Creating Testable Code

Xavier Morera

Helping .NET developers create amazing applications

@xmorera / www.xaviermorera.com / www.bigdatainc.org

Creating Testable Code

Writing unit tests is highly recommended

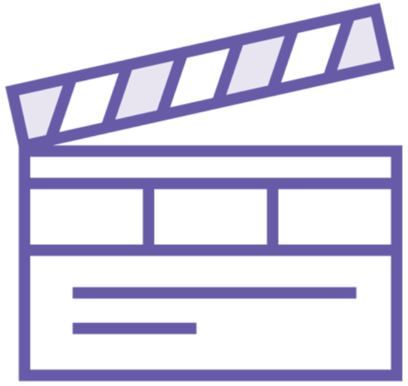
- Maybe even mandatory

Objective

- Make sure that relevant functions provide the correct output

The Three A's

Arrange



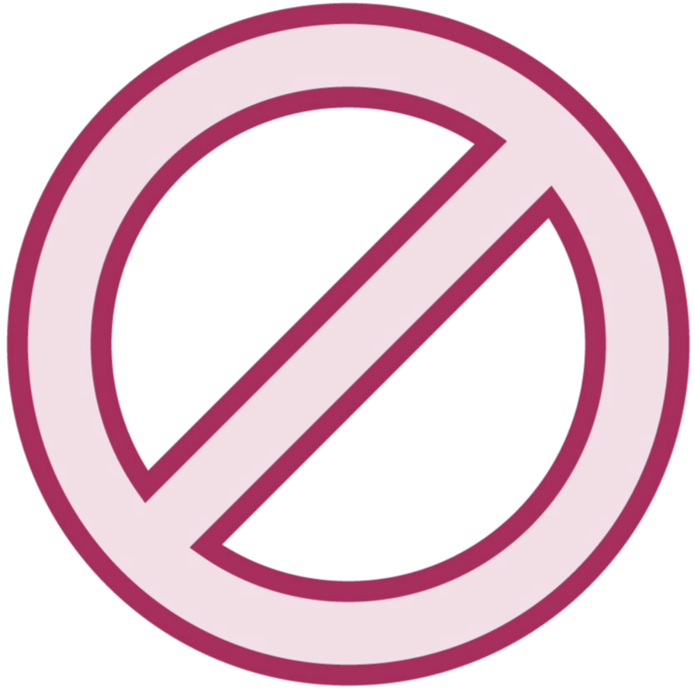
Act

Assert

Why Unit Tests?

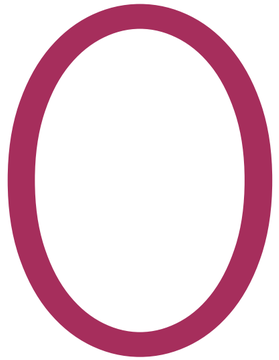
Got Unit Tests?

Yes!



Hmmm... I will create the unit tests tomorrow...

Got Unit Tests?

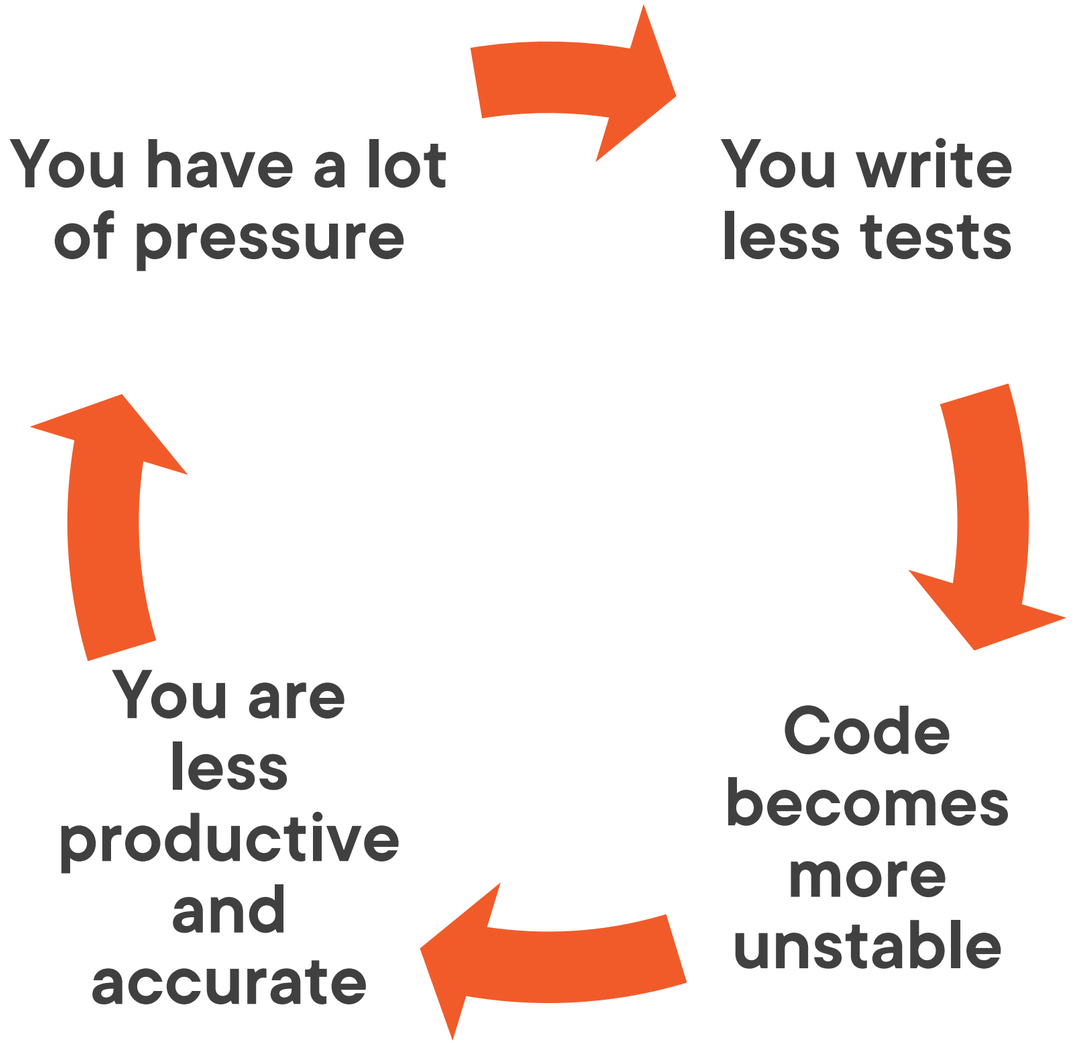


Nope

Highly recommended to write unit tests

Why?

The Cycle


You write less tests
Code becomes more unstable
You are less productive and accurate
You have a lot of pressure

Benefits of Unit Testing

Less functional testing required

Prevent and identify regression bugs

Code decoupling

Help improve design

*** TDD (Different opinions)**

Anatomy of a Unit Test

Disclaimer

This is not a Unit Testing training

A Unit Test

```
public void KilometersToMilesTest()
{









}
```

A Unit Test

```
public void KilometersToMilesTest()
{
```

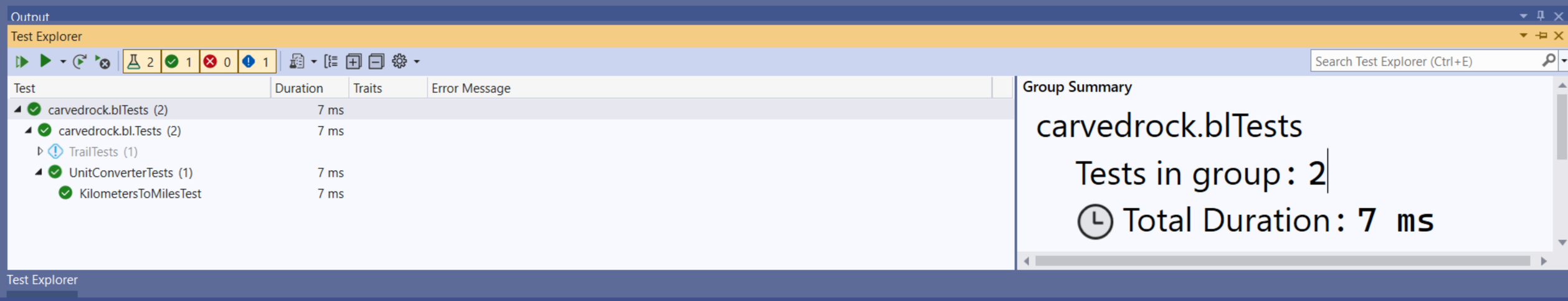
Why is it called unit testing?

Because you break down the functionality of your program into discrete testable behaviors that you can test as individual units

```
}
```

A Unit Test

```
[TestMethod()]
public void KilometersToMilesTest()
{
```



A Unit Test

```
[TestMethod()]
public void KilometersToMilesTest()
{
    // Arrange



    // Act


    // Assert

}
```

A Unit Test

```
[TestMethod()]
public void KilometersToMilesTest()
{
    // Arrange
    decimal kilometers = 50.0M;
    decimal expected = 31.0686M;

    // Act
    decimal actual = UnitConverter.KilometersToMiles(kilometers);

    // Assert
    Assert.AreEqual(expected, actual);
}
```

A Unit Test

```
[TestMethod()]
public void KilometersToMilesTest()
{
    // Arrange
    decimal kilometers = 50.0M;
    decimal expected = 31.0686M;

    // Act
    decimal actual = UnitConverter.KilometersToMiles(kilometers);

    // Assert
    Assert.AreEqual(expected, actual);
}
```

Version
Visual Studio 2022 SDK
Search
Microsoft.VisualStudio.TestTools.UnitTesting
AfterAssemblyCleanupEventArgs
AfterAssemblyInitializeEventArgs
AfterClassCleanupEventArgs
AfterClassInitializeEventArgs
AfterTestCleanupEventArgs
AfterTestInitializeEventArgs
AssemblyCleanupAttribute
AssemblyInitializeAttribute
Assert
Assert
Properties
Methods
AssertFailedException
AssertInconclusiveException
BaseShadow
BaseShadow.ElementConverter
BeforeAssemblyCleanupEventArgs
BeforeAssemblyInitializeEventArgs

... / Microsoft.VisualStudio.TestTools.UnitTesting /
C++

Assert Class

Reference

Feedback

Definition

Namespace: Microsoft.VisualStudio.TestTools.UnitTesting

Assemblies: Microsoft.VisualStudio.QualityTools.UnitTestFramework.dll, Microsoft.VisualStudio.TestPlatform.TestFramework.dll

A collection of helper classes to test various conditions within unit tests. If the condition being tested is not met, an exception is thrown.

C++ Copy

```
public ref class Assert abstract sealed
```

Inheritance Object → Assert

Properties

That	Gets the singleton instance of the Assert functionality.

Version

Visual Studio 2022 SDK

Search

Methods

AreEqual(Double, Double, Double)	Tests whether the specified doubles are equal and throws an exception if they are not equal.
AreEqual(Double, Double, Double, String)	Tests whether the specified doubles are equal and throws an exception if they are not equal.
AreEqual(Double, Double, Double, String, Object[])	Tests whether the specified doubles are equal and throws an exception if they are not equal.
AreEqual(Object, Object)	Tests whether the specified objects are equal and throws an exception if the two objects are not equal. Different numeric types are treated as unequal even if the logical values are equal. 42L is not equal to 42.
AreEqual(Object, Object, String)	Tests whether the specified objects are equal and throws an exception if the two objects are not equal. Different numeric types are treated as unequal even if the logical values are equal. 42L is not equal to 42.
AreEqual(Object, Object, String, Object[])	Tests whether the specified objects are equal and throws an exception if the two objects are not equal. Different numeric types are treated as unequal even if the logical values are equal. 42L is not equal to 42.
AreEqual(Single, Single, Single)	Tests whether the specified floats are equal and throws an exception if they are not equal.
AreEqual(Single, Single, Single, String)	Tests whether the specified floats are equal and throws an exception if they are not equal.
AreEqual(Single, Single, Single, String, Object[])	Tests whether the specified floats are equal and throws an exception if they are not equal.
AreEqual(String, String, Boolean)	Tests whether the specified strings are equal and throws an exception if they are not equal. The invariant culture is used for the comparison.

Version

Visual Studio 2022 SDK

Search

Microsoft.VisualStudio.TestTools.UnitTesting
AfterAssemblyCleanupEventArgs
AfterAssemblyInitializeEventArgs
AfterClassCleanupEventArgs
AfterClassInitializeEventArgs
AfterTestCleanupEventArgs
AfterTestInitializeEventArgs
AssemblyCleanupAttribute
AssemblyInitializeAttribute
Assert
Assert
Properties
Methods
AssertFailedException
AssertInconclusiveException
BaseShadow
BaseShadow.ElementConverter
BeforeAssemblyCleanupEventArgs
BeforeAssemblyInitializeEventArgs
BeforeClassCleanupEventArgs

IsFalse(Boolean, String, Object[])	Tests whether the specified condition is false and throws an exception if the condition is true.
IsInstanceOfType(Object, Type)	Tests whether the specified object is an instance of the expected type and throws an exception if the expected type is not in the inheritance hierarchy of the object.
IsInstanceOfType(Object, Type, String)	Tests whether the specified object is an instance of the expected type and throws an exception if the expected type is not in the inheritance hierarchy of the object.
IsInstanceOfType(Object, Type, String, Object[])	Tests whether the specified object is an instance of the expected type and throws an exception if the expected type is not in the inheritance hierarchy of the object.
IsNotInstanceOfType(Object, Type)	Tests whether the specified object is not an instance of the wrong type and throws an exception if the specified type is in the inheritance hierarchy of the object.
IsNotInstanceOfType(Object, Type, String)	Tests whether the specified object is not an instance of the wrong type and throws an exception if the specified type is in the inheritance hierarchy of the object.
IsNotInstanceOfType(Object, Type, String, Object[])	Tests whether the specified object is not an instance of the wrong type and throws an exception if the specified type is in the inheritance hierarchy of the object.
IsNotNull(Object)	Tests whether the specified object is non-null and throws an exception if it is null.
IsNotNull(Object, String)	Tests whether the specified object is non-null and throws an exception if it is null.
IsNotNull(Object, String, Object[])	Tests whether the specified object is non-null and throws an exception if it is null.
IsNull(Object)	Tests whether the specified object is null and throws an exception if it is not.

Some Assert Methods

AreEqual, AreNotEqual

Fail

IsTrue, IsFalse

IsInstanceOfType, IsNotInstanceOfType

IsNull, IsNotNull

AreSame, AreNotSame

ThrowsException

https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert

Assert AreEqual

```
[TestMethod()]
public void Convert50KilometresToMiles()
{
    // 1.Arrange
    decimal kilometres = 50.0M;
    decimal expected = 31.0686M;

   // 2.Act
    decimal actual = MeasuresConverter.KilometresToMiles(kilometres);

   // 3. Assert
   Assert.AreEqual(expected, actual);
}
```

Assert AreNotEqual

```
[TestMethod()]
public void EstimateTimeNegativeElevation()
{
    // 1.Arrange
    Trail trail = new() { DistanceInMiles = 6, ElevationInFeet = -2000 };
    double notExpected = 1.0;

   // 2.Act
    double actual = trail.EstimateTime();

   // 3.Assert
    Assert.AreNotEqual(notExpected, actual);
}
```

Assert IsTrue

```
[TestMethod()]
public void EstimateTime6Miles2000Elevation()
{
    // 1.Arrange
    Trail trail = new() { DistanceInMiles = 6, ElevationInFeet = 2000 };
    double expected = 3.0;

   // 2.Act
    double actual = trail.EstimateTime();

   // 3.Assert
    Assert.IsTrue(expected == actual);
}
```

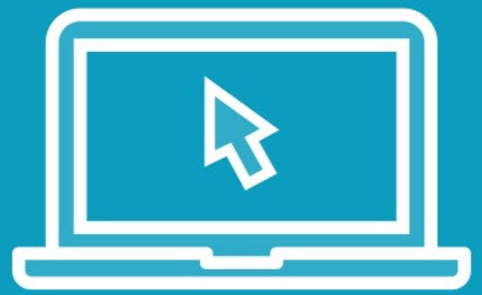
Assert Fail

```
[TestMethod()]
public void EstimateTimeWithNegativeDistance()
{
    // 1.Arrange
    Trail trail = new() { DistanceInMiles = -1, ElevationInFeet = 2000 };

   try
    {
        // 2.Act
        double _ = trail.EstimateTime();

       // 3.Assert
        Assert.Fail(“No exception was thrown, and an exception was expected");
    }
    catch
    {
        Console.WriteLine("Exception thrown, which is the expected scenario");
    }
}
```

Demo



Creating a unit test

Best Practices for Writing Unit Tests

Code Coverage

Code coverage measures how much of your code is being tested

Higher code coverage

- Typically related to higher quality code

However, this metric does not indicate the quality of your code

Law of diminishing returns applies

Characteristics of a Good Unit Test

Tests should run fast, especially on larger projects

Fast

No dependencies on outside factors, like a db

Isolated

Consistent results on every run

Repeatable

Characteristics of a Good Unit Test

Pass or fail automatically detected without human intervention

Self-checking

Writing the unit test should not take a lot more than writing the actual code

Timely

Use the AAA Unit Test Pattern

Use the AAA pattern

Arrange

- Create and set up your objects

Act

- Perform an action on the object

Assert

- Validate the result

Naming Your Tests

Naming standard is important

Name explicitly expresses

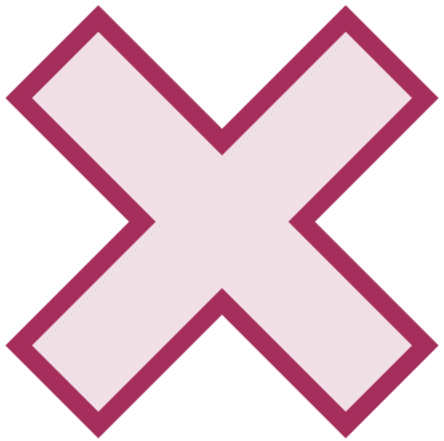
- The intent of the test
- Scenario
- Expected behavior

Use Simple Input

Input used should be as simple as possible

Tests that require complex input have higher chances of introducing errors

Avoid Magic Strings



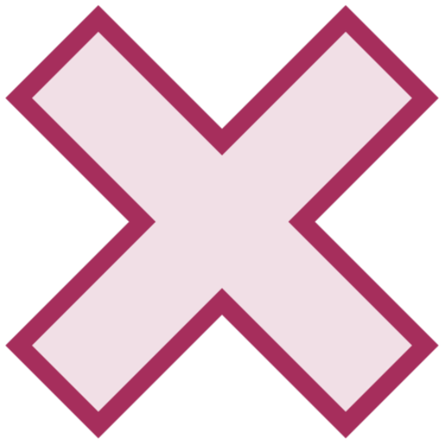
Do not use magic strings

- Values that have a special meaning but have no direct explanation

Instead, assign these values to constants

- With representative names

Avoid Logic in Tests



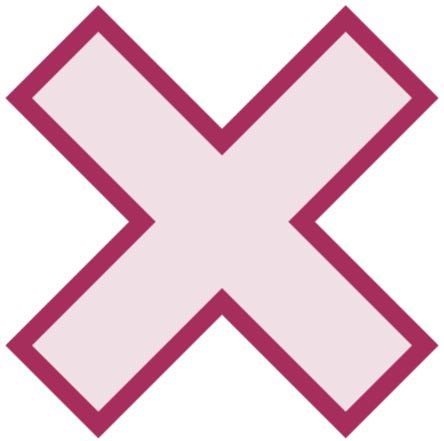
Logic should go in the business logic

Test should focus on testing a scenario

- Instead of implementation details

Use helper methods instead for setup and teardown

Avoid Multiple Acts



Do not have multiple acts per test

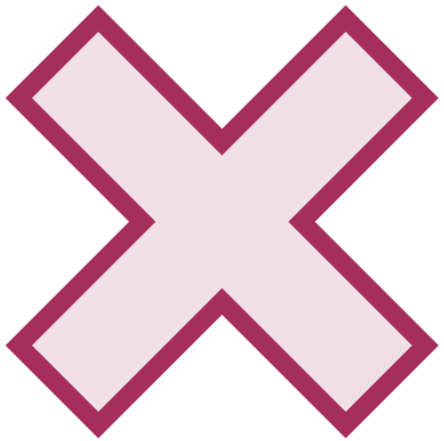
- Will make a failed test unclear

Instead, write one act per test

Validating Private Methods

Validate private methods using public methods

Static References



Unit test must be repeatable and predictable

Static references may cause issues

Try to avoid calls to static references

Takeaway

Writing unit tests is highly recommended

- Even mandatory in some cases

Objective

- Make sure functions provide correct output

Takeaway

Multiple benefits

- Less functional testing required
- Prevent and identify regression bugs
- Code decoupling
- Help improve design

Takeaway

Unit test is a method

- [TestMethod()] attribute

Follows the AAA pattern

- Arrange
  - Prepare for the test
- Act
  - Invoke method being tested
- Assert
  - Validate the result

Takeaway

Assert class used often for verification

- AreEqual
- AreNotEqual
- Fail
- IsInstanceOf

Takeaway

Characteristics of a good unit test

- Fast
- Isolated
- Repeatable
- Self-checking
- Timely

Takeaway

Many best practices for writing unit tests

- Use the AAA pattern
  - Only one act
  - Without complex logic
- Name that explicitly expresses intent, scenario, and expected behavior
- Use simple input
- Avoid magic strings