Removing Duplication in Unit Tests

Articles

Posted by:

Posted on : 2009/4/30 1:00:00

This article is taken from the book The Art of Unit Testing. As part of a chapter on the pillars of good tests, this segment shows how to remove duplication from unit tests. Author: Roy Osherove

This article is based on The Art of Unit Testing, to be published in May 2009. It is being reproduced here by permission from Manning Publications. Manning early access books and ebooks are sold exclusively through Manning. Visit the book's page for more information. Duplication in our unit tests can hurt us as developers just as much (if not more) than duplication in production code. The DRY (Don't Repeat Yourself) Principle should be in effect in test code as if it were production code. Duplicated code means more code to change when one particular aspect we test against may change. Changing a constructor, changing the semantics of using a class, and more, can have a large effect on tests that have a lot of duplicated code. To understand why, let’s begin with a simple example of a test, seen in listing 1. **Listing 1: A class under test and a test that uses it.**

```
 public class LogAnalyzer
  {
     public bool IsValid(string fileName)
     {
        if (fileName.Length           {
           return true;
        }
        return false;
     }
  }

  [TestFixture]
  public class LogAnalyzerTestsMaintainable
1 Don&rsquo;t Repeat Yourself
  {
     [Test]
     public void IsValid_LengthBiggerThan8_IsFalse()
     {
        LogAnalyzer logan = new LogAnalyzer();
        bool valid = logan.IsValid("123456789");
        Assert.IsFalse(valid);
     }
  }
```

The test at the bottom of the listing seems reasonable, until you introduce another test for the same class, and end up with two tests like in listing 2: **Listing 2: Can you spot the duplication?** [Test]

```
     public void IsValid_LengthBiggerThan8_IsFalse()
     {
```

```
        LogAnalyzer logan = new LogAnalyzer();
        bool valid = logan.IsValid("123456789");
        Assert.IsFalse(valid);
    }

    [Test]
    public void IsValid_LengthSmallerThan8_IsTrue()
    {
        LogAnalyzer logan = new LogAnalyzer();
        bool valid = logan.IsValid("1234567");
        Assert.IsTrue(valid);
    }
```

What’s wrong with the tests in listing 2? The main problem is that if the way they use LogAnalyzer change (its semantics), the tests will have to be maintained independently of each other, leading to more maintenance work. Here’s an example of such a change in listing 3:
Listing 3: LogAnalyzer changed semantics and now requires initialization public class LogAnalyzer

```
  {
  private bool initialized=false;

    public bool IsValid(string fileName)
    {
 If(!initialized)
Throw NotInitializedException(
&ldquo;The analyzer.Initialize() method should be called before any other operation!&rdquo;);

        if (fileName.Length          {
            return true;
        }
        return false;
    }
    public void Initialize()
    {
 //initialization logic here
 ...
 Initialized=true;
    }
  }
```

Now, the two tests that we’ve written will both break, because they both neglect to call Initialize() against the LogAnalyzer class. Because we have code duplication (both of the tests create the class within the test) we need to go into each one and change it to call initialize() first. We can refactor the tests to remove the duplication of creating the LogAnalyzer into a single CreateDefaultAnalyzer() method which both tests will call. We could also push the creation and initialization up into a new [setup] method in our test class.