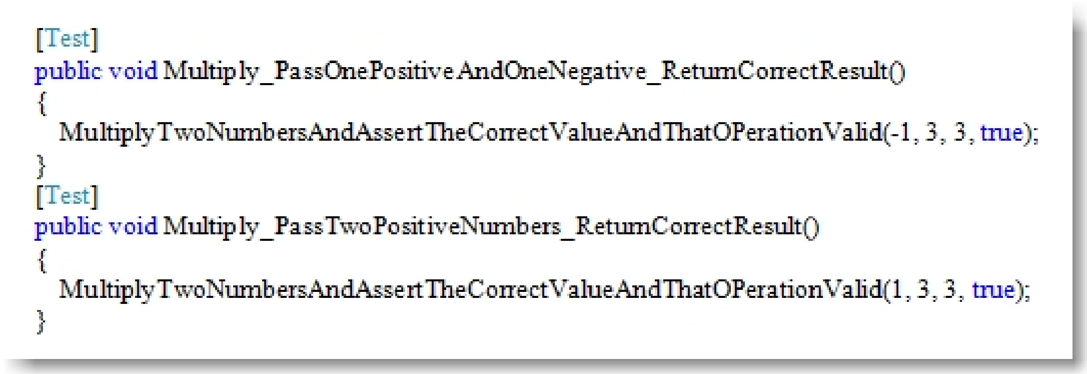
8 Principles of Better Unit Testing

09/24/2012

Guideline #5: Do repeat yourself

One of the first lessons I learned in Computer Science 101 is that writing the same code twice is bad. In production code, you should avoid duplication because it causes maintainability issues. Readability is very important in unit testing, so it is acceptable to have duplicate code. Avoiding duplication in tests creates tests that are difficult to read and understand:



```
[Test]
public void Multiply_PassOnePositiveAndOneNegative_ReturnCorrectResult()
{
  MultiplyTwoNumbersAndAssertTheCorrectValueAndThatOPerationValid(-1, 3, 3, true);
}
[Test]
public void Multiply_PassTwoPositiveNumbers_ReturnCorrectResult()
{
  MultiplyTwoNumbersAndAssertTheCorrectValueAndThatOPerationValid(1, 3, 3, true);
}
```

In other words, having to change 4-5 similar tests is preferable to not understanding one non-duplicated test when it fails. Eliminating duplication is usually a good thing -- as long as it does not obscure anything. Object creating can be refactored to factory methods and custom assertions can be created to check a complex object -- as long as the test's readability does not suffer.

Guideline #6: Test results, not implementation

Successful unit testing requires writing tests that would only fail in case of an actual error or requirement change. There are a few rules that help avoid writing fragile unit tests. These are tests that would fail due to an internal change in the software that does not affect the user.

Since the same developer that wrote the code and knows how the solution was implemented usually writes unit tests, it is difficult not to test the inner workings of how a feature was implemented. The problem is that implementation tends to change and the test will fail even if the result is the same.

Another issue arises when testing internal/private methods and objects. There is a reason that these methods are private -- they are not meant to be "seen" outside of the class and are part of the internal mechanics of the class. Only test private methods if you have a very good reason to do so. Trivial refactoring can cause complication errors and failures in the tests.

Guideline #7: Avoid overspecification

It is tempting to create a well-defined, controlled, and strict test that observes the exact process flow during the test by setting every single object and testing every single aspect being tested. The problem is that this "locks" the scenario under test, preventing it from changing in ways that do not affect the result.

For example, try to avoid writing a test that expects a certain method to be called exactly three times. There are reasons for writing very precise tests, but usually such micromanagement of test execution will only lead to a very fragile test. Use an Isolation framework to set default behavior of external objects and make sure that it is not set to throw an exception if an unexpected method was called. This option is usually referred to as "strict" by several Isolation frameworks.

Guideline #8: Use an Isolation framework

Writing good unit tests can be hard when the class under test has internal or external dependencies. In order to run a test, you may need a connection to a fully populated database or a remote server. In some cases, you may need to instantiate a complex class created by someone else.

These dependencies hinder the ability to write unit tests. When such dependencies need a complex setup for the automated test to run, the result is fragile tests that break, even if the code under test works perfectly.

A mocking framework (or Isolation framework) is a third-party library and a huge time saver. In fact, the savings in lines of code between using a mocking framework and writing hand-rolled mocks for the same code can go up to 90 percent. Instead of creating our fake objects by hand, we can use the framework to create them with only a few API calls. Each mocking framework has a set of APIs for creating and using fake objects without the user needing to maintain irrelevant details of the specific test. If a fake is created for a specific class, then when that class adds a new method, nothing needs to change in the test.

The Bottom Line

Writing good, robust unit tests is not hard. It just takes a little practice. This list is far from comprehensive, but it outlines a few key points that will help you write better unit tests. In addition, remember that if a specific test keeps failing, investigate the root cause, and find a better way to test that feature.

- - -

Dror Helper is a software architect at Better Place. He was previously a software developer at Typemock. You can contact the author at his blog, http://blog.drorhelper.com.