8 Principles of Better Unit Testing

Writing good, robust unit tests is not hard -- it just takes a little practice. These pointers will help you write better unit tests.

- **09/24/2012**

By Dror Helper
Writing unit tests should be easy for software developers – after all, writing tests is just like writing production code. However, this is not always the case. The rules that apply for writing good production code do not always apply to creating a good unit test.
Not many software professionals recognize that they need to follow different rules for writing unit tests, and so software developers continue to write bad unit tests, following best practices for writing production code that are not appropriate for writing unit tests.

What makes a good unit test?

Unit tests are short, quick, and automated tests that make sure a specific part of your program works. They test specific functionality of a method or class that have a clear pass/fail condition. By writing unit tests, developers can make sure their code works, before passing it to QA for further testing.
For example, the following unit test checks for a valid user and password when the method CheckPassword returns true:

```
[Test]
public void CheckPassword_ValidUserAndPassword_ReturnTrue()
{
   UserService classUnderTest = new UserService();
   bool result = classUnderTest.CheckPassword("user", "pass");
   Assert.IsTrue(result);
}
```

In other words, a unit test is just a method written in code.
A "good" unit test follows these rules:

- The test only fails when a new bug is introduced into the system or requirements change
- When the test fails, it is easy to understand the reason for the failure.

To write good unit tests, the developer that writes the tests needs to follow these guidelines:

Guideline #1: Know what you're testing

Although this seems like a trivial guideline, it is not always easy to follow.

A test written without a clear objective in mind is easy to spot. This type of test is long, hard to understand, and usually tests more than one thing.

There is nothing wrong with testing every aspect of a specific scenario/object. The problem is that developers tend to gather several such tests into a single method, creating a very complex and fragile “unit test.” For example:

```
[Test]
public void TestMethod1()
{
   var calc = new Calculator();
   calc.ValidOperation = Calculator.Operation.Multiply;
   calc.ValidType = typeof (int);
   var result = calc.Multiply(-1, 3);
   Assert.AreEqual(result, -3);
   calc.ValidOperation = Calculator.Operation.Multiply;
   calc.ValidType = typeof (int);
   result = calc.Multiply(1, 3);
   Assert.IsTrue(result == 3);
   if (calc.ValidOperation == Calculator.Operation.Invalid)
   {
      throw new Exception("Operation should be valid");
   }
   calc.ValidOperation = Calculator.Operation.Multiply;
   calc.ValidType = typeof (int);
   result = calc.Multiply(10, 3);
   Assert.AreEqual(result, 30);
}
```

One trick is to use the scenario tested and expected result as part of the test method name. When a developer has a problem naming a test, that means the test lacks focus.

Testing only one thing creates a more readable test. When a simple test fails, it is easier to find the cause and fix it than to do so with a long and complex test.

The example above is actually three different tests. Once we define the objective of each test, it is easy to split the code tested:

```
[Test]
public void Multiply_PassOnePositiveAndOneNegative_ReturnCorrectResult()
{
   var calculator = CreateMultiplyCalculator();
   var result = calculator.Multiply(-1, 3);
   Assert.AreEqual(result, -3);
}
[Test]
public void Multiply_PassTwoPositiveNumbers_ReturnCorrectResult()
{
   var calculator = CreateMultiplyCalculator();
   var result = calculator.Multiply(1, 3);
   Assert.AreEqual(result, 3);
}
[Test]
public void Multiply_CreateMultiplyCalculatorAndUseIt_ValidOperationIsNotInvalid()
{
   var calculator = CreateMultiplyCalculator();

   const int firstNumber = 1;
   const int secondNumber = 2;
   calculator.Multiply(firstNumber, secondNumber);
   Assert.AreNotEqual(Calculator.Operation.Invalid, calculator.ValidOperation);
}
```

Guideline #2: Unit tests should be self-sufficient

A good unit test should be isolated. Avoid dependencies such as environment settings, register values, or databases. A single test should not depend on running other tests before it, nor should it be affected by the order of execution of other tests. Running the same unit test 1,000 times should return the same result every time.

Using global states such as static variables, external data (i.e. registry, database), or environment settings may cause "leaks" between tests. Make sure to properly initialize and clean each of the "global states" between test runs or avoid using them completely.

Guideline #3: Tests should be deterministic

The worst test is the one that passes some of the time. A test should either pass all the time or fail until fixed. Having a unit test that passes some of the time is equivalent to not having a test at all.

For example, the following test passes most of the time:

```
[Test]
public void NonDeterministicTest()
{
   var client = new MyClient();
   client.Connect();
   // wait until client connects
   Thread.Sleep(1000);
   Assert.IsTrue(client.IsConnected);
}
```

The test above can fail when running on a slow computer and pass later on another machine. A development team "learns" to ignore when such test fails rendering the test ineffective. A non-deterministic test is irrelevant because when it fails, there is no definitive indication that there is a bug in the code. Another “practice” that must be avoided is writing tests with random input. Using randomized data in a unit test introduces uncertainty. When that test fails, it is impossible to reproduce because the test data changes each time it runs.

Guideline #4: Naming conventions

To know why a test failed, we need to be able to understand it at a glance. The first thing that you notice about a failed test is its name -- the test method name is very important. When a well-named test fails, it is easier to understand what was tested and why it failed.

For example, when testing a calculator class that can divide two numbers there are several options.

```
This is a good test:
[Test]
public void Divide_DivideByZero_ExceptionThrown()

This is not so good:
[Test]
public void DivideThrowException()

This is terrible:
[Test]
public void DivideTest2()
```

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**.