Your book cover goes here.

Testing Rails

Josh Steiner

April 8, 2015

Contents

Introduction

Why test?

As software developers, we are hired to write code that *works*. If our code doesn't work, we have failed.

So how do we ensure correctness?

One way is to manually run your program after writing it. You write a new feature, open a browser, click around to see that it works, then continue adding more features. This works while your application is small, but at some point your program has too many features to keep track of. You write some new code, but it unexpectedly breaks old features and you might not even know it. This is called a **regression**. At one point your code worked, but you later introduced new code which broke the old functionality.

A better way is to have the computer check our work. We write sofware to automate our lives, so why not write programs to test our code as well? **Automated tests** are small scripts that output whether or not your code works as intended. They verify that our program works now, and will continue to work in the future, without humans having to test it by hand. Once you write a test, you should be able to reuse it for the lifetime of the code it tests, although your tests can change as expections of your application change.

Any large scale and long lasting Rails application should have a comprehensive test suite. A **test suite** is the collection of tests that ensure that your system works. Before marking any task as "complete" (i.e. merging into the `master` branch of your Git repository), it is imperative to run your entire test suite to catch regressions.

If you have written an effective test suite, and the test suite passes, you can be confident that your entire application behaves as expected.

A test suite will be comprised of many different kinds of tests, varying in scope and subject matter. Some tests will be high level, testing an entire feature and walking through your application as if they were a real user. Others may be specific to a single line of code. We'll discuss the varying flavors of tests in detail throughout this book.

Saving Time and Money

At the end of the day, testing is about saving time and money. Automated tests catch bugs sooner, preventing them from ever being deployed. By reducing the manpower necessary to test an entire system, you quickly make up the time it takes to implement a test in the first place.

Automated tests also offer a quicker feedback loop to programmers, as they don't have to walk through every path in their application by hand. A well written test can take milliseconds to run, and with a good development setup you don't even have to leave your editor. Compare that to using a manual approach a hundred times a day and you can save a good chunk of time. This enables developers to implement features faster because they can code confidently without opening the browser.

When applications grow without a solid test suite, teams are often discouraged by frequent bugs quietly sneaking into their code. The common solution is to hire dedicated testers; a Quality Assurance (QA) team. This is an expensive mistake. As your application grows, now you have to scale the number of hands on deck, who will never be as effective as automated tests at catching regressions. QA increases the time to implement features, as developers must communicate back and forth with another human. Compared to a test suite, this is costly.

This is not to say that QA is completely useless, but they should be hired in addition to a good test suite, not as a replacement. While manual testers are not as efficient as computers at finding regressions, they are much better at validating subjective qualities of your software, such as user interfaces.

Confidence

Having a test suite you can trust enables you do things you would otherwise not be able to. It allows you to make large, sweeping changes in your codebase without fearing you will break something. It gives you the confidence to deploy code at 5pm on a Friday. Confidence allows you to move faster.

Living Documentation

Since every test covers a small piece of functionality in your app, they serve as something more than just validations of correctness. Tests are a great form of living documentation. Since comments and dedicated documentation are decoupled from your code, they quickly go stale as you change your application. Tests must be up to date, or they will fail. This makes them the second best source of truth, after the code itself, though they are often easier to read. When I am unsure how a piece of functionality works, I'll look first at the test suite to gain insight into how the program is supposed to behave.

Test Driven Development

Automated tests are likely the best way to achieve confidence in a growing codebase. To amplify that confidence and achieve bigger wins in time savings and code cleanliness, we recommend writing code using a process called **Test Driven Development** (TDD). TDD is a process that uses tests to *drive* the design and development of your application. It begins with a development cycle called **Red, Green, Refactor**.

Red, Green, Refactor

Red, Green, Refactor is a series of steps that lead you through developing a given feature or fixing a bug:

Red

Write a test that covers the functionality you would like to see implemented. You don't have to know what your code looks like at this point, you just have to know what it will do. Run the test. You should see it fail. Most test runners will output red for failure and green for success. While we say "failure" do not take this negatively. It's a good sign! By seeing the test fail first, we know that once we make it pass, our code works!

Green

Read the error message from the failing test, and write as little code as possible to fix the current error message. By only writing enough code to see our test pass, we tend to write less code as a whole. Continue this process until the test passes. This may involve writing intermediary features covering lower level functionality which require their own Red, Green, Refactor cycle.

Do not focus on code quality at this point. Be shameless! We simply want to get our new test passing. Strictly speaking, this includes things like returning literal values that are expected to force yourself to write additional tests that cover multiple cases.

Refactor

Clean up your code, reducing any duplication you may have introduced. This includes your code *as well as your tests*. Treat your test suite with as much respect as you would your live code, as it can quickly become difficult to maintain if not handled with care. You should feel confident enough in the tests you've written that you can make your changes without breaking anything.

TDD Approaches

When solving problems with TDD, you must decide where to start testing your code. Should you start from a high level, testing how the user interacts with the system, then drill down to the nitty gritty? Or, should you begin with a low level design and progress outwards to the final feature? The answer to this depends, and will vary person-to-person and feature-to-feature.

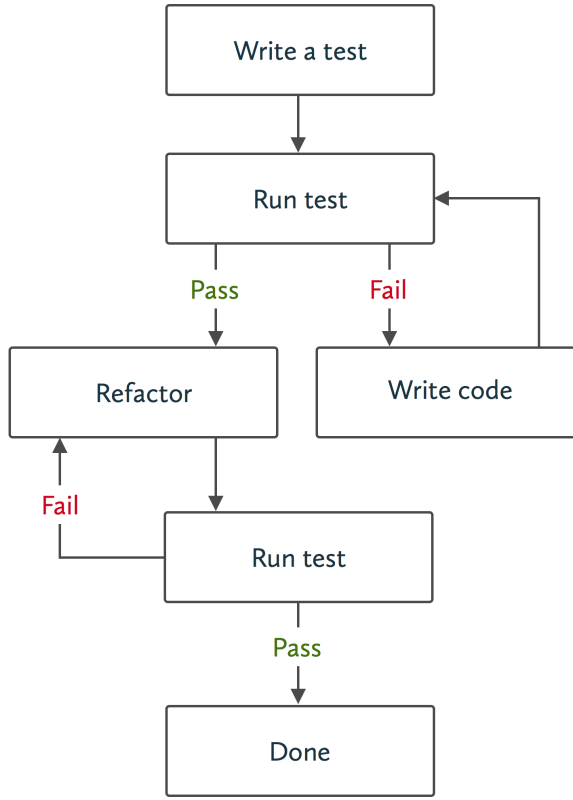

Figure 1.1: TDD Cycle

Outside-In Development

Outside-In Development starts from the highest level of abstraction first. Typically, this will be from the perspective of the user walking through the application in their browser and interacting with elements on the page. We call this an acceptance test, as it ensures that the behavior of the program is acceptable to the end user.

As we develop the feature, we'll gradually need to implement more granular functionality, necessitating intermediate level and lower level tests. These lower level tests may check a single conditional or return value.

By working inwards instead of outwards, you ensure that you never write more code than necessary, because there is a clear end. Once the acceptance test is green, there is no more code to write!

Working outside-in is desireable when you have a good understanding of the problem, and have a rough understanding of how the interface and code will work ahead of time. Because you are starting from a high level, your code will not work until the very end, however your first test will guide your design all the way to completion. You have to trust that your test will bring you there successfully.

Inside-Out Development

Sometimes you don't know what your end solution will look like, so it's better to use an inside-out approach. An inside-out approach helps you build up your code component by component. At each step of the way you will get a larger piece of the puzzle working and your program will be fully functioning at the end of each step. By building smaller parts, one at a time, you can change directions as you get to higher-level components after you build a solid lower-level foundation.

Test Driven vs. Test First

Just because you write your test first, does not mean you are using test driven development. While following the Red, Green, Refactor cycle, it's important to write code only in response to error messages that are provided by test failures. This will ensure that you do not overengineer your solution or implement features that are not tested.

It's also important not to skip the refactor step. This is one of the most important parts of TDD, and ensures that your code is maintainable and easy to change in the future.

Benefits of TDD

Confidence

When it comes down to it, TDD is all about confidence. By writing tests *after* your production code, it's all too easy to forget to test a specific code path. Writing your tests first and only writing code in response to a failing test, you can trust that all of our production code is covered. This confidence gives you power to quickly and easily change your code without fear of it breaking.

Time Savings

Consider automated tests an investment. At first, you will add time by writing tests you would otherwise not be writing. However, most real applications don't stay the same; they grow. An effective test suite will keep your code honest, and save you time debugging over the lifetime of the project. The time savings grow as the project progresses.

TDD can also lead to time savings over traditional testing. Writing your test up front gives you useful error messages to follow to a finished feature. You save time thinking of what to do next, because your test tells you!

Flow

It isn't uncommon for developers to reach a state of "flow" when developing with TDD. Once you write your test, the test failures continuously tell you what to do next, which can almost make programming seem robotic.

Improved Design

That TDD itself improves design is arguable (and many have argued it). In reality, a knowledge of object oriented design priciples paired with TDD aids design.

TDD helps you recognize coupling up front. Object oriented design principles, like dependency injection, help you write your code in ways that reduce this coupling, making your code easier to test. It turns out that code that is easy to test happens to align with well structured code. This makes perfect sense, because our tests run against our code and good code is reusable.

A Pragmatic Approach

There's a lot of dogmatism surrounding the excercise of TDD. We believe that TDD is often the best choice for all the reasons above; however, you must always consider the tradeoffs. Sometimes, TDD doesn't make sense or simply isn't worth it. In the end, the most important thing is that you can feel confident that your program works as it should. If you can achieve that confidence in other ways, that's great!

Here are some reasons you might *not* test drive, or even test, your code:

- The feature you are trying to implement is outside your wheelhouse, and you want to code an exploratory version before you can write your test. We call a quick implementation like this a spike. After writing your spike, you may then choose to implement the associated test. If you implement the test after your production code, you should at the very least toggle some code that would make it fail in an expected way. This way, you can be certain it is testing the correct thing. Alternatively, you may want to start from scratch with your new knowledge and implement it as part of a TDD cycle.
- The entire program is small or unlikely to change. If it's small enough to test by hand efficiently, you may elect to forego testing.
- The program will only be used for a short time. If you plan to throw out the program soon, it will be unlikely to change enough to warrant regression testing, and you may decide not to test it.
- You don't care if the program doesn't behave as expected. If the program is unimportant, it may not be worth testing.

Characteristics of an Effective Test Suite

The most effective test suites share the following characteristics.

Fast

The faster your tests are, the more often you can run them. Ideally, you can run your tests after every change you make to your codebase. Tests give you the feedback you need to change your code. The faster they are the faster you can work and the sooner you can deliver a product.

When you run slow tests, you have to wait for them to complete. If they are slow enough, you may even decide to take a coffee break or check Twitter. This quickly becomes a costly exercise. Even worse, you may decide that running tests is such an inconvenience that you stop running your tests altogether.

Complete

Tests cover all public code paths in your application. You should not be able to remove publicly accessible code from your production app without seeing test failures. If you aren't sufficiently covered, you can't make changes and be confident they won't break things. This makes is difficult to maintain your codebase.

Reliable

Tests do not wrongly fail or pass. If your tests fail intermittently or you get false positives you begin to lose confidence in your test suite. Intermittent failures can be difficult to diagnose. We'll discuss some common symptoms later.

Isolated

Tests can run in isolation. They set themselves up, and clean up after themselves. Tests need to set themselves up so that you can run tests individually. When working on a portion of code, you don't want to have to waste time running the entire suite just to see output from a single test. Tests that don't clean up after themselves may leave data or global state which can lead to failures in other tests when run as an entire suite.

Maintainable

It is easy to add new tests and existing tests are easy to change. If it is difficult to add new tests, you will stop writing them and your suite becomes ineffective. You can use the same principles of good object oriented design to keep your codebase maintainable. We'll discuss some of them later in this book.

Expressive

Tests are a powerful form of documentation because they are always kept up to date. Thus, they should be easy enough to read so they can serve as said documentation. During the refactor phase of your TDD cycle, be sure you remove duplication and abstract useful constructs to keep your test code tidy.

Example Application

This book comes with a bundled example application, a Reddit clone called Reddat. If you are unfamiliar with Reddit, it is an online community for posting links and text posts. People can then comment on and upvote those posts. Make sure that you sign into GitHub before attempting to view the example application and commit links, or you'll receive a 404 error.

Most of the code samples included in the book come directly from commits in the example application. At any point, you can check out the application locally and check out those commits to explore solutions in progress. For some solutions, the entire change is not included in the chapter for the sake of focus and brevity. However, you can see every change made for a solution in the example commits.

The book is broken into chapters for specific topics in testing, which makes it easier to use as a referrence and learn about each part step by step. However, it does make it more challenging to see how a single feature is developed that requires multiple types of tests. To get a sense of how features develop naturally please check out the app's commit history to see the code evolve one feature at a time.

Make sure to take a look at the application's README, as it contains a summary of the application and instructions for setting it up.

RSpec

We'll need a testing framework in order to write and run our tests. The framework we choose determines the format we use to write our tests, the commands we use to execute our tests, and the output we see when we run our tests.

Examples of such frameworks include Test::Unit, Minitest, and RSpec. Minitest is the default Rails testing framework, but we use RSpec for the mature test runner and a syntax that encourages human readable tests. RSpec provides a Domain Specific Language (DSL) specifically written for test writing that makes reading and writing tests feel more natural. The gem is called RSpec, because the tests read like specifications. They describe *what* the software does and how the interface should behave. For this reason, we refer to RSpec tests as *specs*.

While this book uses RSpec, the content will be based in theories and practice that you can use with any framework.

Installation

When creating new apps, we run `rails new` with the `-T` flag to avoid creating any Minitest files. If you have an existing Rails app and forgot to pass that flag, you can always remove `/test` manually to avoid having an unused folder in your project.

Use `rspec-rails` to install RSpec in a Rails app, as it configures many of the things you need for Rails testing out of the box. The plain ol' `rspec` gem is used for testing non-Rails programs.

Be sure to add the gem to *both* the `:development` and `:test` groups in your `Gemfile`. It needs to be in `:development` to expose Rails generators and rake tasks at the command line.

```
group :development, :test do
  gem 'rspec-rails', '~> 3.0'
end
```

Bundle install:

```
bundle install
```

Generate rspec files:

```
rails generate rspec:install
```

This creates the following files:

- `.rspec`

  Configures the default flags that are passed when you run `rspec`. The line `--require spec_helper` is notable, as it will automatically require the spec helper file for every test you run.

- `spec/spec_helper.rb`

  Further customizes how RSpec behaves. Because this is loaded in every test, you can guarantee it will be run when you run a test in isolation. Tests run in isolation should run near instantaneously, so be careful adding any dependencies to this file that won't be needed by every test. If you have configurations that need to be loaded for a subset of your test suite, consider making a separate helper file and load it only in those files.

  At the bottom of this file is a comment block the RSpec maintainers suggest we enable for a good experience. We agree with all of the customizations, so I've commented them in.

- `spec/rails_helper.rb`

  A specialized helper file that loads Rails and its dependencies. Any file that depends on Rails will need to require this file explicitly.

The generated spec helpers come with plenty of comments describing what each configuration does. I encourage you to read those comments to get an idea of how you can customize RSpec to suit your needs. I won't cover them as they tend to change with each RSpec version.