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);
}
```