Extracted from:

Rails 4 Test Prescriptions

Build a Healthy Codebase

This PDF file contains pages extracted from *Rails 4 Test Prescriptions*, 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 printed versions; the content is otherwise identical.

Copyright © 2014 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.

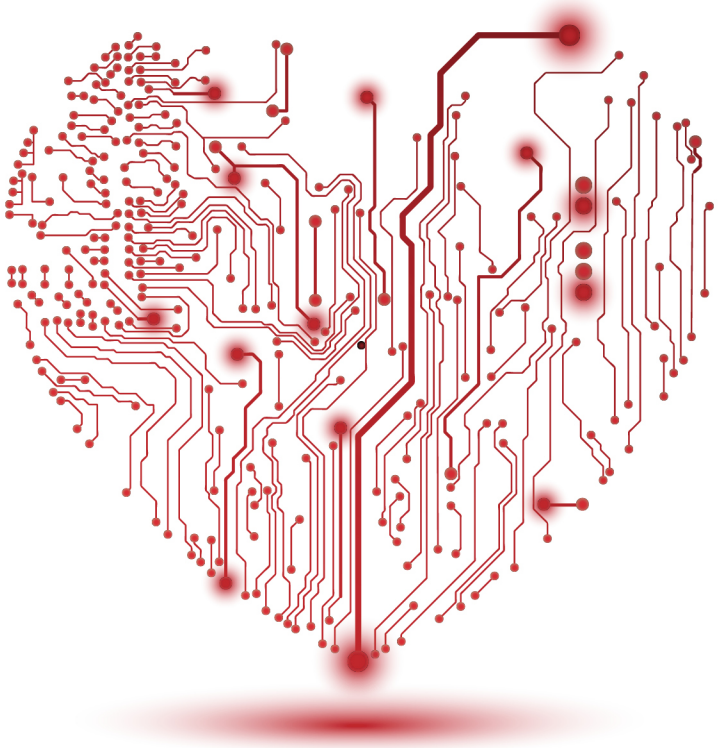
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails 4 Test Prescriptions

Build a Healthy Codebase



Noel Rappin

Edited by Lynn Beighley

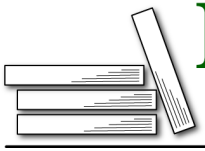
Rails 4 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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, PragProg 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 *https://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Copyright © 2014 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-13: 978-1-941222-19-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2014

You have a problem.

You are the team leader for a development team that is distributed across multiple locations. You'd like to be able to maintain a common list of tasks for the team. For each task, you'd like to maintain data such as the status of the task, which pair of developers the task is assigned to, and so on. You'd also like to be able to use the past rate of task completion to estimate the project's completion date. For some reason none of the existing tools that do this are suitable (work with me here, folks) and so you've decided to roll your own. We'll call it Gatherer.

As you sit down to start working on Gatherer, your impulse is going to be to start writing code immediately. That's a great impulse, and we're just going to turn it about ten degrees east. Instead of starting off by writing code, we're going to start off by writing tests.

In our introductory chapter we talked about why you might work test-first. In this chapter we'll look at the basic mechanics of a TDD cycle by building a feature in a Rails application. We'll start by creating some business logic with our models, because model logic is the easiest part of a Rails application to test—in fact, most of this chapter won't touch Rails at all. In the next chapter we'll start testing the controller and view parts of the Rails framework.

Infrastructure

First off, we'll need a Rails application. We'll be using Rails 4.1.7 and Ruby 2.1.4; use of Ruby 2.0–specific features will be minimal.

We'll start by generating the Rails application from the command line:

```
% rails new gatherer
```

This will create the initial directory structure and code for a Rails application. It will also run `bundle install` to load initial gems. I assume that you are already familiar with Rails core concepts, I won't spend a lot of time re-explaining them. If you are not familiar with Rails, *Agile Web Development with Rails [RTH13]* is still the gold standard for getting started.

We need to create our databases. For ease of setup and distribution we'll stick to the Rails default, which is SQLite. (You'll need to have SQLite installed; see http://www.sqlite.org for details if it is not already on your machine.)

```
% cd gatherer
% rake db:create:all
% rake db:migrate
```

We need the db:migrate call even though we haven't actually created a database migration, because it sets up the schema.rb file that Rails uses to rebuild the test database. In Rails 4.1 the test database is automatically maintained when the schema.rb file changes.

The Requirements

The most complex business logic we need to build concerns forecasting a project's progress. We want to be able to predict the end date of a project and determine whether that project is on schedule or not.

In other words, given a project and a set of tasks, some of which are done and some of which are not, use the rate at which tasks are being completed to estimate the project's end date. Also, compare that projected date to a deadline to determine if the project is on time.

This is a good example problem for TDD because, while I have a sense of what the answer is, I don't have a very strong sense of the best way to structure the algorithm. TDD will help, guiding me toward reasonable code design.

Installing RSpec

Before we start testing, we'll need to load RSpec, our testing library.

We'll be talking about RSpec 3, which has some significant syntactical differences from previous versions. We'll largely ignore those differences and focus on only the new syntax.

To add RSpec to a Rails project, add the rspec-rails gem to your Gemfile:

```
group :development, :test do
  gem 'rspec-rails', '~> 3.1'
end
```

The rspec-rails gem depends on the rspec gem proper. The rspec gem is mostly a list of other dependencies where the real work gets done, including rspec-core, rspec-expectations, and rspec-mocks. Sometimes rspec and rspec-rails are updated separately; you might choose to explicitly specify both versions in the Gemfile. Also, rspec goes in the development group as well as the test group so that you can call rspec from the command line, where development mode is the default. (RSpec switches to the test environment as it initializes.)

Install with bundle install. Then we need to generate some installation files using the rspec:install generator:

```
$ bundle install
$ rails generate rspec:install
```

```
create  .rspec
create  spec
create  spec/spec_helper.rb
create  spec/rails_helper.rb
```

This generator creates the following:

- The .rspec file, where RSpec run options go. In RSpec 3.1 the default currently sets two options, --color, which sets terminal output in color, and --require spec_helper, which ensures that the spec_helper file is always required.
- The spec directory, which is where your specs go. RSpec does not automatically create subdirectories like controller and model on installation. The subdirectories can be created manually or will be created by Rails generators as needed.
- The spec_helper.rb and rails_helper.rb files, which contain setup information. The spec_helper.rb file contains general RSpec settings while the rails_helper.rb file, which requires spec_helper, loads the Rails environment and contains settings that depend on Rails. The idea behind having two files is to make it easier to write specs that do not load Rails.

The rspec-rails gem does a couple of other things when loaded in a Rails project:

- Adds a Rake file that changes the default Rake test to run RSpec instead of Minitest and defines a number of subtasks such as spec:models that filter an RSpec run to a subset of the overall RSpec suite.
- Sets itself up as the test framework of choice for the purposes of future Rails generators. Later, when you set up, say, a generated model or resource, RSpec's generators are automatically invoked to create appropriate spec files.

Where to Start?

"Where do I start testing?" is one of the most common questions that people have when they start with TDD. Traditionally, my answer is a somewhat glib "start anywhere." While true, this is less than helpful.

A good option for starting a TDD cycle is to specify the initialization state of the objects or methods under test. Another is the "happy path"—a single representative example of the error-free version of the algorithm. Which starting point you choose depends on how complicated the feature is. In this case it's sufficiently complex that we will start with the initial state and move to the happy path. As a rule of thumb, if it takes more than a couple of steps to define an instance of the application, I'll start with initialization only.

• Click HERE to purchase this book now. discuss

Prescription 3	Initializing objects is a good starting place for a TDD process. Another good approach is to use the test to design what you want a successful interaction of the feature to look like.

This application is made up of projects and tasks. A newly created project would have no tasks. What can we say about that brand-new project?

If there are no outstanding tasks, then there's nothing more to do. A project with nothing left to do is done. The initial state, then, is a project with no tasks, and we can specify that the project is done. That's not inevitable; we could specify that a project with no tasks is in some kind of empty state.

We don't have any infrastructure in place yet, so we need to create the test file ourselves—we're deliberately not using Rails generators right now. We're using RSpec, so the spec goes in the spec directory using a file name that is parallel to the application code in the app directory. We think this is a test of a project model, which would be in app/models/project.rb, so we'll put the spec in spec/models/project_spec.rb. We're making very small design decisions here, and so far these decisions are consistent with Rails conventions.

Here's our spec of a project's initial state:

basics_rspec/01/gatherer/spec/models/project_spec.rb

```
Line 1  require 'rails_helper'
    2
    3  RSpec.describe Project do
    4    it "considers a project with no tasks to be done" do
    5      project = Project.new
    6      expect(project.done?).to be_truthy
    7    end
    8  end
```

Let's talk about this spec at two levels: the logistics of the code in RSpec and what this test is doing for us in our TDD process.

This file has four interesting RSpec and Rails features:

- Requiring rails_helper
- Defining a suite with describe
- Writing an RSpec example with it
- Specifying a particular state with expect

On line 1, we require the file rails_helper, which contains Rails-related setup common to all tests. We'll peek into that file in the next chapter, when we talk about more Rails-specific test features. The rails_helper file, in turn, requires a file named spec_helper, which contains non-Rails RSpec setup.

What's a Spec?

What do you call the things you write in an RSpec file? If you are used to TDD and Minitest, the temptation to call them tests can be overwhelming. However, as we've discussed, the BDD planning behind RSpec suggests it's better not to think of your RSpec code as tests, which are things happen after the fact. So, what are they?

The RSpec docs and code refer to the elements of RSpec as "examples." The term I hear most often is simply "spec," as in "I need to write some specs for that feature." I've tried to use "spec" and "example" rather than "test" in this book, but I suspect I'll slip up somewhere. Bear with me.

We use the RSpec.describe method on line 3. In RSpec, the describe method defines a suite of tests that can share a common setup. The describe method takes one argument (typically either a class name or a string) and a block. The argument documents what the test suite is supposed to cover, and the block contains the test suite itself.

As you'll see in a little bit, describe calls can be nested. By convention, the outermost call often has the name of the class under test. In RSpec 3, the outermost describe call should be invoked as RSpec.describe, which is part of a general design change in RSpec 3 to avoid adding methods to Ruby's Kernel and Object namespaces. Nested calls can use just plain describe, since RSpec manages those calls internally.

The actual spec is defined with the it method, which takes an optional string argument that documents the spec, and then a block that is the body of the spec. The string argument is not used internally to identify the spec—you can have multiple specs with the same description string.

RSpec also defines specify as an alias for it. Normally, we'd use it when the method takes a string argument to give the spec a readable natural-language name. (Historically the string argument started with "should," so the name would be something like "it should be valid," but that construct has gotten less popular recently.) For single-line tests in which a string description is unnecessary, we use specify to make the single line read more clearly, such as this:

```
specify { expect(user.name).to eq("fred") }
```

On line 6 we make our first testable specification about the code: expect(project.done?) to be_truthy. The general form of an RSpec expectation is expect(actual_value).to(matcher), with the parentheses around the matcher often omitted in practice.

Let's trace through what RSpec does with our first expectation. First is the expect call itself, expect(project.done?). RSpec defines the expect method, which takes in any object as an argument and returns a special RSpec proxy object called an ExpectationTarget.

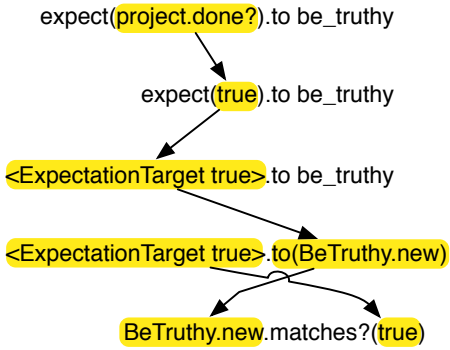
The ExpectationTarget holds on to the object that was the argument to expect, and itself responds to two messages: to and not_to. (Okay, technically three messages, since to_not exists as an alias.) Both to and not_to are ordinary Ruby methods that expect as an argument an RSpec matcher. There's nothing special about an RSpec matcher; at base it's just an object that responds to a matches? method. There are several predefined matchers and you can write your own.

In our case, be_truthy is a method defined by RSpec to return the BeTruthy matcher. You could get the same behavior with

```
expect(project.done?).to(RSpec::BuiltIn::BeTruthy.new)
```

but you probably would agree that the idiomatic version reads better.

The ExpectationTarget is now holding on to two objects: the object being matched (in our case, project.done?) and the matcher (be_truthy). When the spec is executed, RSpec calls the matches? method on the matcher, with the object being matched as an argument. If the expectation uses to, then the expectation passes if matches? is true. If the expectation uses not_to, then it checks for a does_not_match? method in the matcher. If there is no such method it falls back to passing if matches? is false. This is shown in the following diagram.



Compared to other testing libraries, RSpec shifts the tone from an assertion, potentially implying already-implemented behavior, to an expectation implying future behavior. The RSpec version, arguably, reads more smoothly (though some strenuously dispute this). Later in this chapter we'll cover some other tricks RSpec uses to make matchers read like natural language.

From an RSpec perspective we're creating an object and asserting an initial condition. What are we doing from a TDD perspective and why is this useful?

Small as it might seem, we've performed a little bit of design. We are starting to define the way parts of our system communicate with each other, and the tests ensure the visibility of important information in our design.

This small test makes three claims about our program:

- There is a class called Project.
- You can query instances of that class as to whether they are done.
- A brand-new instance of Project qualifies as done.

This last assertion isn't inevitable—we could say that you aren't done unless there is at least one completed task, but it's a choice we're making in our application's business logic.

RSpec Predefined Matchers

Before we run the tests, let's take a quick look at RSpec's basic matchers. RSpec predefines a number of matchers. Here's a list of the most useful ones; for a full list visit https://relishapp.com/rspec/rspec-expectations/v/3-0/docs/built-in-matchers.

```
expect(array).to all(matcher)
expect(actual).to be_truthy
expect(actual).to be_falsy
expect(actual).to be_nil
expect(actual).to be_between(min, max)
expect(actual).to be_within(delta).of(actual)
expect { block }.to change(receiver, message, &block)
```

```
expect(actual).to contain_exactly(expected)
expect(actual).to eq(actual)
expect(actual).to have_attributes(key/value pairs)
expect(actual).to include(*expected)
expect(actual).to match(regex)
expect { block }.to raise_error(exception)
expect(actual).to satisfy { block }
```

Most of these mean what they appear to say. The all matcher takes a different matcher as an argument and passes if all elements of the array pass that internal matcher, as in expect([1, 2, 3]).to all(be_truthy). The change matcher passes if the value of receiver.message changes when the block is evaluated. The contain_exactly matcher is true if the expected array and the actual array contain the same elements, regardless of order. The satisfy matcher passes if the block evaluates to true. The matchers that take block arguments are for specifying a side effect of the block's execution—that it raises an error or that it changes a different value—rather than the state of a particular object. Any of these except raise_error can be negated by using not_to instead of to.

RSpec 3 allows you to compose matchers to express compound behavior, and most of these matchers have alternate forms that allow them to read better when composed. Composing matchers allows you to specify, for example, multiple array values in a single statement and get useful error messages.

Here is a contrived example:

```
expect(["cheese", "burger"]).to contain_exactly(
    a_string_matching(/ch/), a_string_matching(/urg/))
```

In this case a_string_matching is an alias for match, and the arguments to contain_exactly are themselves matchers that must match individual elements of the array to allow the entire compound matcher to pass.

Running Our Test

Having written our first test, we'd like to execute it. Although RSpec provides Rake tasks for executing RSpec, I recommend using the rspec command directly to avoid the overhead of starting up Rake. If you use rspec with no arguments, then RSpec will run over the entire spec directory. You can also give RSpec an individual file, directory, or line to run. For full details on those options, see Chapter 15, *Running Tests Faster and Running Faster Tests*, on page ?.

What Happens When We Run the Test?

It fails. We haven't written any code yet.

That's Funny. What Really Happens—Internally?

When you run rspec with no arguments, RSpec loads every file in the spec directory. The following things happen (this process is slightly simplified for clarity):

1. Each file in the spec directory is loaded. Usually these files will contain just these specs, but sometimes you'll define extra helper methods or dummy classes that exist just to support the tests.

2. Each RSpec file typically requires the rails_helper.rb file. The rails_helper.rb file loads the Rails environment itself, as well as the spec_helper.rb, which contains non-Rails RSpec setup. In the default Rails configuration the .rspec file automatically loads spec_helper.rb.

3. By default the rails_helper.rb file sets up transactional fixtures. *Fixtures* are a Rails mechanism that defines global ActiveRecord data that is available to all tests. By default fixtures are added once inside a database transaction that wraps all the tests. At the end of the test the transaction is rolled back, allowing the next test to continue with a pristine state. More on fixtures in *Fixtures*, on page ?.

1. Each top-level call to RSpec.describe creates an internal RSpec object called an *example group*. The creation of the example group causes the block argument to describe to be executed. This may include further calls to describe to create nested example groups.

1. The block argument to describe may also contain calls to it. Each call to it results in the creation of an individual test, which is internally called an "example." The block arguments to it are stored for later execution.

2. Each top-level example group runs. By default the order in which the groups run is random.

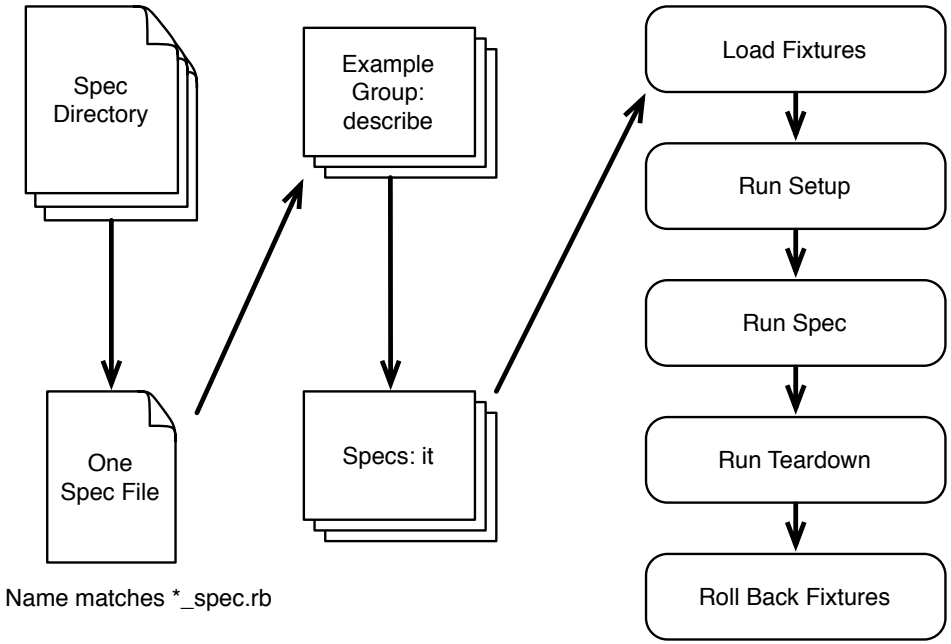
Running an example group involves running each example that it contains, and that involves a few steps:

1. Run all before(:example) setup blocks. We'll talk about those more in a moment, when they become useful.
2. Run the example, which is the block argument to it. The method execution ends when a runtime error or a failed assertion is encountered. If neither of those happens, the test method passes. Yay!
3. Run all after(:example) teardown blocks. Teardown blocks are declared similarly to setup blocks, but their use is much less common.

• Click HERE to purchase this book now. discuss

4. Roll back or delete the fixtures as described earlier. The result of each example is passed back to the test runner for display in the console or IDE window running the test.

The following diagram shows the flow.



In our specific case, we have one file, one example group, and one spec, and if we run things we fail pretty quickly. Here's the slightly edited output:

```
$ rspec
gatherer/spec/models/project_spec.rb:3:in `<top (required)>':
    uninitialized constant Project
```

We're not even getting to the test run; the use of `describe Project` at the beginning of our test is failing because we haven't defined `Project` yet.

Making Our Test Pass

Now it's time to make our first test pass.

But how?

It seems like a straightforward question, but it has a few different answers.

- The purist way: *Do the simplest thing that could possibly work.* In this case "work" means "minimally pass the test without regard to the larger

context." Or it might even mean "write the minimum amount of code to clear the current error without regard to the larger context."
- The "practical" way, scare quotes intended: Write the code you know you need to eventually write, effectively skipping steps that seem too small to be valuable.
- The teaching way, which is somewhere in between the other two and lets me best explain how and why test-driven development works without getting bogged down in details or skipping too many steps.

Ultimately, there isn't a one-size-fits-all answer to the question. The goal is to make the test pass in a way that allows us to best discover the solution to the problem and design our code. In practice, the more complicated the problem is and the less I feel I understand the solution, the more purist I get, taking slow steps.

Let's make this test pass. The first error we need to clear is the uninitialized constant: Project error, so put this in app/models/project.rb:

```
class Project
end
```

This is a minimal way to clear the error. (Well, that's technically not true; I could just declare a constant Project = true or something like that, but there's purist and then there's crazy.) But the test still doesn't pass. If we run the tests now, we get this:

```
rspec
F

Failures:

  1) Project considers a project with no tasks to be done
     Failure/Error: expect(project.done?).to be_truthy
     NoMethodError:
       undefined method `done?' for #<Project:0x00000107ce67d0>
     # ./spec/models/project_spec.rb:6:in `block (2 levels) in <top (required)>'

Finished in 0.00104 seconds (files took 1.29 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/models/project_spec.rb:4 #
    Project considers a project with no tasks to be done
```

See that last line starting with rspec? That's where RSpec usefully gives us the exact command-line invocation we need to run just the failing spec.

Our error is that we are calling project.done? and the done? method doesn't exist yet.

That's simple to clear, still in app/models/project.rb:

```
class Project
  def done?
  end
end
```

And when we do this and run rspec again, we finally get a more interesting error:

```
Failure/Error: expect(project.done?).to be_truthy
       expected: truthy value
            got: nil
```

We've now passed out of the realm of syntax and runtime errors and into the realm of assertion failures—our test runs, but the code does not behave as expected. We've expected that the value of project.done? will be truthy, which is to say any Ruby value that evaluates to true. But since our method doesn't return any value, we get nil.

Luckily, that has a simple fix:

basics_rspec/01/gatherer/app/models/project.rb

```
class Project

  def done?
    true
  end

end
```

Which results in this:

```
$ rspec
.

Finished in 0.00105 seconds (files took 1.2 seconds to load)
1 example, 0 failures
```

And the test passes! We're done! Ship it!

Okay, we're not exactly done. We have made the test pass, which actually only gets us two-thirds of the way through the TDD cycle. We've done the failing test step (sometimes this step is called "red") and the passing test step (sometimes called "green") and now we are at the refactoring step. However, we've written almost no code, so we can safely say there are no refactorings indicated at this point.

I suspect that if you are inclined to be skeptical of test-driven development, I haven't convinced you yet. We've gone on for a few pages and written one line of code, and that line of code clearly isn't even final. I reiterate that in practice this doesn't take much time. If we weren't stopping to discuss each step this would take only a couple of minutes, and some of that time—like setting up the Project class—would need to be spent anyway.

In fact, we haven't exactly done nothing—we've defined and documented a subtle part of how our Project class behaves, and we will find out immediately if the class ever breaks that behavior. As I've said, though, documentation and regression are only part of what makes test-driven development powerful. We need to get to the design part. And for that we need to write more tests.

• Click HERE to purchase this book now. discuss