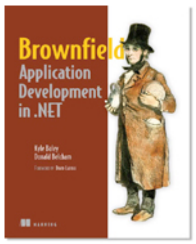

Writing New Tests in Brownfield Applications

By Kyle Baley and Donald Belcham, author of *Brownfield Application Development in .NET*

You may find that after integrating your new test project into the automated build, the build fails because there are no tests to execute. If this is the case, just write a test. In this article, based on chapter 4 of Brownfield Application Development in .NET*, the author shows you how to write a new test.*

Let's assume you have a test project in place with all the tests within it being executed as part of your automated build in your continuous integration (CI) process. We now have the framework in place to write some new tests.

One of the key parts in the definition of a brownfield application is that the codebase is in active development. As part of that work, we expect that there will be refactorings as well as completely new features added. And as we've mentioned before, it's difficult to have confidence in code if it isn't being tested. After all, how can we be sure that modifying existing code or adding new code won't break something?

As you might expect with two different scenarios (writing new code vs. modifying existing code), there are multiple ways to approach the test writing. First, let's look at writing new tests when you're modifying existing code. It's this scenario that will offer the most challenges.

Tests on existing code

Modifying existing code to add new or changed functionality is much like fixing bugs. It's likely you're going into an area of the application that may not be covered by tests at all. Making changes with confidence in that scenario is hard to do. So how do we resolve that?

The scenario we discuss here is slightly different than when you fix a bug. Here, we're adding a new feature or changing existing functionality. It's a subtle difference but the method to attack it, testing, is the same. You should never make a change to code until there's a test in place first to prove that your change will be successful when you're done. Don't worry about testing the current functionality; it's going to go away. Instead, create tests that verify the functionality you *intend* to have after making your changes. Write a test as if the code is already doing what you want it to do. This test will fail, but that's good. Now you'll know when your modifications are done: when the test passes.

Let's look at a simple example to explain what we mean. Say you have a service that retrieves the display name for a customer, as we do here:

```
public string GetDisplayNameFor( Customer customer )
{
    string format = "{0} {1}";
    return String.Format( format, customer.FirstName, customer.LastName );
}
```

Because ours is a typical brownfield application, there are no tests for this method. For now, we're fine with this because no bugs have been reported as yet.

Now, the client says that they'd like to change the format of the customer's name to LastName, FirstName. After searching through the code, we zero in on this function and…then what?

This is a benign change, so you'd probably just change the format string and be done with it. But let's rein our instincts in for a moment and try to make sure this issue doesn't come back to haunt us.

Instead of diving into the code, let's write a test to verify that this function behaves the way we want it to (listing 1).

Listing 1 Testing to verify desired functionality

```
[Test]
public void Display_name_should_show_last_name_comma_first_name( )
{
    MyNameService sut = new MyNameService( );
    Customer customer = new Customer( "Tony", "Danza" );
    string displayName = sut.GetDisplayNameFor( customer );
    Assert.That( displayName, Is.EqualTo( "Danza, Tony" );
}
```

Listing 1 is a test that verifies that our name display service returns names in the format LastName, FirstName.

You may be asking yourself, "Won't this test fail in the current code?" The answer is, "You better believe it'll fail." The current implementation of this function displays the name in FirstName, LastName format.

But we know we're about to change the current functionality, so there's no point verifying that it's correct. Rather, we want to make sure that the new functionality is correct. And we do that by writing the test first.

By writing the test for the modified functionality first and having it fail, we've now given ourselves a specific goal. Setting it up in this way makes it unambiguous as to when we're finished. We aren't done until the test passes.

For completeness, let's modify the `GetDisplayNameFor` method so that it passes our test:

```
public string GetDisplayNameFor( Customer customer )
{
    string format = "{1}, {0}";
    return String.Format( format, customer.FirstName, customer.LastName );
}
```

We run the tests in our automated build and voilà: the tests pass. We can now commit both the tests and the newly modified code to our source code repository.

The reality of testing existing code

Our example was just shy of being utopian (read: unrealistic) in its simplicity. In many cases, maybe even most, it will be nearly impossible to test existing code in isolation in this manner. Perhaps you have a single function that (a) retrieves a connection string from the app.config, (b) opens a connection to the database, (c) executes some embedded SQL, and (d) returns the resulting `DataSet`.

If you need to modify such a function in any way, you're looking at a fairly significant refactoring just to get this one piece of code properly layered so that you can test each of its tasks.

In the end, you achieve the modified functionality you wanted, confidence in that functionality, and the ability to perform ongoing regression testing for that functionality if anything changes. Instead of just adding the changes that the client needs, you've added long-term reinforcement as well.

With our existing code covered (so to speak), we'll now talk about approaches for testing against new code.

Tests on new code

When you're working on your brownfield application, you'll run into one more newtest scenario: that glorious time when you get to write all new code, such as a new module for your application. More likely than not, you'll be tying into existing code, but you'll be adding, not modifying, most of the time. It's like your own little greenfield oasis in the middle of a brownfield desert.

Compared with testing existing code, writing tests for new code should be a breeze. You have more control over the code being tested and chances are that you'll run into less "untestable" code.

There are two ways to write unit tests for new code: *before* the new code is written or *after*. Traditionally, test-after development (TAD) has been the most popular because it's intuitive and can be easily understood.

Writing tests before you write your code is more commonly known as test-driven design (TDD). We won't pretend this topic is even remotely within the scope of this chapter (or even this book), but since we have your attention, it's our opinion that

TDD is worth looking into and adopting as a design methodology.

Test-driven design

Although it has little bearing on brownfield application development specifically, we can't resist at least a sidebar on test-driven design. It's worth your effort to evaluate it at your organization and try it on a small scale.

Using TDD allows for the creation of code in response to a design that you've determined first. Yes, folks, you read it here: TDD is about designing first. Unlike some schools of thought, this design process doesn't usually include UML, flow charts, state diagrams, data flow diagrams, or any other design tool supported by a GUI-driven piece of software. Instead, the design is laid out through the process of writing code in the form of tests. It's still design, but it's done in a different medium. The tests, while valuable, are more of a side effect of the methodology and provide ongoing benefit throughout the life of the project.

Let it not be said that it's an easy thing to learn. But there are a lot of resources available in the form of books, articles, screencasts, and blogs. Help is never more than an email away.

Whether you choose to write your tests before or after you write your new code, the important thing is that you write them. And, execute them along with your automated build.

Summary

Automated tests can vastly improve the confidence everyone feels in the codebase. By writing tests to expose bugs before we fix them, for example, we close the feedback loop between the time a bug is written and the time it's discovered.

Last updated: August 2, 2013