

# 單元測試簡介

## 一、參考

1. 簡單介紹：HiSKIO 程式語言線上教學
  - [#1 什麼是單元測試 & 整合測試 & 端對端測試？](#)
  - [#2 單元測試：用 Nunit 寫 Test](#)
  - [#3 單元測試：用 MSTest 寫 Test](#)
  - [#4 單元測試：自動生成 MSTest](#)
  - [#5 單元測試：如何 Debug Unit Test](#)
  - [#6 單元測試：使用 NUnit 重構](#)若連結失效再看[參考教學影片](#) 資料夾的影片
2. 詳細介紹及認識 TDD：[30 天快速上手 TDD](#)

## 二、單元測試介紹

1. 軟體測試最小單位為單元測試，單元測試最小測試單位即為一個方法
2. 利用 3A 以及 Fake Object 來完成單元測試
3. 單元測試通常會有一個正向測試(正常執行方法流程)以及逆向測試(測試例外或者特殊流程)
4. TDD 概念：開發功能前先寫單元測試
5. 私有方法(Partive)不需做單元測試
6. 如果很難做單元測試(單元測試寫得太複雜)，則需思考是否是物件方法寫得不夠好(相依太多、邏輯太過複雜等)，而考慮要不要做重構

## 三、3A(Arrange、Act、Assert)介紹

1. **Arrange**(安排)：宣告傳入的參數測試值、預期值以及定義要 mock 的物件或方法。
2. **Act**(測試動作)：要測試執行的方法。
3. **Assert**(宣稱)：檢查實際測試的輸出與預期結果是否相同。

## 四、Fake Object(假物件)介紹

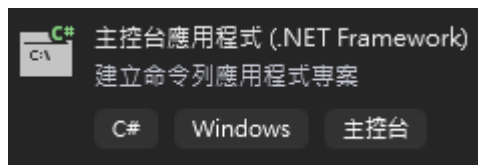
1. 單元測試最小測試單位即為一個方法，但測試的方法可能有一些外在因素(如專案外在環境設定、DB 操作、相依其他物件等)都會影響單元測試，因此我們需要將這些外在因素都用假物件代替，這樣才能達到單元測試的可測試性。
2. 假物件有很多種，常見的有 Mock、Stub、Fake，各有其使用方式及差別(介紹略)。

## 五、常見單元測試 Nuget 套件介紹

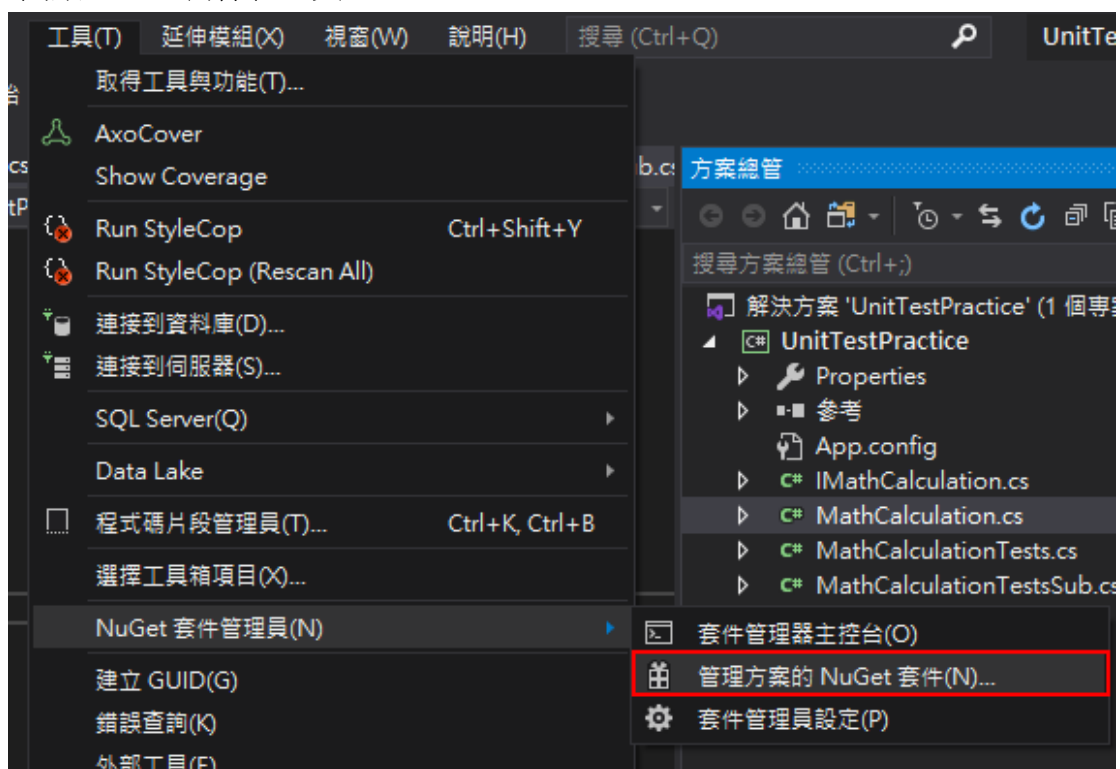
1. MSTest：微軟提供的套件
2. NUnit：第三方套件，其用法和 MSTest 大同小異，只有一些語法差異
3. Xunit：第三方套件

## 六、單元測試範例一(單一物件單一方法)

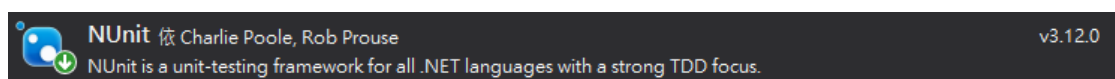
1. 新建專案
  - I. 建立一個主控台應用程式(C#)專案



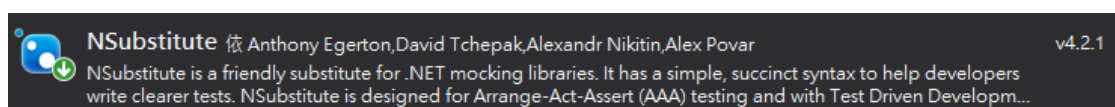
2. 單元測試套件安裝
  - I. 開啟 NuGet 套件管理員



- II. 安裝 NUnit 套件(單元測試用)



- III. 安裝 Nsubstitute 套件(Mock 用)



#### IV. 安裝 NUnit 3 Test Adapter(執行單元測試用)



##### NUnit 3 Test Adapter

NUnit 3 adapter for running tests in Visual Studio. Works with NUnit 3.x.

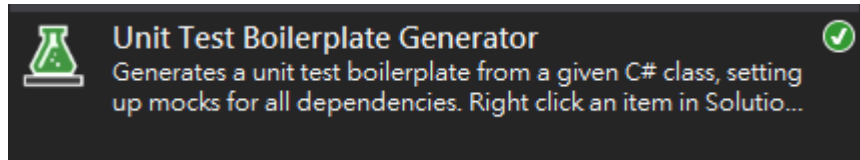
#### 3. 建立一個數學運算類別並添加一些方法

```
namespace UnitTestPractice
{
    /// <summary>
    /// 數學運算
    /// </summary>
    public class MathCalculation
    {
        /// <summary>
        /// 加法
        /// </summary>
        /// <param name="num1">值一</param>
        /// <param name="num2">值二</param>
        /// <returns></returns>
        public double Add(double num1, double num2)
        {
            return num1 + num2;
        }

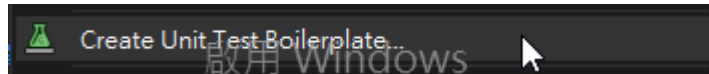
        /// <summary>
        /// 乘法(私有方法)
        /// </summary>
        /// <param name="num1">值一</param>
        /// <param name="num2">值二</param>
        /// <returns></returns>
        private double Multi(double num1, double num2)
        {
            return num1 * num2;
        }
    }
}
```

#### 4. 建立該 class 的單元測試(若先寫完開發可以用下列方法快速建立單元測試，也可自行從頭撰寫)

##### I. 先安裝 Unit Test Boilerplate Generator 套件

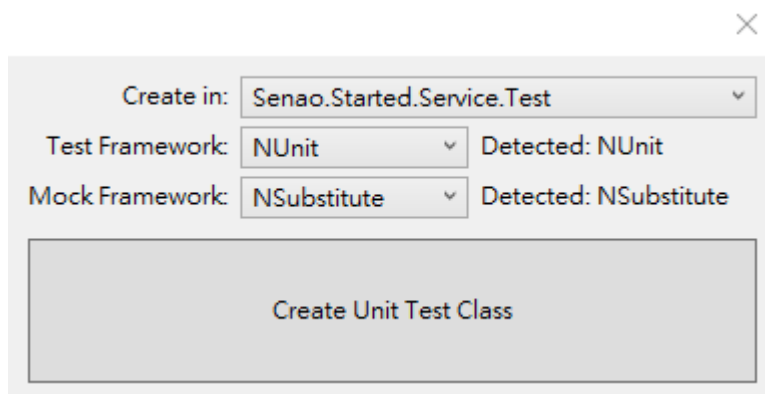


II. 對要建單元測試的 Service 按右鍵>Create Unit Test Boilerplate



III. 選擇建立的地方、單元測試套件、Mock 方式

注意這裡的套件選擇會影響單元測試撰寫的語法，故使用前先去了解各套件用法，此處的單元測試選擇 **NUnit** 套件、Mock 方式選擇 **Nsubstitute** 套件



IV. 按下 Create Unit Class 後就會生成一個單元測試檔

其中測試的類別應加上[TestFixture]屬性、測試的方法加上[Test]、設定初始值的方法加上[SetUp]，其他屬性介紹參考 NUnit 套件介紹，類別私有方法(如數學運算的乘法)不用做單元測試，故不會產生私有方法的單元測試

```
using NSubstitute;
using NUnit.Framework;
using System;
using UnitTestPractice;

namespace UnitTestPractice
{
    [TestFixture]
    public class MathCalculationTests
    {
        [SetUp]
        public void Setup()
        {

        }
    }
}
```

```

private MathCalculation CreateMathCalculation()
{
    return new MathCalculation();
}

[Test]
public void Add_StateUnderTest_ExpectedBehavior()
{
    // Arrange
    var mathCalculation = this.CreateMathCalculation();
    double num1 = 0;
    double num2 = 0;

    // Act
    var result = mathCalculation.Add(
        num1,
        num2);

    // Assert
    Assert.Fail();
}
}
}

```

#### V. 修改單元測試內容

```

using NUnit.Framework;

namespace UnitTestPractice
{
    /// <summary>
    /// 數學運算單元測試
    /// </summary>
    [TestFixture]
    public class MathCalculationTests
    {
        /// <summary>
        /// 初始化(在測試方法前所執行的處理)
        /// </summary>

```

```

[SetUp]
public void SetUp()
{
}

/// <summary>
/// 建立數學運算物件
/// </summary>
/// <returns>數學運算實體化物件</returns>
private MathCalculation CreateMathCalculation()
{
    return new MathCalculation();
}

/// <summary>
/// Add_兩數相加_取得兩數相加的結果
/// </summary>
[Test]
public void Add_兩數相加_取得兩數相加的結果()
{
    // Arrange
    var mathCalculation = this.CreateMathCalculation(); //取得數學運
算類別物件

    double num1 = 3;    //預設傳入值1
    double num2 = 7;    //預設傳入值2
    double expectedNum = 10;    //預期結果

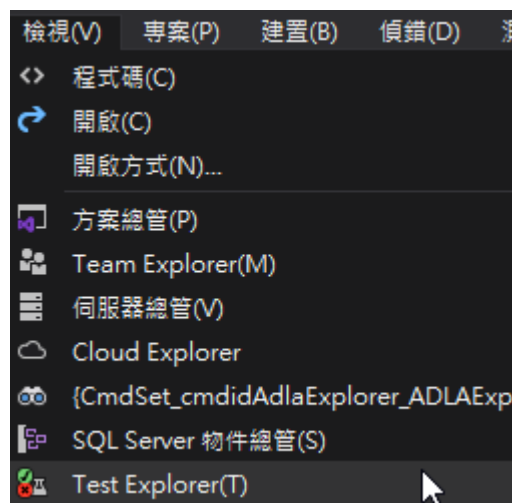
    // Act
    var result = mathCalculation.Add(num1, num2);    //測試數學運算
的加法並取得結果



    // Assert
    Assert.AreEqual(expectedNum, result);    //比對預期值和實際執
行值是否相同，若不同則拋出例外
}
}
}


```

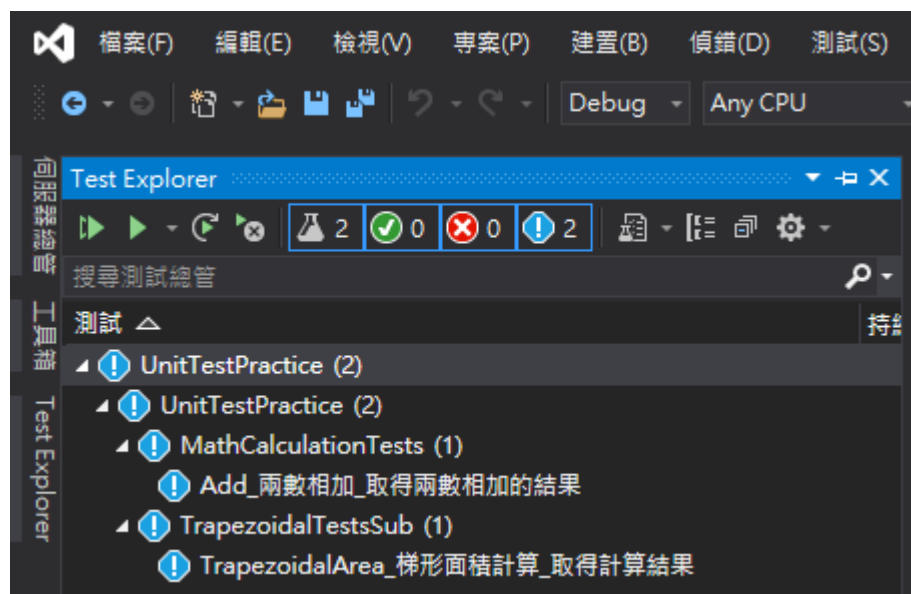
## 5. 執行單元測試

### I. 使用 Test Explorer(單元測試工具)

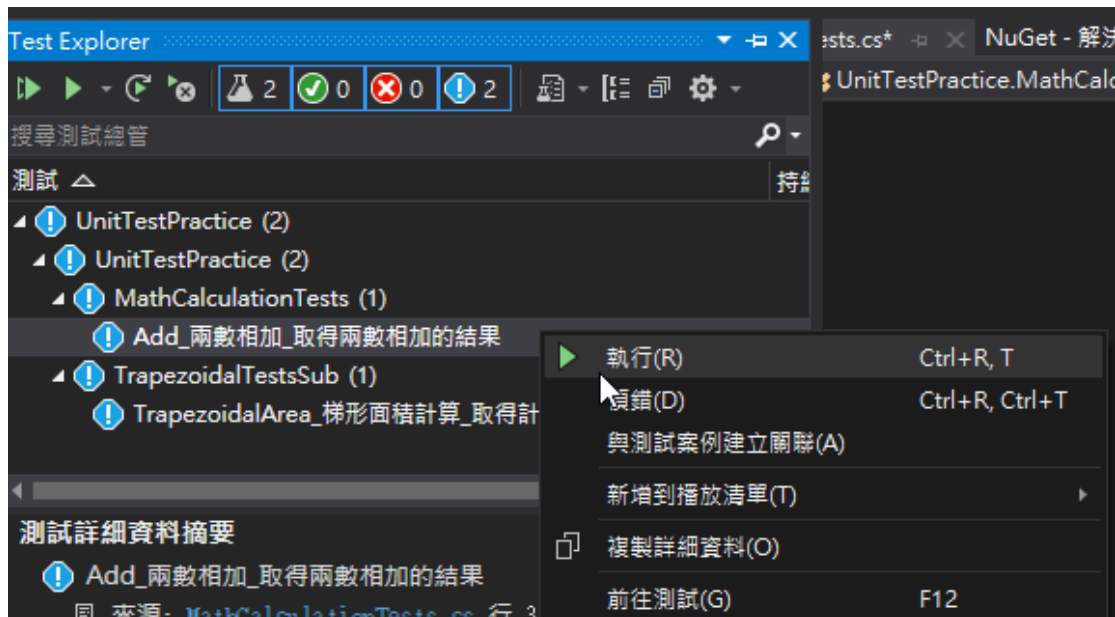


II. 未測前的符號為，請先重建專案後再點選左上角的執行單元測試

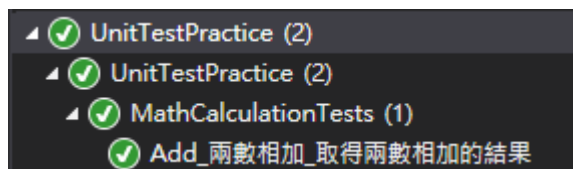
注意：若一直為，則檢查是否有安裝 NUnit 3 Test Adapter 套件



III. 對要做單元測試的方法按右鍵>執行



IV. 成功就會顯示 ，失敗則會顯示  並在底下顯示失敗訊息



## 七、 單元測試範例二(相依物件、單一方法內包含其他方法)

1. 先新建專案、裝套件，同範例一前兩步驟
2. 建立數學運算介面並添加一些方法，另外讓新增數學運算類別實做該介面
  - I. 數學運算介面：

```
namespace UnitTestPractice
{
    public interface IMathCalculation
    {
        /// <summary>
        /// 加法
        /// </summary>
        /// <param name="num1">值一</param>
        /// <param name="num2">值二</param>
        /// <returns></returns>
        double Add(double num1, double num2);
    }
}
```



```

    /// <summary>
    /// 乘法
    /// </summary>
    /// <param name="num1">值一</param>
    /// <param name="num2">值二</param>
    /// <returns></returns>
    double Multi(double num1, double num2);
}
}

```

## II. 數學運算類別：

```

namespace UnitTestPractice
{
    /// <summary>
    /// 數學運算
    /// </summary>
    public class MathCalculation : IMathCalculation
    {
        private readonly IMathCalculation _mathCalculation;
        public MathCalculation(IMathCalculation mathCalculation)
        {
            this._mathCalculation = mathCalculation;
        }

        /// <summary>
        /// 加法
        /// </summary>
        /// <param name="num1">值一</param>
        /// <param name="num2">值二</param>
        /// <returns></returns>
        public double Add(double num1, double num2)
        {
            return num1 + num2;
        }

        /// <summary>
        /// 乘法
        /// </summary>
        /// <param name="num1">值一</param>

```

```

    /// <param name="num2">值二</param>
    /// <returns></returns>
    public double Multi(double num1, double num2)
    {
        return num1 * num2;
    }
}

```

3. 再建立一個梯形類別，並撰寫一個梯形面積運算方法，該方法會呼叫(相依)數學運算類別

```

namespace UnitTestPractice
{
    /// <summary>
    /// 梯形類別
    /// </summary>
    public class Trapezoidal
    {
        private readonly IMathCalculation _mathCalculation;
        public Trapezoidal(IMathCalculation mathCalculation)
        {
            _mathCalculation = mathCalculation;
        }

        /// <summary>
        /// 梯形面積計算
        /// </summary>
        /// <param name="num1">上底</param>
        /// <param name="num2">下底</param>
        /// <param name="num3">高</param>
        /// <returns></returns>
        public double TrapezoidalArea(double num1, double num2, double
num3)
        {
            double sum1 = _mathCalculation.Add(num1, num2); //上底加下
底
            double sum2 = _mathCalculation.Multi(sum1, num3); //(上底加
下底)乘高

```

```

        double result = sum2 / 2;    //除2

        return result;
    }
}

```

#### 4. 建立該類別的單元測試並修改

```

using NSubstitute;
using NUnit.Framework;

namespace UnitTestPractice
{
    /// <summary>
    /// 梯形單元測試
    /// </summary>
    [TestFixture]
    public class TrapezoidalTests
    {
        /// <summary>
        /// 數學運算
        /// </summary>
        private IMathCalculation subMathCalculation;

        /// <summary>
        /// 初始化
        /// </summary>
        [SetUp]
        public void SetUp()
        {
            //產生數學運算Stub物件(模擬物件)
            this.subMathCalculation = Substitute.For<IMathCalculation>();
        }

        /// <summary>
        /// 建立梯形物件
        /// </summary>
        /// <returns>梯形實體化物件</returns>
        private Trapezoidal CreateTrapezoidal()
    }
}

```

```

    {
        return new Trapezoidal(this.subMathCalculation);
    }

    /// <summary>
    /// TrapezoidalArea_梯形面積計算_取得計算結果
    /// </summary>
    [Test]
    public void TrapezoidalArea_梯形面積計算_取得計算結果()
    {
        // Arrange
        var trapezoidal = this.CreateTrapezoidal();
        double num1 = 1;    //預設傳入值1
        double num2 = 2;    //預設傳入值2
        double num3 = 5;    //預設傳入值3
        double addExpected = 10;    //預期加法值
        double multiExpected = 50; //預期乘法值
        double expectedResult = 25; //預期結果

        //將方法內的外在因素都用模擬物件替代並取得預期值
        //傳入兩個模擬參數做加法運算，最後回傳預期加法值
        this.subMathCalculation.Add(Arg.Any<double>()),
        Arg.Any<double>()).Returns(addExpected);
        //傳入兩個模擬參數做乘法運算，最後回傳預期乘法值
        this.subMathCalculation.Multi(Arg.Any<double>()),
        Arg.Any<double>()).Returns(multiExpected);

        // Act
        var result = trapezoidal.TrapezoidalArea(num1, num2, num3); //執行梯形面積計算

        // Assert
        Assert.AreEqual(expectedResult, result);    //判斷預期值與實際執行值是否相等，不相等則拋出例外
        Arg.Is<double>(result); //判斷實際執行值的型別是否為double
    }
}

```

## 5. 步驟 4 注意事項

- I. 在執行單元測試時，如果有其他相依物件，則需先將相依給替換掉，用模擬物件代替，如上述程式的 **IMathCalculation**，注意**此處需用介面宣告**
- II. 當測試的方法內須呼叫其他方法時，也需要將其他方法都替換掉，避免測試結果不準確(可能受其他方法影響測試結果)，如上述的這段程式碼

```
//將方法內的外在因素都用模擬物件替代並取得預期值
//傳入兩個模擬參數做加法運算，最後回傳預期加法值
this.subMathCalculation.Add(Arg.Any<double>(),
Arg.Any<double>()).Returns(addExpected);
//傳入兩個模擬參數做乘法運算，最後回傳預期乘法值
this.subMathCalculation.Multi(Arg.Any<double>(),
Arg.Any<double>()).Returns(multiExpected);
```

**Arg.Any<T>()**代表傳入一個**模擬參數**，如上述的 **Arg.Any<double>()**，即為一個模擬的 **double** 值，而 **Returns** 代表前面描述的方法最後回傳值，宣告了這段方法後，在執行梯形面積方法時，若有呼叫到加法、乘法，就會自動替換成該處的執行並回傳 **Returns** 的結果

- III. 另外有時也會驗證執行結果的型別是否為預期型別，則可以下

**Arg.Is<T>(value)**，如上述的這段

```
Arg.Is<double>(result); //判斷實際執行值的型別是否為 double
```

- IV. 其他還有很多驗證方法，可自行上網查詢。

## 6. 執行單元測試同範例一的步驟 5