FROM ZERO TO HERO

UNIT TESTING 101

CESAR AGUIRRE

Introduction

DO YOU WANT TO START WRITING UNIT TESTS?

Do you want to start writing unit tests? But, you don't know where to start? If you're a beginner or a seasoned developer new to unit testing, this is the place for you.

In this ebook, you will find how to start writing your first unit tests, the most common mistakes when writing them and how to name your unit tests. This ebook is based on a series of posts I wrote on my blog about unit testing every two weeks for more than two months.

In *Chapter 1*, you will learn what a unit test is and why you should write them. I won't try to sell you the concept of unit testing. I will only give you one single reason: the confidence to change your codebase. In this chapter, you will write your first tests for Stringie, a (fictional) library to manipulate strings, using C# and MSTest.

In *Chapter 2*, you will learn 4 common mistakes we make when writing our first unit tests. Among them, you will learn not to duplicate the tested logic inside your test suite. I've seen this mistake a lot.

In *Chapter 3*, you will zoom in to one of the mistakes: naming conventions. You will learn 5 common naming conventions. You will see the same tests named following 5 different naming conventions.

In the *Appendix*, you will find some of my notes on the book "The Art of Unit Testing", a must-read on the topic of unit testing. Also, you will find the tips and best practices from all my posts on this series: from naming to mocking, in a single place.

You can follow along with the code in every chapter. Clone the GitHub repository at: Testing101. You will find Stringie source code and exercises for each chapter.

I hope this ebook can help you write your first unit tests and include them in your own codebase. If you want to learn more about unit testing, visit my blog.

If this eBook helps you, feel free to share it.

Happy testing!

Let's write our first unit tests

LET'S SEE WHAT A UNIT TEST IS AND HOW TO WRITE ONE IN C#

The book "The Art of Unit Testing" defines a unit test as...

> ***"an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work"***

From the previous definition, a unit of work is any logic exposed through public methods. Often, a unit of work returns a value, changes the internals of the system, or makes an external invocation.

If that definition answers how to test public methods, we might ask: 'What about private methods?' Short answer: we don't test them. We test private methods when we call our code through its public methods.

In short, **a unit test is code that invokes some code under test and verifies a given behavior of that code**.

WHY SHOULD WE WRITE UNIT TESTS?

Have you ever needed to change your code, but you were concerned about breaking something? I've been there too.

The main reason to write unit tests is to gain confidence. Unit tests allow us to make changes with confidence that they will work. **Unit tests allow change**.

Unit tests work like a "safety net" to prevent us from breaking things when we add features or change our codebase.

In addition, unit tests work like living documentation. **The first end-user of our code is our unit tests.** If we want to know what a library does, we should check its unit tests. Often, we will find not-documented features in the tests.

WHAT MAKES A GOOD UNIT TEST?

Now, we know what a unit test is and why we should write them. The next question we need to answer is: 'What makes a test a good unit test?' Let's see what all good unit tests have in common.

- **OUR TESTS SHOULD RUN QUICKLY**

The longer our tests take to run, the less frequent we run them. And, if we don't run our tests often, we have doors opened to bugs.

- **OUR TESTS SHOULD RUN IN ANY ORDER**

Tests shouldn't depend on the output of previous tests to run. A test should create its own state and not rely upon the state of other tests.

> ***"It could be considered unprofessional to write code without tests"***
> ***- Robert Martin, The Clean Coder***

- **OUR TESTS SHOULD BE DETERMINISTIC**

No matter how many times we run our tests, they should either fail or pass every time. We don't want our test to use random input, for example.

- **OUR TESTS SHOULD VALIDATE THEMSELVES**

We shouldn't debug our tests to make sure they passed or failed. Each test should determine the success or failure of the tested behavior. Imagine we have hundreds of tests and to make sure they pass, we have to debug every one of them. What's the point, then?

LET'S WRITE OUR FIRST UNIT TEST

Let's write some unit tests for Stringie, a (fictional) library to manipulate strings with more readable methods.

One of Stringie methods is **Remove()**. It removes chunks of text from a string. For example, **Remove()** receives a substring to remove. Otherwise, it returns an empty string if we don't pass any parameters.

```
"Hello, world!".Remove("Hello");
// ", world!"

"Hello, world!".Remove();
// ""
```

Here's the implementation of the **Remove()** method for the scenario without parameters.

```
namespace Stringie
{
    public static class RemoveExtensions
    {
        public static RemoveString Remove(this string source)
        {
            return new RemoveString(source);
        }
    }

    public class RemoveString
    {
        private readonly string _source;

        internal RemoveString(string source)
        {
            _source = source;
        }

        public static implicit operator string(RemoveString removeString)
        {
            return removeString.ToString();
        }

        public override string ToString()
        {
            return _source != null ? string.Empty : null;
        }
    }
}
```

Let's write some tests for the **Remove()** method. We can write a Console program to test these two scenarios.

```
using Stringie;
using System;

namespace TestProject
{
    class Program
    {
        static void Main(string[] args)
        {
            var helloRemoved = "Hello, world!".Remove("Hello");
            if (helloRemoved == ", world!")
            {
                Console.WriteLine("Remove Hello OK");
            }
            else
            {
                Console.WriteLine($"Remove Hello failed. Expected: ', world!'."
                     + "But it was: '{helloRemoved}'");
            }

            var empty = "Hello, world!".Remove();
            if (string.IsNullOrEmpty(empty))
            {
                Console.WriteLine("Remove: OK");
            }
            else
            {
                Console.WriteLine($"Remove failed. Expected: ''."
                    + " But it was: {empty}");
            }

            Console.ReadKey();
        }
    }
}
```

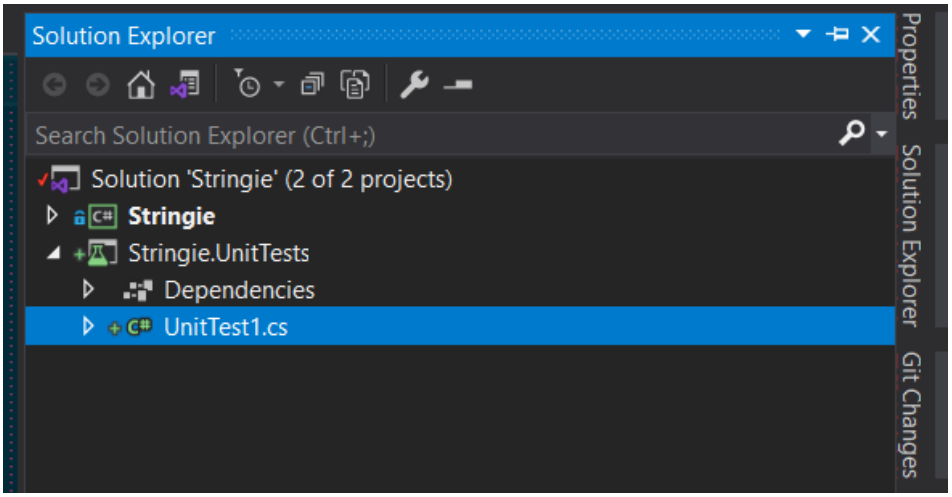
However, these aren't real unit tests. They run quickly, but they don't run in any order, and they don't validate themselves. We have to inspect the Console to check if our tests pass or fail.

WHERE SHOULD WE PUT OUR TESTS?

Let's create a new project. Let's add to the solution containing Stringie a new project of type "MSTest Test Project (.NET Core)". Since we're adding tests for the **Stringie** project, let's name our new test project **Stringie.UnitTests**.

It's my recommendation, to put our unit tests in a test project named after the project they test. We can add the suffix "Tests" or "UnitTests". For example, if we have a library called **MyLibrary**, we should name our test project: **MyLibrary.UnitTests**.

In our new test project, let's add a reference to the Stringie project.



Visual Studio Solution Explorer with our new test project

After adding the new test project, Visual Studio created a file **UnitTest1.cs**. Let's rename it! We are adding tests for the **Remove()** method, let's name this file: **RemoveTests.cs**.

One way of making our tests easy to find and group is to put our unit tests separated in files named after the unit of work or entry point of the code we're testing. Let's add the suffix "Tests". For a class **MyClass**, let's name our file: **MyClassTests**.

> If you want to follow along, check my Unit Testing 101 repository over on GitHub.

MSTEST: MICROSOFT TEST FRAMEWORK

Now, let's see what's inside our **RemoveTests.cs** file.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Stringie.UnitTests
{
    [TestClass]
    public class RemoveTests
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

It contains one normal class and method. However, they're annotated with two unusual attributes: **[TestClass]** and **[TestMethod]**. These attributes tell Visual Studio that our file contains unit tests to run.

The **[TestClass]** and **[TestMethod]** attributes belong to a project called MSTest. Microsoft Test Framework (MSTest) is an open-source unit testing framework. MSTest comes installed with Visual Studio.

Unit testing frameworks help us to write and run unit tests. Also, they create reports with the results of our tests. Other common unit testing frameworks include NUnit and XUnit.

> ***"Unit tests work like a 'safety net' to prevent us from breaking things when we add features or change our codebase."***

HOW SHOULD WE NAME OUR TESTS?

Let's replace the name **TestMethod1** with a name that follows a naming convention.

We should use naming conventions to show the feature tested and the purpose behind our tests. Tests names should tell what they're testing.

A name like **TestMethod1** doesn't say anything about the code under test and the expected result.

Tests names should tell what they're testing.

- ## IT SHOULD

One naming convention for our test names uses a sentence to tell what they're testing. Often these names start with the prefix "ItShould" followed by an action.

For our **Remove()** method, it could be: **ItShouldRemoveASubstring()** and **ItShouldReturnEmpty()**.

- **UNITOFWORK_SCENARIO_EXPECTEDRESULT**

Another convention uses underscores to separate the unit of work, the test scenario, and the expected behavior in our test names.

If we follow this convention for our example tests, we name our tests: **Remove_ASubstring_RemovesThatSubstring()** and **Remove_NoParameters_ReturnsEmpty()**.

With this convention, we can read our test names out loud like this: "*When calling Remove with a substring, then it removes that substring*".

Following the second naming convention, our tests look like this:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Stringie.UnitTests
{
    [TestClass]
    public class RemoveTests
    {
        [TestMethod]
        public void Remove_ASubstring_RemovesThatSubstring()
        {
        }

        [TestMethod]
        public void Remove_NoParameters_ReturnsEmpty()
        {
        }
    }
}
```

These names could look funny at first glance. We should use compact names in our code. However, when writing unit tests, readability is important. Every test should state the scenario under test and the expected result. We shouldn't worry about long test names.

HOW SHOULD WE WRITE OUR TESTS?

Now, let's write the body of our tests.

To write our tests, let's follow the **Arrange/Act/Assert** (AAA) principle. Each test should contain these three parts.

In the **Arrange** part, we create input values to call the entry point of the code under test.

In the **Act** part, we call the entry point to trigger the logic being tested.

In the **Assert** part, we verify the expected behavior of the code under test.

Let's use the AAA principle to replace our two examples with real tests. Also, let's use line breaks to visually separate the AAA parts.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Stringie.UnitTests
{
    [TestClass]
    public class RemoveTests
    {
        [TestMethod]
        public void Remove_ASubstring_RemovesThatSubstring()
        {
            string str = "Hello, world!";

            string transformed = str.Remove("Hello");

            Assert.AreEqual(", world!", transformed);
        }

        [TestMethod]
        public void Remove_NoParameters_ReturnsEmpty()
        {
            string str = "Hello, world!";

            string transformed = str.Remove();

            Assert.AreEqual(0, transformed.Length);
        }
    }
}
```

We used the **Assert** class from MSTest to write the Assert part of our test. This class contains methods like **AreEqual()**, **IsTrue()** and **IsNull()**.

The **AreEqual()** method checks if the result from a test is equal to an expected value. In our test, we used it to verify the length of the transformed string. We expect it to be zero.

DON'T REPEAT LOGIC IN THE ASSERTIONS

Let's use a known value in the Assert part instead of repeating the logic under test in the assertions. It's OK to hardcode some expected values in our tests. We shouldn't repeat the logic under test in our assertions. For example, we can use well-named constants for our expected values.

Here's an example of how not to write the Assertion part of our second test.

```
[TestMethod]
public void Remove_ASubstring_RemovesThatSubstring()
{
    string str = "Hello, world!";

    string transformed = str.Remove("Hello");

    var position = str.IndexOf("Hello");
    var expected = str.Substring(position + 5);
    Assert.AreEqual(expected, transformed);
}
```

Notice how it uses the **Substring()** method in the Assert part, to find the string without the **Hello** substring. A better alternative is to use the expected result, **", world!"**, in the **AreEqual()** method.

> ***"Unit tests work like living documentation. If you want to know what a library does, check its unit tests."***

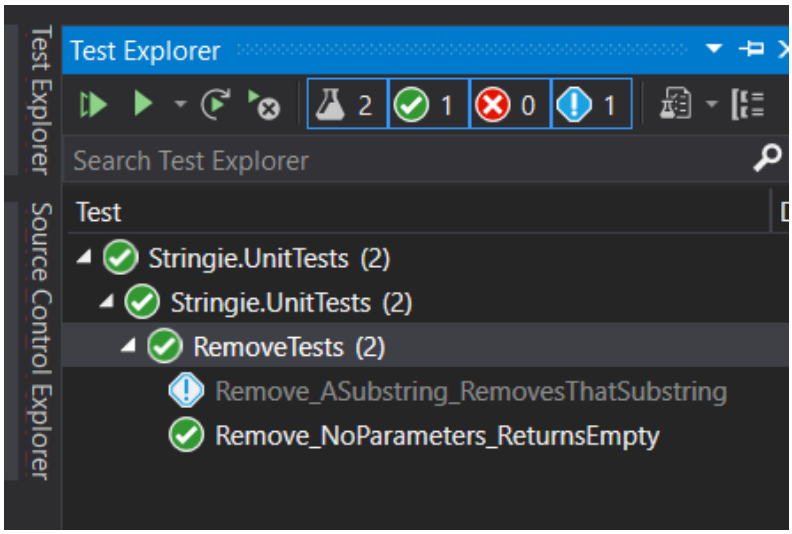
Let's rewrite our last test to use an expected value instead of repeating the logic being tested.

```
[TestMethod]
public void Remove_ASubstring_RemovesThatSubstring()
{
    string str = "Hello, world!";

    string transformed = str.Remove("Hello");

    // Here we use the expected result ", world!"
    Assert.AreEqual(", world!", transformed);
}
```

HOW CAN WE RUN A TEST INSIDE VISUAL STUDIO?

To run a test, let's right-click on the **[TestMethod]** attribute of the test and use "Run Test(s)". Visual Studio will compile your solution and run the test you clicked on.

After the test runs, let's go to the "Test Explorer" menu. There we will find the list of tests. A passed test has a green icon. If we don't have the "Test Explorer", we can use the "View" menu in Visual Studio and click "Test Explorer" to display it.



Visual Studio 'Test Explorer' showing our first passing test

That's a passing test! Hurray!

If the result of a test isn't what was expected, the Assertion methods will throw an **AssertFailedException**. This exception or any other unexpected exception flags a test as failed.

MSTEST CHEATSHEET

These are some of the most common Assertion methods in MSTest.

Assert.AreEqual **Assert.AreNotEqual**	Check if the expected value is equal to the found value Check if the expected value isn't equal to the found value
Assert.IsTrue **Assert.IsFalse**	Check if the found value is **true** Check if the found value is **false**
Assert.IsNull **Assert.IsNotNull**	Check if the found value is **null** Check if the found value isn't **null**
Assert.ThrowsException **Assert.ThrowsExceptionAsync**	Check if a method throws an exception Check if an async method throws an exception
StringAssert.Contains **StringAssert.Matches** **StringAssert.DoesNotMatch**	Check if a string contains a substring Check if a string matches a regular expression Check if a string doesn't matches a regular expression
CollectionAssert.AreEquivalent **CollectionAssert.AreNotEquivalent**	Check if two collections contain the same elements Check the opposite of the previous assertion
CollectionAssert.Contains **CollectionAssert.DoesNotContain**	Check if a collection contains an element Check if a collection doesn't contain an element

SUMMARY

- A unit test is code that invokes some code under test and verifies a given behavior of that code.
- The main reason to write unit tests is to gain confidence.
- Tests names should tell what they’re testing. Every test should state the scenario under test and the expected result.
- Follow the Arrange/Act/Assert (AAA) principle.
- Use known values in the Assert part instead of repeating the logic under test in the assertions.

Let's fix our tests

4 COMMON MISTAKES WHEN WRITING OUR FIRST UNIT TESTS

In Chapter 1, we covered how to write our first unit tests with C# and MSTest. We started from a Console program and converted it into our first unit tests. We wrote those tests for Stringie, a (fictional) library to manipulate strings with more readable methods.

In this chapter, we will cover how NOT to write unit tests. Let's learn four common mistakes we should avoid when writing our first unit tests.

- ### DO NOT FOLLOW A NAMING CONVENTION

First, keep your tests in the right place. Have one test project per project, one test class per class. Add the suffix "Tests" in the name of your test projects and classes.

Choose a naming convention for your test names and stick to it.

In Chapter 1, we covered two naming conventions. An "*ItShould*" sentence and the "*UnitOfWork_Scenario_ExpectedResult*", a three-part name separated with underscores. You can choose the one you like the most.

That time, for Stringie **Remove()** method, following the "*UnitOfWork_Scenario_ExpectedResult*" convention, we wrote test names like these ones:

```
[TestClass]
public class RemoveTests
{
    [TestMethod]
    public void Remove_ASubstring_RemovesThatSubstring() { }

    [TestMethod]
    public void Remove_NoParameters_ReturnsEmpty() { }
}
```

Every test name should tell the scenario under test and the expected result. We shouldn't worry about long test names. But, let's stop naming our tests: **Test1**, **Test2,** and so on.

Don't prefix your test names with "Test". If we're using a testing framework that doesn't need keywords in our test names, let's stop doing that. With MSTest, we have attributes like **[TestClass]** and **[TestMethod]** to mark our methods as tests.

Also, **don't use filler words like "Success" or "IsCorrect" in your test names**. Instead, let's tell what "success" and "correct" means for that test. Is it a successful test because it doesn't throw exceptions? Is it successful because it returns a value? Make your test names easy to understand.

> For more naming conventions, check Chapter 3: "*Let's name our tests*"

- **DO NOT USE THE RIGHT ASSERTION METHODS**

Follow the Arrange/Act/Assert principle. Separate the body of your tests to visually differentiate these three parts.

For the Assert part of your tests, make sure to use an assertion library. MSTest, NUnit, and XUnit are the three most popular ones for C#.

Use the right assertion methods of your library. For example, MSTest has assertion methods for strings, collections, and other objects. For a list of the most common MSTest assertions methods, check the Cheatsheet at the end of Chapter 1.

Please, don't do.

```
Assert.AreEqual(null, result);
// or
Assert.AreEqual(true, anotherResult);
```

Do, instead.

```
Assert.IsNull(result);
// or
Assert.IsTrue(anotherResult);
```

- **DO NOT HAVE A SINGLE ASSERTION PER TEST**

Have only one Act and Assert part in your tests. Don't repeat the same Act part with different test values in a single test.

Please, avoid writing tests like this one.

```
[TestMethod]
public void Remove_SubstringWithDifferentCase_RemovesSubstring()
{
    var str = "Hello, world!";

    var transformed = str.RemoveAll("Hello").IgnoringCase();
    Assert.AreEqual(", world!", transformed);

    transformed = str.RemoveAll("HELLO").IgnoringCase();
    Assert.AreEqual(", world!", transformed);

    transformed = str.RemoveAll("HeLlO").IgnoringCase();
    Assert.AreEqual(", world!", transformed);
}
```

Here, we tested the same method with different test values in a single test.

> ***"Every test name should tell the scenario under test and the expected result"***

Also, avoid writing tests like this one.

```
[TestMethod]
public void Remove_SubstringWithDifferentCase_RemovesSubstring()
{
    var str = "Hello, world!";

    var testCases = new string[]
    {
        "Hello",
        "HELLO",
        "HeLlO"
    };
    string transformed;
    foreach (var str in testCases)
    {
        transformed = str.RemoveAll("Hello").IgnoringCase();
        Assert.AreEqual(", world!", transformed);
    }
}
```

This time, to avoid repetition, we put the test values in an array and looped through them to test each value.

If we want to test the same scenario with different test values, let's use parameterized tests.

PARAMETERIZED TESTS WITH MSTEST

To write a parameterized test with MSTest, we can follow these steps:

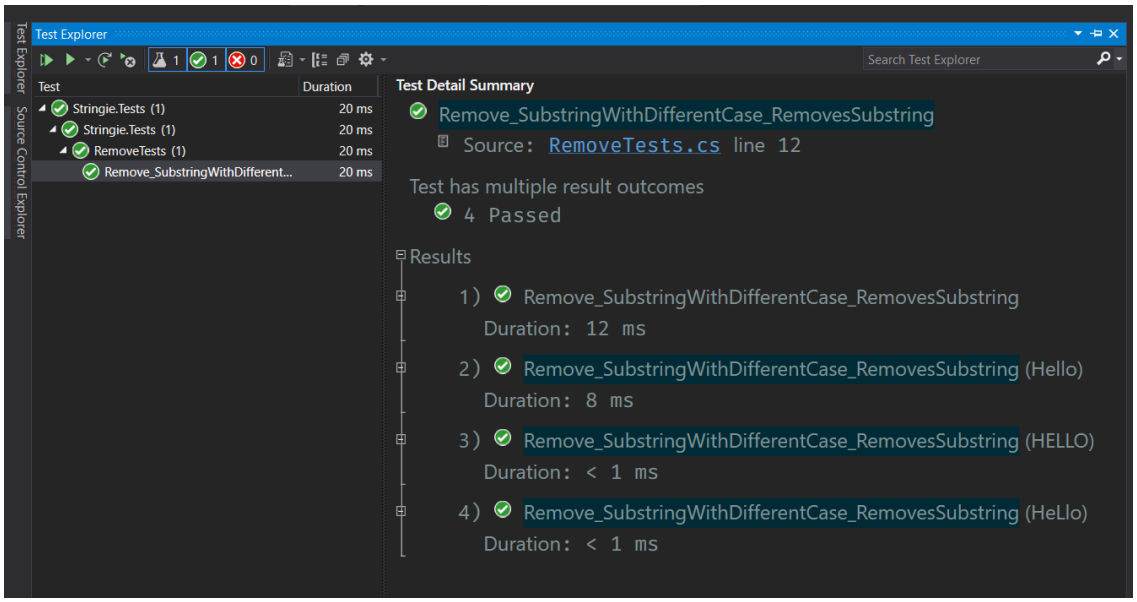
- Replace the **[TestMethod]** attribute with the **[DataTestMethod]** attribute in your test.
- Add **[DataRow]** attributes for each set of test values.
- Add parameters for each test value inside the **[DataRow]** attributes.
- Use the input parameters in your test to arrange, act or assert.

Let's convert the previous test with many test values into a parameterized test.

```
[DataTestMethod]
[DataRow("Hello")]
[DataRow("HELLO")]
[DataRow("HeLlo")]
public void Remove_SubstringWithDifferentCase_RemovesSubstring(
      string substringToRemove)
{
    var str = "Hello, world!";

    var transformed = str.RemoveAll(substringToRemove).IgnoringCase();

    Assert.AreEqual(", world!", transformed);
}
```

With parameterized tests, we have separate tests. Inside Visual Studio, in the "Test Explorer" menu, we will have one result per each **[DataRow]** attribute in the parent test.

Parameterized tests make troubleshooting easier when we have a test that fails for a single test value.



Visual Studio 'Test Explorer' showing the result outcomes for our parameterized test

- **REPEAT LOGIC IN YOUR ASSERTIONS**

I can't stress this enough. **Don't repeat the logic under test in your assertions.** Use known, hard-coded, pre-calculated values instead.

We shouldn't copy the tested logic and paste it in a private method in our tests to use it in our assertions. We will have code and bugs in two places.

Please, don't write assertions like the one in this test.

```
[TestMethod]
public void Remove_ASubstring_RemovesThatSubstringFromTheEnd()
{
    string str = "Hello, world!";

    string transformed = str.Remove("world!").From(The.End);

    var position = str.IndexOf("world!");
    var expected = str.Substring(0, position);
    Assert.AreEqual(expected, transformed);
}
```

For this test, instead of using the **Substring()** method to remove the input string, use a known expected value. Write **Assert.AreEqual("Hello,", transformed)**. For example,

```
[TestMethod]
public void Remove_ASubstring_RemovesThatSubstringFromTheEnd()
{
    string str = "Hello, world!";

    string transformed = str.Remove("world!").From(The.End);

    // Let's use a known value in our assertions
    Assert.AreEqual("Hello,", transformed)
}
```

Voilà! These are four common mistakes we make when writing our first unit tests.

Remember to put your test in the right places following a naming convention. Also, keep one assertion per test, and don't repeat logic in your assertions. You will have better tests avoiding these four mistakes.

SUMMARY

- Choose a naming convention for your test names and stick to it.
- Don't prefix test names with "Test". Don't use filler words like "Success" or "IsCorrect".
- Separate the body of your tests to visually differentiate the three AAA parts.
- Have only one Act and Assert part in your tests.
- Use parameterized tests to test the same scenario with different test values.
- Don't repeat the logic under test in your assertions. Use known/pre-calculated values instead.

Let's name our tests

5 NAMING CONVENTIONS FOR BETTER TEST NAMES

In Chapter 2, we learned 4 common mistakes we make when writing our first unit tests. One of them is not to follow a naming convention. In Chapter 1, we covered two naming conventions, let's bring back those two and see three new naming conventions for our unit tests.

Test names should tell the scenario under test and the expected results. Long names are acceptable when writing unit tests since test names show the purpose behind what they're testing. Write better test names instead of assertions messages.

Let's continue to use Stringie, a (fictional) library to manipulate string. Stringie has a **Remove()** method to remove a substring from the beginning or the end of an input string. When **Remove()** receives no parameters, it returns an empty string.

- ### ITSHOULD

```
[TestClass]
public class RemoveTests
{
    [TestMethod]
    public void ItShouldRemoveASubstring() { }

    [TestMethod]
    public void ItShouldReturnEmpty() { }
}
```

This first naming convention uses a sentence to tell what we're testing in each test. We start these names with the prefix "ItShould" followed by an action.

For our **Remove()** method, it could be: **ItShouldRemoveASubstring** and **ItShouldReturnEmpty**.

- **UNITOFWORK_SCENARIO_EXPECTEDRESULT**

```
[TestClass]
public class RemoveTests
{
    [TestMethod]
    public void Remove_NoParameters_ReturnsEmpty() {}

    [TestMethod]
    public void Remove_ASubstring_RemovesOnlyASubstring() {}
}
```

We find this naming convention in the book "The Art of Unit Testing". This convention uses underscores to separate the unit of work or entry point, the test scenario, and the expected behavior.

With this convention, we can read our test names out loud like this: "*When calling Remove with no parameters, then it returns empty*".

Don't worry about long test names

- **PLAIN ENGLISH SENTENCE**

```
[TestClass]
public class RemoveTests
{
    [TestMethod]
    public void Returns_empty_with_no_parameters() {}

    [TestMethod]
    public void Removes_only_a_substring() {}
}
```

Unlike the "UnitOfWork_Scenario_ExpectedResult" convention, this convention strives for a less rigid name structure.

This convention uses sentences in plain English for test names. We describe in a sentence what we're testing in a language easy to understand even for non-programmers. For more readability, we separate each word in our sentence with underscores.

This convention considers smells adding method names and filler words like "should" or "should be" in our test names. For example, instead of writing, **should_remove_only_a_substring()**, we should write **removes_only_a_substring()**.

You could read more about this convention in You are naming your tests wrong!

> ***"Don't use assertion messages. Write better test names"***

- **SENTENCE FROM CLASSES AND METHODS NAMES**

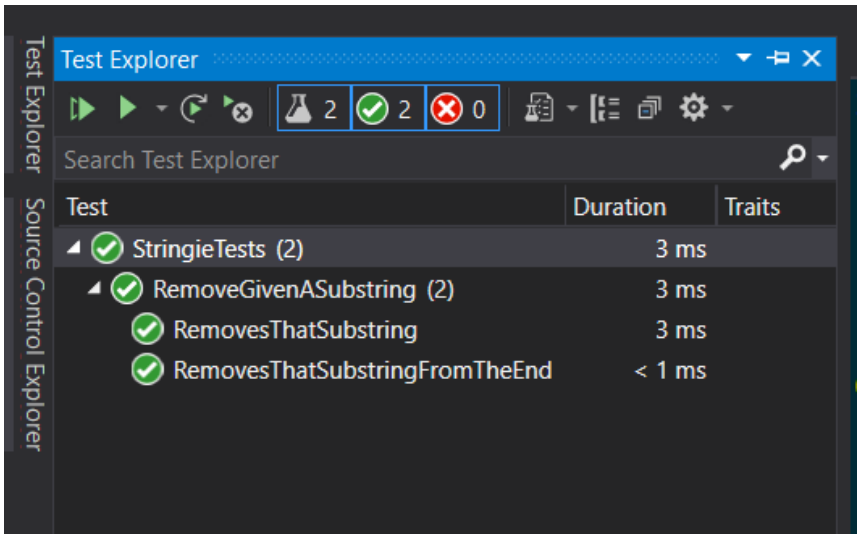
```
[TestClass]
public class RemoveGivenASubstring
{
    [TestMethod]
    public void RemovesThatSubstring() {}

    [TestMethod]
    public void RemovesThatSubstringFromTheEnd() {}
}
```

This naming convention uses sentences in plain English too. In this case, class names will act as the subject of our sentences and method names as the verb and the complement. We write the unit of work or entry point in class names and the expected result in method names.

Also, we can split different scenarios into separate classes. We add the scenarios in class names with the work **Given** followed by the scenario under test.

For our **Remove()** method, we can name our test class **RemoveGivenASubstring** and our test methods **RemovesOnlyASubstring()** and **RemovesSubstringFromTheEnd()**.

With this convention, we can read our test names like full sentences in the "Test Explorer" menu in Visual Studio when we group our tests by class. Like this: "*Remove, given a substring, removes that substring*".



Visual Studio 'Solution Explorer' showing a our sample tests grouped by class

- **NESTED CLASSES AND METHODS**

```
[TestClass]
public class RemoveTests
{
    [TestMethod]
    public void ReturnsEmpty() {}

    [TestClass]
    public class GivenASubstring
    {
        [TestMethod]
        public void RemovesThatSubstring() {}

        [TestMethod]
        public void RemovesThatSubstringFromTheEnd() {}
    }
}
```

This last convention uses sentences split into class and method names too. Unlike the previous naming convention, each scenario has its own nested class.

For example, instead of having a test class **RemoveGivenASubstring**, we create a nested class **GivenASubstring** inside a **RemoveTests** class.

Voilà! That's how we can name our unit tests. Remember naming things is hard. Pick one of these five naming conventions and stick to it. But, if you inherit a codebase, prefer the convention already in use. I hope you can write more readable test names now.

SUMMARY

- Choose a naming convention for your test names and stick to it.
- Long names are acceptable when writing unit tests since test names show the purpose behind what they're testing.
- Don't write assertion messages, write better test names instead.
- Write test names easy to understand even for non-programmers.

Let's learn more

THE ART OF UNIT TESTING: TAKEAWAYS

This is THE book to learn how to write unit tests. It starts from the definition of a unit test to how to implement them at the organization level. Although it covers extensively the subject, it doesn't advocate writing unit tests before or after the production code.

"The Art of Unit Testing" teaches us to treat tests the same way we treat production code. Sometimes, we write unit tests without the same care and attention we put into production code.

These are some of the main ideas from "The Art Of Unit Testing".

WRITE TRUSTWORTHY TESTS

A test is trustworthy if you don't have to debug it to make sure it passes.

To write trustworthy tests, avoid any logic in your tests. If you have conditionals and loops in your tests, you have logic in them.

You can find logic in helper methods, fakes, and assert statements. To avoid logic in the assert statements, use hardcoded values instead.

Tests with logic are hard to read and replicate. A unit test should consist of method calls and assert statements.

HAVE A SAFE GREEN ZONE

Keep a set of always-passing unit tests. You will need some configurations for your integration tests: a database connection, environment variables, or some files in a folder. Integration tests will fail if those configurations aren't in place. So, developers could ignore some failing tests, and real issues, because of those missing configurations.

Therefore, **separate your unit tests from your integration tests**. Put them in different projects. This way, you will distinguish between a missing setup and an actual problem with your code.

ORGANIZE YOUR TESTS

Have a unit test project per project and a test class per class. You should easily find tests for your classes and methods.

Create separate projects for your unit and integration tests. Add the suffix "*UnitTests*" and "*IntegrationTests*" accordingly.

Create tests inside a file with the same name as the tested code adding the suffix "*Tests*". You can group features in separate files adding the name of the feature as a suffix. For example, "*MyClassTests.AnAwesomeFeature*".

BUILDERS VS SETUP METHODS

Use builders instead of SetUp methods. Tests should be isolated from other tests. Sometimes, **SetUp** methods create shared state among our tests. We will find tests that pass in isolation but don't pass alongside other tests and tests that need to be run many times to pass.

Often **SetUp** methods end up with initialization for only some tests. Tests should create their own world. Therefore, initialize what's needed inside every test using builders.

Voilà! These are some of my takeaways. The main lesson from this book is to write readable, maintainable and trustworthy tests. Remember the next person reading your tests will be you.

> ***"Write readable, maintainable and trustworthy tests"***

Unit Testing Best Practices

ON NAMING

- [ ] Every test name should tell the scenario under test and the expected result
- [ ] Write your test names in a language easy to understand even for non-programmers
- [ ] Don't prefix your test names with "*Test*"
- [ ] Don't use filler words like "*Success*" or "*IsCorrect*" in test names

ON ORGANIZATION

- [ ] Put your tests in a test project named after the project they test
- [ ] Put your tests in files named after the entry point of the code you're testing

ON ASSERTIONS

- [ ] Use line breaks to visually separate the three AAA parts in the body of your tests
- [ ] Don't repeat the logic under test in your assertions
- [ ] Don't make private methods public to test them
- [ ] Have a single Act and Assert parts in your tests
- [ ] Use the right assertion methods of your testing framework: IsNull vs IsTrue
- [ ] Prefer assertion methods for strings like Contains, Matches and StartsWith

ON TEST DATA

- [ ] Use factory methods to reduce complex Arrange scenarios
- [ ] Make your scenario under test and test values extremely obvious
- [ ] Use object mothers to create input test values
- [ ] Prefer Builders to create complex object graphs

ON STUBS AND MOCKS

- [ ] Use fakes when you depend on external systems you don't control
- [ ] Avoid complex logic inside your fakes
- [ ] Don't write assertions for stubs
- [ ] Keep one mock per test
- [ ] Make tests set their own values for fakes

CESAR AGUIRRE

THAT'S ALL FOLKS!

If you have reached this point, you know the basics to start your unit testing journey. We covered how to write your first unit tests with C# and MSTest, how to structure your tests, what mistakes to avoid, and how to name your tests. But, this isn't the end. Unit testing is a broad subject.

For more content about unit testing, stay tuned to my Unit Testing 101 series on my blog.

I didn't have the chance to introduce myself. Hola, I'm Cesar Aguirre! A software engineer, lifelong learner, language enthusiast, and vivid reader. I'm a software engineer working remotely from Colombia (*not Columbia*). I help teams to write maintainable and performant backend code. You can find me online on my blog, dev.to, GitHub and LinkedIn.

Feel free to share, email, and redistribute this ebook.

Happy testing!