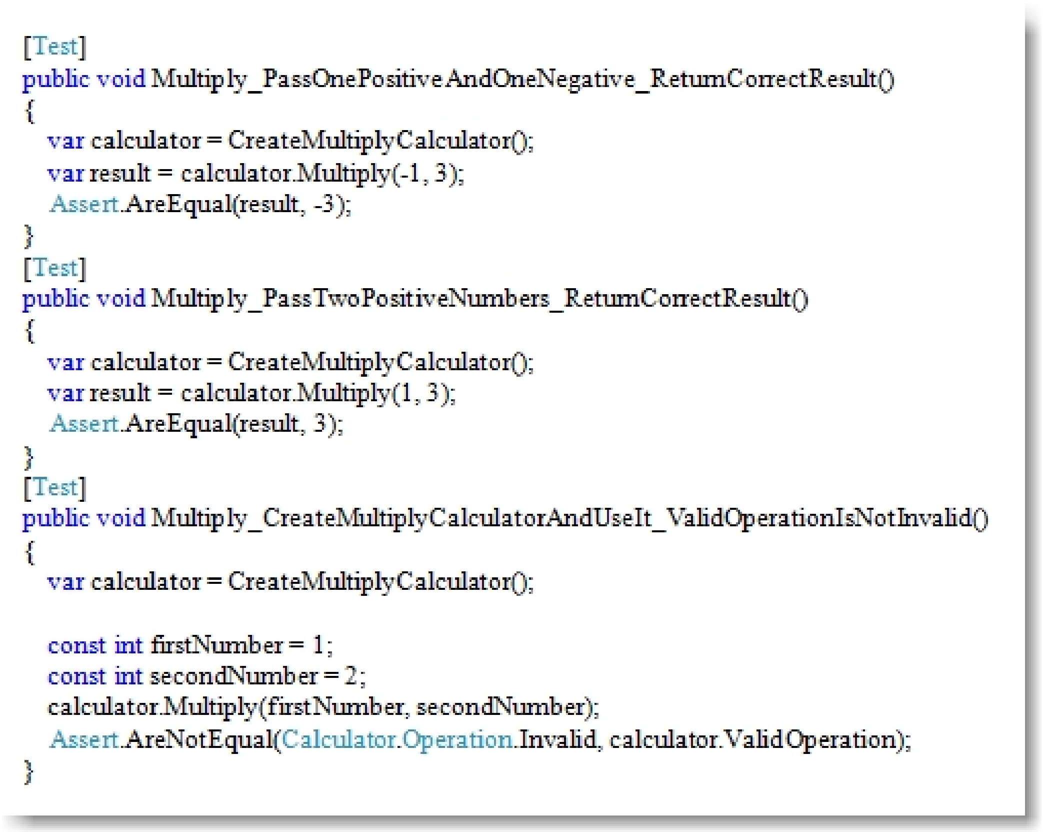
8 Principles of Better Unit Testing

09/24/2012

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

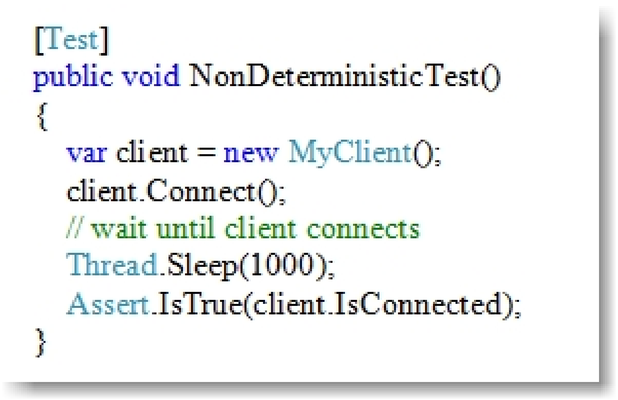
A good unit test should be isolated. Avoid dependencies such as environment settings, register values, or databases. A single test should not depend on running other tests before it, nor should it be affected by the order of execution of other tests. Running the same unit test 1,000 times should return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

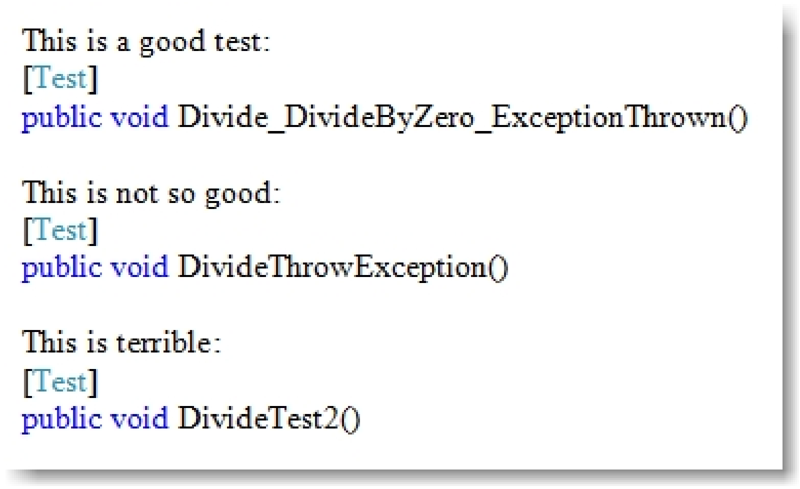
The test above can fail when running on a slow computer and pass later on another machine. A development team "learns" to ignore when such test fails rendering the test ineffective. A non-deterministic test is irrelevant because when it fails, there is no definitive indication that there is a bug in 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()
```