Nick Hodges Follow
Oct 7, 2019 · 5 min read ★ · Listen

Save

13 Tips for Writing Useful Unit Tests

How you write your tests is as important as writing them

Photo by Ben Mullins on Unsplash

In my previous article, I made the case that you should be unit testing your code. I think I made the case pretty well.

In this article, I'll cover some tips and pointers about how to unit test your code. Here we go.

1. Test One Thing at a Time in Isolation

This is probably the baseline rule to follow when it comes to unit tests. All classes should be tested in isolation. They should not depend on anything other than mocks and stubs.

They shouldn't depend on the results of other tests. They should be able to run on any machine. You should be able to take your unit test executable and run it on your mother's computer when it isn't even connected to the internet.

2. Follow the AAA Rule: Arrange, Act, Assert

When it comes to unit testing, AAA stands for *Arrange, Act, Assert*. It is a general pattern for writing individual tests to make them more readable and useful.

First, you arrange. In this step, you set things up to be tested. You set variables, fields, and properties to enable the test to be run, as well as define the expected result.

Then you act — that is, you call the method that you are testing.

Finally, you assert— call the testing framework to verify that the result of your "Act" is what was expected. Follow the AAA principle, and your test will be clear and easy to read.

3. Write Simple "Fastball-Down-the-Middle" Tests First

The first tests you write should be the simplest — the happy path. They should be the ones that easily and quickly illustrate the functionality you are trying to write.

If you are writing an addition algorithm, the early tests that you write should make sure that your code can do 2 + 2 = 4. Then, once those tests pass, you should start writing the more complicated tests (as discussed below) that test the edges and boundaries of your code.

4. Test Across Boundaries

Unit tests should test both sides of a given boundary. If you are building some tests for date and time utilities, try testing one second before midnight and one second after. Check across the date value of 0.0.

If you are dealing with a structure that holds a rectangle, then test what happens to points inside and outside the rectangle. What about above or below? To the left or right? Above and to the right? Below and to the left?

Moving across boundaries are places where your code might fail or perform in unpredictable ways.

5. If You Can, Test the Entire Spectrum

If it is practical, test the whole set of possibilities for your functionality. If it involves an enumerated type, test the functionality with every one of the items in the enumeration.

It might be impractical to check every possible string or every integer, but if you can test every possibility, do it.

6. If Possible, Cover Every Code Path

This one is challenging as well, but if your code is designed for testing, and you make use of a code coverage tool, you can ensure that every line of your code is covered by unit tests at least once.

If your language of choice has a code coverage tool, use it in concert with your unit tests. Covering every code path won't guarantee that there aren't any bugs, but it surely gives you valuable information about the state of every line of code.

7. Write Tests That Reveal a Bug, Then Fix It

This is a powerful and useful technique. If you find a bug, write a test that reveals it. Then, you can quickly fix the bug by debugging the test.

Then, you have an excellent regression test to make sure that if that bug comes back for any reason, you'll know right away. It's easy to fix a bug when you have a simple, straightforward test to run in the debugger.

A side benefit here is that you've "tested your test". Because you've seen the test fail and then have seen it pass, you know that the test is valid in that it has proven to work correctly. This makes it an even better regression test.

8. Make Each Test Independent

Tests should never depend on each other. If your tests have to be run in a specific order, then you need to change your tests.

Instead, you should make proper use of the `Setup` and `TearDown` features of your unit-testing framework to ensure each test is ready to run individually.

Unit test frameworks don't guarantee that tests are going to be run in any particular order. If your tests depend on tests running in a specific order, then you may find yourself with some subtle, hard to track down bugs in your tests themselves.

Make sure each test stands alone and you won't have this problem.

9. Name Your Tests Clearly and Don't Be Afraid of Long Names

As you are doing one assert per test, each test can end up being very specific. Thus, don't be hesitant to use a long, complete test name. It is better to have `TestDivisionWhenNumPositiveDenomNegative` than `DivisionTest3`.

A long, complete name lets you know immediately which test failed and what exactly what the test was trying to do. Long, clearly named tests also can document your tests.

For example, a test called `DivisionByZeroShouldThrowException` documents precisely what the code does when you try to divide by zero.

10. Test That Every Raised Exception Is Raised

If your code raises exceptions, then write tests to ensure that every exception you raise gets raised when it is supposed to.

Most xUnit testing frameworks can test for an exception being raised, so you should use that feature to ensure that every exception your code raises is indeed raised under the proper circumstances.

11. Avoid the Use of Assert.IsTrue

Avoid checking for a boolean condition.

For instance, instead of checking if two things are equal with `Assert.IsTrue` , use `Assert.AreEqual` instead. Why? Because this:

```
Assert.IsTrue(Expected = Actual);
```

Will report something like `Some test failed: Expected True, but the actual result was False` . That doesn't tell you anything. Instead, use `Assert.AreEqual` :

```
Assert.AreEqual(Expected, Actual)
```

Which will tell you the actual values involved, such as `Some test failed: Expected 7, but the actual result was 3`, which is much more valuable as an error message.

12. Constantly Run Your Tests

Run your tests while you are writing code. Your tests should run fast, enabling you to run them after even minor changes.

If you can't run your tests as part of your normal development process, then something is going wrong — unit tests are supposed to run almost instantly. If they aren't, it's probably because you aren't running them in isolation.

13. Run Your Tests as Part of Every Automated Build

Just as you should be running your tests as you develop, they should also be an integral part of your continuous integration process. A failed test should mean that your build is broken.

Don't let a failing test linger — consider it a build failure and fix it immediately.

Conclusion

Well, there are 13 ways to write useful unit tests. Remember, unless you are writing unit tests, your code will end up hard to maintain and hard to fix. Well-written, thorough unit tests are just a big win all around.