

The
Pragmatic
Programmers

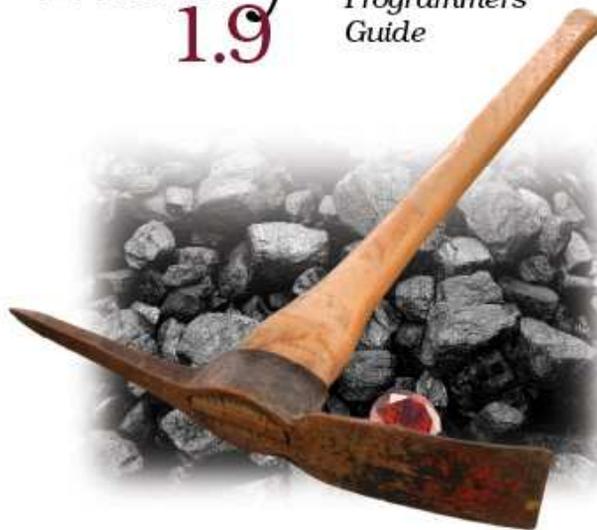
10th

Anniversary

Updated for
Ruby 1.9.2

Programming Ruby 1.9

*The Pragmatic
Programmers'
Guide*



Dave Thomas
with Chad Fowler and Andy Hunt

The Facets  of Ruby Series

Extracted from:

Programming Ruby 1.9

The Pragmatic Programmers' Guide

This PDF file contains pages extracted from Programming Ruby 1.9, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Programming Ruby 1.9

The Pragmatic Programmers' Guide

Dave Thomas

with Chad Fowler
Andy Hunt

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <http://www.pragprog.com>.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-08-5

ISBN-13: 978-1-934356-08-1

Printed on acid-free paper.

3.0 printing, November 2010

Version: 2010-11-5

Namespaces, Source Files, and Distribution

As your programs grow (and they all seem to grow over time), you'll find that you'll need to start organizing your code—simply putting everything into a single huge file becomes unworkable (and makes it hard to reuse chunks of code in other projects). So, we need to find a way to split our project into multiple files and then to knit those files together as our program runs.

There are two major aspects to this organization. The first is internal to your code: how do you prevent different things with the same name from clashing? The second area is related: how do you conveniently organize the source files in your project?

16.1 Namespaces

We've already encountered a way that Ruby helps you manage the names of things in your programs. If you define methods or constants in a class, Ruby ensures that their names can be used only in the context of that class (or its objects, in the case of instance methods):

```
class Triangle
  SIDES = 3

  def area
    # ..
  end
end

class Square
  SIDES = 4

  def initialize(side_length)
    @side_length = side_length
  end
  def area
    @side_length * @side_length
  end
end

puts "A triangle has #{Triangle::SIDES} sides"

sq = Square.new(3)
puts "Area of square = #{sq.area}"
```

produces:

```
A triangle has 3 sides
Area of square = 9
```

Both classes define a constant called `SIDES` and an instance method `area`, but these things don't get confused. You access the instance method via objects created from the class, and you access the constant by prefixing it with the name of the class followed by a double colon. The double colon (`::`) is Ruby's namespace resolution operator. The thing to the left must be a class or module, and the thing to the right is a constant defined in that class or module.¹

So, putting code inside a module or class is a good way of separating it from other code. Ruby's `Math` module is a good example—it defines constants such as `Math::PI` and `Math::E` and methods such as `Math.sin` and `Math.cos`. You can access these constants and methods via the `Math` module object:

```
Math::E # => 2.718281828459045
Math.sin(Math::PI/6.0) # => 0.49999999999999994
```

(Modules have another, significant use—they implement Ruby's *mix-in* functionality, which we discussed on page 94.)

Ruby has an interesting little secret. The names of classes and modules are themselves just constants.² And that means that if you define classes or modules inside other classes and modules, the names of those inner classes follow the same namespacing rules as other constants:

```
module Formatters
  class Html
    # ...
  end
  class Pdf
    # ...
  end
end

html_writer = Formatters::Html.new
```

You can nest classes and modules inside other classes and modules to any depth you want (although it's rare to see them more than three deep).

So, now we know that we can use classes and modules to partition the names used by our programs. The second question to answer is, what do we do with the source code?

16.2 Organizing Your Source

This section covers two related issues: how do we split our source code into separate files, and where in the file system do we put those files?

1. The thing to the right of the `::` can also be a class or module method, but this use is falling out of favor—using a period makes it clearer that it's just a regular old method call.
2. Remember that we said that most everything in Ruby is an object. Well, classes and modules are, too. The name that you use for a class, such as `String`, is really just a Ruby constant containing the object representing that class.

Some languages, such as Java, make this easy. They dictate that each outer-level class should be in its own file and that file should be named according to the name of the class. Other languages, such as Ruby, have no rules relating source files and their content. In Ruby, you're free to organize your code as you like.

But, in the real world, you'll find that some kind of consistency really helps. It will make it easier for you to navigate your own projects, and it will also help when you read (or incorporate) other people's code.

So, the Ruby community is gradually adopting a kind of de facto standard. In many ways, it follows the spirit of the Java model, but without some of the inconveniences suffered by our Java brethren. Let's start with the basics.

Small Programs

Small, self-contained scripts can be in a single file. However, if you do this, you won't easily be able to write automated tests for your program, because the test code won't be able to load the file containing your source without the program itself running. So, if you want to write a small program that also has automated tests, split that program into a trivial driver that provides the external interface (the command-line part of the code) and one or more files containing the rest. Your tests can then exercise these separate files without actually running the main body of your program.

Let's try this for real. Here's a simple program that finds anagrams in a dictionary. Feed it one or more words, and it gives you the anagrams of each. Here's an example:

```
$ ruby anagram.rb teaching code
Anagrams of teaching: cheating, teaching
Anagrams of code: code, coed
```

If we were typing in this program for casual use, we might just enter it into a single file (perhaps `anagram.rb`). It would look something like this:

[Download](#) packaging/anagram.rb

```
#!/usr/bin/env ruby

require 'optparse'

dictionary = "/usr/share/dict/words"

OptionParser.new do |opts|
  opts.banner = "Usage: anagram [ options ] word..."

  opts.on("-d", "--dict path", String, "Path to dictionary") do |dict|
    dictionary = dict
  end

  opts.on("-h", "--help", "Show this message") do
    puts opts
    exit
  end
end
```

```

begin
  ARGV << "-h" if ARGV.empty?
  opts.parse!(ARGV)
rescue OptionParser::ParseError => e
  STDERR.puts e.message, "\n", opts
  exit(-1)
end
end

# convert "wombat" into "abmotw". All anagrams share a signature
def signature_of(word)
  word.unpack("c*").sort.pack("c*")
end

signatures = Hash.new

File.foreach(dictionary) do |line|
  word = line.chomp
  signature = signature_of(word)
  (signatures[signature] ||= []) << word
end

ARGV.each do |word|
  signature = signature_of(word)
  if signatures[signature]
    puts "Anagrams of #{word}: #{signatures[signature].join(', ')}"
  else
    puts "No anagrams of #{word} in #{dictionary}"
  end
end
end

```

Then someone asks us for a copy, and we start to feel embarrassed. It has no tests, and it isn't particularly well packaged.

Looking at the code, there are clearly three sections. The first twenty-five or so lines do option parsing, the next ten or so lines read and convert the dictionary, and the last few lines look up each command-line argument and report the result. Let's split our file into four parts:

- An option parser
- A class to hold the lookup table for anagrams
- A class that looks up words given on the command line
- A trivial command-line interface

The first three of these are effectively library files, used by the fourth.

Where do we put all these files? The answer is driven by some strong Ruby conventions, first seen in Minero Aoki's `setup.rb` and later enshrined in the RubyGems system. We'll create a directory for our project containing (for now) three subdirectories:

```

anagram/    <- top-level
  bin/      <- command-line interface goes here
  lib/      <- three library files go here
  test/     <- test files go here

```

Now let's look at the library files. We know we're going to be defining (at least) three classes. Right now, these classes will be used only inside our command-line program, but it's conceivable that other people might want to include one or more of our libraries in their own code. This means that we should be polite and not pollute the top-level Ruby namespace with the names of all our classes and so on. We'll create just one top-level module, `Anagram`, and then place all our classes inside this module. This means that the full name of (say) our options-parsing class will be `Anagram::Options`.

This choice informs our decision on where to put the corresponding source files. Because class `Options` is inside the module `Anagram`, it makes sense to put the corresponding file, `options.rb`, inside a directory named `anagram/` in the `lib/` directory. This helps people who read your code in the future; when they see a name like `A::B::C`, they know to look for `c.rb` in the `b/` directory in the `a/` directory of your library. So, we can now flesh out our directory structure with some files:

```
anagram/
  bin/
    anagram      <- command-line interface
  lib/
    anagram/
      finder.rb
      options.rb
      runner.rb
  test/
    ... various test files
```

Let's start with the option parser. Its job is to take an array of command-line options and return to us the path to the dictionary file and the list of words to look up as anagrams. The source, in `lib/anagram/options.rb`, looks like this:

[Download](#) `packaging/anagram/lib/anagram/options.rb`

```
require 'optparse'

module Anagram
  class Options

    DEFAULT_DICTIONARY = "/usr/share/dict/words"

    attr_reader :dictionary
    attr_reader :words_to_find

    def initialize(argv)
      @dictionary = DEFAULT_DICTIONARY
      parse(argv)
      @words_to_find = argv
    end

    private

    def parse(argv)
      OptionParser.new do |opts|
        opts.banner = "Usage: anagram [ options ] word..."

        opts.on("-d", "--dict path", String, "Path to dictionary") do |dict|
```

```

        @dictionary = dict
      end

      opts.on("-h", "--help", "Show this message") do
        puts opts
        exit
      end

      begin
        argv = ["-h"] if argv.empty?
        opts.parse!(argv)
      rescue OptionParser::ParseError => e
        STDERR.puts e.message, "\n", opts
        exit(-1)
      end
    end
  end
end
end
end

```

Notice how we define the Options class inside a top-level Anagram module.

Let's write some unit tests for this code. This should be relatively easy, because options.rb is self-contained—the only external dependency is to the standard Ruby OptionParser. We'll use the standard Ruby Test::Unit framework, extended using the Shoulda gem.³ We'll put the source of this test in the file test/test_options.rb:

[Download](#) packaging/anagram/test/test_options.rb

```

require 'test/unit'
require 'shoulda'
require_relative '../lib/anagram/options'

```

```

class TestOptions < Test::Unit::TestCase

```

```

  context "specifying no dictionary" do
    should "return default" do
      opts = Anagram::Options.new(["someword"])
      assert_equal Anagram::Options::DEFAULT_DICTIONARY, opts.dictionary
    end
  end
end

```

```

  context "specifying a dictionary" do
    should "return it" do
      opts = Anagram::Options.new(["-d", "mydict", "someword"])
      assert_equal "mydict", opts.dictionary
    end
  end
end

```

```

  context "specifying words and no dictionary" do
    should "return the words" do
      opts = Anagram::Options.new(["word1", "word2"])
      assert_equal ["word1", "word2"], opts.words_to_find
    end
  end
end

```

3. We talk about Shoulda starting on page 200.

```

context "specifying words and a dictionary" do
  should "return the words" do
    opts = Anagram::Options.new(["-d", "mydict", "word1", "word2"])
    assert_equal ["word1", "word2"], opts.words_to_find
  end
end
end
end

```

The line to note in this file is as follows:

```
require_relative '../lib/anagram/options'
```

This is where we load in the source of the Options class we just wrote. We use the new Ruby 1.9 ^{1.9} feature, `require_relative`. This is like regular old `require`, but it always loads from a path relative to the directory of the file that invokes it.

```

$ ruby test/test_options.rb
Loaded suite test/test_options
Started
....
Finished in 0.001342 seconds.
4 tests, 4 assertions, 0 failures, 0 errors, 0 skips

```

The finder code (in `lib/anagram/finder.rb`) is modified slightly from the original version. To make it easier to test, we'll have the default constructor take a list of words, rather than a filename. We'll then provide an additional factory method, `from_file`, that takes a filename and constructs a new Finder from that file's contents:

[Download](#) `packaging/anagram/lib/anagram/finder.rb`

```

module Anagram
  class Finder

    def self.from_file(file_name)
      new(File.readlines(file_name))
    end

    def initialize(dictionary_words)
      @signatures = Hash.new
      dictionary_words.each do |line|
        word = line.chomp
        signature = Finder.signature_of(word)
        (@signatures[signature] ||= []) << word
      end
    end

    def lookup(word)
      signature = Finder.signature_of(word)
      @signatures[signature]
    end

    def self.signature_of(word)
      word.unpack("c*").sort.pack("c*")
    end
  end
end
end

```

Again, we embed the Finder class inside the top-level Anagram module. And, again, this code is self-contained, allowing us to write some simple unit tests:

[Download](#) packaging/anagram/test/test_finder.rb

```
require 'test/unit'
require 'shoulda'
require_relative '../lib/anagram/finder'

class TestFinder < Test::Unit::TestCase

  context "signature" do
    { "cat" => "act", "act" => "act", "wombat" => "abmotw" }.each do
      |word, signature|
        should "be #{signature} for #{word}" do
          assert_equal signature, Anagram::Finder.signature_of(word)
        end
      end
    end
  end

  context "lookup" do
    setup do
      @finder = Anagram::Finder.new(["cat", "wombat"])
    end

    should "return word if word given" do
      assert_equal ["cat"], @finder.lookup("cat")
    end

    should "return word if anagram given" do
      assert_equal ["cat"], @finder.lookup("act")
      assert_equal ["cat"], @finder.lookup("tca")
    end

    should "return nil if no word matches anagram" do
      assert_nil @finder.lookup("wibble")
    end
  end
end
```

These go in lib/test_finder.rb:

```
$ ruby test/test_finder.rb
Loaded suite test/test_finder
Started
.....
Finished in 0.000916 seconds.
6 tests, 7 assertions, 0 failures, 0 errors, 0 skips
```

So, now we have all the support code in place. We just need to run it. We'll make the command-line interface—the thing the end user actually executes—really thin. It's in the bin/ directory in a file called anagram (no .rb extension, because that would be unusual in a command).⁴

4. If you're on Windows, you might want to wrap the invocation of this in a .cmd file.

[Download](#) packaging/anagram/bin/anagram

```
#!/usr/local/rubybook/bin/ruby
require 'anagram/runner'

runner = Anagram::Runner.new(ARGV)
runner.run
```

The code that this script invokes (lib/anagram/runner.rb) knits our other libraries together:

[Download](#) packaging/anagram/lib/anagram/runner.rb

```
require_relative 'finder'
require_relative 'options'

module Anagram
  class Runner

    def initialize(argv)
      @options = Options.new(argv)
    end

    def run
      finder = Finder.from_file(@options.dictionary)
      @options.words_to_find.each do |word|
        anagrams = finder.lookup(word)
        if anagrams
          puts "Anagrams of #{word}: #{anagrams.join(', ')}"
        else
          puts "No anagrams of #{word} in #{@options.dictionary}"
        end
      end
    end
  end
end
```

In this case, the two libraries `finder` and `options` are in the same directory as the `runner`, so `require_relative` finds them perfectly.

Now that all our files are in place, we can run our program from the command line:

```
$ ruby -I lib bin/anagram teaching code
Anagrams of teaching: cheating, teaching
Anagrams of code: code, coed
```

There's nothing like a cheating coed teaching code.

16.3 Distributing and Installing Your Code

Now that we have our code a little tidier, it would be nice to be able to distribute it to others. We could just zip or tar it up and send them our files, but then they'd have to run the code the way we do, remembering to add the correct `-I lib` options and so on. They'd also have some problems if they wanted to reuse one of our library files—it would be sitting in some random directory on their hard drive, not in a standard location used by Ruby.

So, we're really looking for a way to take our little application and *install* it in a standard way.

Now, Ruby already has a standard installation structure on your computer. When Ruby is installed, it puts its commands (`ruby`, `ri`, `irb`, and so on) into a directory of binary files. It puts its libraries into another directory tree and documentation somewhere else. So, one option would be to write an installation script that you distribute with your code that copies components of your application to the appropriate directories on the system that's installing it.

Being a Good Packaging Citizen

So, I've ignored some stuff that you'd want to do before distributing your code to the world. Your distributed directory tree really should have a `README` file, outlining what it does and probably containing a copyright statement; an `INSTALL` file, giving installation instructions; and a `LICENSE` file, giving the license it is distributed under.

You'll probably want to distribute some documentation, too. This would go in a directory called `doc/`, parallel with the `bin` and `lib` directories.

You might also want to distribute native C-language extensions with your library. These extensions would go into your project's `ext/` directory.

Using RubyGems

The RubyGems package management system (which is also just called *Gems*) has become the standard for distributing and managing Ruby code packages. As of Ruby 1.9, it comes bundled with Ruby itself.⁵

RubyGems is also a great way to package your own code. If you want to make your code available to the world, RubyGems is the way to go. Even if you're just sending code to a few friends or within your company, RubyGems gives you dependency and installation management—one day you'll be grateful for that.

RubyGems needs to know information about your project that isn't contained in the directory structure. Instead, you have to write a short RubyGems specification: a GemSpec. Create this in a separate file named *project-name.gemspec* in the top-level directory of your application (in our case, the file is `anagram.gemspec`):

[Download](#) `packaging/anagram/anagram.gemspec`

```
Gem::Specification.new do |s|
  s.name           = "anagram"
  s.summary        = "Find anagrams of words supplied on the command line"
  s.description    = File.read(File.join(File.dirname(__FILE__), 'README'))
  s.requirements  =
    [ 'An installed dictionary (most Unix systems have one)' ]
  s.version       = "0.0.1"
  s.author        = "Dave Thomas"
  s.email         = "dave@pragprog.com"
  s.homepage      = "http://pragdave.pragprog.com"
  s.platform      = Gem::Platform::RUBY
```

5. Prior to RubyGems, folks often distributed a tool called `setup.rb` with their libraries. This would install the library into the standard Ruby directory structure on a user's machine.

```

s.required_ruby_version = '>=1.9'
s.files                 = Dir['**/**']
s.executables          = [ 'anagram' ]
s.test_files           = Dir["test/test*.rb"]
s.has_rdoc              = false
end

```

The first line of the spec gives our gem a name. This is important—it will be used as part of the package name, and it will appear as the name of the gem when installed. Although it can be mixed case, we find that confusing, so do our poor brains a favor and use lowercase for gem names.

The version string is significant, because RubyGems will use it both for package naming and for dependency management. Stick to the *x.y.z* format.⁶

The platform field tells RubyGems that (in this case) our gem is pure Ruby code. It's also possible to package (for example) Windows *.exe* files inside a gem, in which case you'd use `Gem::Platform::Win32`.

The next line is also important (and oft-forgotten by package developers). Because we use `require_relative`, our gem will run only with Ruby 1.9 and newer.

We then tell RubyGems which files to include when creating the gem package. Here we've been lazy and included everything. You can be more specific.

The `s.executables` line tells RubyGems to install the `anagram` command-line script when the gem gets installed on a user's machine.

To save space, we haven't added RDoc documentation comments to our source files (RDoc is described in Chapter 19, *Documenting Ruby*, on page 276). The last line of the spec tells RubyGems not to try to extract documentation when the gem is installed.

Obviously I've skipped a lot of details here. A full description of GemSpecs is available online,⁷ along with other documents on RubyGems.⁸

Packaging Your RubyGem

Once the gem specification is complete, you'll want to create the packaged `.gem` file for distribution. This is as easy as navigating to the top level of your project and typing this:

```

$ gem build anagram.gemspec
WARNING: no rubyforge_project specified
Successfully built RubyGem
Name: anagram
Version: 0.0.1
File: anagram-0.0.1.gem

```

You'll find you now have a file called `anagram-0.0.1.gem`.

6. And read <http://www.rubygems.org/read/chapter/7> for information on what the numbers mean.

7. <http://www.rubygems.org/read/book/4>

8. <http://www.rubygems.org/>

```
$ ls *gem
anagram-0.0.1.gem
```

You can install it:

```
$ sudo gem install pkg/anagram-0.0.1.gem
Successfully installed anagram-0.0.1
1 gem installed
```

And check to see that it is there:

```
$ gem list anagram -d

*** LOCAL GEMS ***
anagram (0.0.1)
  Author: Dave Thomas
  Homepage: http://pragdave.pragprog.com
  Installed at: /usr/local/lib/ruby/gems/1.9.0
```

Find anagrams of words supplied on the command line

Now you can send your .gem file to friends and colleagues or share it from a server. Or, you could go one better and share it from a RubyGems server.

If you have RubyGems installed on your local box, you can share them over the network to others. Simply run this:

```
$ gem server
Server started at http://[::ffff:0.0.0.0]:8808
Server started at http://0.0.0.0:8808
```

This starts a server (by default on port 8808, but the --port option overrides that). Other people can connect to your server to list and retrieve RubyGems:

```
$ gem list --remote --source http://dave.local:8808

*** REMOTE GEMS ***

anagram (0.0.1)
builder (2.1.2, 0.1.1)
..
```

This is particularly useful in a corporate environment.

You can speed up the serving of gems by creating a static index—see the help for `gem generate_index` for details.

Serving Public RubyGems

RubyGems.org (<http://rubygems.org>) has become the main repository for public Ruby libraries and projects. And, if you create a RubyGems.org account, you can push your .gem file to their public servers.

```
$ gem push anagram-0.0.1.gem
Enter your RubyGems.org credentials.
Don't have an account yet? Create one at http://rubygems.org/sign_up
Email: dave@pragprog.com
```

```

Password:
Signed in.
Pushing gem to RubyGems.org...
Successfully registered gem: anagram (0.0.1)

```

And, at that point, any Ruby user in the world can do this:

```
$ gem search -r anagram
```

```
*** REMOTE GEMS ***
```

```
anagram (0.0.1)
```

```
..
```

and, even better, can do this:

```
$ gem install anagram
```

Adding Even More Automation

The Jewler library⁹ can create a new project skeleton that follows the layout guidelines in this chapter. It also provides a set of Rake tasks that will help create and manage your project as a gem.

If you're a Rails user, you'll have come across bundler, a utility that manages the gems used by your application. Bundler is more general than this: it can be used to manage the gems used by any piece of Ruby code.

Some folks like the extra features of these utilities, while others prefer the leaner “roll-your-own” approach. Whatever route you take, taking the time to package your applications and libraries will pay you back many times over.

See You on GitHub

Finally, if you're developing a Ruby application or library that you'll be sharing, you'll probably want to store it on GitHub.¹⁰ Although it started as a public Git repository, GitHub is now a community in its own right. It's a home away from home for many in the Ruby community.

9. <http://github.com/technicalpickles/jeweler>

10. <http://github.com>