FREE eBook

LEARNING unit-testing

Free unaffiliated eBook created from
Stack Overflow contributors.

#unit-testing

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unit-testing

It is an unofficial and free unit-testing ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unit-testing.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with unit-testing

Remarks

Unit testing describes the process of testing individual units of code in isolation from the system that they are a part of. What constitutes a unit can vary from system to system, ranging from an individual method to a group of closely related classes or a module.

The unit is isolated from its dependencies using test doubles when necessary and setup into a known state. Its behaviour in reaction to stimuli (method calls, events, simulated data) is then tested against the expected behaviour.

Unit testing of entire systems can be done using custom written test harnesses, however many test frameworks have been written to help streamline the process and take care of much of the plumbing, repetitive and mundane tasks. This allows developers to concentrate on what they want to test.

When a project has enough unit tests any modification of adding new functionality or performing a code refactoring can be done easily by verifying at the end that everything works as before.

Code Coverage, normally expressed as a percentage, is the typical metric used to show how much of the code in a system is covered by Unit Tests; note that there is no hard and fast rule about how high this should be, but it is generally accepted that the higher, the better.

Test Driven Development (TDD) is a principle that specify that a developer should start coding by writing a failing unit test and only then to write the production code that make the test pass. When practicing TDD, it can be said that the tests themselves are the first consumer of the code being created; therefore they help to audit and drive the design of the code so that it is as simple to use and as robust as possible.

Versions

Unit testing is a concept that does not have version numbers.

Examples

A basic unit test

At its simplest, a unit test consists of three stages:

- Prepare the environment for the test
- Execute the code to be tested
- Validate the expected behaviour matches the observed behaviour

These three stages are often called 'Arrange-Act-Assert', or 'Given-When-Then'.

Below is example in C# that uses the NUnit framework.

```
[TestFixture]
public CalculatorTest
{
   [Test]
   public void Add_PassSevenAndThree_ExpectTen()
   {
       // Arrange - setup environment
       var systemUnderTest = new Calculator();

       // Act - Call system under test
       var calculatedSum = systemUnderTest.Add(7, 3);

       // Assert - Validate expected result
       Assert.AreEqual(10, calculatedSum);
  }
}
```

Where necessary, an optional fourth clean up stage tidies up.

A unit test with stubbed dependency

Good unit tests are independent, but code often has dependencies. We use various kinds of test doubles to remove the dependencies for testing. One of the simplest test doubles is a stub. This is a function with a hard-coded return value called in place of the real-world dependency.

```
// Test that oneDayFromNow returns a value 24*60*60 seconds later than current time

let systemUnderTest = new FortuneTeller()       // Arrange - setup environment
systemUnderTest.setNow(() => {return 10000})    //   inject a stub which will
                                                //   return 10000 as the result

let actual = systemUnderTest.oneDayFromNow()    // Act - Call system under test

assert.equals(actual, 10000 + 24 * 60 * 60)     // Assert - Validate expected result
```

In production code, `oneDayFromNow` would call Date.now(), but that would make for inconsistent and unreliable tests. So here we stub it out.

A unit test with a spy (interaction test)

Classic unit tests test *state*, but it can be impossible to properly test methods whose behavior depends on other classes through state. We test these methods through *interaction tests*, which verify that the system under test correctly calls its collaborators. Since the collaborators have their own unit tests, this is sufficient, and actually a better test of the actual responsibility of the tested method. We don't test that this method returns a particular result given an input, but instead that it correctly calls its collaborator(s).

```
// Test that squareOfDouble invokes square() with the doubled value

let systemUnderTest = new Calculator()          // Arrange - setup environment
let square = spy()
```

```
systemUnderTest.setSquare(square)               //   inject a spy

let actual = systemUnderTest.squareOfDouble(3)  // Act - Call system under test

assert(square.calledWith(6))                    // Assert - Validate expected interaction
```

Simple Java+JUnit Test

JUnit is the leading testing framework used for testing Java code.

The class under test models a simple bank account, that charges a penalty when you go overdrawn.

```
public class BankAccount {
    private int balance;

    public BankAccount(int i){
        balance = i;
    }

    public BankAccount(){
        balance = 0;
    }

    public int getBalance(){
        return balance;
    }

    public void deposit(int i){
        balance += i;
    }

    public void withdraw(int i){
        balance -= i;
        if (balance < 0){
            balance -= 10; // penalty if overdrawn
        }
    }
}
```

This test class validates the behaviour of some of the `BankAccount` public methods.

```
import org.junit.Test;
import static org.junit.Assert.*;

// Class that tests
public class BankAccountTest{

    BankAccount acc;

    @Before                         // This will run **before** EACH @Test
    public void setUptestDepositUpdatesBalance(){
        acc = new BankAccount(100);
    }

    @After                         // This Will run **after** EACH @Test
    public void tearDown(){
```

```
    // clean up code
    }

    @Test
    public void testDeposit(){
       // no need to instantiate a new BankAccount(), @Before does it for us

        acc.deposit(100);

        assertEquals(acc.getBalance(),200);
    }

    @Test
    public void testWithdrawUpdatesBalance(){
        acc.withdraw(30);

        assertEquals(acc.getBalance(),70); // pass
    }

    @Test
    public void testWithdrawAppliesPenaltyWhenOverdrawn(){

        acc.withdraw(120);

        assertEquals(acc.getBalance(),-30);
    }
}
```

Unit Test with Parameters using NUnit and C#

```
using NUnit.Framework;

namespace MyModuleTests
{
    [TestFixture]
    public class MyClassTests
    {
        [TestCase(1, "Hello", true)]
        [TestCase(2, "bye", false)]
        public void MyMethod_WhenCalledWithParameters_ReturnsExpected(int param1, string
param2, bool expected)
        {
        //Arrange
        var foo = new MyClass(param1);

        //Act
        var result = foo.MyMethod(param2);

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

A basic python unit test

```
import unittest
```

```
def addition(*args):
    """ add two or more summands and return the sum """

    if len(args) < 2:
        raise ValueError, 'at least two summands are needed'

    for ii in args:
        if not isinstance(ii, (int, long, float, complex )):
            raise TypeError

    # use build in function to do the job
    return sum(args)
```

Now the test part:

```
class Test_SystemUnderTest(unittest.TestCase):

    def test_addition(self):
        """test addition function"""

        # use only one summand - raise an error
        with self.assertRaisesRegexp(ValueError, 'at least two summands'):
            addition(1)

        # use None - raise an error
        with self.assertRaises(TypeError):
            addition(1, None)

        # use ints and floats
        self.assertEqual(addition(1, 1.), 2)

        # use complex numbers
        self.assertEqual(addition(1, 1., 1+2j), 3+2j)

if __name__ == '__main__':
    unittest.main()
```

An XUnit test with parameters

```
using Xunit;

public class SimpleCalculatorTests
{
    [Theory]
    [InlineData(0, 0, 0, true)]
    [InlineData(1, 1, 2, true)]
    [InlineData(1, 1, 3, false)]
    public void Add_PassMultipleParameters_VerifyExpected(
        int inputX, int inputY, int expected, bool isExpectedCorrect)
    {
        // Arrange
        var sut = new SimpleCalculator();

        // Act
        var actual = sut.Add(inputX, inputY);

        // Assert
```

```
        if (isExpectedCorrect)
        {
            Assert.Equal(expected, actual);
        }
        else
        {
            Assert.NotEqual(expected, actual);
        }
    }
}

public class SimpleCalculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Read Getting started with unit-testing online: https://riptutorial.com/unit-testing/topic/570/getting-started-with-unit-testing

Chapter 2: Assertion Types

Examples

Verifying a Returned Value

```
[Test]
public void Calculator_Add_ReturnsSumOfTwoNumbers()
{
    Calculator calculatorUnderTest = new Calculator();

    double result = calculatorUnderTest.Add(2, 3);

    Assert.AreEqual(5, result);
}
```

State Based Testing

Given this simple class, we can test that the `ShaveHead` method is working correctly by asserting state of the `HairLength` variable is set to zero after the `ShaveHead` method is called.

```
public class Person
{
    public string Name;
    public int HairLength;

    public Person(string name, int hairLength)
    {
        this.Name = name;
        this.HairLength = hairLength;
    }

    public void ShaveHead()
    {
        this.HairLength = 0;
    }
}

[Test]
public void Person_ShaveHead_SetsHairLengthToZero()
{
    Person personUnderTest = new Person("Danny", 10);

    personUnderTest.ShaveHead();

    int hairLength = personUnderTest.HairLength;

    Assert.AreEqual(0, hairLength);
}
```

Verifying an Exception is Thrown

Sometimes it is necessary to assert when an exception is thrown. Different unit testing frameworks

have different conventions for asserting that an exception was thrown, (like NUnit's Assert.Throws method). This example does not use any framework specific methods, just built in exception handling.

```
[Test]
public void GetItem_NegativeNumber_ThrowsArgumentInvalidException
{
    ShoppingCart shoppingCartUnderTest = new ShoppingCart();
    shoppingCartUnderTest.Add("apple");
    shoppingCartUnderTest.Add("banana");

    double invalidItemNumber = -7;

    bool exceptionThrown = false;

    try
    {
        shoppingCartUnderTest.GetItem(invalidItemNumber);
    }
    catch(ArgumentInvalidException e)
    {
        exceptionThrown = true;
    }

    Assert.True(exceptionThrown);
}
```

Read Assertion Types online: https://riptutorial.com/unit-testing/topic/6330/assertion-types

Chapter 3: Dependency Injection

Remarks

One approach that can be taken to writing software is to create dependencies as they are needed. This is quite an intuitive way to write a program and is the way that most people will tend to be taught, partly because it is easy to follow. One of the issues with this approach is that it can be hard to test. Consider a method that does some processing based on the current date. The method might contain some code like the following:

```
if (DateTime.Now.Date > processDate)
{
    // Do some processing
}
```

The code has a direct dependency on the current date. This method can be hard to test because the current date cannot be easily manipulated. One approach to making the code more testable is to remove the direct reference to the current date and instead supply (or inject) the current date to the method that does the processing. This dependency injection can make it much easier to test aspects of code by using test doubles to simplify the setup step of the unit test.

IOC systems

Another aspect to consider is the lifetime of dependencies; in the case where the class itself creates its own dependencies (also known as invariants), it is then responsible for disposing of them. Dependency Injection inverts this (and this is why we often refer to an injection library as an "Inversion of Control" system) and means that instead of the class being responsible for creating, managing and cleaning up its dependencies, an external agent (in this case, the IoC system) does it instead.

This makes it much simpler to have dependencies that are shared amongst instances of the same class; for example consider a service that fetches data from an HTTP endpoint for a class to consume. Since this service is stateless (i.e. it doesn't have any internal state) therefore we really only need a single instance of this service throughout our application. Whilst it is possible (for example, by using a static class) to do this manually, it's much simpler to create the class and tell the IoC system that it's to be created as a *Singleton*, whereby only one instance of the class every exists.

Another example would be database contexts used in a web application, whereby a new Context is required per request (or thread) and not per instance of a controller; this allows the context to be injected in every layer executed by that thread, without having to be manually passed around.

This frees the consuming classes from having to manage the dependencies.

Examples

Constructor Injection

Constructor injection is the safest way of injecting dependencies that a whole class depends upon. Such dependencies are often referred to as *invariants*, since an instance of the class cannot be created without supplying them. By requiring the dependency to be injected at construction, it is guaranteed that an object cannot be created in an inconsistent state.

Consider a class that needs to write to a log file in error conditions. It has a dependency on a `ILogger`, which can be injected and used when necessary.

```
public class RecordProcessor
{
    readonly private ILogger _logger;

    public RecordProcessor(ILogger logger)
    {
        _logger = logger;
    }

    public void DoSomeProcessing() {
        // ...
        _logger.Log("Complete");
    }
}
```

Sometimes while writing tests you may note that constructor requires more dependencies than it is actually needed for a case being tested. The more such tests you have the more likely it is that your class breaks *Single Responsibility Principle* (SRP). That is why it it is not a very good practice to define the default behavior for all mocks of injected dependencies at test class initialization phase as it can mask the potential warning signal.

The unittest for this would look like the following:

```
[Test]
public void RecordProcessor_DependencyInjectionExample()
{
    ILogger logger = new FakeLoggerImpl(); //or create a mock by a mocking Framework

    var sut = new RecordProcessor(logger); //initialize with fake impl in testcode

    Assert.IsTrue(logger.HasCalledExpectedMethod());
}
```

Property Injection

Property injection allows a classes dependencies to be updated after it has been created. This can be useful if you want to simplify object creation, but still allow the dependencies to be overridden by your tests with test doubles.

Consider a class that needs to write to a log file in an error condition. The class knows how to construct a default `Logger`, but allows it to be overridden through property injection. However it worths noting that using property injection this way you are tightly coupling this class with an exact

implementation of `ILogger` that is `ConcreteLogger` in this given example. A possible workaround could be a factory that returns the needed ILogger implementation.

```
public class RecordProcessor
{
    public RecordProcessor()
    {
        Logger = new ConcreteLogger();
    }

    public ILogger Logger { get; set; }

    public void DoSomeProcessing()
    {
        // ...
        _logger.Log("Complete");
    }
}
```

In most cases, Constructor Injection is preferable to Property Injection because it provides better guarantees about the state of the object immediately after its construction.

Method Injection

Method injection is a fine grained way of injecting dependencies into processing. Consider a method that does some processing based on the current date. The current date is hard to change from a test, so it is much easier to pass a date into the method that you want to test.

```
public void ProcessRecords(DateTime currentDate)
{
    foreach(var record in _records)
    {
        if (currentDate.Date > record.ProcessDate)
        {
            // Do some processing
        }
    }
}
```

Containers / DI Frameworks

Whilst extracting dependencies out of your code so that they can be injected makes your code easier to test, it pushes the problem further up the hierarchy and can also result in objects that are difficult to construct. Various dependency injection frameworks / Inversion of Control Containers have been written to help overcome this issue. These allow type mappings to be registered. These registrations are then used to resolve dependencies when the container is asked to construct an object.

Consider these classes:

```
public interface ILogger {
    void Log(string message);
}
```

```
public class ConcreteLogger : ILogger
{
    public ConcreteLogger()
    {
        // ...
    }
    public void Log(string message)
    {
        // ...
    }
}
public class SimpleClass
{
    public SimpleClass()
    {
        // ...
    }
}

public class SomeProcessor
{
    public SomeProcessor(ILogger logger, SimpleClass simpleClass)
    {
        // ...
    }
}
```

In order to construct `SomeProcessor`, both an instance of `ILogger` and `SimpleClass` are required. A container like Unity helps to automate this process.

First the container needs to be constructed and then mappings are registered with it. This is usually done only once within an application. The area of the system where this occurs is commonly known as the *Composition Root*

```
// Register the container
var container = new UnityContainer();

// Register a type mapping.  This allows a `SimpleClass` instance
// to be constructed whenever it is required.
container.RegisterType<SimpleClass, SimpleClass>();

// Register an instance.  This will use this instance of `ConcreteLogger`
// Whenever an `ILogger` is required.
container.RegisterInstance<ILogger>(new ConcreteLogger());
```

After the container is configured, it can be used to create objects, automatically resolving dependencies as required:

```
var processor = container.Resolve<SomeProcessor>();
```

Read Dependency Injection online: https://riptutorial.com/unit-testing/topic/597/dependency-injection

Chapter 4: Guide unit testing in Visual Studio for C#

Introduction

How to create unit test project and unit tests and how to run the unit tests and code coverage tool.

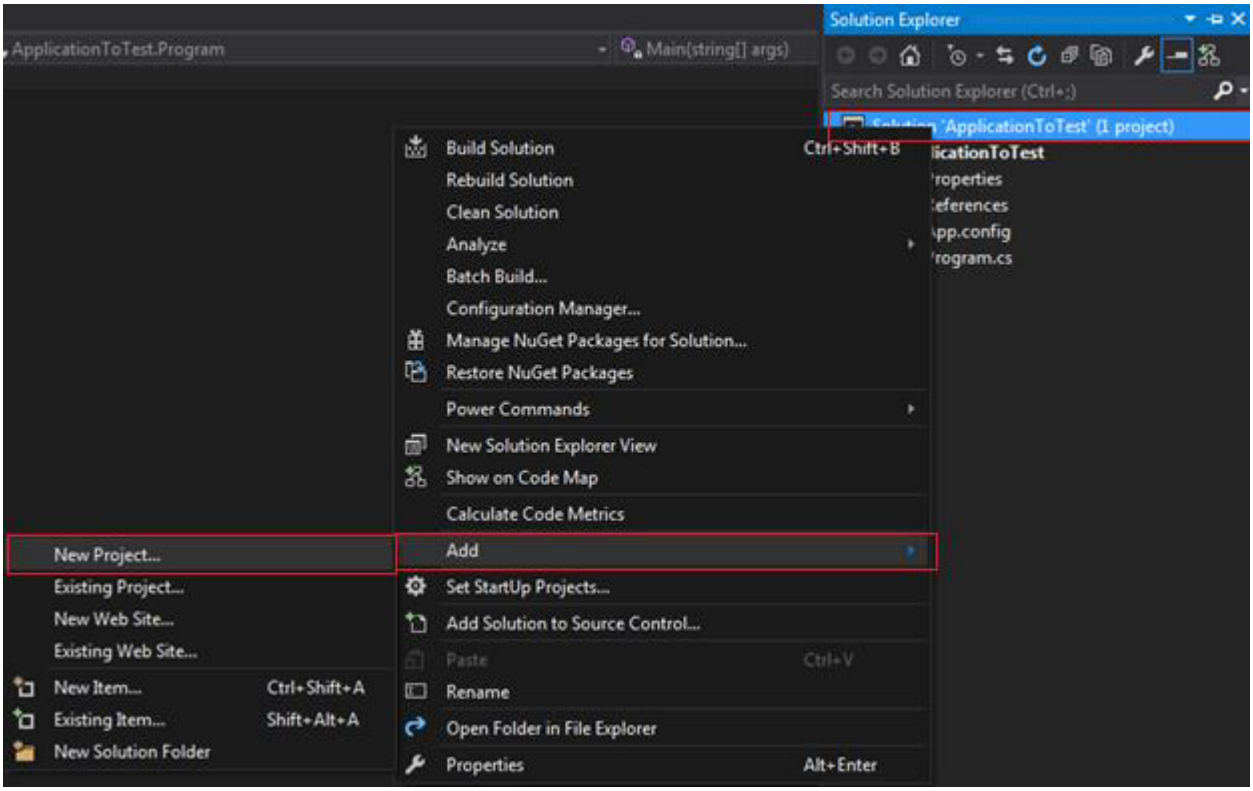
In this guide the standard MSTest framework will be used and the standard Code Coverage Analyses tool which are available in Visual Studio.

The guide was written for Visual Studio 2015, so it's possible some things are different in other versions.

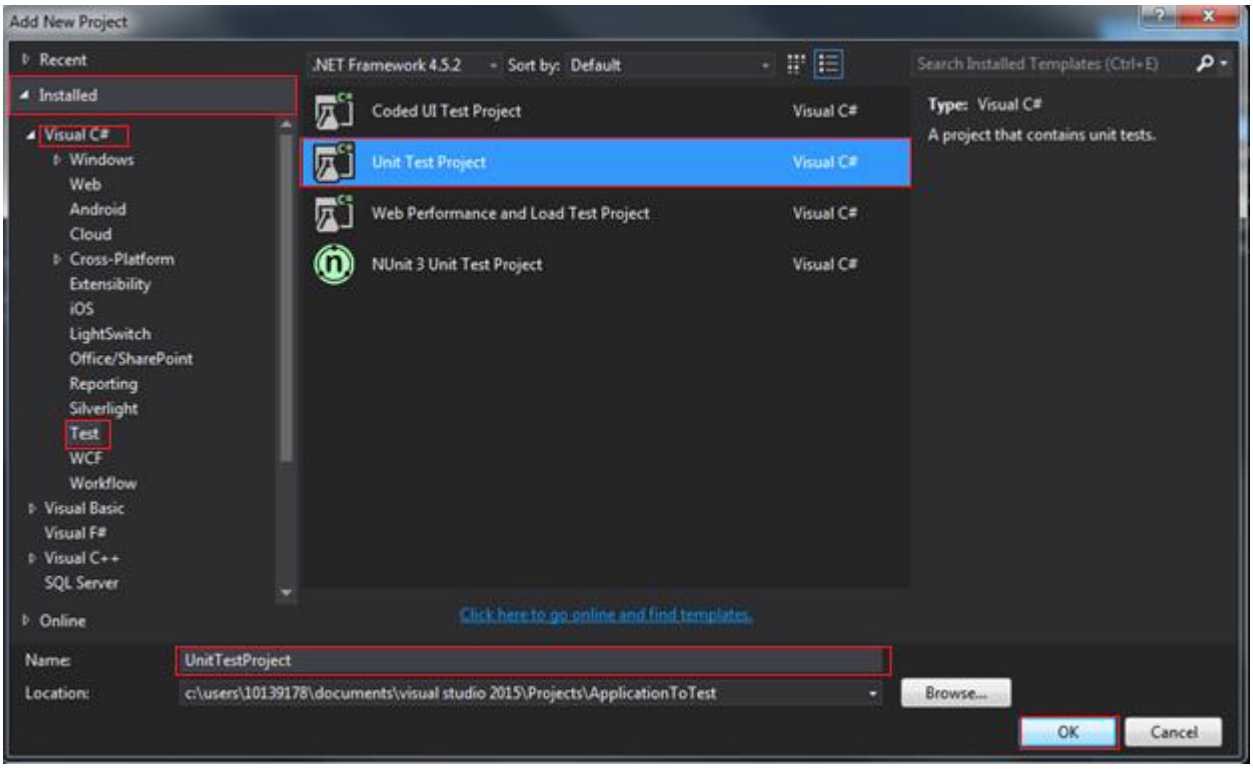
Examples

Creating a unit test project

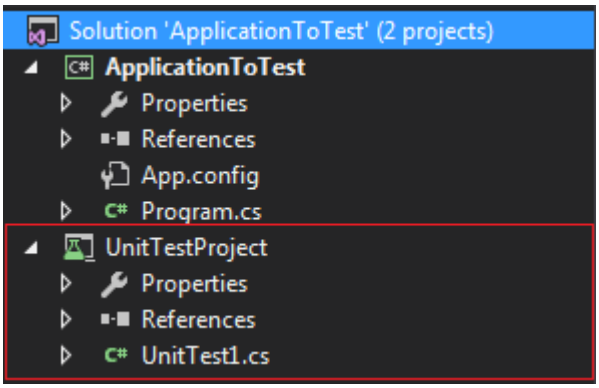
- Open the C# project
- Right-click on the solution -> Add -> New Project…
- (Figure 1)



- Go to Installed -> Visual C# -> Test
- Click on Unit Test Project
- Give it a name and click OK
- (Figure 2)

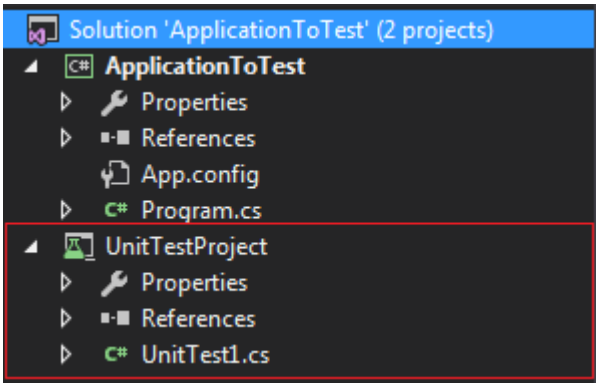


- The unit test project is added to the solution
- (Figure 3)

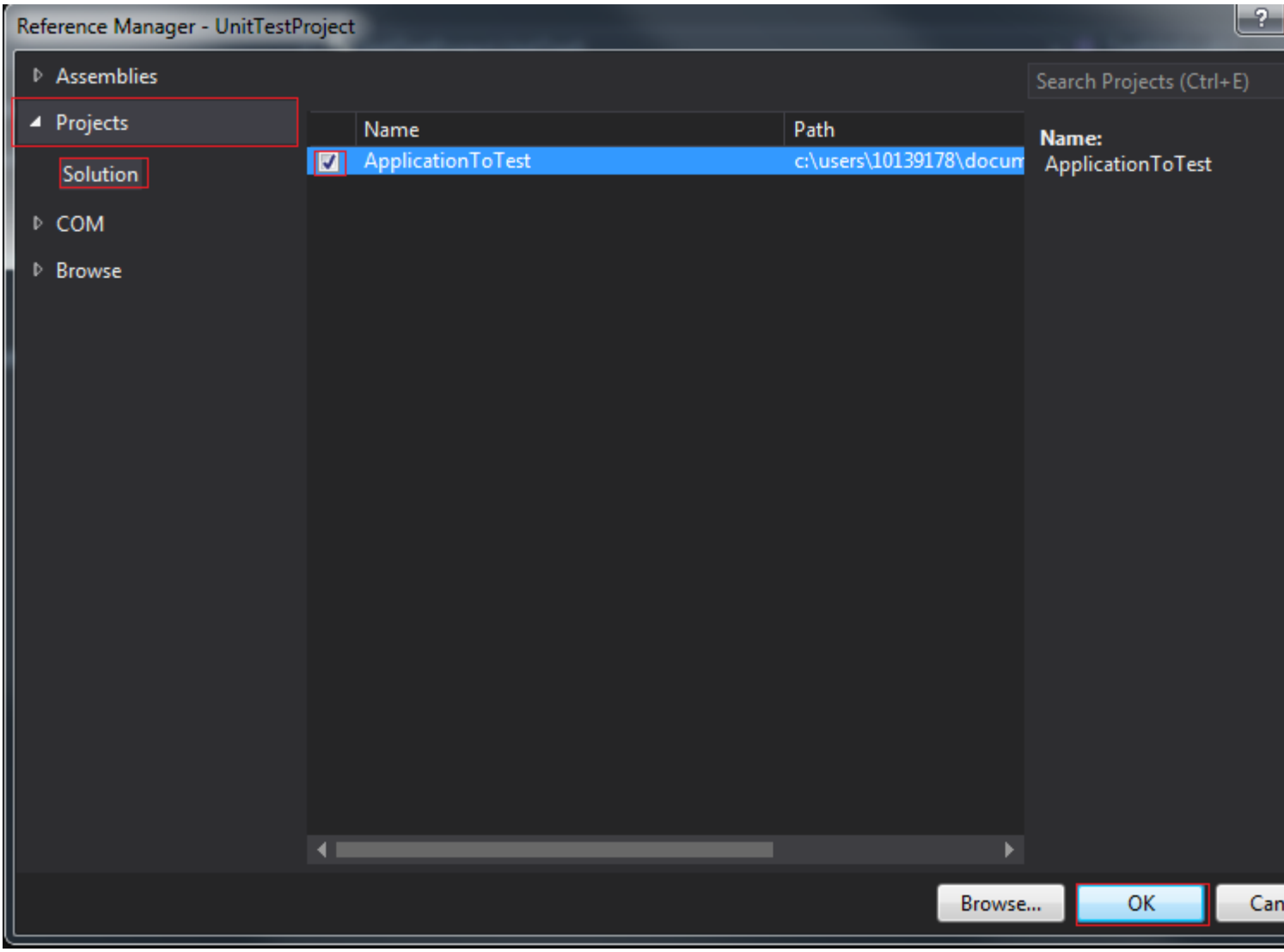


Adding the reference to the application you want to test

- In the unit test project, add a reference to the project you want to test
- Right-click on References -> Add Reference…
- (Figure 3)



- Select the project you want to test
- Go to Projects -> Solution
- Check the checkbox of the project you want to test -> click OK
- (Figure 4)



Two methods to create unit tests

Method 1

- Go to your unit test class in the unit test project
- Write a unit test

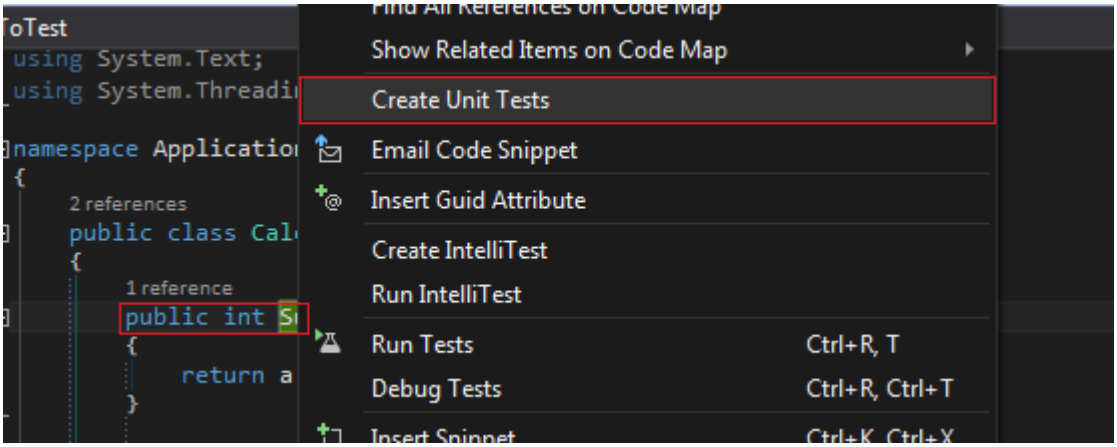
```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;
```

```
        //Act
        int result = ClassCalc.Sum(2,3);

        //Assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

Method 2

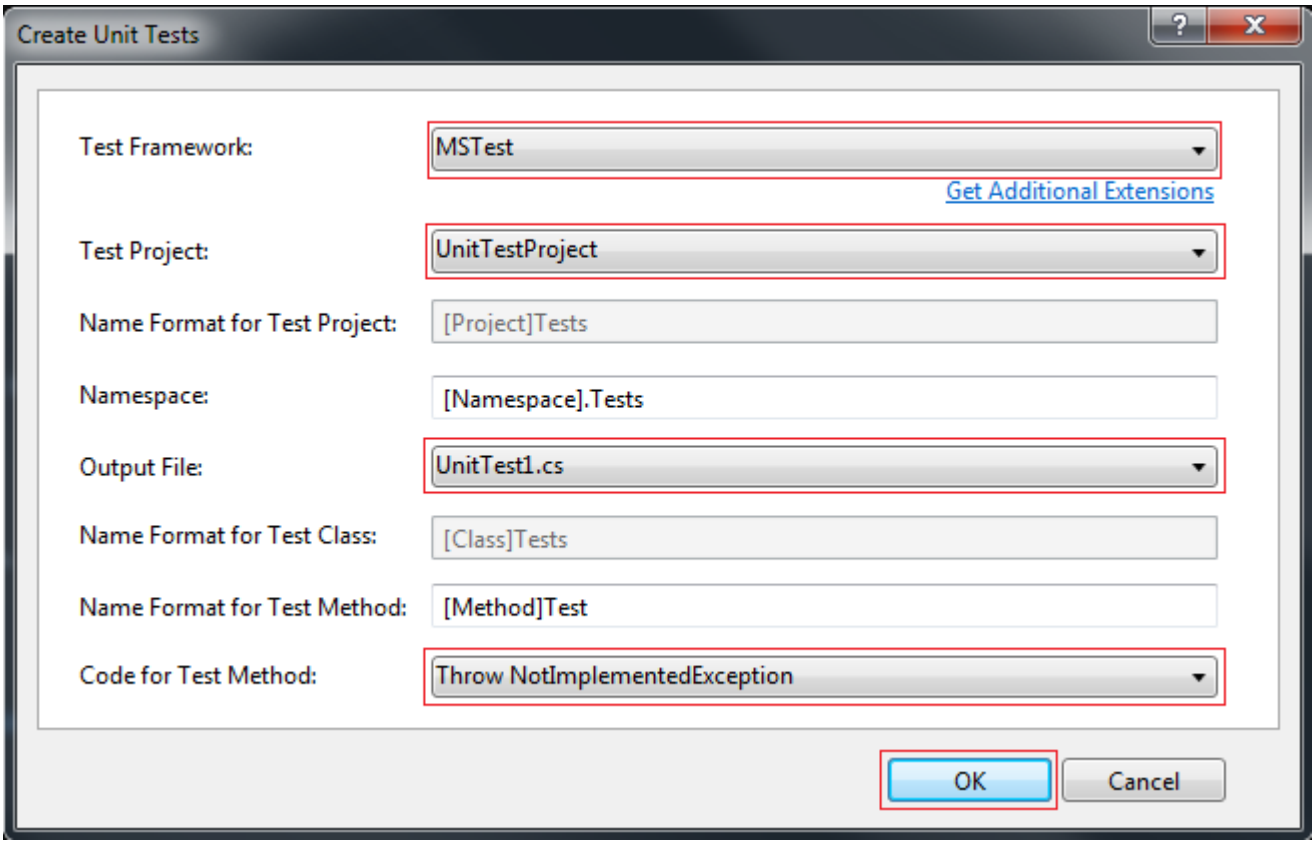
- Go the method you want to test
- Right-click on the method -> Create Unit Tests
- (Figure 4)



- Set Test Framework to MSTest
- Set Test Project to the name of your unit test project
- Set Output File to the name of the class of the unit tests
- Set Code for Test Method to one of the options listed which you prefer
- The other options can be edited but it’s not necessary

(Tip: If you haven’t made a unit tests project yet, you can still use this option. Just set Test Project to and Output File to . It will create the unit test project and it will add the reference of the project to the unit test project)

- (Figure 5)

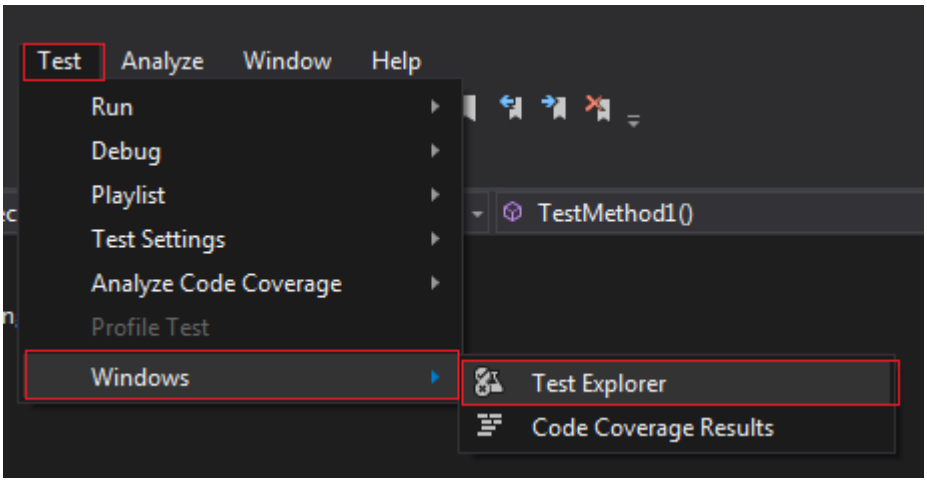


- As you see below it creates the base of the unit test for you to fill in
- (Figure 6)

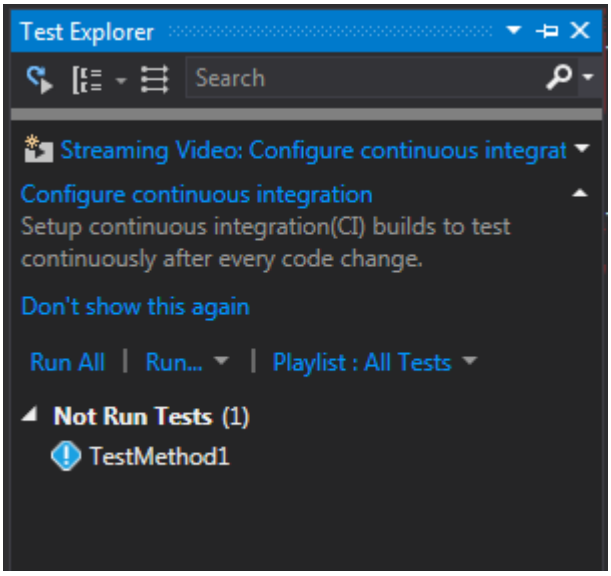
```
namespace ApplicationToTest.Tests
{
    [TestClass()]
    0 references
    public class UnitTest1
    {
        [TestMethod()]
        0 references
        public void SumTest()
        {
            throw new NotImplementedException();
        }
    }
}
```

Running unit tests within Visual Studio

- To see you unit tests go to Test -> Windows -> Test Explorer
- (Figure 1)



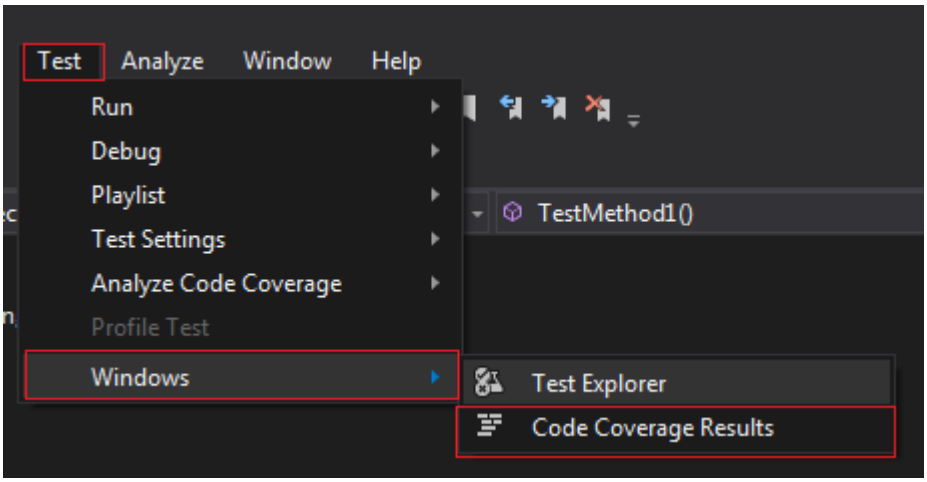
- This will open an overview of all the tests in the application
- (Figure 2)



- In the figure above you can see that the example has one unit test and it hasn't been run yet
- You can double-click on a test to go to the code where the unit test is defined
- You can run single or multiple tests with the Run All or Run…
- You can also run tests and change settings from the Test menu (Figure 1)

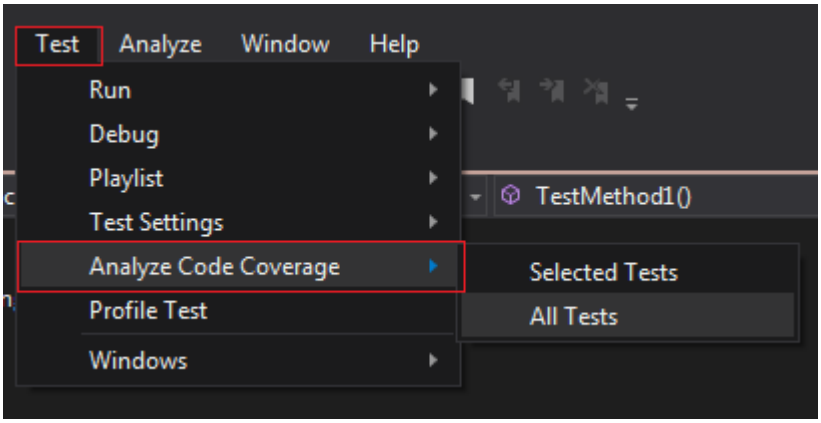
Running code coverage analysis within Visual Studio

- To see you unit tests go to Test -> Windows -> Code Coverage Results
- (Figure 1)

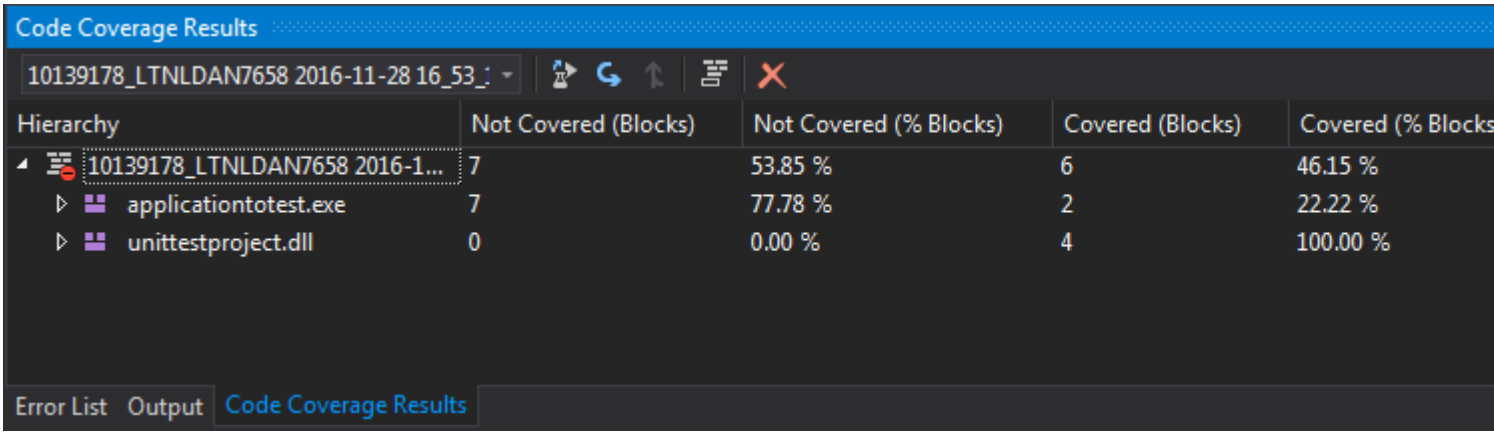


- It will open the following window
- (Figure 2)

- The window is now empty
- Go to the Test menu -> Analyze Code Coverage
- (Figure 3)



- The tests will now be run as well (See the results in the Test Explorer)
- The results will be shown in a table in with you can see which classes and methods are covered with unit tests and which aren't
- (Figure 4)



Read Guide unit testing in Visual Studio for C# online: https://riptutorial.com/unit-testing/topic/9953/guide-unit-testing-in-visual-studio-for-csharp

Chapter 5: Test Doubles

Remarks

When testing, it is sometimes useful to use a test double to manipulate or verify the behaviour of the system under test. The doubles are passed or injected into the class or method under test instead of instances of production code.

Examples

Using a stub to supply canned responses

A stub is a light weight test double that provides canned responses when methods are called. Where a class under test relies on an interface or base class an alternative 'stub' class can be implemented for testing which conforms to the interface.

So, assuming the following interface,

```
public interface IRecordProvider {
    IEnumerable<Record> GetRecords();
}
```

If the following method was to be tested

```
public bool ProcessRecord(IRecordProvider provider)
```

A stub class that implements the interface can be written to return known data to the method being tested.

```
public class RecordProviderStub : IRecordProvider
{
    public IEnumerable<Record> GetRecords()
    {
        return new List<Record> {
            new Record { Id = 1, Flag=false, Value="First" },
            new Record { Id = 2, Flag=true, Value="Second" },
            new Record { Id = 3, Flag=false, Value="Third" }
        };
    }
}
```

This stub implementation can then be provided to the system under test, to influence it's behaviour.

```
var stub = new RecordProviderStub();
var processed = sut.ProcessRecord(stub);
```

Using a mocking framework as a stub

The terms Mock and Stub can often become confused. Part of the reason for this is that many mocking frameworks also provide support for creating Stubs without the verification step associated with Mocking.

Rather than writing a new class to implement a stub as in the "Using a stub to supply canned responses" example, mocking frameworks can be used instead.

Using Moq:

```
var stub = new Mock<IRecordProvider>();
stub.Setup(provider => provider.GetRecords()).Returns(new List<Record> {
    new Record { Id = 1, Flag=false, Value="First" },
    new Record { Id = 2, Flag=true, Value="Second" },
    new Record { Id = 3, Flag=false, Value="Third" }
});
```

This achieves the same behaviour as the hand coded stub, and can be supplied to the system under test in a similar way:

```
var processed = sut.ProcessRecord(stub.Object);
```

Using a mocking framework to validate behaviour

Mocks are used when it is necessary to verify the interactions between the system under test and test doubles. Care needs to be taken to avoid creating overly brittle tests, but mocking can be particularly useful when the method to test is simply orchestrating other calls.

This test verifies that when the method under test is called (`ProcessRecord`), that the service method (`UseValue`) is called for the `Record` where `Flag==true`. To do this, it sets up a stub with canned data:

```
var stub = new Mock<IRecordProvider>();
stub.Setup(provider => provider.GetRecords()).Returns(new List<Record> {
    new Record { Id = 1, Flag=false, Value="First" },
    new Record { Id = 2, Flag=true, Value="Second" },
    new Record { Id = 3, Flag=false, Value="Third" }
});
```

Then it sets up a mock which implements the `IService` interface:

```
var mockService = new Mock<IService>();
mockService.Setup(service => service.UseValue(It.IsAny<string>())).Returns(true);
```

These are then supplied to the system under test and the method to be tested is called.

```
var sut = new SystemUnderTest(mockService.Object);

var processed = sut.ProcessRecord(stub.Object);
```

The mock can then be interrogated to verify that the expected call has been made to it. In this case, a call to `UseValue`, with one parameter "Second", which is the value from the record where `Flag==true`.

```
mockService.Verify(service => service.UseValue("Second"));
```

Read Test Doubles online: https://riptutorial.com/unit-testing/topic/615/test-doubles

Chapter 6: The general rules for unit testing for all languages

Introduction

When starting with unit-testing all kinds of questions come up:

What is unit-testing? What is a SetUp and TearDown? How do I deal with dependencies? Why do unit-testing at all? How do I make good unit tests?

This article will answer all these questions, so you can start unit-testing in any language you want.

Remarks

What is unit testing?

Unit testing is the testing of code to ensure that it performs the task that it is meant to perform. It tests code at the very lowest level possible - the individual methods of your classes.

What is a unit?

Any discrete module of code that can be tested in isolation. Most of the time classes and their methods. This class is generally referred to as the "Class Under Test" (CUT) or the "System Under Test" (SUT)

The difference between unit testing and integration testing

Unit testing is the act of testing a single class in isolation, completely apart from any of its actually dependencies. Integration testing is the act of testing a single class along with one or more of its actual dependencies.

The SetUp and TearDown

When made the SetUp method is run before every unit test and the TearDown after every test.

In general you add all prerequisite steps in the SetUp and all the clean-up steps in the TearDown. But you only make these method if these steps are needed for every test. If not, than these steps are taken within the specific tests in the "arrange" section.

How to deal with dependencies

Many times a class has the dependency of other classes to execute its methods. To be able to not

depend on these other classes, you have to fake these. You can make these classes yourself or use an isolation or mockup framework. An isolation framework is a collection of code that enables the easy creation of fake classes.

Fake classes

Any class that provides functionality sufficient to pretend that it is a dependency needed by a CUT. There are two types of fakes: Stubs and Mocks.

- A stub: A fake that has no effect on the passing or failing of the test and that exists purely to allow the test to run.
- A mock: A fake that keeps track of the behavior of the CUT and passes or fails the test based on that behavior.

Why do unit testing?

1. Unit testing will find bugs

When you write a full suite of tests that define what the expected behavior is for a given class, anything that isn't behaving as expected is revealed.

2. Unit testing will keep bugs away

Make a change that introduces a bug and your tests can reveal it the very next time you run your tests.

3. Unit testing saves time

Writing unit tests helps ensure that your code is working as designed right from the start. Unit tests define what your code should do and thus you won't be spending time writing code that does things it shouldn’t do. No one checks in code that they don't believe works and you have to do something to make yourself think that it works. Spend that time to write unit tests.

4. Unit testing gives peace of mind

You can run all those tests and know that your code works as it is supposed to. Knowing the state of your code, that it works, and that you can update and improve it without fear is a very good thing.

5. Unit testing documents the proper use of a class

Unit tests become simple examples of how your code works, what it is expected to do and the proper way to use your code being tested.

General rules for unit testing

1. For the structure of a unit test, follow the AAA rule

Arrange:

Set up thing to be tested. Like variables, fields and properties to enable the test to be run as well as the expected result.

Act: Actually call the method you're testing

Assert:

Call the testing framework to verify that the result of your "act" is what was expected.

2. Test one thing at the time in isolation

All classes should be tested in isolation. They shouldn’t depend on anything other than the mocks and stubs. They shouldn’t depend on the results of other tests.

3. Write simple "right down the middle" tests first

The first tests you write should be the simplest tests. They should be the ones that basically and easily illustrate the functionality you are trying to write. Then, once those tests pass, you should start write the more complicated tests that test the edges and boundaries of your code.

4. Write tests that test the edges

Once the basics are tested and you know your basic functionality works, you should test the edges. A good set of tests will explore the outer edges of what might happen to a given method.

For example:

- What happens if an overflow occurs?
- What if values go to zero or below?
- What if they go to MaxInt or MinInt?
- What if you create an arc of 361 degrees?
- What happens if you pass an empty string?
- What happens if a string is 2GB in size?

5. Test across boundaries

Unit tests should test both sides of a given boundary. Moving across boundaries are places where your code might fail or perform in unpredictable ways.

6. If you can, test the entire spectrum

If it's practical, test the entire set of possibilities of for your functionality. If it involves an enumerated type, test the functionality with every one of the items in the enumeration. It might be impractical to test every possibility, but if you can test every possibility, do it.

7. If possible, cover every code path

This one is challenging as well, but if your code is designed for testing, and you make use of a code coverage tool, you can ensure that every line of your code is covered by unit tests at least once. Covering every code path won't guarantee that there aren't any bugs, but it surely gives you valuable information about the state of every line of code.

8. Write tests that reveal a bug, then fix it

If you find a bug, write a test that reveals it. Then, you van easily fix the bug by debugging the test. Then you have a nice regression test to make sure that if the bug comes back for any reason, you'll know right away. It's really easy to fix a bug when you have a simple, straight forward test to run in the debugger.

A side benefit here is that you've tested your test. Because you've seen the test fail and then when you have seen it pass, you know that the test is valid in that it has been proven to work correctly. This makes it an even better regression test.

9. Make each test independent of each other

Tests should never depend on each other. If your tests have to run in a certain order, you need to change the tests.

10. Write one assert per test

You should write one assert per test. If you can't do that, then refractor your code so your SetUp and TearDown events are used to correctly create the environment so that each test can be run individually.

11. Name your tests clearly. Don't be afraid of long names

Since you're doing one assert per test, each test can end up being very specific. Thus, don't be afraid to use long, complete test names.

A long complete name lets you know immediately what test failed and exactly what the test was trying to do.

Long, clearly named tests can also document your tests. A test named "DividedByZeroShouldThrowException" documents exactly what the code does when you try to divide by zero.

12. Test that every raised exception is actually raised

If your code raises an exception, then write a test to ensure that every exception you raise in fact gets raised when it is supposed to.

13. Avoid the use of CheckTrue or Assert.IsTrue

Avoid checking for a Boolean condition. For instance, instead if checking if two things are equal with CheckTrue or Assert.IsTrue, use CheckEquals or Assert.IsEqual instead. Why? Because of this:

CheckTrue (Expected, Actual) This will report something like: "Some test failed: Expected was True but actual result was False."

This doesn’t tell you anything.

CheckEquals (Expected, Actual)

This will tell you something like: "Some test failed: Expected 7 but actual result was 3."

Only use CheckTrue or Assert.IsTrue when your expected value is actually a Boolean condition.

14. Constantly run your tests

Run your tests while you are writing code. Your tests should run fast, enabling you to run them after even minor changes. If you can’t run your tests as part of your normal development process then something is going wrong. Unit tests are supposed to run almost instantly. If they aren't, it's probably because you aren’t running them in isolation.

15. Run your tests as part of every automated build

Just as you should be running test while you develop, they should also be an integral part of your continuous integration process. A failed test should mean that your build is broken. Don’t let failing tests linger. Consider it a build failure and fix it immediately.

Examples

Example of simple unit test in C#

For this example we will test the sum method of a simple calculator.

In this example we will test the application: ApplicationToTest. This one has a class called Calc. This class has a method Sum().

The method Sum() looks like this:

```
public void Sum(int a, int b)
{
    return a + b;
}
```

The unit test to test this method looks like this:

```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
```

```
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;

        //Act
        int result = ClassCalc.Sum(2,3);

        //Assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

Read The general rules for unit testing for all languages online: https://riptutorial.com/unit-testing/topic/9947/the-general-rules-for-unit-testing-for-all-languages

Chapter 7: Unit testing of Loops (Java)

Introduction

Loops considered as one of the important control structures in any programming language. There are different ways in which we can achieve loop coverage.

These methods differ based on type of loop.

Single loops

Nested Loops

Concatenated loops

Examples

Single loop test

These are loops in which their loop body contains no other loops (the innermost loop in case of nested).

In order to have loop coverage, testers should exercise the tests given below.

Test 1 :Design a test in which loop body shouldn’t execute at all (i.e. zero iterations)

Test 2 :Design a test in which loop–control variable be negative (Negative number of iterations)

Test 3 :Design a test in which loop iterates only once

Test 4 :Design a test in which loop iterates twice

Test 5 :Design a test in which loop iterates certain number of times , say m where m < maximum number of iterations possible

Test 6 :Design a test in which loop iterates one less than the maximum number of iterations

Test 7 :Design a test in which loop iterates the maximum number of iterations

Test 8 :Design a test in which loop iterates one more than the maximum number of iterations

Consider the below code example which applies all the conditions specified.

public class SimpleLoopTest {

private int[] numbers = {5,-77,8,-11,4,1,-20,6,2,10};

```
/** Compute total of  positive numbers in the array
```

```
 *  @param numItems number of items to total.
 */
public int findSum(int numItems)
{
    int total = 0;
    if (numItems <= 10)
    {
        for (int count=0; count < numItems; count = count + 1)
        {
          if (numbers[count] > 0)
             {
                total = total + numbers[count];
             }
        }
    }
    return total;
}
```

}

public class TestPass extends TestCase {

```
public void testname() throws Exception {

    SimpleLoopTest s = new SimpleLoopTest();
    assertEquals(0, s.findSum(0));    //Test 1
    assertEquals(0, s.findSum(-1));   //Test 2
    assertEquals(5, s.findSum(1));    //Test 3
    assertEquals(5, s.findSum(2));    //Test 4
    assertEquals(17, s.findSum(5));   //Test 5
    assertEquals(26, s.findSum(9));   //Test 6
    assertEquals(36, s.findSum(10));  //Test 7
    assertEquals(0, s.findSum(11));   //Test 8
}
```

}

Nested Loops Test

A nested loop is a loop within a loop.

The outer loop changes only after the inner loop is completely finished / interrupted.

In this case, test cases should be designed in such a way that

Start at the innermost loop. Set all the outer loops to their minimum values. Perform Simple loop testing on the innermost loop (Test3 / Test4 / Test5 / Test6 / Test7). Continue till all the loops tested

Concatenated loops Test

Two loops are concatenated if it’s possible to reach one after exiting the other on same path from entrance to exit. Sometimes these two loops are independent to each other. In those cases we can apply the design techniques specified as part of single loop testing.

But if the iteration values in one loop are directly or indirectly related to the iteration values of another loop and they can occur on the same path, then we can consider them as nested loops.

Read Unit testing of Loops (Java) online: https://riptutorial.com/unit-testing/topic/10116/unit-testing-of-loops--java-

Chapter 8: Unit Testing: Best Practices

Introduction

A unit test is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. Unit tests are basically written and executed by software developers to make sure that code meets its design and requirements and behaves as expected.

Examples

Good Naming

The importance of good naming, can be best illustrated by some bad examples:

```
[Test]
Test1() {...} //Cryptic name - absolutely no information

[Test]
TestFoo() {...} //Name of the function - and where can I find the expected behaviour?

[Test]
TestTFSid567843() {...} //Huh? You want me to lookup the context in the database?
```

Good tests need goodn names. Good test do not test methods, the test scenarios or requirements.

Good naming also provides information about context and expected behaviour. Ideally, when the test fails on your build machine, you should be able to decide what is wrong, without looking at the test code, or even harder, having the necessity to debug it.

Good naming spares you time for reading code and debugging:

```
[Test]
public void GetOption_WithUnkownOption_ReturnsEmptyString() {...}
[Test]
public void GetOption_WithUnknownEmptyOption_ReturnsEmptyString() {...}
```

For beginners it may be helpful to start the test name with **EnsureThat_** or similar prefix. Start with an "EnsureThat_" helps to begin thinking about the scenario or requirement, that needs a test:

```
[Test]
public void EnsureThat_GetOption_WithUnkownOption_ReturnsEmptyString() {...}
[Test]
public void EnsureThat_GetOption_WithUnknownEmptyOption_ReturnsEmptyString() {...}
```

Naming is important for test fixtures too. Name the test fixture after the class being tested:

```
[TestFixture]
```

```
public class OptionsTests //tests for class Options
{
    ...
}
```

The final conclusion is:

Good naming leads to good tests which leads to good design in production code.

From simple to complex

Same as, with writing classes - start with the simple cases, then add requirement (aka tests) and implementation (aka production code) case by case:

```
[Test]
public void EnsureThat_IsLeapYearIfDecimalMultipleOf4() {...}
[Test]
public void EnsureThat_IsNOTLeapYearIfDecimalMultipleOf100 {...}
[Test]
public void EnsureThat_IsLeapYearIfDecimalMultipleOf400 {...}
```

Don't forget the refactoring step, when finished with requirements - first refactor the code, then refactor the tests

When finished, you should have a complete, up to date and READABLE documentation of your class.

MakeSut concept

Testcode has the same quality demands, as production code. MakeSut()

- improves readability
- can be easily refactored
- perfectly supports dependency injection.

Here's the concept:

```
[Test]
public void TestSomething()
{
    var sut = MakeSut();

    string result = sut.Do();
    Assert.AreEqual("expected result", result);
}
```

The simplest MakeSut() just returns the tested class:

```
private ClassUnderTest MakeSUT()
{
    return new ClassUnderTest();
}
```

When dependencies are needed, they can be injected here:

```
private ScriptHandler MakeSut(ICompiler compiler = null, ILogger logger = null, string
scriptName="", string[] args = null)
{
    //default dependencies can be created here
    logger = logger ?? MockRepository.GenerateStub<ILogger>();
    ...
}
```

One might say, that MakeSut is just a simple alternative for setup and teardown methods provided by Testrunner frameworks and might cosider these methods a better place for test specific setup and teardown.

Everybody can decide on her own, which way to use. For me MakeSut() provides better readability and much more flexibility. Last but not least, the concept is independent from any testrunner framework.

Read Unit Testing: Best Practices online: https://riptutorial.com/unit-testing/topic/6074/unit-testing--best-practices

Credits

S. No	Chapters	Contributors
1	Getting started with unit-testing	Andrey, Carl Manaster, Community, Farukh, forsvarir, Fred Kleuver, mahei, mark_h, Quill, silver, Stephen Byrne, Thomas Weller, zhon
2	Assertion Types	Danny
3	Dependency Injection	forsvarir, kayess, mrAtari, Pavel Voronin, Stephen Byrne
4	Guide unit testing in Visual Studio for C#	DarkAngel
5	Test Doubles	forsvarir
6	The general rules for unit testing for all languages	DarkAngel
7	Unit testing of Loops (Java)	Remya
8	Unit Testing: Best Practices	mrAtari, RamenChef, Shrinivas Patgar, user2314737