

5. Testing Rescues

There are two possible situations when you look at the tests in an application that's come to you for rescue:

1. There are no tests
2. There are bad tests

Oh, I suppose it's possible that some day there will be an application that needs to be rehabilitated despite having an excellent and up-to-date test suite - but I've never seen it. More typically, there are no tests at all, or tests that haven't been maintained and that are falling into disarray.

Diagnosing the Problems

Start by figuring out where things stand. The most critical information is how well the application is tested. While you can get detailed information by running `rcov` (as detailed in Section 3, "Digging Into Quality"), you usually don't have to go into that much depth to see whether the application has a test coverage problem. A simple `rake stats` will tell you most of the story:

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Controllers	1425	1069	13	90	6	9
Helpers	235	183	0	21	0	6
Models	1316	883	18	128	7	4
Libraries	615	469	3	66	22	5
Components	0	0	0	0	0	0
Model specs	249	139	0	0	0	0
View specs	0	0	0	0	0	0
Controller specs	117	6	0	3	0	0
Helper specs	0	0	0	0	0	0
Total	3957	2749	34	308	9	6

Code LOC: 2604 Test LOC: 145 Code to Test Ratio: 1:0.1

That's pretty typical: just a few lines of test code in the entire application, and a dismal code to test ratio. Opening up the tests in this particular application also revealed that the bulk of those lines of code were useless - assertions of truth, for the most part, that were created by generators and never edited. What we have here is an application that, for all intents and purposes, has no tests (though at one point it did have fantasies about using RSpec).

If you do find tests, the next question is how good they are. You can run `rcov` to look at test coverage, but before you get to that point, just try to run the tests themselves. You may well end up with output like this:

```
Errors running test:units and test:functionals!
```

Or this:

```
.F...EEE..FF..FEEEEEF...FFF
```

Either of these situations indicates that the tests in the project, while they seemed like a good idea at some point, were not kept up as the code evolved.

Step 1: Take out the Trash

Before you can make the situation better, you have to make sure you've got a solid starting point. A good place to start is by ruthlessly getting rid of files and code that, while they give the illusion of testing, don't actually contribute to worthwhile test coverage.

TIP: Don't lose sleep worrying that you're going to get rid of something that you later find out you need after all. That's why you have the whole project under source code management, right? If you don't, go back and reread Section 2, "The Opening Investigation."

If the tests are hopelessly failing, the decision is simple: get rid of everything. Delete all the tests, all the specs, all the fixtures (they probably haven't been maintained anyhow), and any other auxiliary files such as Machinist blueprints. Leave yourself with a completely test-less project so you have a clean slate to start from.

If the project has partial test coverage, then you'll have to go through it in more depth:

- Get rid of any scaffolded test that simply asserts truth.
- Look at all failing tests, and strictly timebox your look - 5 minutes per test is good. If the test is a complete muddle, delete it. If it appears to be testing some part of the application and you can figure out how to fix the test, fix it (remember, at this stage the application and not the test is the source of the spec). If it appears to be a legitimate test but you can't immediately see how to fix it, mark it as pending or comment it out.
- Get rid of any fixture that isn't being used in a test that you kept.

At the conclusion of taking out the trash, any test that's left in the application should be passing. You may not have any tests, but if you do, they should be good.

NOTE: If you're not familiar with timeboxing, I've written a quick introduction at "Use Timeboxing to Slay the Perfectionist Beast" (<http://webworkerdaily.com/2007/08/07/use-timeboxing-to-slay-the-perfectionist-beast/>).

Step 2: Make a Plan

There are a lot of choices in Rails testing:

- Test::Unit vs. RSpec
- Shoulda vs. Contexts
- Fixtures vs. Factory Girl vs. Machinist vs. Object Daddy
- Cucumber vs. pure TDD

If you were lucky enough to preserve a large number of existing tests, your choice is made for you: use whatever testing libraries the application came to you with. But in the case where you've flushed the tests to start over, you get to pick the tools. Go ahead and pick whichever testing strategy and infrastructure you personally are most familiar with and happiest with. A project that you're trying to bring back from the brink of death is no place to experiment with learning new tools. Install the tools you like, and move on.

Step 3: Don't Let it Get Worse

Do you have the luxury of doing nothing but writing tests for a few weeks? Nope? Didn't think so.

The reality is that you're probably tinkering with parts of the application that don't work quite right, or adding new features that the previous developer didn't get around to (or couldn't figure out). This is the perfect time to make sure that you don't make any existing problems worse. As you add new code, add new tests - even if they are the first tests in the application. In fact, add them even if you're not entirely sure you've found the best testing strategy. Bad (but passing) tests can be refactored. Nonexistent tests are no good to anyone. At the very least you should be writing unit and functional tests as you add code. If you're in favor of full-blown BDD, then you should be adding Cucumber stories and going from there.

Existing legacy code that is in need of refactoring due to mammoth methods is a special case. It often does not make sense to write tests for a giant method that will be broken up. Here you can use a modified TDD strategy: identify a piece of functionality that you intend to pull out to a new object or new method, and write tests for that chunk of code first. Then do the refactoring to pull that code out and pass the tests. As you do this, the untested code in the original method will shrink, and will eventually reach the point where it makes sense to add test coverage the next time you touch it.

Step 4: Acceptance Tests

Sooner or later - hopefully sooner - you should be able to find some time to start adding test coverage for untested code that you were handed. By this time, you should have some idea of what the application is supposed to do, even if you haven't dug through all of the code to figure out how it works. A good place to start writing tests for existing code is with high-level

acceptance tests: Rails integration tests or Cucumber scenarios. You can worry about testing down into the guts later, but if you can at least test the behavior of the application you can be sure that your continuing refactoring efforts don't break the way it works now.

Step 5: Extending Coverage

Between adding tests for new code, and building acceptance tests, you should gradually find your test coverage numbers getting better. Unless you're terribly budget-constrained, you'll find yourself wanting to complete the job. For many developers, simply wanting to see reasonable numbers out of `rcov` is a driver for this task.

One way to sneak up on better test coverage is to start writing tests for related code, rather than just for the code that you're adding or changing. If you need to add a method to a controller, test a few other methods from the same controller. If you're refactoring a model, add coverage for some of the methods that you haven't touched.

Another thing you can do is try to seed some test coverage in untested parts of the code. With `rcov`'s reports, it's easy to see which areas of the application are least-tested. If there's a whole model completely untested, it's worth opening it up, finding one chunk of code, and writing tests for that. Having at least a couple of tests for each model, controller, and helper vastly lowers the chance that a change in one part of the application will break some other (apparently-unrelated) part of things.