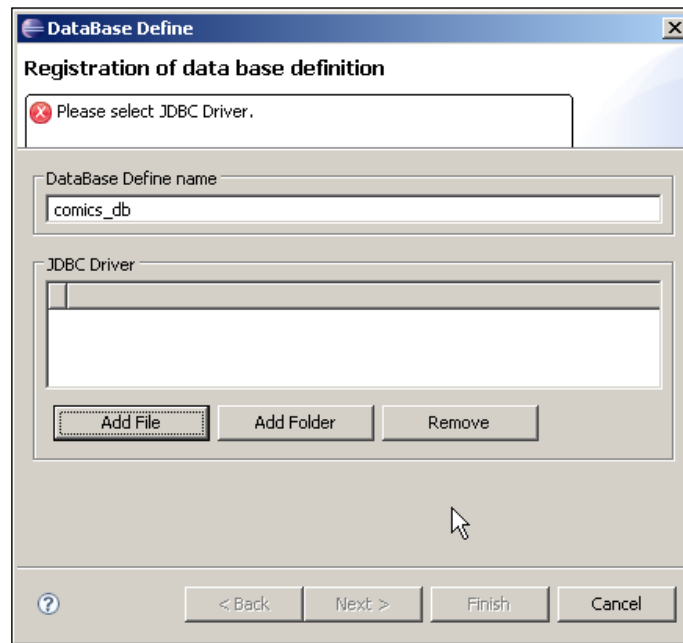
When using a JDBC driver, the connection is always defined by a connection string. This string is very similar to a URL and it's composed either of a host, port, and database name, or a filename pointing to the database contents. The format of this connection string is different for every JDBC driver, but DBViewer will assist us, so we don't have to worry about that.

When you click on **Next**, DBViewer will ask you if you want to replace the connection string with one registered in a template. Select **Yes** so a sample connection string for your driver will be provided.

In the next dialog, we have to fill in the information about our connection. The first field is the **JDBC Driver Name**. The combo will be loaded with different options depending on your driver. Leave this combo unchanged and do the same with the 'Driver Type' radio button.

If you are using a JDBC driver for a database other than SQLite or MySQL, check the documentation to see if it's type-4 compliant. Most modern drivers will be type 4, but if you are experiencing problems change it to type 2, since all the type 4 drivers should be type-2 compatible.
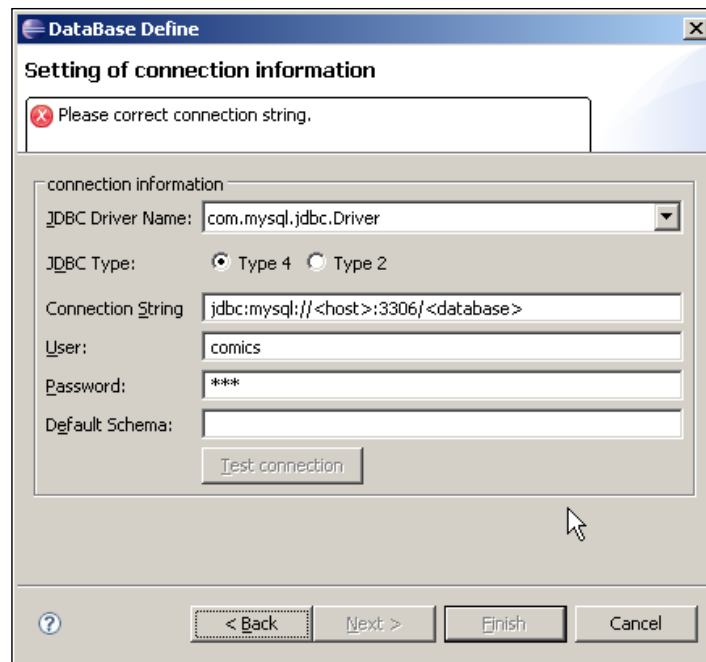
You will see the **Connection String** is pre-entered for you. You only have to provide the information between angular brackets.

In the case of MySQL we have to enter the host name or IP—localhost, most probably—and the database name. You will also have to provide the information for the **User** and **Password** fields. You can leave the **Default Schema** box blank, since this is not used by MySQL.

If you are using SQLite, you only have to provide the path to your `.db` file. A sample connection string would look like:

```
jdbc:sqlite:\temp\test.db
```

In SQLite, no **User**, **Password**, or **Default Schema** are needed, so you can safely leave all these fields blank.

Use the **Test Connection** button to see if there are any problems and correct the entered information if necessary. If the 'Test Connection' button appears disabled, it means you didn't provide a valid connection string. In that case, correct the connection string to activate the test button.

If you click on **Next** you will be able to further configure your connection. You can enter the charset you want to use in this connection, and choose if you want to use Auto Commit.

For SQLite I would recommend to always use Auto Commit in order to avoid potential locking problems. For MySQL, you can safely choose whether you prefer to work in Manual or Auto Commit mode.

After finishing with the creation of the connection, it will appear as a new icon on your **DB Tree** view. Now we will learn how to use this connection.

DB Tree View

The first thing we need is to establish the connection to the database. You can directly double-click on the name of your connection or you can right-click on it and select the **Connect** option. After waiting a few moments, the connection will open and two folders will appear on the tree for this connection, one for the database views (which you will probably not use) and one for your tables. If you open the **TABLE** folder, the names of all the tables in your database should be listed.

We can perform two operations with the tables in the **DB Tree** view. If you click on the plus sign preceding the name of the table, the list of the columns will appear, displaying the name and type for each of them. The primary key will be marked by a small key icon and the NOT NULL columns will be marked with a label reading XN.
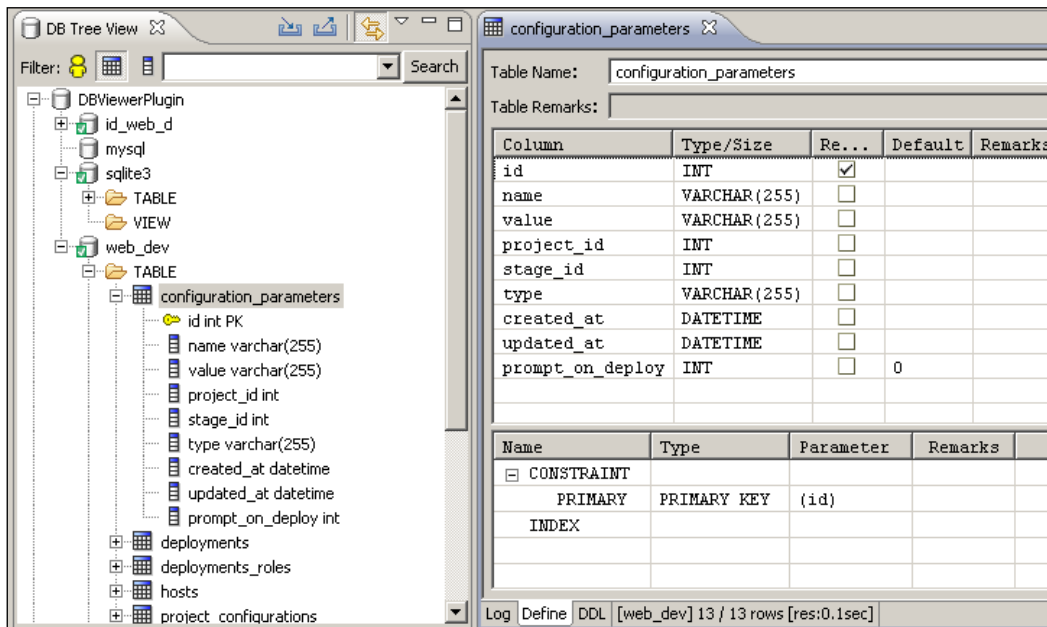
If you double-click on the name of the table, the Table Editor view will open, displaying the rows for that table. You can move up and down the data and you can enter edit mode just by clicking on the cell you want to change. Type the new contents for that cell and press enter. Please note that the changes you make this way will have direct effect on your database, even if you configured your connection not to Auto Commit.

When you double-clicked on the name of the table, all the rows were retrieved without any conditions. If you want to limit the results, you can type the conditions of the WHERE clause in the **where** text box at the top of the row list. That box provides content assist, so if you press *Ctrl+Space* (or *Command+Space* on the Mac) you will be presented with a list of possible options, including the names of the columns in your table.

If your table has a large number of columns, maybe you want to limit which ones will be displayed. You can do so by using the **Filter** button at the top-right corner of this view and checking only the columns you want to show.

Apart from selecting and updating the existing rows, you can remove, copy, or add new rows to your table. Just right-click with your mouse over the row to delete or copy and select the appropriate option from the context menu.

But from this Table Editor view you can do more than selecting and updating the table contents. You can consult and edit the structure of the tables too. In a typical Rails project all the operations related do the structure of the database (the so-called DDL or Data Definition Language) will be done via migrations, so we will use this view for informational purposes only.

At the foot of the Editor view, you will see there are four tabs: **Log**, **Define**, **DDL**, and the Rows Result pane we just explained. The **Log** tab is only available for privileged users and is not available for all databases. If your database supports it, you will see here the log entries for this table.

In the **Define** tab—displayed in the screenshot opposite—we can see the definition of your table. The upper pane shows the column information and the lower one shows the indexes and constraints, such as Primary or Foreign Keys.

If you want to modify the definition of any column, you can double-click on it. For adding, copying, or removing a column you have to right-click and select the appropriate option.

You can also add or drop new indexes and constrains by right-clicking on the lower pane and selecting the operation you want to perform.

Finally, in the tab of the Table Editor view labeled **DDL**, you will see the SQL statement for creating your table. Depending on your technical background and your personal preferences, you may find this more convenient than the **Define** tab for consulting your table structure.

SQL Execute View

In the **SQL Execute View** you can run any SQL statement against your database, and it features a built-in formatter, code assist, and code templates, in the same way as we saw for the Ruby editor.
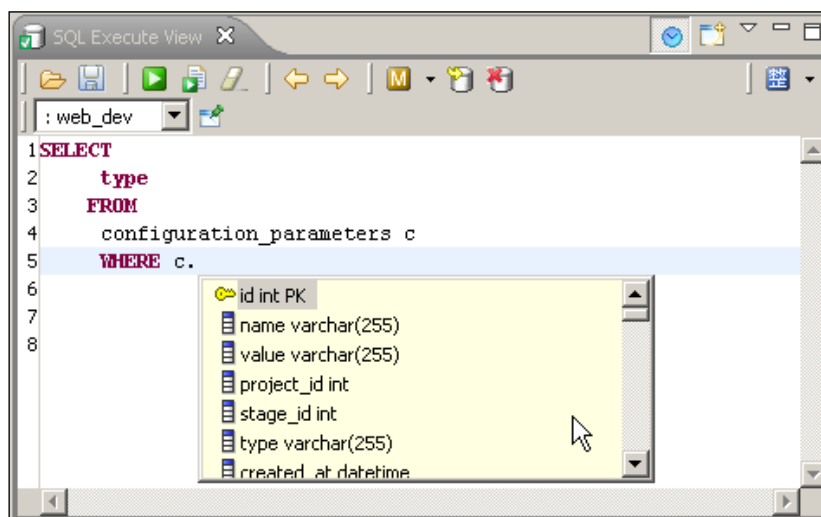
If you have added more than one connection, you will have to select the target connection in the drop-down list in this view. If you try to execute an SQL statement against a closed connection, the plugin will display a message for you to confirm you want to establish the connection.

Once you have selected the target database, you can start writing in the editor. You can try to write just **sel** and press *Ctrl+Space* to get suggestions of code completion or to use one of the available templates. Depending on which part of the statement you are in, you will be presented with SQL keywords, table names, column names, SQL functions, or pre-defined templates. As happened with the Ruby code, you can create your own SQL templates in the preferences dialog.

This editor also features syntax highlighting for the SQL reserved keywords as you type, so the statement will be more readable.

Once you have finished typing the statement, you can run it from this view's toolbar by using the green 'execute all' button. You can also right-click on the editor and select **Execute All** or you can use the shortcut *Shift+Enter*. If you want to execute just a part of the code in the editor, you can select it, right-click, and choose **Execute Selected SQL**.

The rest of the icons in this view are pretty straightforward. With the **Open** and **Save** controls you can load an SQL script in the editor or to save its contents. Then you will find the **Execute** and **Execute as Script** icons and the **Clear** control, to remove the current contents of the editor. The back and next arrows are used to navigate the history of the executed statements.

The next three icons are related to transactions. The first icon will represent an **A** or an **M**, standing for 'Auto' or 'Manual' commit. If using auto-commit, the next two icons will be disabled, since they are the **Commit** and **Rollback** controls, used to persist or undo the changes to your database.

Finally you will find the format icon. You can choose between manual formatting (the default) in which you have to click this icon or right-click on the editor to format, and auto-formatting, in which the code will be automatically formatted on every execution.

In general, this view is pretty simple to use, but it provides all the power for executing SQL in the same comfortable way you usually find in full database clients.

SQL History View

The **SQL History View** is a convenient way to keep track of the history of statements executed. There is a folder for each day, and then the statements appear sorted by date. When you click on any statement, you will see the whole SQL code in the lower pane. If you want to bring the code to the **SQL Execute View**, just double-click on the statement in the history list.
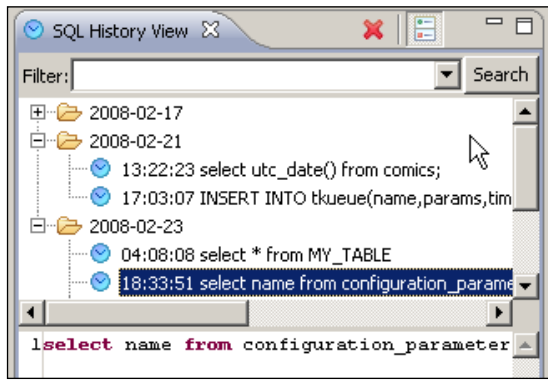
If your history list is growing a lot and you find it difficult to find the statements you are looking for, you have two options. You can click on any statement and select the **Delete** icon of this view's toolbar—so the list will not be so long—or you can use the **Filter** box to type some letters (like the name of a table or column) and let DBViewer search for them in the history.

DBViewer Configuration

As with any other plugin, you can set some preferences for DBViewer in the **Windows | Preferences** menu. It's no surprise that the section in which the preferences appear is named **DBViewer Plugin**.

Most of the preferences should look familiar, since they are related to syntax coloring, code assist, and formatting. These settings work in the same way as we saw in the configuration reference chapter of this book.

There are only two sections in the DBViewer preferences that might not be clear at first glance: **Generate Value Object** and **Connection String**. We will not be using the first one since it is intended for auto-generating Java code.
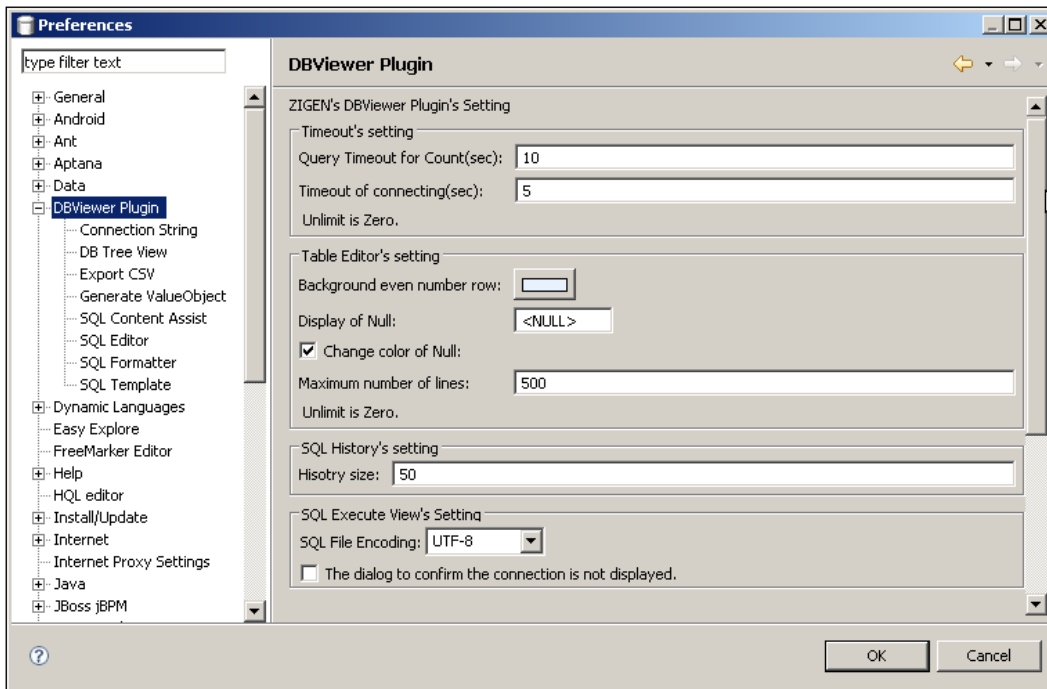
The **Connection String** preferences are used to provide connection strings template for each JDBC driver. You will probably not need to add a new connection string template since out of the box the plugin supports connections for MySQL, SQLite, Oracle, Apache Derby, PostgreSQL, Interbase, DB2, and SQLServer, amongst others.

Should you need to use a different JDBC driver, you could create your own connection string template by using the **New** button and entering the connection string sample in the text box.

Version Control with Eclipse

Version Control—also referred as Revision Control or Software Configuration Management—is a central part of software development. If several developers are working on a project, it's inconceivable not to use a version control system to avoid conflicts and share the code. But even if there is a single developer, version control should be used for keeping track of changes, sharing code between different environments, or tagging releases, for example.

Eclipse understands the importance of a version control system, and provides support for CVS by default. Besides, it allows for any other version control software to be integrated with Eclipse in a convenient way, keeping the same interface. In the Eclipse jargon, the operations related to version control are usually found under the **Team** preferences, the **Team** context-menu, and the **Synchronize** perspective.

There are plugins available for almost any version control system in the market. During the years, I have successfully managed CVS, Subversion, Visual Source Safe, and Perforce repositories with Eclipse. Some minor details apart, the user experience is almost identical.

In the Ruby on Rails community, the most widely used version control system is Subversion—abbreviated as SVN. For over three years, the source code of Rails itself has been hosted in a Subversion repository, most of the projects in RubyForge use SVN, and SVN is the only version control system available for Ruby projects hosted at Google Code.

Very recently—not even a month ago at the moment of writing this book—Rails was moved to a Git repository. As you probably know, Git is a version control system developed originally for managing the Linux code base. Git is a very good option when you have a distributed project with many developers. Thus, due to the large number of contributors, moving Rails to Git makes a lot of sense. However, as of today, Git lacks of a good integration with any IDE, and it must be used from the command line or with a built-in GUI.

Shawn Pearce is currently developing an Eclipse plugin for Git. Unfortunately, the project is at a very early stage and it will probably take a while until the user experience is similar to that of the other version control systems. If you want more information about this plugin, you can visit `http://git.or.cz/gitwiki/ EclipsePlugin`.

Since Subversion is still the version control system of choice for Ruby on Rails development, we will explain in this section how to use Subversion when working with Eclipse.

> Should you decide to use a plugin other than Subversion, many of the things we will explain here will still be useful for you. All the version control plugins for Eclipse are modeled after the CVS one, so all of them are used almost identically.

There are two popular plugins for using Subversion from Eclipse: Subclipse (`http://subclipse.tigris.org`) and Subversive (`http://www.eclipse.org/ subversive`). Both can work perfectly with your subversion repositories and in the end it's mainly a matter of taste using one or the other.

I prefer Subclipse for a number of reasons: It has been around longer than Subversive, it was created by **Tigris.org**, the same Open Source community that created and maintains Subversion, and Mark Phippard—the Lead Developer of Subclipse—is a full committer for Subversion. Thus, you can always be sure that the plugin follows the philosophy of Subversion itself, and the new features are implemented early. This being said, I know of many people that are successfully working with Subversive, and I tested it during one of my projects to see how it performed and found it flawless.

This section assumes you have working knowledge of Subversion, and will focus only on how to use subversion from Eclipse, not explaining the different concepts or the internals of working with Subversion. Also, we will not explain all the details of the version control features you can get with Eclipse, since some topics like branch management can get very complex and fall out of the scope of this book. Please refer to the documentation of Subversion and Subclipse if you want more information.

Installing Subclipse

We will use Eclipse's Update Manager for installing Subclipse. If you are behind a firewall, you can also download the package from the Subclipse website and install it manually, but I strongly recommend to use Update Manager whenever possible.

As we did when installing DBViewer, go to the **Help** menu and select **Software Updates | Find and Install**. In that select **Search for new features for install** and then choose the **New Remote Site**. The Update Manager site for Subclipse is `http://subclipse.tigris.org/update_1.2.x`.
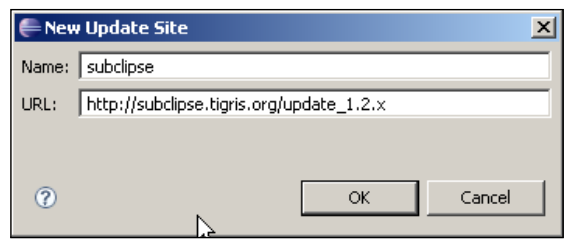
> In Aptana Studio 1.2 you will be able to start the installation of this plugin directly from the plugins section of the **Aptana Studio Start Page** (available in the **Help** menu). The automatic installation will save you from entering the URL for the plugin, but you will still need to finish the next steps in order to install the plugin.

When asked about the components to install, don't check the optional components, since they have dependencies on other plugins that we are not going to be using. Complete the installation process and restart your Eclipse workbench when you are prompted to do so.

You don't need to have Subversion installed in your system in order to use Subclipse, but if you don't have it installed, you would probably need to take an additional step before starting using Subclipse.

In order to connect to your repositories, Subclipse can use either a native library called JavaHL, or a pure-Java library called SVNKit. If you are using Windows, the JavaHL library comes bundled with the Subclipse installation, but in other OS, if you don't have a local copy of subversion in your system, then you cannot use the JavaHL native library and you should change the preferences of Subclipse.

To change this setting open the **Preferences** dialog, go to the **Team | SVN** section and check the radio button for the SVNKit in the **SVN Interface** box. Using SVNKit is the recommended option—even for Windows users—because it performs better than the native API.

Now you are ready to connect to your repositories from Eclipse. Note that this plugin is intended for developers, and does not provide administrative features. If you want to create a brand-new repository, you will have to use the command-line tool **svnadmin** and manually configure the repository privileges. Administration of repositories is out of the scope of this book, so for the examples we will connect to the Rails repository itself and we will download the code of the `acts_as_tree` plugin.

SVN Repository Exploration

The first thing to do in Eclipse when you want to work with a repository is create a repository location pointing to the URL where the repository lives. For that, we have to open the perspective **SVN Repository Location**, which is available as usual in the **Window** menu under the **Open Perspective | Other** option.

In this perspective, we will create a new location pointing to `http://svn.rubyonrails.org/rails`, which was the former root of the Rails repository. Subversion access is still available for reference, even though the development was moved to Git.

To create the new location you can use the **Add SVN location** icon, represented with a plus sign on the SVN Repository toolbar, or you can directly right-click and choose **New | Repository location**. A dialog will open asking you for the URL of your repository.
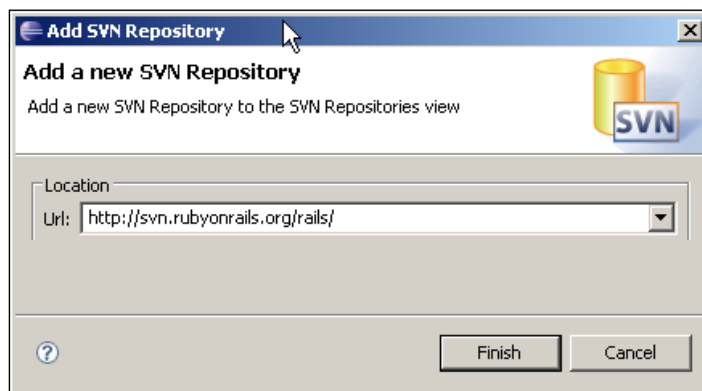
After accepting this, a new folder will appear in your SVN Repository view. If you open the folder by double-clicking on it you will see the contents of the remote repository. Notice that several projects can be found under a single repository, for example in this case the Rails repository contains the Rails code itself, as well as a number of plugins, database adapters, spin-offs, and tools.

Notice that we didn't have to authenticate to use the repository, but if the repository you are connecting to didn't allow anonymous access, Eclipse would prompt you for a user and password, giving you the option of saving the credentials so you don't have to enter them every time you try to connect.

If you navigate to the **plugins** folder and open it, you will see several plugin projects inside. When opening the contents of **acts_as_tree**, you can see the names of the files inside that project, together with the revision number with which they were last updated.
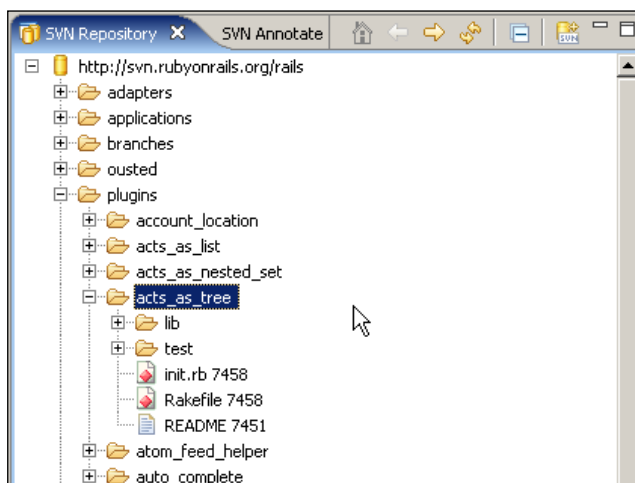
If you want to see the contents of any file at the remote repository, you can double-click on it and it will be opened in the Editor view as usual. Since these are the contents of the remote location, the editor will open in read-only mode. Before we can change the contents of the source code, we will have to get a local copy—a so-called Working Copy—of the repository contents.

Projects and Repositories

To manage a repository with Eclipse, we will have to establish a relationship between an Eclipse project and a repository location. There are basically two scenarios for this. In the first scenario, you want to connect to an existing repository and download an existing project, creating an Eclipse project to work with. This would be the equivalent to the `checkout` command of Subversion. In the second case, you have created a new project in Eclipse and you want to upload it to a repository, so you can put it under control of Subversion. This would be the equivalent to the `import` command.

Since the number of developers is bigger than the number of projects, it's more usual to check out an existing project than to import a new one into a new repository. However, depending on your profile, importing can be a very frequent task too.

Checking out an Existing Project

Before checking out a project you must create a repository location as we saw in the previous step. In the SVN Repository view, navigate to the project you want to check out (in our case the **plugins** | **acts_as_tree** project), right-click on it and select **checkout**.
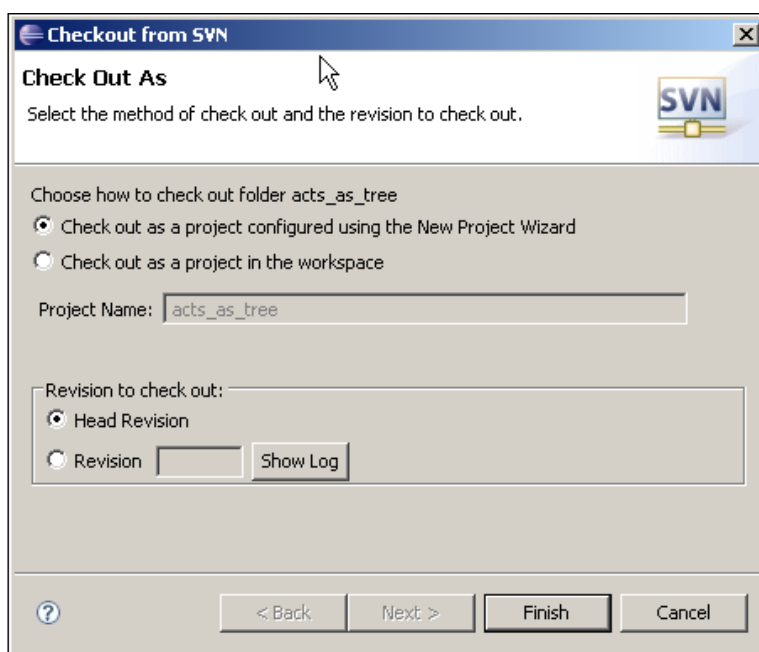
You can choose to check out by using the **New Project** wizard, which will take you to a dialog like the one we saw when creating a new project, or to directly create a new project in the workspace. In either case, when you click on **Finish** the code of the repository will be copied into your local working copy.

If you don't want to fetch the latest revision of the project, you can specify which revision number you want to check out. If you know you want a revision other than the head version, but you are not sure about which one, you can use the **Show Log** button to see the log of the project so you can see the comments associated with the revisions and decide which one you want.

You can also check out a project by using the **New Project** option in the **File** menu or the Ruby Explorer. At the opening dialog select that you want to check out a project from SVN, as shown in the following screenshot:
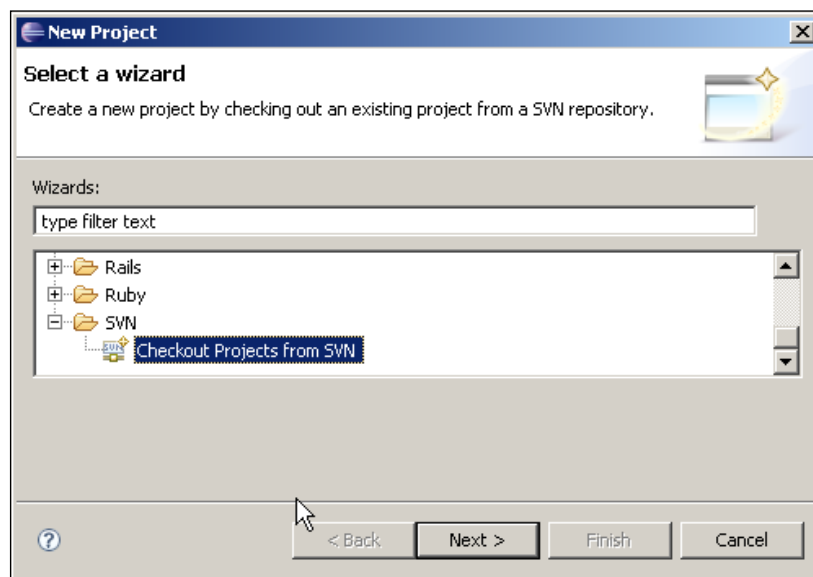
In this case, the wizard will present you with a list of your configured repository locations, for you to choose from where to perform the checkout. After this step, the process is identical to the one of checking out from the SVN Repository view.

Importing a New Project into a Repository

If you have created a new project in Eclipse and you want to import it into a repository, you have to right-click on its name in the Ruby Explorer and select **Share Project**.

In the opening dialog, you will have to select the 'SVN' type and then the repository location in which you are going to import the project. Of course your user must have privileges to write into that repository. The **Share Project** dialog will then prompt you to enter a name for the project in the repository. By default, the same name as your Eclipse project will be used.
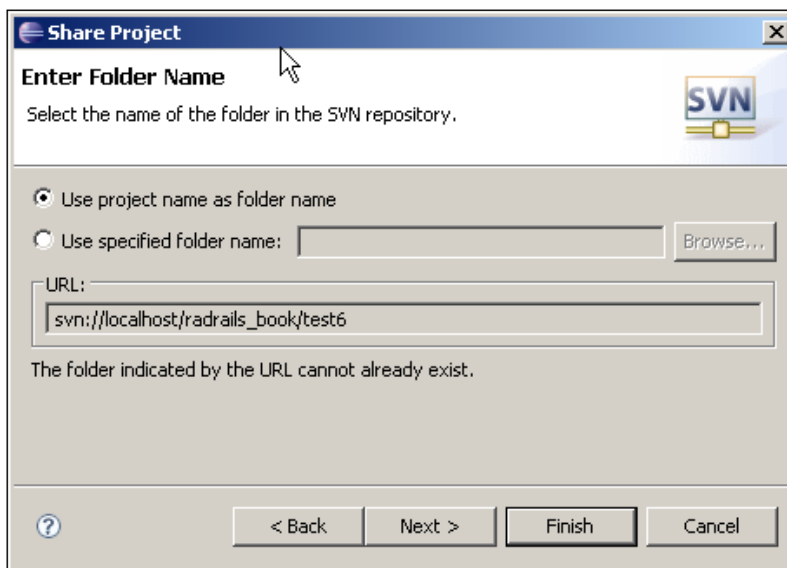
If you click on **Next**, Eclipse will ask you for a comment for the initial import, and will display all the files that are going to be imported into the repository. If there are some files or directories you don't want to import, just uncheck the box by their side. After all the changes are committed, your project will be under version control.

Update, Edit, Compare, and Commit

The work flow when working with any—or almost any—version control system is updating your local copy with the latest repository contents, editing the files, comparing your changes with the repository in case there are any conflicts, and committing back the modified files

All of these Subversion-related operations are available under the **Team** option of the context-menu in the Ruby Explorer. You can see this menu if you switch to the Rails Perspective and right-click on top of the name of the **acts_as_tree** project we previously checked out.

If you wanted to update—meaning to fetch the latest version of the repository contents—the whole project, you could select the **Update** option in the **Team** menu. At this moment, nothing will be updated, since we already have the latest version. However, you will see in the **Console** view that a connection was made and the output of the `svn update` command will be displayed, saying something like **At revision XXX**.

You should be aware that the operations you perform from the **Team** menu are recursive, so if you select **Update** at the project level, the contents of the whole project will be updated.
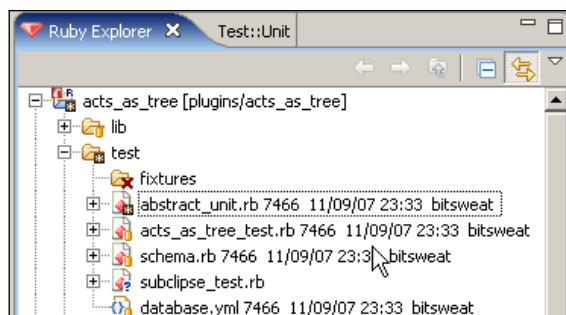
Subclipse will decorate the items in the **Ruby Explorer** with different icons depending on their status. A file or directory under version control will be marked with a small yellow icon. An item under version control with local changes will be marked with an asterisk. The new items will display a question mark icon, and the deleted directories will be marked with a red cross. By default, after each item's name, the revision number, timestamp, and author of the last commit will be shown.

At any time, you can compare the contents of a file with the base revision—the last version you downloaded from the repository—or with the latest version in the repository. To see how this works, you can navigate to the **tree.rb** file under **lib | active_record | acts** and delete some of the lines in the file. After saving, you will see that now this file is marked with an asterisk in the **Ruby Explorer**.

To compare the file with the original, right-click on its name and go to the **Compare with** option. Here you can choose **Base Revision**, so you will see which changes you have made since we last updated it. A compare editor will be displayed so you can see the changes side by side.

If you wanted to undo the changes, you could do so in two different ways. You can use the **Revert** option from the **Team** menu to get back the contents of the last updated revision, or you can right-click and open the **Replace with** option, in which you can choose the **Base Revision**, the latest from the repository, or any other Revision in the history of this resource.

After you have edited your source files, you will want to send the changes back to the repository. Right-click on the file or directory to commit, open the **Team** menu, and select **Commit**. A dialog will display showing all the resources you will be committing.

By default, the new files that are not yet under version control will not be included in the commit, but you can mark the checkboxes by their names to include them, or simply use the **Select All** button to mark them all at once. In the top part of this dialog you can write a message to describe the changeset you are committing, which is always a good practice. In the drop-down list you will find the history of the last messages in case you want to reuse them.
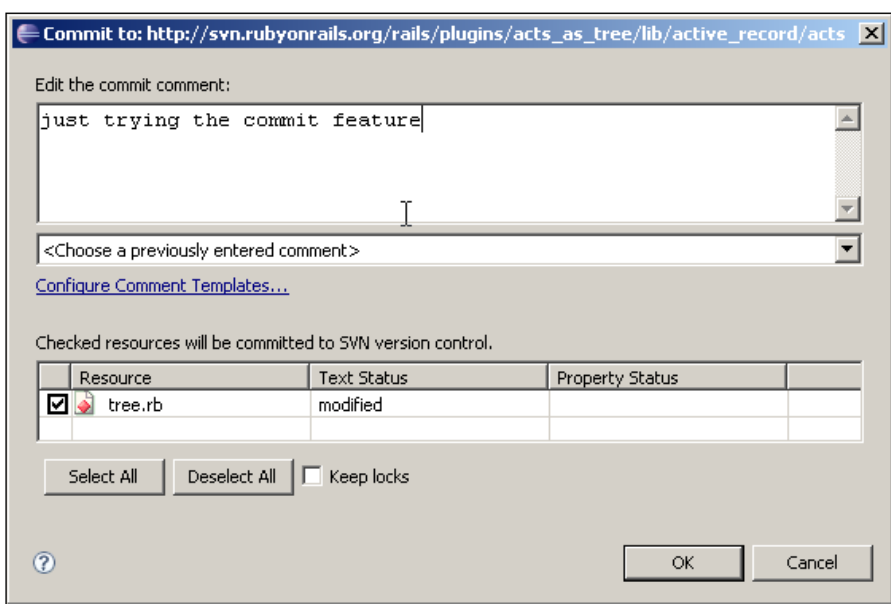
When you are ready to commit the changes, click on the **OK** button. In our case, we will not be able to commit since we don't have privileges for writing to the **acts_as_tree** project, and an error message will be displayed.

Even when you are a privileged user, if you are trying to commit any resources that are **out-of-sync** with the repository, you will also get an error message. Out-of-sync means you are trying to commit a file that was modified by a different person since the last time you updated it. In this case, you will have to make an update to your local file, and resolve any potential conflicts, before being able to commit your changes.

The Synchronize View

We saw in the previous section that you can update a single file or a whole directory, but with this approach it is difficult to see which changes you are going to be updating until you actually download them. Also, in the case of conflicts or out-of-sync resources, you can get merges in your files that you will have to manually correct.

Subclipse provides the **Synchronize** view as a convenient way for comparing the status of our local copy against the latest version of the repository, so we can see which changes will be made before doing anything. We can even compare and modify the contents of our conflicting files in a comfortable way.

The easiest way for starting the synchronize view is right-clicking on the name of your project in the **Ruby Explorer** and choosing **Synchronize with repository** in the **Team** menu. Eclipse will then prompt you to change to the Synchronize perspective.
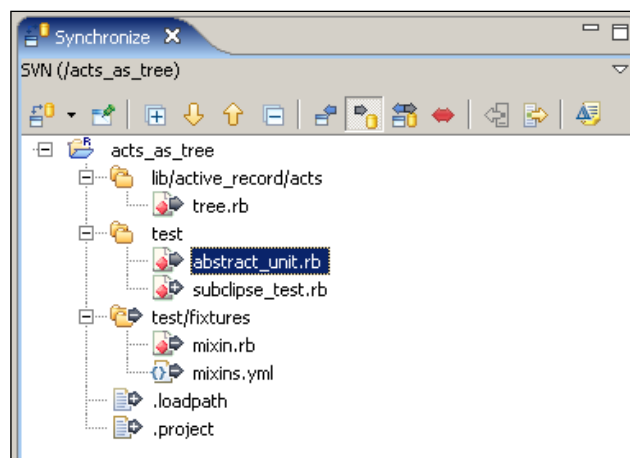
The contents of this view are similar to the **svn status -u** command but with a more convenient presentation. Incoming changes will be displayed with a left-arrow, outgoing changes will be displayed with a right-arrow, and added or deleted files will be represented with a plus or minus sign. In the case of conflicts, a red double-arrow will be shown.

The toolbar of this view has some interesting controls. For example, you can use the up and down arrows to navigate one by one through the differences between the files in your working copy and the repository version.

By default, all the changes will be shown in this view. If you are interested only in inbound, outbound, or conflicting changes, you can use the toolbar icons to limit the results.

You can accept all the incoming or outgoing changes with the two icons in the toolbar, or you can right-click item by item and select **Update** or **Commit** accordingly. If there is an outgoing change you want to revert, you can right-click on that file and select **Override and Update**. The result will be the same as when selecting the **Replace with latest from repository** option from the **Team** menu.

In the case of conflicting files, you can double-click on the name of the file to open a compare editor. You can then incorporate the changes of the remote version—either by hand, or using the built-in controls of the comparing editor, as we learned when comparing Ruby source files—and save the file.

Since the conflicts are already resolved, you can right-click on the name of the file and select the **Mark as merged** option. This way, Eclipse will remove the conflict mark and will let you commit the changes.

History View

The last of the features of Subclipse we will look at is the **History** view. To see the history of any file or directory under version control, switch to the **Ruby Explorer**—or to the **SVN Repository View**—right-click on the item you want to consult, and select the **Show History** option of the **Team** menu.

This view is divided into three panes. The upper one displays the list of revisions for the selected file. The lower left area displays the list of files that were committed in the same revision, and the lower right area shows the comment entered when submitting the selected revision.

You can select any of the revisions in the list and right-click to get a context menu from which you can get the contents of that revision into the current one, or revert your working copy to that revision.

If you select two different revisions—by holding the *Ctrl* key—you can choose to compare those versions from the context menu.
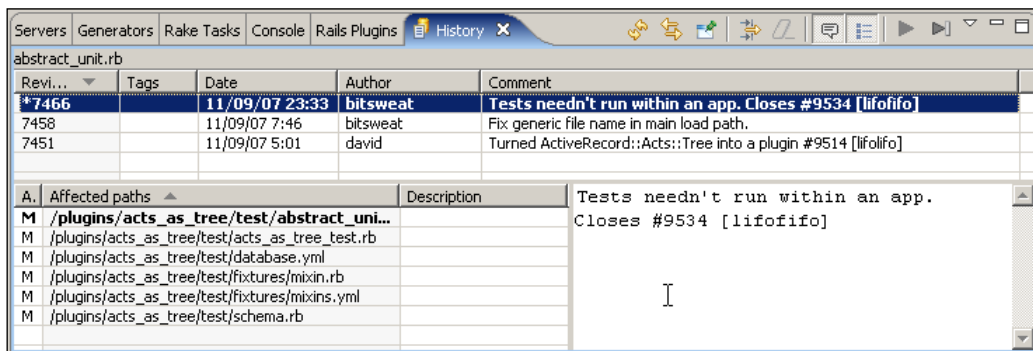
Summary

RadRails provides several tools and specialized editors for working with Ruby and Rails from Eclipse. But in the development of any software project, there are some aspects not directly related to the programming language or the framework being used, and these aspects are not covered by a framework-oriented plugin like RadRails.

Fortunately, one of the advantages of the Eclipse IDE is its extensibility and popularity within the developers community. This means there are plugins for almost any task you need to carry out as a part of the software development cycle.

Out of all the useful plugins available, we have learned how to use DBViewer and Subclipse to help us deal with our databases and repositories without having to leave the workbench. Thus, we can work in a more comfortable way and we will see how our productivity increases.

We also provided some URLs where you can find up-to-date lists of Eclipse plugins that you might want to incorporate to your installation.

Index

Ruby on Rails Enterprise Application Development

ISBN: 978-1-847190-85-7   Paperback: 528 pages

Building a complete Ruby on Rails business application from start to finish

1. Create a non-trivial, business-focused Rails application

2. Solve the real-world problems of developing and deploying Rails applications in a business environment

3. Apply the principles behind Rails development to practical real-world situations

Building Dynamic Web 2.0 Websites with Ruby on Rails

ISBN: 978-1-847193-41-4   Paperback: 216 pages

Create database-driven dynamic websites with this open-source web application framework

1. Create a complete Web 2.0 application with Ruby on Rails

2. Learn rapid web development

3. Enhance your user interface with AJAX

Please check **www.PacktPub.com** for information on our titles