Extracted from:

Programming Ruby

The Pragmatic Programmers' Guide

Second Edition

Chad Fowler *is a leading figure in the Ruby community. He's on the board of Ruby Central, Inc. He's one of the organizers of RubyConf. And he's one of the writers of RubyGems. All this makes him uniquely qualified to write this chapter.*

Chapter 17

Package Management with RubyGems

RubyGems is a standardized packaging and installation framework for libraries and applications, making it easy to locate, install, upgrade, and uninstall Ruby packages. It provides users and developers with four main facilities.

1. A standardized package format,
2. A central repository for hosting packages in this format,
3. Installation and management of multiple, simultaneously installed versions of the same library,
4. End-user tools for querying, installing, uninstalling, and otherwise manipulating these packages.

Before RubyGems came along, installing a new library involved searching the Web, downloading a package, and attempting to install it—only to find that its dependencies haven't been met. If the library you want is packaged using RubyGems, however, you can now simply ask RubyGems to install it (and all its dependencies). Everything is done for you.

In the RubyGems world, developers bundle their applications and libraries into single files called *gems*. These files conform to a standardized format, and the RubyGems system provides a command-line tool, appropriately named *gem*, for manipulating these gem files.

In this chapter, we'll see how to

1. Install RubyGems on your computer.
2. Use RubyGems to install other applications and libraries.
3. Write your own gems.

Installing RubyGems

To use RubyGems, you'll first need to download and install the RubyGems system from the project's home page at http://rubygems.rubyforge.org. After downloading and unpacking the distribution, you can install it using the included installation script.

```
% cd rubygems-0.7.0
% ruby install.rb
```

Depending on your operating system, you may need suitable privileges to write files into Ruby's site_ruby/ and bin/ directories.

The best way to test that RubyGems was installed successfully also happens to be the most important command you'll learn.

```
% gem help
RubyGems is a sophisticated package manager for Ruby.  This is
a basic help message containing pointers to more information.

  Usage:
    gem -h/--help
    gem -v/--version
    gem command [arguments...] [options...]

  Examples:
    gem install rake
    gem list --local
    gem build package.gemspec
    gem help install

  Further help:
    gem help commands            list all 'gem' commands
    gem help examples            show some examples of usage
    gem help <COMMAND>           show help on COMMAND
                                  (e.g. 'gem help install')
  Further information:
    http://rubygems.rubyforge.org
```

Because RubyGems' help is quite comprehensive, we won't go into detail about each of the available RubyGems commands and options in this chapter.

Installing Application Gems

Let's start by using RubyGems to install an application that is written in Ruby. Jim Weirich's Rake (http://rake.rubyforge.org) holds the distinction of being the first application that was available as a gem. Not only that, but it's generally a great tool to have around, as it is a build tool similar to Make and Ant. In fact, you can even use Rake to build gems!

Locating and installing Rake with RubyGems is simple.

```
% gem install -r rake
Attempting remote installation of 'Rake'
Successfully installed rake, version 0.4.3
% rake --version
rake, version 0.4.3
```

RubyGems downloads the Rake package and installs it. Because Rake is an application, RubyGems downloads both the Rake libraries and the command-line program `rake`.

You control the gem program using subcommands, each of which has its own options and help screen. In this example, we used the `install` subcommand with the `-r` option, which tells it to operate remotely. (Many RubyGems operations can be performed either locally or remotely. For example, you can to use the `query` command either to display all the gems that are available remotely for installation or to display a list of gems you already have installed. For this reason, subcommands accept the options `-r` and `-l`, specifying whether an operation is meant to be carried out remotely or locally.)

If for some reason—perhaps because of a potential compatibility issue—you wanted an older version of Rake, you could use RubyGems' version requirement operators to specify criteria by which a version would be selected.

```
% gem install -r rake -v "< 0.4.3"
Attempting remote installation of 'rake'
Successfully installed rake, version 0.4.2
% rake --version
rake, version 0.4.2
```

Table 17.1 on the following page lists the version requirement operators. The `-v` argument in our previous example asks for the highest version lower than 0.4.3.

There's a subtlety when it comes to installing different versions of the same application with RubyGems. Even though RubyGems keeps separate versions of the application's library files, it does not version the actual command you use to run the application. As a result, each install of an application effectively overwrites the previous one.

During installation, you can also add the `-t` option to the RubyGems `install` command, causing RubyGems to run the gem's test suite (if one has been created). If the tests fail, the installer will prompt you to either keep or discard the gem. This is a good way to gain a little more confidence that the gem you've just downloaded works on your system the way the author intended.

```
% gem install SomePoorlyTestedProgram -t
Attempting local installation of 'SomePoorlyTestedProgram-1.0.1'
Successfully installed SomePoorlyTestedProgram, version 1.0.1
23 tests, 22 assertions, 0 failures, 1 errors...keep Gem? [Y/n] n
Successfully uninstalled SomePoorlyTestedProgram version 1.0.1
```

Had we chosen the default and kept the gem installed, we could have inspected the gem to try to determine the cause of the failing test.

Table 17.1. Version operators

Both the `require_gem` method and the `add_dependency` attribute in a `Gem::Specification` accept an argument that specifies a version dependency. RubyGems version dependencies are of the form `operator major.minor.patch_level`. Listed below is a table of all the possible version operators.

Operator	Description
=	Exact version match. Major, minor, and patch level must be identical.
!=	Any version that is not the one specified.
>	Any version that is greater (even at the patch level) than the one specified.
<	Any version that is less than the one specified.
>=	Any version greater than or equal to the specified version.
<=	Any version less than or equal to the specified version.
~>	"Boxed" version operator. Version must be greater than or equal to the specified version *and* less than the specified version after having its minor version number increased by one. This is to avoid API incompatibilities between minor version releases.

Installing and Using Gem Libraries

Using RubyGems to install a complete application was a good way to get your feet wet and to start to learn your way around the `gem` command. However, in most cases, you'll use RubyGems to install Ruby libraries for use in your own programs. Since RubyGems enables you to install and manage multiple versions of the same library, you'll also need to do some new, RubyGems-specific things when you require those libraries in your code.

Perhaps you've been asked by your mother to create a program to help her maintain and publish a diary. You have decided that you would like to publish the diary in HTML format, but you are worried that your mother may not understand all of the ins and outs of HTML markup. For this reason, you've opted to use one of the many excellent templating packages available for Ruby. After some research, you've decided on Michael Granger's BlueCloth, based on its reputation for being very simple to use.

You first need to find and install the BlueCloth gem.

```
% gem query -rn Blue
*** REMOTE GEMS ***
BlueCloth (0.0.4, 0.0.3, 0.0.2)
   BlueCloth is a Ruby implementation of Markdown, a text-to-HTML
   conversion tool for web writers. Markdown allows you to write using
   an easy-to-read, easy-to-write plain text format, then convert it
   to structurally valid XHTML (or HTML).
```

This invocation of the `query` command uses the `-n` option to search the central gem repository for any gem whose name matches the regular expression `/Blue/`. The results show that three available versions of BlueCloth exist (0.0.4, 0.0.3, and 0.0.2). Because you want to install the most recent one, you don't have to state an explicit version on the `install` command; the latest is downloaded by default.

```
% gem install -r BlueCloth
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
```

Generating API Documentation

Being that this is your first time using BlueCloth, you're not exactly sure how to use it. You need some API documentation to get started. Fortunately, with the addition of the `--rdoc` option to the `install` command, RubyGems will generate RDoc documentation for the gem it is installing. For more information on RDoc, see Chapter 16 on page 187.

```
% gem install -r BlueCloth --rdoc
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
Installing RDoc documentation for BlueCloth-0.0.4...
WARNING: Generating RDoc on .gem that may not have RDoc.
        bluecloth.rb: cc.............................
Generating HTML...
```

Having generated all this useful HTML documentation, how can you view it? You have at least two options. The hard way (though it really isn't that hard) is to open RubyGems' documentation directory and browse the documentation directly. As with most things in RubyGems, the documentation for each gem is stored in a central, protected, RubyGems-specific place. This will vary by system and by where you may explicitly choose to install your gems. The most reliable way to find the documents is to ask the `gem` command where your RubyGems main directory is located. For example:

```
% gem environment gemdir
/usr/local/lib/ruby/gems/1.8
```

RubyGems stores generated documentation in the `doc/` subdirectory of this directory, in this case `/usr/local/lib/ruby/gems/1.8/doc`. You can open the file `index.html` and view the documentation. If you find yourself using this path often, you can create a shortcut. Here's one way to do that on Mac OS X boxes.

```
% gemdoc=`gem environment gemdir`/doc
% ls $gemdoc
BlueCloth-0.0.4
% open $gemdoc/BlueCloth-0.0.4/rdoc/index.html
```

To save time, you could declare `$gemdoc` in your login shell's profile or rc file.

The second (and easier) way to view gems' RDoc documentation is to use RubyGems' included `gem_server` utility. To start `gem_server`, simply type

```
% gem_server
[2004-07-18 11:28:51] INFO  WEBrick 1.3.1
[2004-07-18 11:28:51] INFO  ruby 1.8.2 (2004-06-29) [i386-mswin32]
[2004-07-18 11:28:51] INFO  WEBrick::HTTPServer#start: port=8808
```

`gem_server` starts a Web server running on whatever computer you run it on. By default, it will start on port 8808 and will serve gems and their documentation from the default RubyGems installation directory. Both the port and the gem directory are overridable via command-line options, using the `-p` and `-d` options, respectively.

Once you've started the `gem_server` program, if you are running it on your local computer, you can access the documentation for your installed gems by pointing your Web browser to `http://localhost:8808`. There, you will see a list of the gems you have installed with their descriptions and links to their RDoc documentation.

Let's Code!

Now you've got BlueCloth installed and you know how to use it, you're ready to write some code. Having used RubyGems to download the library, we can now also use it to load the library components into our application. Prior to RubyGems, we'd say something like

```
require 'bluecloth'
```

With RubyGems, though, we can take advantage of its packaging and versioning support. To do this, we use `require_gem` in place of `require`.

```
require 'rubygems'
require_gem 'BlueCloth', ">= 0.0.4"
doc = BlueCloth::new <<MARKUP
 This is some sample [text][1].  Just learning to use [BlueCloth][1].
 Just a simple test.
 [1]: http://ruby-lang.org
MARKUP
puts doc.to_html
```

produces:

```
<p>This is some sample <a href="http://ruby-lang.org">text</a>.  Just
 learning to use <a href="http://ruby-lang.org">BlueCloth</a>.
 Just a simple test.</p>
```

The first two lines are the RubyGems-specific code. The first line loads the RubyGems core libraries that we'll need in order to work with installed gems.

```
require 'rubygems'
```

The second line is where most of the magic happens.

```
require_gem 'BlueCloth', '>= 0.0.4'
```

This line adds the BlueCloth gem to Ruby's `$LOAD_PATH` and uses `require` to load any libraries that the gem's creator specified to be autoloaded. Let's say that again a slightly different way.

Each gem is considered to be a bundle of resources. It may contain one library file or one hundred. In an old-fashioned, non-RubyGems library, all these files would be copied into some shared location in the Ruby library tree, a location that was in Ruby's predefined load path.

RubyGems doesn't work this way. Instead, it keeps each version of each gem in its own self-contained directory tree. The gems are not injected into the standard Ruby library directories. As a result, RubyGems needs to do some fancy footwork so that you can get to these files. It does this by adding the gem's directory tree to Ruby's load path. From inside a running program, the effect is the same: `require` just works. From the outside, though, RubyGems gives you far better control over what's loaded into your Ruby programs.

In the case of BlueCloth, the templating code is distributed as one file, `bluecloth.rb`; that's the file that `require_gem` will load. `require_gem` has an optional second argument, which specifies a version requirement. In this example, you've specified that BlueCloth version 0.0.4 or greater be installed to use this code. If you had required version 0.0.5 or greater, this program would fail, because the version you've just installed is too low to meet the requirement of the program.

```
require 'rubygems'
require_gem 'BlueCloth', '>= 0.0.5'
```

produces:

```
/usr/local/lib/ruby/site_ruby/rubygems.rb:30:
          in `require_gem':  (LoadError)
RubyGem version error: BlueCloth(0.0.4 not >= 0.0.5)
from prog.rb:2
```

As we said earlier, the version requirement argument is optional, and this example is obviously contrived. But, it's easy to imagine how this feature can be useful as different projects begin to depend on multiple, potentially incompatible, versions of the same library.

Dependent on RubyGems?

Astute readers (that's all of you) will have noticed that the code we've created so far is dependent on the RubyGems package being installed. In the long term, that'll be a fairly safe bet (we're guessing that RubyGems will make its way into the Ruby core distribution). For now, though, RubyGems is not part of the standard Ruby distribution,

The Code Behind the Curtain

So just what does happen behind the scenes when you call the magic `require_gem` method?

First, the gems library modifies your `$LOAD_PATH`, including any directories you have added to the gemspec's `require_paths`. Second, it calls Ruby's `require` method on any files specified in the gemspec's `autorequires` attribute (described on page 212). It's this `$LOAD_PATH`-modifying behavior that enables RubyGems to manage multiple installed versions of the same library.

so users of your software may not have RubyGems installed on their computers. If we distribute code that has `require 'rubygems'` in it, that code will fail.

You can use at least two techniques to get around this issue. First, you can wrap the RubyGems-specific code in a block and use Ruby's exception handling to rescue the resultant `LoadError` should RubyGems not be found during the `require`.

```
begin
  require 'rubygems'
  require_gem 'BlueCloth', ">= 0.0.4"
rescue LoadError
  require 'bluecloth'
end
```

This code first tries to require in the RubyGems library. If this fails, the `rescue` stanza is invoked, and your program will try to load BlueCloth using a conventional `require`. This latter require will fail if BlueCloth isn't installed, which is the same behavior users see now if they're not using RubyGems.

Alternatively, RubyGems can generate and install a stub file during gem installation. This stub file is inserted into the standard Ruby library location and will be named after the gem package contents (so the stub for BlueCloth will be called `bluecloth.rb`). People using this library can then simply say

```
require 'bluecloth'
```

This is exactly what they would have said in pre-RubyGems days. The difference now is that rather than loading BlueCloth directly, they'll instead load the stub, which will in turn call `require_gem` to load the correct package. A stub file for BlueCloth would look something like this.

```
require 'rubygems'
$".delete('bluecloth.rb')
require_gem 'BlueCloth'
```

The stub keeps all the RubyGems-specific code in one place, so dependent libraries won't need to include any RubyGems code in their source. The `require_gem` call will load whatever library files the gem maintainer has specified as being autoloaded.

As of RubyGems 0.7.0, stub installation is enabled by default. During installation, you can disable it with the `--no-install-stub` option. The biggest disadvantage of using the library stubs is that you lose RubyGems' ability to manage multiple installed versions of the same library. If you need a specific version of a library, it's better to use the `LoadError` method described previously.

Creating Your Own Gems

By now, you've seen how easy RubyGems makes things for the users of an application or library and are probably ready to make a gem of your own. If you're creating code to be shared with the open-source community, RubyGems are an ideal way for end-users to discover, install, and uninstall your code. They also provide a powerful way to manage internal, company projects, or even personal projects, since they make upgrades and rollbacks so simple. Ultimately, the availability of more gems makes the Ruby community stronger. These gems have to come from somewhere; we're going to show you how they can start coming from you.

Let's say you've finally gotten your mother's online diary application, MomLog, finished, and you have decided to release it under an open-source license. After all, other programmers have mothers, too. Naturally, you want to release MomLog as a gem (moms love it when you give them gems).

Package Layout

The first task in creating a gem is organizing your code into a directory structure that makes sense. The same rules that you would use in creating a typical tar or zip archive apply in package organization. Some general conventions follow.

- Put all of your Ruby source files under a subdirectory called `lib/`. Later, we'll show you how to ensure that this directory will be added to Ruby's `$LOAD_PATH` when users load this gem.
- If it's appropriate for your project, include a file under `lib/yourproject.rb` that performs the necessary `require` commands to load the bulk of the project's functionality. Before RubyGems' autorequire feature, this made things easier for others to use a library. Even with RubyGems, it makes it easier for others to explore your code if you give them an obvious starting point.
- Always include a `README` file including a project summary, author contact information, and pointers for getting started. Use RDoc format for this file so you

can add it to the documentation that will be generated during gem installation. Remember to include a copyright and license in the README file, as many commercial users won't use a package unless the license terms are clear.

- Tests should go in a directory called `test/`. Many developers use a library's unit tests as a usage guide. It's nice to put them somewhere predictable, making them easy for others to find.
- Any executable scripts should go in a subdirectory called `bin/`.
- Source code for Ruby extensions should go in `ext/`.
- If you've got a great deal of documentation to include with your gem, it's good to keep it in its own subdirectory called `docs/`. If your README file is in the top level of your package, be sure to refer readers to this location.

This directory layout is illustrated in Figure 17.1 on page 220.

The Gem Specification

Now that you've got your files laid out as you want them, it's time to get to the heart of gem creation: the gem specification, or *gemspec*. A gemspec is a collection of metadata in Ruby or YAML (see page 737) that provides key information about your gem. The gemspec is used as input to the gem-building process. You can use several different mechanisms to create a gem, but they're all conceptually the same. Here's your first, basic MomLog gem.

```
require 'rubygems'
SPEC = Gem::Specification.new do |s|
  s.name     = "MomLog"
  s.version  = "1.0.0"
  s.author   = "Jo Programmer"
  s.email    = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/MomLog"
  s.platform = Gem::Platform::RUBY
  s.summary  = "An online Diary for families"
  candidates = Dir.glob("{bin,docs,lib,tests}/**/*")
  s.files    = candidates.delete_if do |item|
                 item.include?("CVS") || item.include?("rdoc")
               end
  s.require_path     = "lib"
  s.autorequire      = "momlog"
  s.test_file        = "tests/ts_momlog.rb"
  s.has_rdoc         = true
  s.extra_rdoc_files = ["README"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
end
```

Let's quickly walk through this example. A gem's metadata is held in an object of class `Gem::Specification`. The gemspec can be expressed in either YAML or Ruby code. Here we'll show the Ruby version, as it's generally easier to construct and more flexible in use. The first five attributes in the specification give basic information such as the gem's name, the version, and the author's name, e-mail, and home page.

In this example, the next attribute is the platform on which this gem can run. In this case, the gem is a pure Ruby library with no operating system–specific requirements, so we've set the platform to `RUBY`. If this gem were written for Windows only, for example, the platform would be listed as `WIN32`. For now, this field is only informational, but in the future it will be used by the gem system for intelligent selection of precompiled native extension gems.

The gem's summary is the short description that will appear when you run a `gem query` (as in our previous BlueCloth example).

The `files` attribute is an array of pathnames to files that will be included when the gem is built. In this example, we've used `Dir.glob` to generate the list and filtered out CVS and RDoc files.

Runtime Magic

The next two attributes, `require_path` and `autorequire`, let you specify the directories that will be added to the `$LOAD_PATH` when `require_gem` loads the gem, as well as any files that will automatically be loaded using `require`. In this example, `lib` refers to a relative path under the MomLog gem directory, and the `autorequire` will cause `lib/momlog.rb` to be required when `require_gem "MomLog"` is called. For each of these two attributes, RubyGems provides corresponding plural versions, `require_paths` and `autorequires`. These take arrays, allowing you to have many files automatically loaded from different directories when the gem is loaded using `require_gem`.

Adding Tests and Documentation

The `test_file` attribute holds the relative pathname to a single Ruby file included in the gem that should be loaded as a `Test::Unit` test suite. (You can use the plural form, `test_files`, to reference an array of files containing tests.) For details on how to create a test suite, see Chapter 12 on page 143 on unit testing.

Finishing up this example, we have two attributes controlling the production of local documentation of the gem. The `has_rdoc` attribute specifies that you have added RDoc comments to your code. It's possible to run RDoc on totally uncommented code, providing a browsable view of its interfaces, but obviously this is a lot less valuable than running RDoc on well-commented code. `has_rdoc` is a way for you to tell the world, "Yes. It's worth generating the documentation for this gem."

RDoc has the convenience of being very readable in both source and rendered form, making it an excellent choice for an included `README` file with a package. By default, however, the `rdoc` command will run only on source code files. The `extra_rdoc_file` attribute takes an array of paths to non-source files in your gem that you would like to be included in the generation of RDoc documentation.

Adding Dependencies

For your gem to work properly, users are going to need to have BlueCloth installed.

We saw earlier how to set a load-time version dependency for a library. Now we need to tell our gemspec about that dependency, so the installer will ensure that it is present while installing MomLog. We do that with the addition of a single method call to our `Gem::Specification` object.

```
s.add_dependency("BlueCloth", ">= 0.0.4")
```

The arguments to our `add_dependency` method are identical to those of `require_gem`, which we explained earlier.

After generating this gem, attempting to install it on a clean system would look something like this.

```
% gem install pkg/MomLog-1.0.0.gem
Attempting local installation of 'pkg/MomLog-1.0.0.gem'
/usr/local/lib/ruby/site_ruby/1.8/rubygems.rb:50:in `require_gem':
    (LoadError)
Could not find RubyGem BlueCloth (>= 0.0.4)
```

Because you are performing a local installation from a file, RubyGems won't attempt to resolve the dependency for you. Instead, it fails noisily, telling you that it needs BlueCloth to complete the installation. You could then install BlueCloth as we did before, and things would go smoothly the next time you attempted to install the MomLog gem.

If you had uploaded MomLog to the central RubyGems repository and then tried to install it on a clean system, you would be prompted to automatically install BlueCloth as part of the MomLog installation.

```
% gem install -r MomLog
Attempting remote installation of 'MomLog'
Install required dependency BlueCloth? [Yn]   y
Successfully installed MomLog, version 1.0.0
```

Now you've got both BlueCloth and MomLog installed, and your mother can start happily publishing her diary. Had you chosen not to install BlueCloth, the installation would have failed as it did during the local installation attempt.

As you add more features to MomLog, you may find yourself pulling in additional external gems to support those features. The `add_dependency` method can be called multiple times in a single gemspec, supporting as many dependencies as you need it to support.

Ruby Extension Gems

So far, all of the examples we've looked at have been pure Ruby code. However, many Ruby libraries are created as native extensions (see Chapter 21 on page 261). You have two ways to package and distribute this kind of library as a gem. You can distribute the gem in source format and have the installer compile the code at installation time. Alternatively, you can precompile the extensions and distribute one gem for each separate platform you want to support.

For source gems, RubyGems provides an additional `Gem::Specification` attribute called `extensions`. This attribute is an array of paths to Ruby files that will generate Makefiles. The most typical way to create one of these programs is to use Ruby's `mkmf` library (see Chapter 21 on page 261 and the appendix about `mkmf` on page 755). These files are conventionally named `extconf.rb`, though any name will do.

Your mom has a computerized recipe database that is near and dear to her heart. She has been storing her recipes in it for years, and you would like to give her the ability to publish these recipes on the Web for her friends and family. You discover that the recipe program, MenuBuilder, has a fairly nice native API and decide to write a Ruby extension to wrap it. Since the extension may be useful to others who aren't necessarily using MomLog, you decide to package it as a separate gem and add it as an additional dependency for MomLog.

Here's the gemspec.

```
require 'rubygems'
spec = Gem::Specification.new do |s|
  s.name = "MenuBuilder"
  s.version = "1.0.0"
  s.author = "Jo Programmer"
  s.email = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/projects/MenuBuilder"
  s.platform = Gem::Platform::RUBY
  s.summary = "A Ruby wrapper for the MenuBuilder recipe database."
  s.files = ["ext/main.c", "ext/extconf.rb"]
  s.require_path = "."
  s.autorequire = "MenuBuilder"
  s.extensions = ["ext/extconf.rb"]
end
if $0 == __FILE__
  Gem::manage_gems
  Gem::Builder.new(spec).build
end
```

Note that you have to include source files in the specification's `files` list so they'll be included in the gem package for distribution.

When a source gem is installed, RubyGems runs each of its `extensions` programs and then executes the resultant Makefile.

```
% gem install MenuBuilder-1.0.0.gem
Attempting local installation of 'MenuBuilder-1.0.0.gem'
ruby extconf.rb inst MenuBuilder-1.0.0.gem
creating Makefile
make
gcc -fPIC -g -O2  -I. -I/usr/local/lib/ruby/1.8/i686-linux \
     -I/usr/local/lib/ruby/1.8/i686-linux -I.   -c main.c
gcc -shared  -L"/usr/local/lib" -o MenuBuilder.so main.o  \
     -ldl -lcrypt -lm   -lc
make install
install -c -p -m 0755 MenuBuilder.so \
     /usr/local/lib/ruby/gems/1.8/gems/MenuBuilder-1.0.0/.
Successfully installed MenuBuilder, version 1.0.0
```

RubyGems does not have the capability to detect system library dependencies that source gems may have. Should your source gems depend on a system library that is not installed, the gem installation will fail, and any error output from the `make` command will be displayed.

Distributing source gems obviously requires that the consumer of the gem have a working set of development tools. At a minimum, they'll need some kind of `make` program and a compiler. Particularly for Windows users, these tools may not be present. You can get around this limitation by distributing precompiled gems.

Creation of precompiled gems is simple—add the compiled shared object files (DLLs on Windows) to the gemspec's `files` list, and make sure these files are in one of the gem's `require_path` attributes. As with pure Ruby gems, the `require_gem` command will modify Ruby's `$LOAD_PATH`, and the shared object will be accessible via `require`.

Since these gems will be platform specific, you can also use the `platform` attribute (remember this from the first gemspec example?) to specify the target platform for the gem. The `Gem::Specification` class defines constants for Windows, Intel Linux, Macintosh, and pure Ruby. For platforms not included in this list, you can use the value of the `RUBY_PLATFORM` variable. This attribute is purely informational for now, but it's a good habit to acquire. Future RubyGems releases will use the `platform` attribute to intelligently select precompiled gems for the platform on which the installer is running.

Building the Gem File

The MomLog gemspec we just created is runnable as a Ruby program. Invoking it will create a gem file, `MomLog-0.5.0.gem`.

```
% ruby momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

Alternatively, you can use the `gem build` command to generate the gem file.

```
% gem build momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

Now that you've got a gem file, you can distribute it like any other package. You can put it on an FTP server or a Web site for download or e-mail it to your friends. Once your friends have got this file on their local computers (downloading from your FTP server if necessary), they can install the gem (assuming they have RubyGems installed too) by calling

```
% gem install MomLog-0.5.0.gem
Attempting local installation of 'MomLog-0.5.0.gem'
Successfully installed MomLog, version 0.5.0
```

If you would like to release your gem to the Ruby community, the easiest way is to use RubyForge (`http://rubyforge.org`). RubyForge is an open-source project management Web site. It also hosts the central gem repository. Any gem files released using RubyForge's file release feature will be automatically picked up and added to the central gem repository several times each day. The advantage to potential users of your software is that it will be available via RubyGems' remote query and installation operations, making installation even easier.

Building with Rake

Last but certainly not least, we can use Rake to build gems (remember Rake, the pure-Ruby build tool we mentioned back on page 204). Rake uses a command file called a `Rakefile` to control the build. This defines (in Ruby syntax!) a set of *rules* and *tasks.* The intersection of make's rule-driven concepts and Ruby's power make for a build and release automator's dream environment. And, what release of a Ruby project would be complete without the generation of a gem?

For details on how to use Rake, see `http://rake.rubyforge.org`. Its documents are comprehensive and always up-to-date. Here, we'll focus on just enough Rake to build a gem. From the Rake documentation:

Tasks are the main unit of work in a Rakefile. Tasks have a name (usually given as a symbol or a string), a list of prerequisites (more symbols or strings), and a list of actions (given as a block).

Normally, you can use Rake's built-in `task` method to define your own named tasks in your Rakefile. For special cases, it makes sense to provide helper code to automate some of the repetitive work you would have to do otherwise. Gem creation is one of

these special cases. Rake comes with a special *TaskLib*, called GemPackageTask, that helps integrate gem creation into the rest of your automated build and release process.

To use GemPackageTask in your Rakefile, create the gemspec exactly as we did previously, but this time place it into your Rakefile. We then feed this specification to GemPackageTask.

```
require 'rubygems'
Gem::manage_gems
require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.name     = "MomLog"
  s.version  = "0.5.0"
  s.author   = "Jo Programmer"
  s.email    = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/MomLog"
  s.platform = Gem::Platform::RUBY
  s.summary  = "An online Diary for families"
  s.files = FileList["{bin,tests,lib,docs}/**/*"].exclude("rdoc").to_a
  s.require_path     = "lib"
  s.autorequire      = "momlog"
  s.test_file        = "tests/ts_momlog.rb"
  s.has_rdoc         = true
  s.extra_rdoc_files = ["README"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
  s.add_dependency("MenuBuilder", ">= 1.0.0")
end
Rake::GemPackageTask.new(spec) do |pkg|
    pkg.need_tar = true
end
```

Note that you'll have to require the `rubygems` package into your Rakefile. You'll also notice that we've used Rake's `FileList` class instead of `Dir.glob` to build the list of files. `FileList` is smarter than `Dir.glob` for this purpose in that it automatically ignores commonly unused files (such as the `CVS` directory that the CVS version control tool leaves lying around).

Internally, the GemPackageTask generates a Rake target with the identifier

```
package_directory/gemname-gemversion.gem
```

In our case, this identifier will be `pkg/MomLog-0.5.0.gem`. You can invoke this task from the same directory where you've put the Rakefile.

```
% rake pkg/MomLog-0.5.0.gem
(in /home/chad/download/gembook/code/MomLog)
  Successfully built RubyGem
  Name: MomLog
  Version: 0.5.0
  File: MomLog-0.5.0.gem
```

Now that you've got a task, you can use it like any other Rake task, adding dependencies to it or adding it to the dependency list of another task, such as deployment or release packaging.

Maintaining Your Gem (and One Last Look at MomLog)

You've released MomLog, and it's attracting new, adoring users every week. You have taken great care to package it cleanly and are using Rake to build your gem.

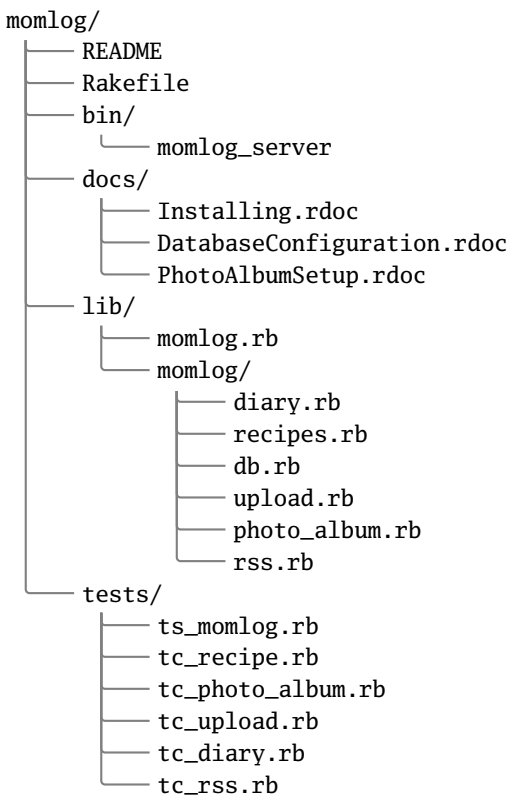
Your gem being "in the wild" with your contact information attached to it, you know that it's only a matter of time before you start receiving feature requests (and fan mail!) from your users. But, your first request comes via a phone call from none other than dear old Mom. She has just gotten back from a vacation in Florida and asks you how she can include her vacation pictures in her diary. You don't think an explanation of command-line FTP would be time well spent, and being the ever-devoted son or daughter, you spend your evening coding a nice photo album module for MomLog.

Since you have added functionality to the application (as opposed to just fixing a bug), you decide to increase MomLog's version number from 1.0.0 to 1.1.0. You also add a set of tests for the new functionality and a document about how to set up the photo upload functionality.

Figure 17.1 on the next page shows the complete directory structure of your final MomLog 1.1.0 package. The final gem specification (extracted from the Rakefile) looks like this.

```
spec = Gem::Specification.new do |s|
  s.name     = "MomLog"
  s.version  = "1.1.0"
  s.author   = "Jo Programmer"
  s.email    = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/MomLog"
  s.platform = Gem::Platform::RUBY
  s.summary  = "An online diary, recipe publisher, " +
               "and photo album for families."
  s.files = FileList["{bin,tests,lib,docs}/**/*"].exclude("rdoc").to_a
  s.require_path     = "lib"
  s.autorequire      = "momlog"
  s.test_file        = "tests/ts_momlog.rb"
  s.has_rdoc         = true
  s.extra_rdoc_files = ["README", "docs/DatabaseConfiguration.rdoc",
                "docs/Installing.rdoc", "docs/PhotoAlbumSetup.rdoc"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
  s.add_dependency("MenuBuilder", ">= 1.0.0")
end
```

Figure 17.1. MomLog package structure



You run Rake over your Rakefile, generating the updated MomLog gem, and you're ready to release the new version. You log into your RubyForge account, and upload your gem to the "Files" section of your project. While you wait for RubyGems' automated process to release the gem into the central gem repository, you type a release announcement to post to your RubyForge project.

Within about an hour, you log in to your mother's Web server to install the new software for her. RubyGems makes things easy, but we have to take special care of Mom.

```
% gem query -rn MomLog

*** REMOTE GEMS ***

MomLog (1.1.0, 1.0.0)
    An online diary, recipe publisher, and photo album for families.
```

Great! The query indicates that there are two versions of MomLog available now. You type the `install` command without specifying a version argument, because you know that the default is to install the most recent version.

```
% gem install -r MomLog
Attempting remote installation of 'MomLog'
Successfully installed MomLog, version 1.1.0
```

You haven't changed any of the dependencies for MomLog, so your existing BlueCloth and MenuBuilder installations meet the requirements for MomLog 1.1.0.

Now that Mom's happy, it's time to go try some of her recently posted recipes.