AMDARIS

Continous staff improvement project

WRITING TESTS. TDD

YURII HOHAN

INTRODUCTION

- WRITING TESTS USING NUNIT.
- TEST, TESTFIXTURE, TESTCASE, ETC.
- ORGANIZING UNIT-TESTS.
- MOQ.
- TDD.
- INTRODUCTION. BENEFITS. TEST DRIVEN APPROACH.

AMDARIS

NUNIT

- NUnit is a unit-testing framework for all .Net languages.
- Initially ported from JUnit.
- It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities.
- Unit testing is used to test a small piece of workable code (operational) called unit.

AAA PATTERN

- **Arrange:** setup everything needed for the running the tested code. This includes any initialization of dependencies, mocks and data needed for the test to run.
- **Act:** Invoke the code under test.
- **Assert:** Specify the pass criteria for the test, which fails it if not met.

EXAMPLE

```
public class MathsHelper
{
    public MathsHelper() { }
    public int Add(int a, int b)
    {
        int x = a + b;
        return x;
    }

    public int Subtract(int a, int b)
    {
        int x = a - b;
        return x;
    }
}
```

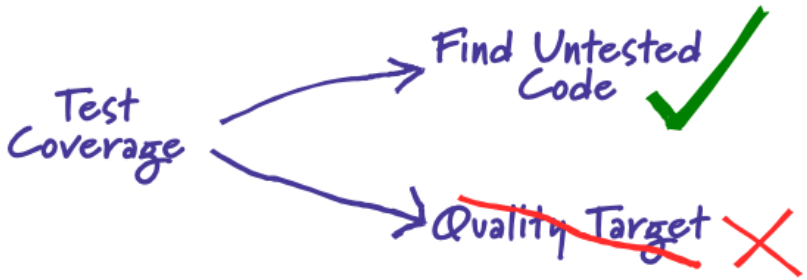
```
[TestFixture]
public class TestClass
{
    [TestCase]
    public void AddTest()
    {
        MathsHelper helper = new MathsHelper();
        int result = helper.Add(20, 10);
        Assert.AreEqual(30, result);
    }

    [TestCase]
    public void SubtractTest()
    {
        MathsHelper helper = new MathsHelper();
        int result = helper.Subtract(20, 10);
        Assert.AreEqual(10, result);
    }
}
```

WAYS TO RUN

- NUnit Test Runner.
- ReSharper Test Runner in IDE.
- Visual Studio Runner.

TEST COVERAGE. COVERING EACH CASE.

```
int m(int m){
      if (m%2 ==0) return m*2;
      return m*3;
}
```




```
[Test]
public void ShouldReturnExpectedNumberForEvenParam()
{
    int n = 2;
    var actual = m(n);
    Assert.AreEqual(n*2, actual);
}
```

TEST COVERAGE. COVERING EACH CASE.

```
[Test]
public void ShouldReturnExpectedNumberForOddParam()
{
    int n = 3;
    var actual = m(n);
    Assert.AreEqual(n*3, actual);
}
```

I expect a high level of coverage. Sometimes managers require one. There's a subtle difference.

-- Brian Marick

TEST QUALITY.

- Tests will be more qualitative:
  - if it will catch more corner cases.
  - if it will focus on user requirements.
  - if it might be used to deduce the specs and serve as a doc.
  - if it influences the design of the system.

REFACTORING.

- **Code refactoring** is the process of restructuring existing computer code – changing the *factoring* – without changing its external behavior.
- Three steps:
  1. find bad code aka “code smell”
  2. Refactoring itself
  3. Running unit-tests

REFACTORING AND TESTS.

- "*Walking on water and developing software from a specification are easy if both are frozen*". Edward V. Berard
- Quality tests should freeze the specs of the system as below zero temperature freezes the water.
- Tests are a safety net, not a guarantee, still you need to test manually.

IMPORTANT ATTRIBUTES

1. [SetUp]

SetUp is generally used for initialization purposes. Any code that must be initialized or set prior to executing a test is put in functions marked with this attribute. As a consequence, it avoids the problem of code repetition in each test.

2. [TearDown]

This is an attribute that acts the opposite of [SetUp]. It means the code written with this attribute is executed last

IMPORTANT ATTRIBUTES

3. [TestCase]

TestCaseAttribute serves the dual purpose of marking a method with parameters as a test method and providing inline data to be used when invoking that method

```
[TestCase(12,3,4)]
[TestCase(12,2,6)]
[TestCase(12,4,3)]
public void DivideTest(int n, int d, int q)
{
  Assert.AreEqual( q, n / d );
}
```

IMPORTANT ATTRIBUTES

4. [TestCaseSource]

TestCaseSourceAttribute is used on a parameterized test method to identify the property, method or field that will provide the required arguments.

```
[Test, TestCaseSource("DivideCases")]
public void DivideTest(int n, int d, int q)
{
    Assert.AreEqual( q, n / d );
}

static object[] DivideCases =
{
    new object[] { 12, 3, 4 },
    new object[] { 12, 2, 6 },
    new object[] { 12, 4, 3 }
};
```

ORGANIZING UNIT-TESTS

Fixture per class pattern

- Fixture should be named according to type being tested, and suffixed with "Fixture" e.g. “MathFixture".
- Test should contain with which method it is you are testing, e.g. “DivideThrowsWhenDivizorIsZero" or “DivideShouldReturnExpectedResult".

ORGANIZING UNIT-TESTS

Fixture per method pattern

- Base fixture class should be named according to type being tested, and suffixed with "Fixture" e.g. “MathFixture".
- Method fixture class should be named according to method being tested, and suffixed with "Fixture" e.g. “DivideFixture".
- Test should not contain method being tested since that is implied via its fixture. E.g. name test as "ItThrowsWhenDivizorIsZero" or “ItReturnsExpectedResult".

ORGANIZING UNIT-TESTS

Fixture per class with nested class

- CREATE 1 TEST CLASS FOR EACH CLASS YOU WANT TO TEST.
- CREATE NESTED CLASSES FOR EACH MEMBER OF THE CLASS UNDER TEST.
- NAME NESTED CLASSES AND TEST METHODS SO THEY CONVEY WHAT ATTRIBUTE A MEMBER SHOULD HAVE.

ORGANIZING UNIT-TESTS

Fixture per class with nested class example

```
8  namespace MyApplication.Tests.Unit.Shared.Infra.FileSystemChannels
9  {
10     class FileSystemChannelTests
11     {
12         [TestFixture]
13         public class HasFilesProperty
14         {
15             [Test]
16             public void ReturnsTrueIfSomeFilesExist()...
24
25
26             [Test]
27             public void ReturnsFalseIfNoFilesExist()...
36         }
37
38         [TestFixture]
39         public class FileIdFromStringMethod
40         {
41             [Test]
42             public void ReturnsTextFileIdForNormalFileName()...
52
53
54             [Test]
55             public void ReturnsZippedTextFileIdForZippedTextFileName()...
65         }
66
67         [TestFixture]
68         public class GetFileIdsMethod
69         {
70         }
71     }
72 }
```

AMDARIS

MOQ

- Moq is the most popular and friendly mocking framework for .NET

```
public class ProductBusiness {
    private readonly IProductDataAccess _productDataAccess;

    public ProductBusiness(IProductDataAccess productDataAccess) {
            _productDataAccess = productDataAccess;
    }

    public bool CreateProduct(Product newProduct) {
            bool result = _productDataAccess.CreateProduct(newProduct);
            return result;
    }
}
```

MOQ

```
[Test]
public void ItShouldCallCreateProduct() {
   var mockDataAccess = new Mock <IProductDataAccess>();
   mockDataAccess.Setup(m = > m.CreateProduct(It.IsAny <Product> )).Returns(true);
   var productBusiness = new ProductBusiness(mockDataAccess.Object);

   //act
   productBusiness.CreateProduct(new Product());

   //assert

   mockDataAccess.Verify(m = > m.CreateProduct(It.IsAny <Product>  ), Times.Once())
}
```

TDD

- In principle, it is just about writing the test before the program.
- But in consequence, it leads the developer to first think about “how to use” the component (why do we need the component, what’s it for?)
  1. and only then about “how to implement”.
  2. So, it’s a testing technique as well as a design technique
- It results into components that are easy to test. It results into components that are easy to enhance and adapt.
- In the end, there is no code without a test.

TDD

- The developer can tell at any time
  1. whether everything still works as it should, or
  2. what exactly does no longer work as it once did.

TDD. MOTIVATION

- If you intend to test after you‘ve developed the system, you won‘t have the time for testing. Write the tests before the code!
- If things get complicated, you might fear that „the system“ doesn‘t work.Execute the tests and get positive feedback (everything still works) or get pointed to the bit that does not / no longer work.
- If you‘re overwhelmed by the complexity, you get frustrated. Start with the simplest thing and proceed in tiny steps!

RED – GREEN – REFACTOR

- Red. Write a little test that doesn‘t work (and perhaps doesn‘t even compile at first).
- Green. Make the test work quickly (committing whatever sins necessary)
- Refactor. Eliminate all of the duplication created in merely getting the test to work, improve the design.

TEST EXAMPLES

```
public class ProductFactory
{
    private readonly INotifyProductCreation _notifyProductCreation;

    6 references
    public ProductFactory(INotifyProductCreation notifyProductCreation)
    {
        _notifyProductCreation = notifyProductCreation;
    }

    2 references
    public Product CreateNewProduct(long price, IList<long> categoryIds,
        Action<IProductOptions> optionalParams)
    {
        return CreateProduct(price, ranking: 0, categoryIds, optionalParams);
    }

    6 references
    public Product CreateExportedProduct(long price, long ranking, IList<long> categoryIds,
        Action<IProductOptions> optionalParams)
    {
        return CreateProduct(price, ranking, categoryIds, optionalParams);
    }

    2 references
    private Product CreateProduct(long price, long ranking, IList<long> categoryIds,
        Action<IProductOptions> optionalParams)
    {
        var options = new ProductOptions();
        if (optionalParams != null)
            optionalParams(options);

        var description :string = options.GetDescription();
        if (string.IsNullOrWhiteSpace(description))
            description = "No Description available";

        var name :string = options.GetName();
        if (string.IsNullOrWhiteSpace(name))
        {
            name = "No Name";
        }

        var product = new Product(name, description, price, ranking, categoryIds);

        OnProductCreation(product);

        return product;
    }
```

TEST EXAMPLES

```
[TestFixture]
0 references
public class ProductFactoryFixture
{

    private Mock<INotifyProductCreation> _notifyProductCreationMock;

    [SetUp]
    0 references
    public void Setup()
    {
        _notifyProductCreationMock = new Mock<INotifyProductCreation>();
    }

    [Test]
    0 references
    public void WhenCreateNewProductDefaultDescriptionIsSet()
    {
        //Arrange
        var productfactory = new ProductFactory(_notifyProductCreationMock.Object);

        //Act
        var product = productfactory.CreateNewProduct( price: 23,  categoryIds: new List<long>() {1},  optionalParams: null);

        //Assert
        Assert.AreEqual( expected: "No Description available",  actual: product.Description);

    }
```

TEST EXAMPLES

```
[Test]
0 references
public void WhenExportProductDefaultDescriptionIsSet()
{
    //Arrange
    var productfactory = new ProductFactory(_notifyProductCreationMock.Object);

    //Act
    var product = productfactory.CreateExportedProduct(price: 23, ranking: 2, categoryIds: new List<long>() { 1 }, optionalParams: null);

    //Assert
    Assert.AreEqual(expected: "No Description available", actual: product.Description);

}

[TestCase(value: "MyTest", expected: "MyTest")]
[TestCase(value: "", expected: "No Description available")]
[TestCase(value: null, expected: "No Description available")]
0 references
public void CanCreateProductWithOptions(string value, string expected)
{
    var productfactory = new ProductFactory(_notifyProductCreationMock.Object);

    var product = productfactory.CreateExportedProduct(price: 23, ranking: 2, categoryIds: new List<long>() { 1 }, optionalParams: x :IProductOptions => x.WithDescription(value));
    Assert.AreEqual(expected, actual: product.Description);
}

[Test]
0 references
public void WhenProductIsCreatedNotifyIsCalled()
{
    _notifyProductCreationMock.Setup(expression: x :INotifyProductCreation => x.Notify(product: It.IsAny<Product>()));
    var productfactory = new ProductFactory(_notifyProductCreationMock.Object);

    //Act
    var product = productfactory.CreateExportedProduct(price: 23, ranking: 2, categoryIds: new List<long>() { 1 }, optionalParams: null);

    _notifyProductCreationMock.Verify(expression: x :INotifyProductCreation => x.Notify(product), Times.Once);

}
```

TEST EXAMPLES

```
[TestCase( value: "MyName", expected: "MyName")]
[TestCase( value: "", expected: "No Name")]
[TestCase( value: null, expected: "No Name")]
0 references
public void WhenCreateProductIshouldBeAbleToSetNameInOptions(string value, string expected)
{
    var productfactory = new ProductFactory(_notifyProductCreationMock.Object);

    var product = productfactory.CreateExportedProduct( price: 23, ranking: 2, categoryIds: new List<long>() { 1 }, optionalParams: x :IProductOptions => x.WithName(value));
    Assert.AreEqual(expected, actual: product.Name);
}

[Test]
0 references
public void WhenCreateproductCanSetDescriptionAndName()
{
    var productfactory = new ProductFactory(_notifyProductCreationMock.Object);

    var value = "name";
    var description = "descr";
    var product = productfactory.CreateExportedProduct( price: 23, ranking: 2, categoryIds: new List<long>() { 1 }, optionalParams: x :IProductOptions => x.WithName(value) // IProductOptions
        .WithDescription(description));
    Assert.AreEqual( expected: value, actual: product.Name);
    Assert.AreEqual( expected: description, actual: product.Description);
}
```

WHY PEOPLE LIKE IT?

- The test is the executable specification.
  - You start thinking about the goal first, then about the possible implementations.
  - You understand the program‘s behavior by looking at the tests.
- You develop just enough.
  - You get to the goal as quick as possible.
  - You don‘t develop unnecessary code.
  - There is no code without a test.
  - There is no test without a user requirement.
- Once you get one test working, you know it is working now and forever.
  - You use the tests as regression tests.
  - The tests give us the courage to refactor.

ASSIGNMENT

- UNIT-TEST A DEVELOPED CLASS OF YOUR CHOICE. USE AT LEAST TWO ORGANIZATIONAL STRUCTURES
- USE ATTRIBUTES TESTFIXTURE, TEST, TESTCASE, SETUP.
- CREATE A TEST USING MOCK.
- TRY WRITING A FIXTURE USING TDD.

AMDARIS