Visual Studio 自習書シリーズ

Visual Studio 2012 による単体テスト

発行日：2013 年 5 月 9 日

最終更新日：2013 年 5 月 29 日

日本マイクロソフト株式会社

目次

テスト自動化の重要性

ソフトウェアがビジネスや生活に多大な影響を与え、欠かすことができない存在となった今、ソフトウェアの品質はビジネスにとってもますます重要な存在になってきています。しかしその一方で、開発プロジェクトのコスト削減や開発期間の短縮が求められており、品質を確保するための作業に十分な工数をかけられない現状があります。

そのような現状において、近年注目を集めつつあるのが、テストの自動化です。テストの自動化には以下のようなメリットが挙げられます。

テストの効率化

仕様変更やバグ修正などの理由により、開発プロジェクトの途中でプログラムを何度も修正することがあります。プログラムを修正した場合は、その変更によって予想外の新たな問題が発生していないかを確認する必要があります。(これを「回帰テスト」「レグレッション テスト」と言います)

手動で単体テストを実施すると、その都度人の手によりテストを実施する必要がありますが、自動化のためのテストコードを用意しておけば、2 回目以降はツールを活用し、ボタン1 つでテストを実行することができます。

また　Visual Studio においてチーム開発を支えるサーバー機能を提供している Team Foundation Server を利用すると、ソースコードのチェックイン時にビルドと単体テストを実施し、コードの変更により単体テストが失敗した場合にはいち早くその問題を検出することが可能です。
つまり、自動化により効率的にテストを行うことができるようになったことで、頻繁なテストの実施が可能になり、より素早く問題を発見し、対応することが可能になるのです。

確実性と一貫性

手動（人力）によるテストの実施では工数がかかるだけでなく、人によって実施の制度にばらつきが出てしまい、また操作方法や入力においてミスがうまれることもあります。テストをツールで自動化することによってテストに確実性と一貫性が出ます。

保守性の向上

近年ソフトウェアを1 からスクラッチから作成せず、既存のソフトウェアを保守したり、拡張する機会が多くなってきました。他の開発者が作成したプログラムを修正する場合、一部のコード修正であっても影響範囲が不明確なため広範囲なコードを確認する必要があります。そのような場合にソフトウェアとともにテストコードを保持することで、ソフトウェアを修正した際にテストを実施することで副作用が発生しないか都度確認しながら、より確実に保守、拡張を進めることができます。近年ではソフトウェアを納品する際に実装コードだけでなくテストコードも合わせて納品してソフトウェアの保守、拡張に備える企業も増えています。

品質の透明性

ソフトウェアの品質を確保するためには、テストの実施だけでなくテストの実施データや、実施結果等の管理も重

要となります。例えば Team Foundation Server を利用すると、テストの実施結果をデータ化し、自動的に収集、共有することができます。この結果、テストの進捗やソフトウェアの品質を把握することがより容易となります。

この自習書においては、テストの自動化の第一歩として単体テストを取り上げます。
単体テストに関しては、Visual Studio のような有償ツールはもちろん、オープンソースの NUnit のような無償のツールも普及しており、また様々な技術書も存在しているため、最も身近で導入を行いやすい分野だといえます。

ぜひこの自習書で、単体テストの基本的な概念と実施のための手順をご確認ください。

※ Visual Studio 2012 試用版の入手方法

Visual Studio 2012 の試用版は無償で以下のWeb サイトよりダウンロード入手いただけます。
■Visual Studio 2012 試用版
http://www.microsoft.com/visualstudio/jpn/downloads

（本自習書の内容をお試しいただくには Premium もしくは Ultimate の環境をご用意ください）

単体テストの基本的な手順

１つめのハンズオンとして、ここでは Visual Studio 2012 による単体テストの作成と、実施のための基本的な手順を学びます。

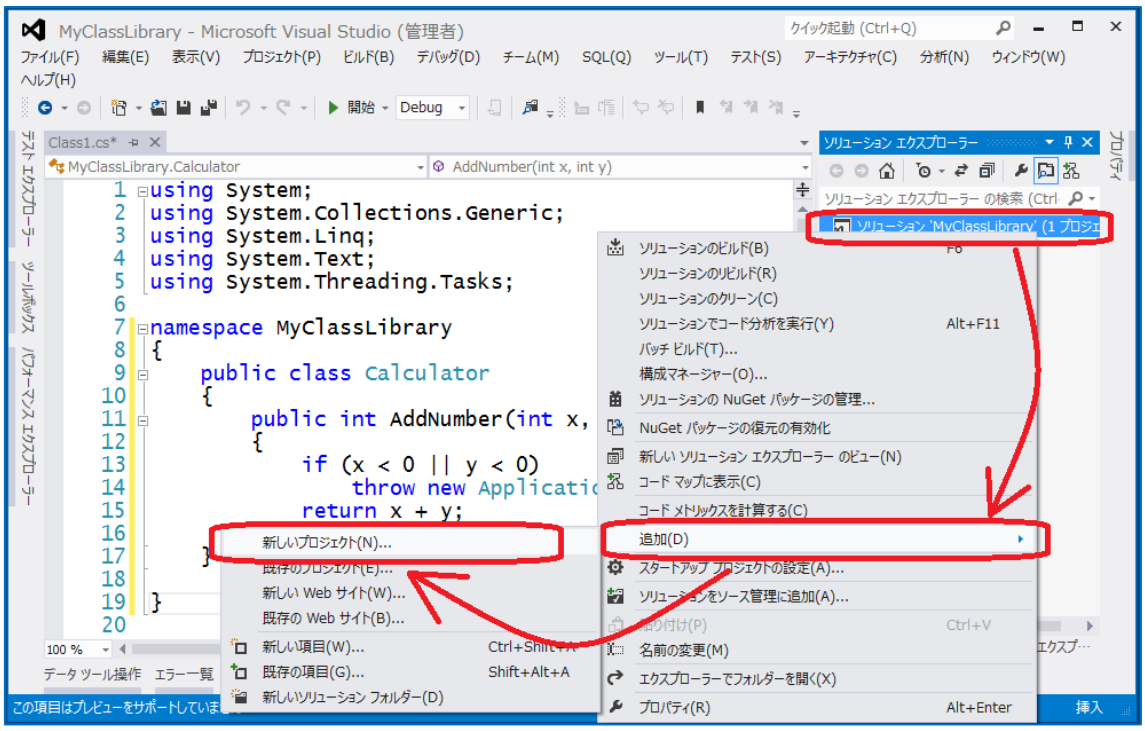
まず、事前の準備として、本ハンズオンではテスト対象のコードとして、“MyClassLibrary” プロジェクト（クラスライブラリのプロジェクト。名前空間もおなじく “MyClassLibrary” としています ）を作成し、以下の C# のクラスライブラリをテスト対象コードとして作成しています。

```
public class Calculator
{
    public int AddNumber(int x, int y)
    {
        if (x < 0 || y < 0)
            throw new ApplicationException("0 以上の数値を指定してください");
        return x + y;
    }
}
```

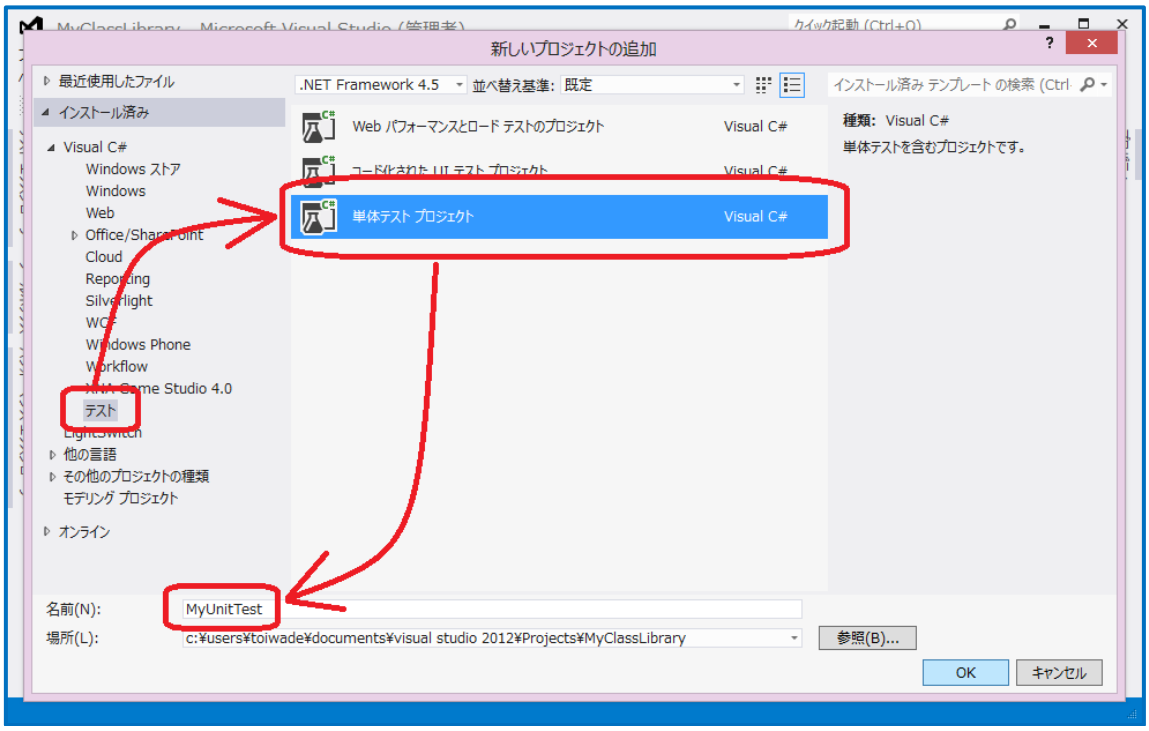
このAddNumber メソッドは x と y という int 型の2 つの引数を取り、その加算結果を int 型で返却する処理です。また、x もしくは y が0 未満の場合は例外が発生するようにしています。

上記コードに対して、単体テストを作成し、実施する手順を説明します。

1. Visual Studio 2012 のソリューション エクスプローラーにおいて、該当のコードを含むソリューションを右クリックし、メニューを表示し、「追加」→ 「新しいプロジェクト」を選択します。

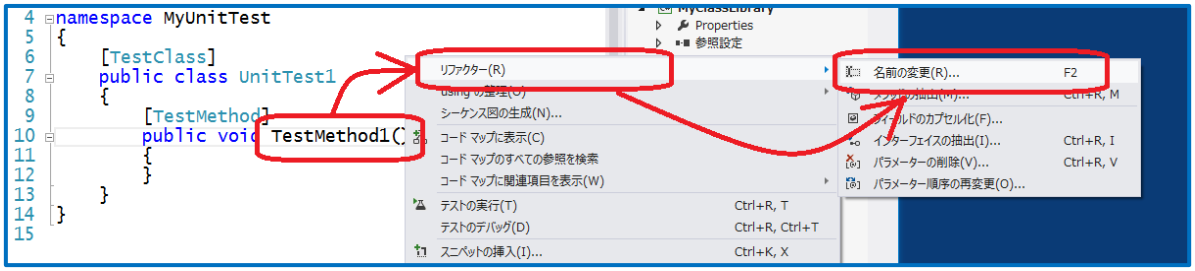


2. プロジェクトテンプレートが表示されるので、「Visual C#」の「テスト」を選択します。表示されたテスト用のテンプレートから「単体テスト プロジェクト」を選択し、またプロジェクト名として “MyUnitTest” と入力し、「OK」をクリックします。



3.　　単体テスト用のスケルトンコードが生成されます。Visual Studio 2012 の単体テスト機能では、単体テストを行うクラスには、**[TestClass] 属性**が、また実際のテストを行うテスト メソッドには **[TestMethod] 属性**が指定されています。テスト用のメソッドや、クラスを追加したい場合は、同様にこれらの属性を持つメソッドやクラスを追加します。

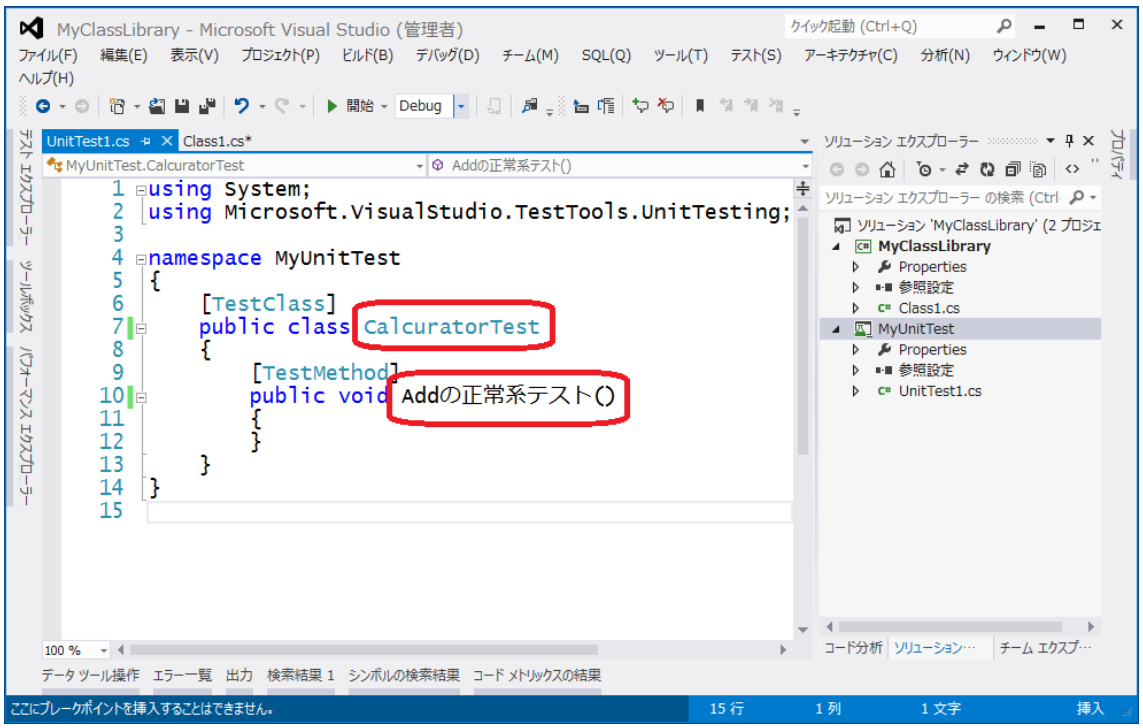
さて、今回作成されたスケルトンコードにおいて、テストのクラス名、メソッド名を適切な名称にリファクタリングしましょう。リファクタリングのメニューは、例えば対象のテストメソッドにおいて、右クリックでメニューを開き、「リファクター」を選択することで確認することができます。今回は「リファクター」メニューから「名前の変更」を選択します。



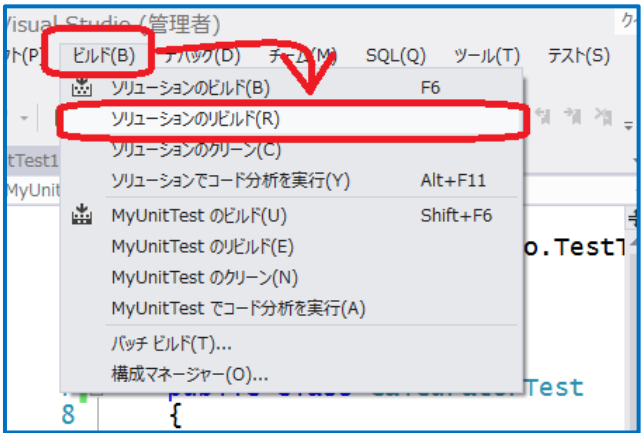
4.　　今回はクラス名を「CalcuratorTest」とし、メソッド名を「Addの正常系テスト」と変更します。

※　単体テストにおいては、通常テスト対象となるクラス（今回は “Calcurator” クラス）に対して、テストクラスを一つ用意します。その際に、テストクラスの名称としては、対象となるクラスの名称に “Test” を付け加えたものにする、といったルールを用意することで、単体テストのコードの保守性を向上させることが可能です（今回は、”Calcurator” + “Test” で、”CalcuratorTest” というクラス名にしています）。

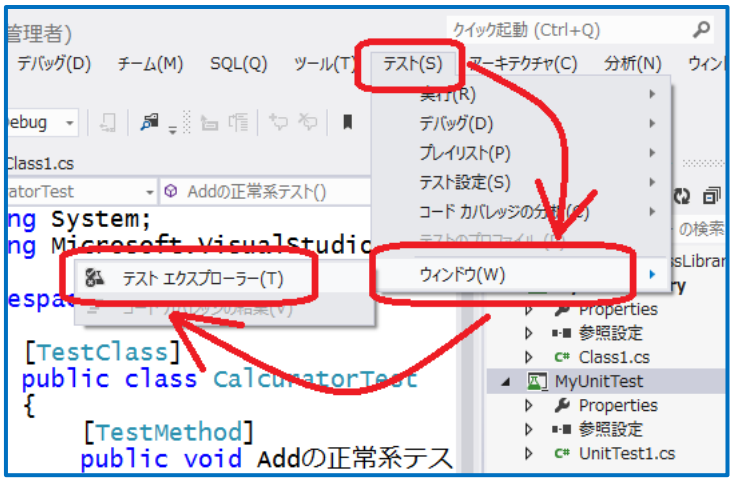
※　同じくメソッド名に関しては、テストを実施する際にメソッド名だけでなにをテストするメソッドなのかがわかるようにしましょう。今回は、日本語を利用しメソッド名を命名しています。



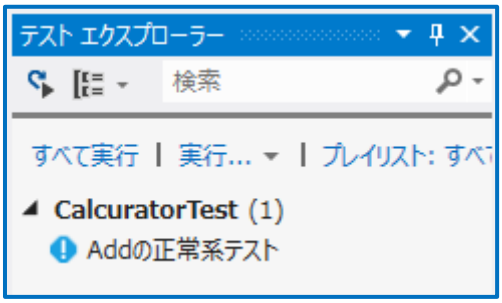
5.　「ビルド」メニューから「ソリューションのリビルド」を実行します。



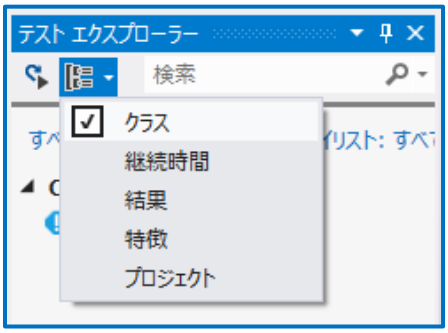
6.　ビルドが正常に終了したら、「テスト」メニューから「ウィンドウ」→「テスト エクスプローラー」を選択します。



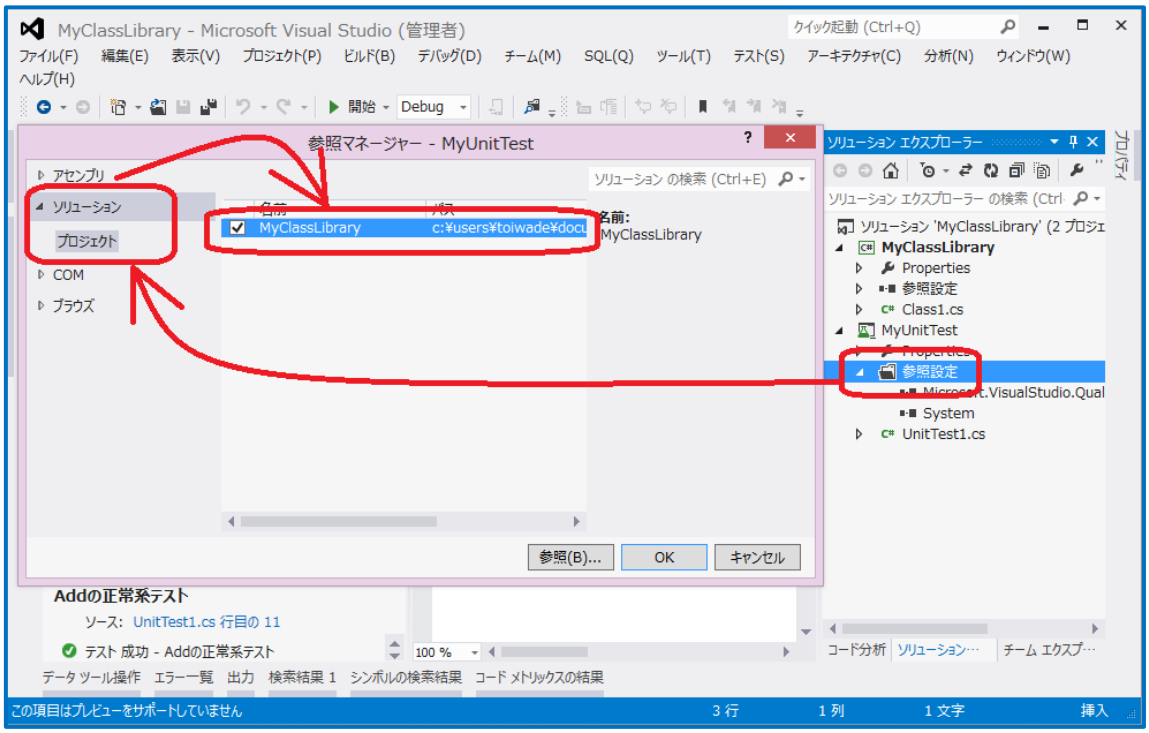
7.　テスト エクスプローラーは Visual Studio において作成した単体テストを一覧し、またそこからテストの実行や、実行結果の確認、あるいはテストの分類を行うための機能を提供するウィンドウです。
ソリューションのビルドが完了すると、Visual Studio がソリューション中に含まれる単体テストの情報を自動的に収集し、テスト エクスプローラーに表示します。
この段階では、以下のように、CalcuratorTest クラスにおいて、”Addの正常系テスト” のメソッドが表示されていることを確認します。



※　テスト エクスプローラーでは、テストの分類をおこないための機能として、「クラス」の他、テスト実施時の「継続時間」やテストの「結果」、あるいは「特徴」や「プロジェクト」といった分類軸でテストを表示することが可能です（一部の機能は Visual Studio 2012 の Update を適用することで利用いただける内容です。Visual Studio 2012 を最大限に活用いただくためにも、最新の Visual Studio Update の適用をおすすめいたします）。



8.　MyUnitTest プロジェクトの参照設定において、テスト対象となるプロジェクト（下記の画面においては MyClassLibrary プロジェクト）への参照を追加します。手順としては、ソリューション エクスプローラーにて、単体テストの MyUnitTest プロジェクトの「参照設定」にて右クリックでメニューを表示し、「参照の追加」を選択します。その後、表示されたウィンドウにて、「ソリューション」の「プロジェクト」を選択し、「MyClassLibrary」にチェックを入れ、「OK」をクリックします。



9. CalucratorTest クラスを以下のように書き換えます。

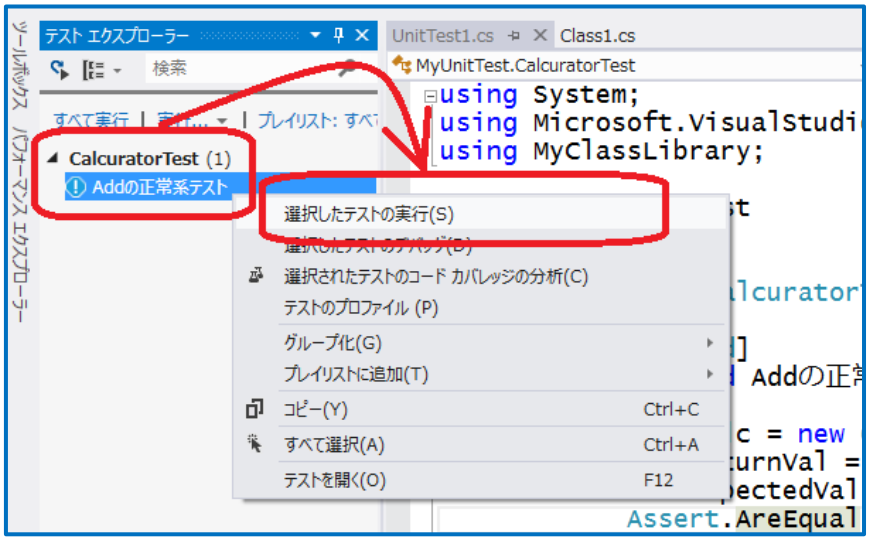
```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MyClassLibrary;

namespace MyUnitTest
{
    [TestClass]
    public class CalcuratorTest
    {
        [TestMethod]
        public void Addの正常系テスト()
        {
            var calc = new Calculator();
            var returnVal = calc.AddNumber(1, 2);
            var expectedVal = 3;
            Assert.AreEqual(expectedVal, returnVal);
        }
    }
}
```

このテストメソッドでは、Calcurator クラスのインスタンスを生成し、AddNumber メソッドを第一引数に 1 を、第二引数に 2 を渡して呼び出し、その結果を returnVal 変数に格納しています。
また、この呼び出しを行った際に期待される結果を expectedVal 変数に格納しています（1と2の足し算なので、結果として 3 を期待しています）。

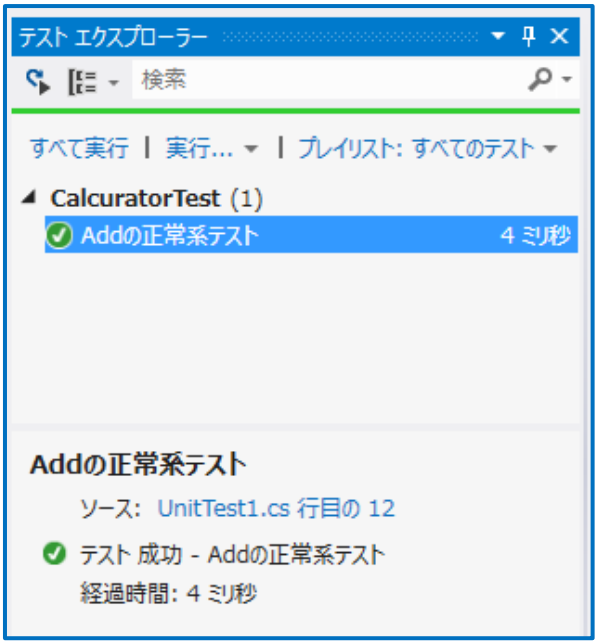
最後に、Assert (“アサート“ と読みます)クラスの AreEqual メソッドにて、期待される結果(expectedVal)と実際に帰ってきた値(returnVal)と比較しています。
Assert 文は、テストの結果として ”こうであるべき”、”こうでならなければならない” という状態を判定するための条件文で、この Assert 文に成功するということが、テスト対象のコードの振る舞い（今回は Calcurator クラスの AddNumber メソッドの振る舞い）が単体テストに合格したこと（今回は “Addの正常系テスト” に合格したこと）、を示します。

10. 「ビルド」メニューから「ソリューションのリビルド」を実行します。

11.　「テスト エクスプローラー」から、「Addの正常系テスト」を選択したまま右クリックを行い、表示されたメニューから「選択したテストの実行」を選択します。



12.　Visual Studio によって、”Addの正常系テスト” メソッドが実行され、テスト エクスプローラーにテストの結果が表示されます。成功した場合は、テスト メソッドの前に、緑色のインジケーターが表示されます。

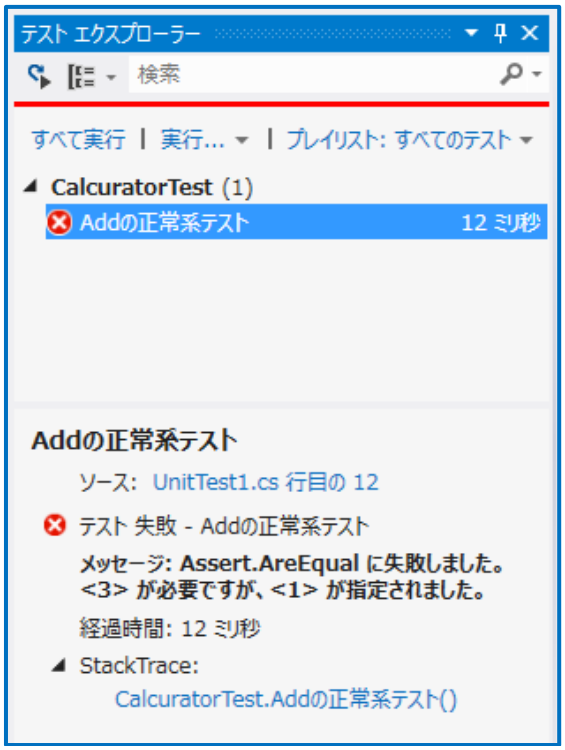


13.　次にテストに失敗した場合の挙動を確認するために、テスト対象となる AddNumberメソッドを以下のように書き換えます（x+y ではなく、x を返すように変更）。

```
public class Calculator
{
    public int AddNumber(int x, int y)
    {
        if (x < 0 || y < 0)
            throw new ApplicationException("0 以上の数値を指定してください");
        return x;
        //return x + y;
    }
}
```

14.　「ビルド」メニューから「ソリューションのリビルド」を実行し、続けて「テスト エクスプローラー」から、「Addの正常系テスト」を選択したまま右クリックを行い、表示されたメニューから「選択したテストの実行」を選択します。

15.　テスト エクスプローラーにおいて、以下のようにテストが失敗したことが表示されます。テスト メソッドの前には、単体テストの失敗を表す赤色のインジケーターが表示され、また、Assert に失敗した理由が表示されます。



16.　手順「13」で変更した Calcurator のコードを元に戻して下さい。リビルドを実施し、再度 “Addの正常系” の単体テストを実行し、テストが成功するのを確認しましょう。

以上で、１つ目のハンズオンは終了です。

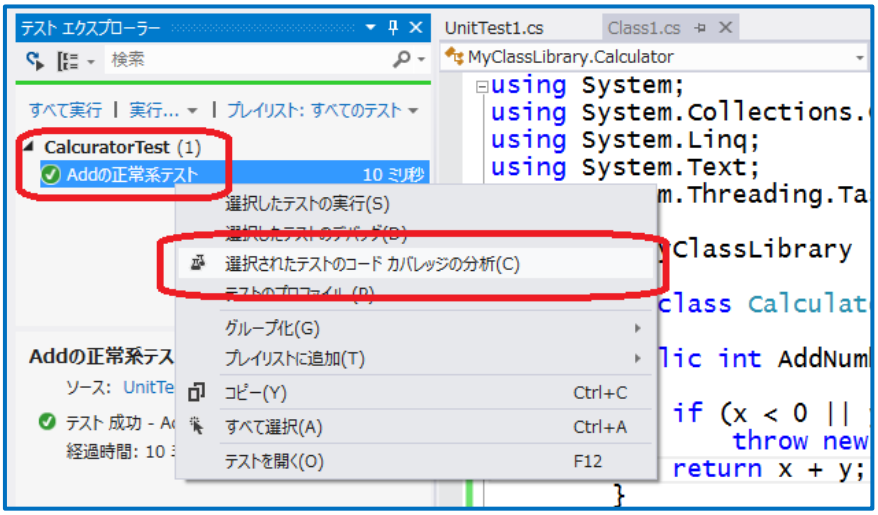
1つ目のハンズオンでは、Visual Studio 2012 を使用した単体テストの作成と、実施のための基本的な手順として、以下の内容を確認しました。

(1) ソリューションに対する**単体テストプロジェクト**の追加方法
(2) 単体テストを作成するための基本的な属性の理解（**TestClass 属性**と、**TestMethod 属性**）
(3) **テスト エクスプローラー**による単体テストの表示
(4) **Assert クラス**による期待される振る舞いの指定
(5) **テスト エクスプローラー**による単体テストの実行
(6) 対象コードが期待される振る舞いを行った際の動作（テストに成功した場合の動作）
(7) 対象コードが期待される振る舞いを行わなかった際の動作（テストに失敗した場合の動作）

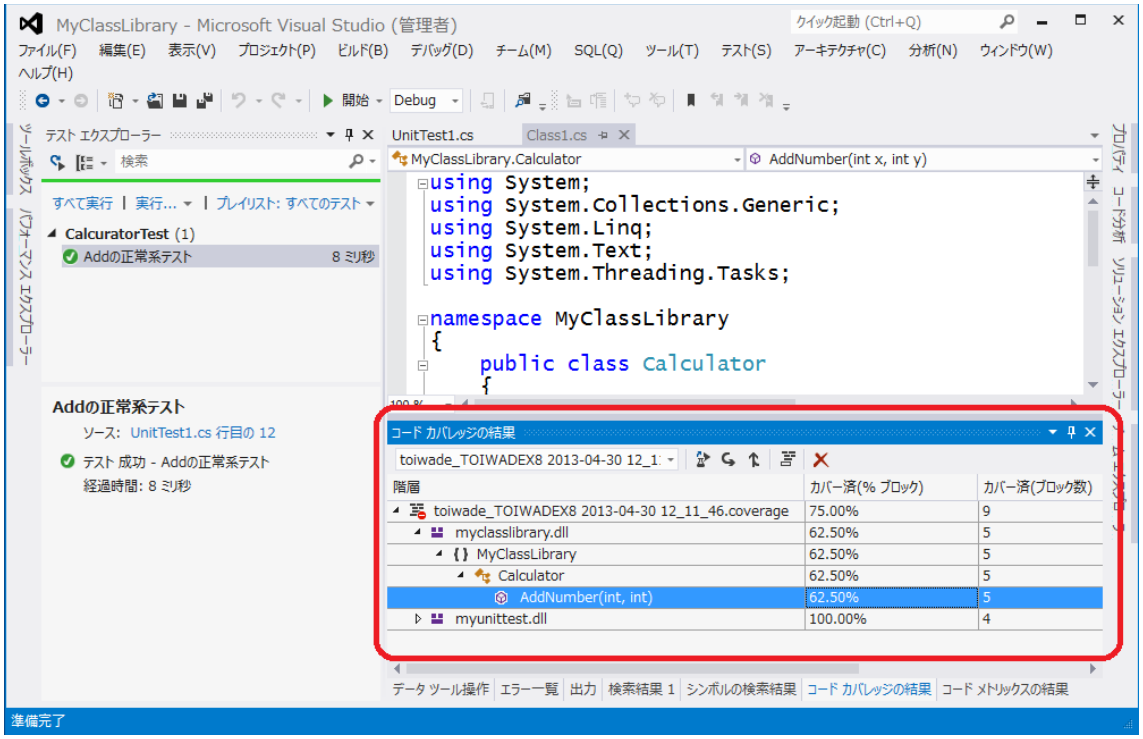
カバレッジ分析を活用した単体テスト

2つ目のハンズオンでは、より質の高い単体テストを作成するために、Visual Studio 2012 のコードカバレッジ機能を活用する方法を学びます。

1. テスト エクスプローラーにて、”Addの正常系テスト” を実行する際に、「選択されたテストのコードカバレッジ分析」を指定し、テストを実行します。

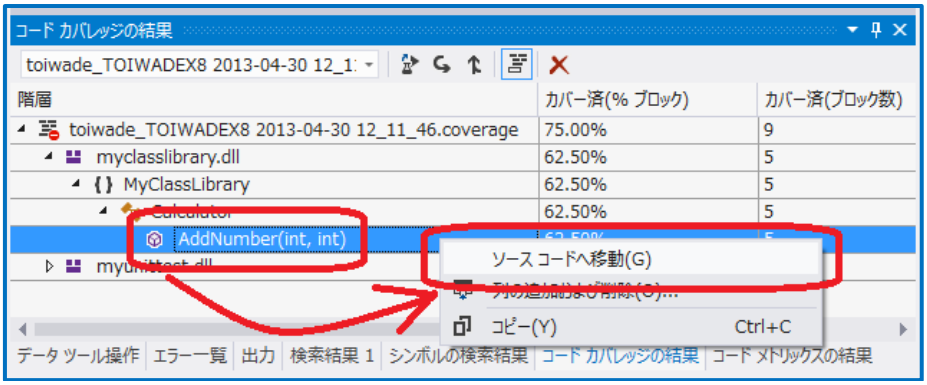


2. Visual Studio の「コードカバレッジの結果」ウィンドウに、実施した単体テストによって、テスト対象コードの何%がカバーされたかが数値で表示されます。

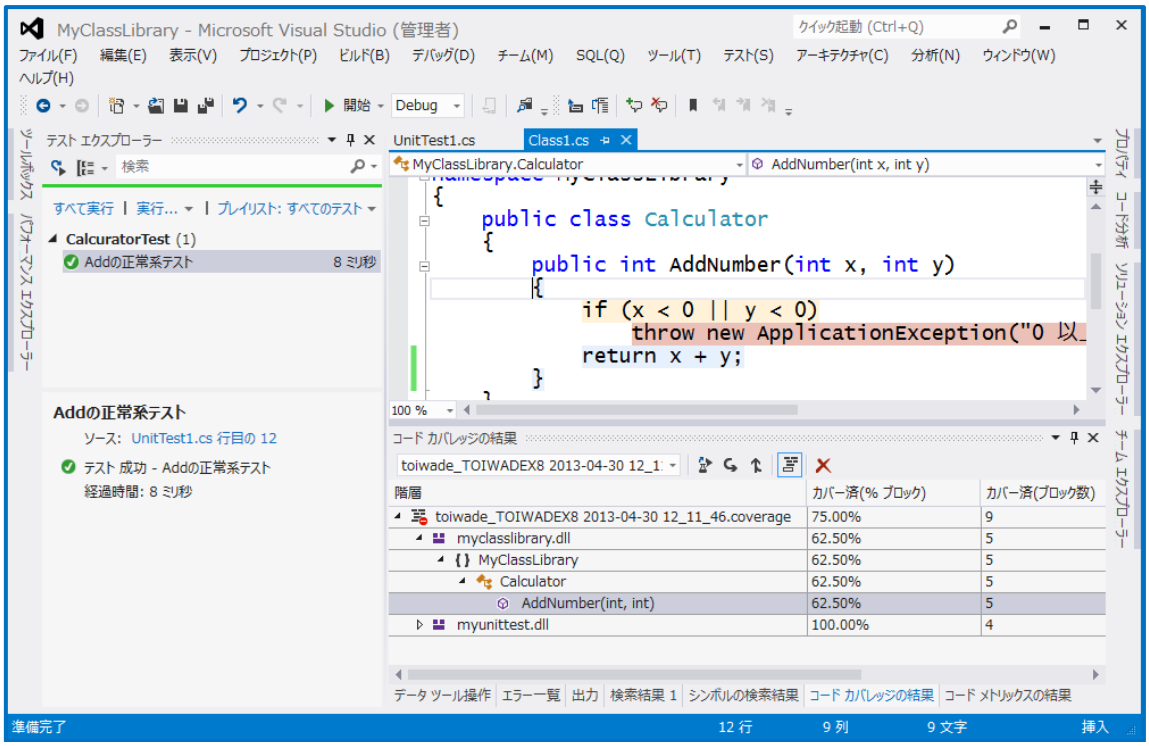


3.　　「コードカバレッジの結果」ウィンドウにて、AddNumber メソッドを選択し、右クリックを押し、表示されたメニューから「ソースコードへ移動」を選択します。

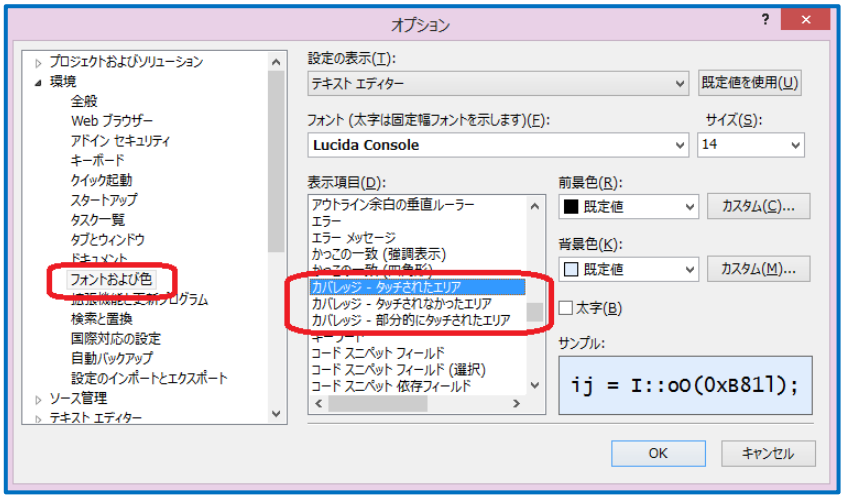
※　ダブルクリックでも同様にソースコードへ移動します。



4.　　AddNumber メソッドのソースコードが、コードカバレッジの情報を付加した状態で表示されます。具体的には、実施したテストによりカバーされたコードが「青色」で、カバーされなかったコードが「赤色」のマーカーが付いた状態で表示されます。また、部分的にカバーされたコードは「黄色」のマーカーが付けられます。



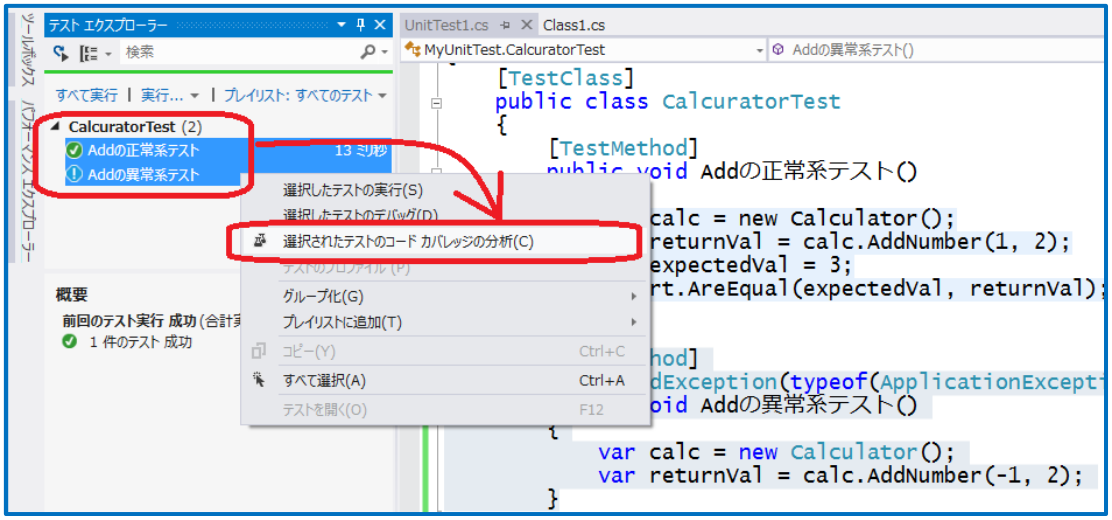
コードカバレッジの結果表示における配色は「ツール」→「オプション」メニューにおける、「環境」の「フォントおよび色」の設定項目で変更を行うことが可能です。



5. 単体テストによるカバレッジ率を上げるために、if 文における条件評価が True となるテストパターンを追加しましょう。以下のテスト メソッドをテストクラス(CalcuratorTest クラス)に追加します。

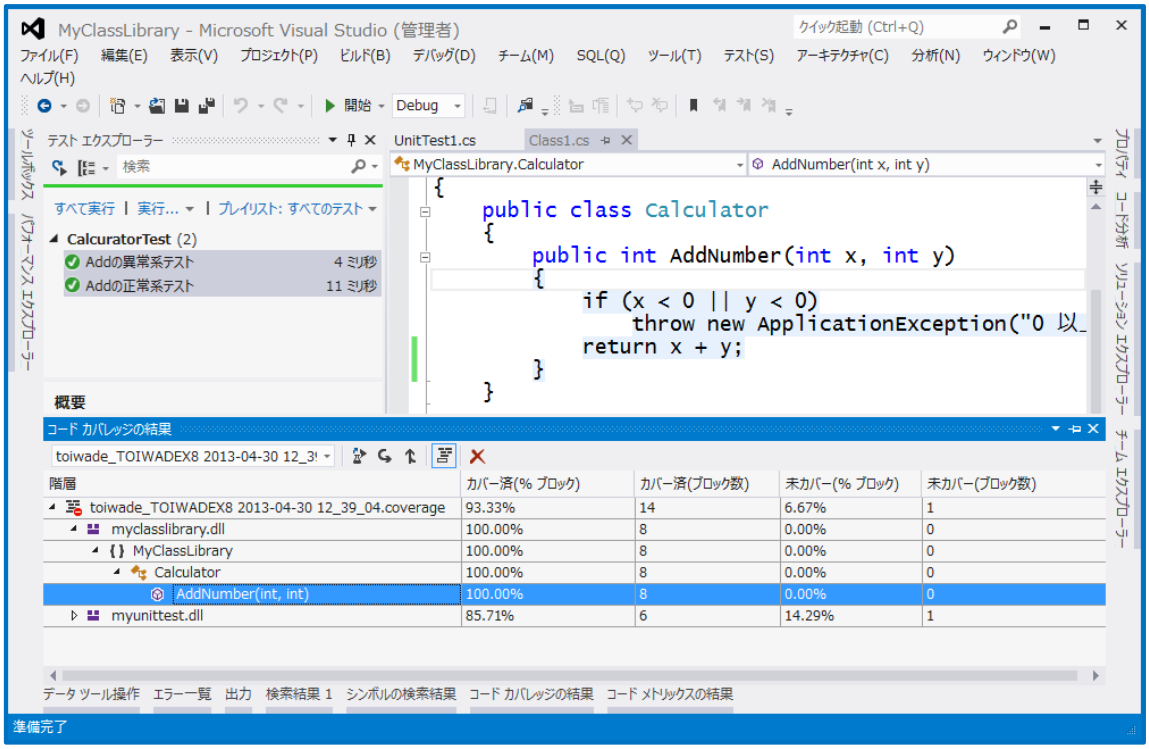
```
[TestMethod]
[ExpectedException(typeof(ApplicationException))]
public void Addの異常系テスト()
{
    var calc = new Calculator();
    var returnVal = calc.AddNumber(-1, 2);
}
```

6. ソリューションをリビルドすると、以下のように、テスト エクスプローラーに “Addの異常系テスト” が追加されますので、”Addの正常系テスト” と “Addの異常系テスト“ 両方を選択した状態で、「選択されたテストのコードカバレッジの分析」を実行します。



今回のテストにおいては、AddNumber の第一引数として、「-1」を与えることにより、例外が発生し、処理が中断してしまう、という動作が期待されます。
従って、AddNumber に対する戻り値を期待し、戻り値に対して Assert 文を用意するのではなく、テストメソッドの属性で **[ExpectedException]** を指定することで、メソッド呼び出しにおいて、「例外が起こるべき」テストとして作成しています。

7. 「コードカバレッジの結果」ウィンドウを確認すると、AddNumber のカバレッジ率が 100% になったことを確認できます。



※ Visual Studio で提供しているコードのカバレッジ測定機能は、コード行に対するカバレッジの測定となります（2つのテスト条件によりカバー率が100%となります）。if 文における論理的なカバレッジを考慮する場合、今回の条件式に対しては以下のように4つのパターンのテストデータを作成し、テストを行います（x, y の値は参考としてのサンプルです）。

	y がマイナス	y が0以上
x がマイナス	X=-1, y=-2	x=-1, y=2
x が0以上	x=1, y=-2	x=1, y=2

単体テストにおいては、「常に 100% のカバレッジを必要とする」わけではありません。カバレッジ率は一つの指標であり、テストにおける抜けや漏れを防ぐためのガイドとして使用しましょう。

以上で、２つ目のハンズオンは終了です。

２つ目のハンズオンでは、単体テストにおいて Visual Studio 2012 のコードカバレッジ機能を活用する方法として以下の内容を確認しました。

(1) **コード カバレッジ 分析の実行**

(2) **コード カバレッジ結果の確認**（数値による確認と、コードにおける視覚的確認）

(3) 例外を発生するロジックのテスト(**ExpectedException 属性の利用**)

(4) **複数の単体テストによるカバレッジ 分析の実行と結果の確認**

Fake を利用した高度な単体テスト

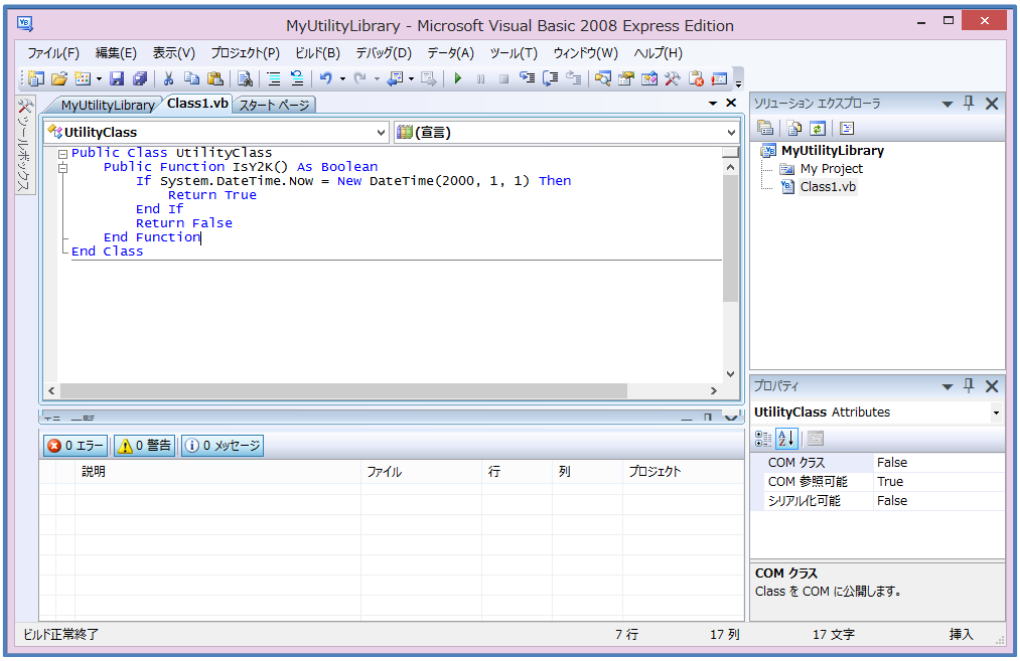
3つ目のハンズオンでは、Visual Studio 2012 の Fake Framework を利用した高度な単体テストを紹介します。Fake Framework は Visual Studio 2012 の Premium および Ultimate エディションにて利用いただける機能です（Premium エディションでは、Update 2 を適用いただくことで Fake Framework を利用いただけます）。

今回は、過去に作成したコードに対するメンテナンスの一環として、単体テストを作成する、といったシナリオで単体テストの作成を行います（Fake Framework 自体は過去のコードに対するメンテナンスだけでなく、テスト対象における外部コードへの依存性をなくし、純粋な単体テストを実施するために活用できる機能です）。

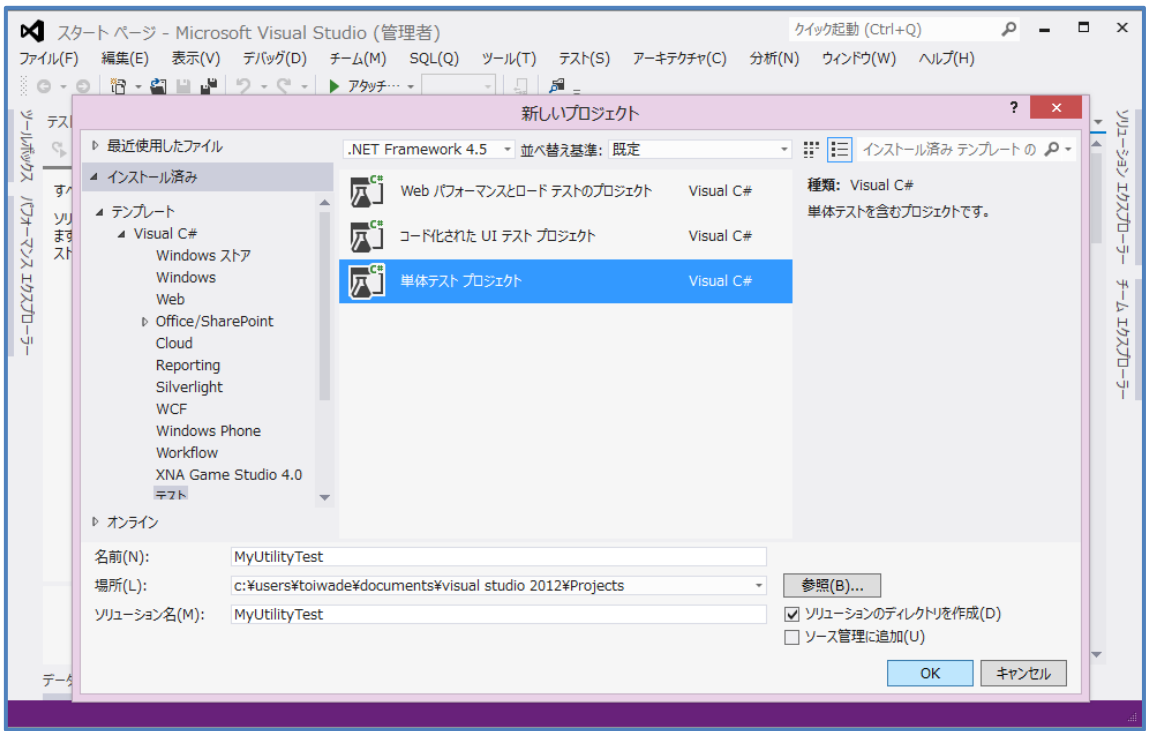
1. 最初に過去のコードのサンプルとして、Visual Basic 2008 にて以下のクラスライブラリを作成します。（Visual Basic 2008 の環境がない場合は、Visual Studio 2012 で通常の Visual Basic のクラスライブラリとして .NET Framework 4.5 を使用して作成いただいても結構です）

```
Public Class UtilityClass
    Public Function IsY2K() As Boolean
        If System.DateTime.Now = New DateTime(2000, 1, 1) Then
            Return True
        End If
        Return False
    End Function
End Class
```

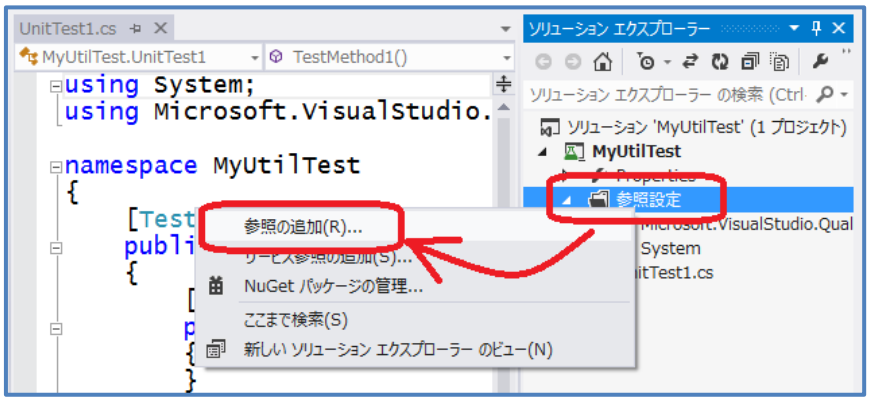
2. Visual Basic 2008 の環境において、該当コードをコンパイルし、DLL を作成します。



3.　　Visual Studio 2012 を立ち上げ、「ファイル」→「新規作成」→「プロジェクト」を選択します。表示されたダイアログにおいて、Visual C# のテンプレートから、「テスト」を選択し、「単体テスト プロジェクト」を指定し、プロジェクトの名前を “MyUtilTest” として 「OK」 をクリックします。

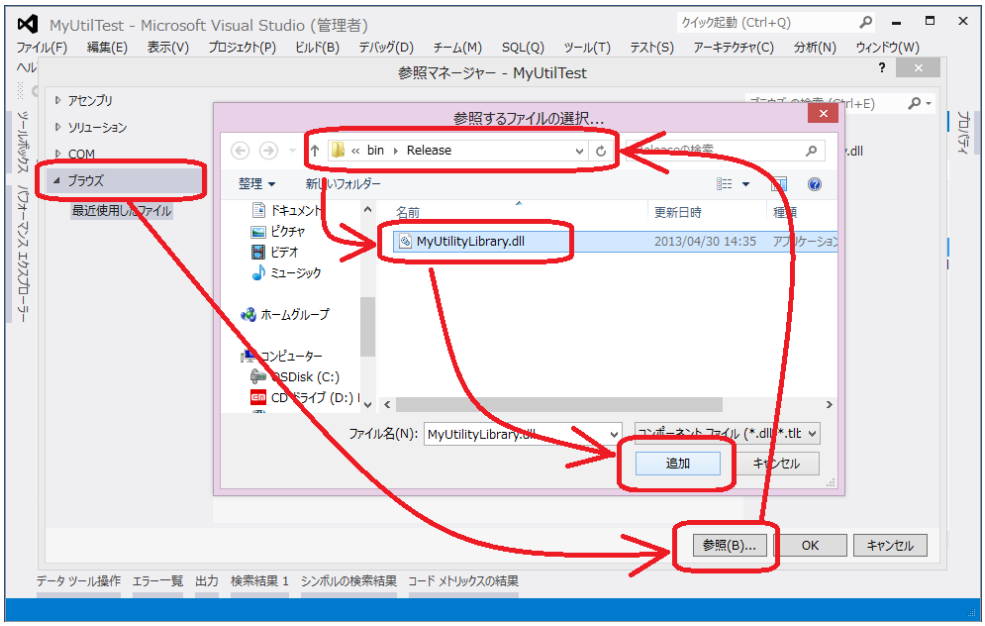


4.　　単体テストプロジェクトが作成されるので、ソリューション エクスプローラーの MyUtilTest プロジェクトにおける「参照設定」を右クリックし、表示されるメニューにおいて「参照の追加」を選択します。



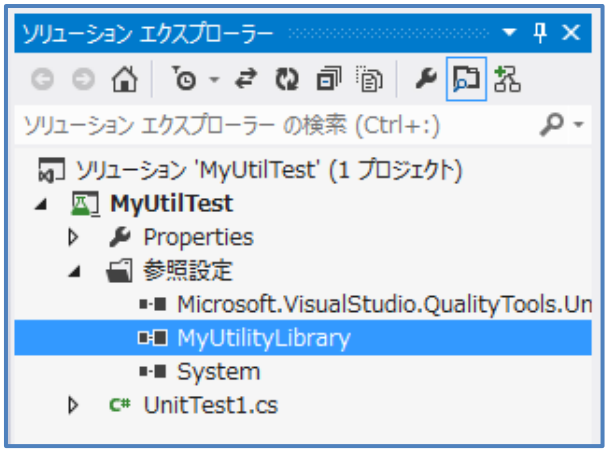
5.　　「参照マネージャー」のウィンドウが表示されるので、手順2 で作成した DLL を指定します。

手順としては、「参照マネージャー」のブラウズメニューを選択後、「参照」ボタンを押し、「参照するファイルの選択」ダイアログで、手順2で作成したDLL （例えば、C:¥Users¥<<ユーザー名>>¥Documents¥Visual Studio 2008¥Projects¥MyUtilityLibrary¥MyUtilityLibrary¥bin¥Release といったフォルダ）を指定します。

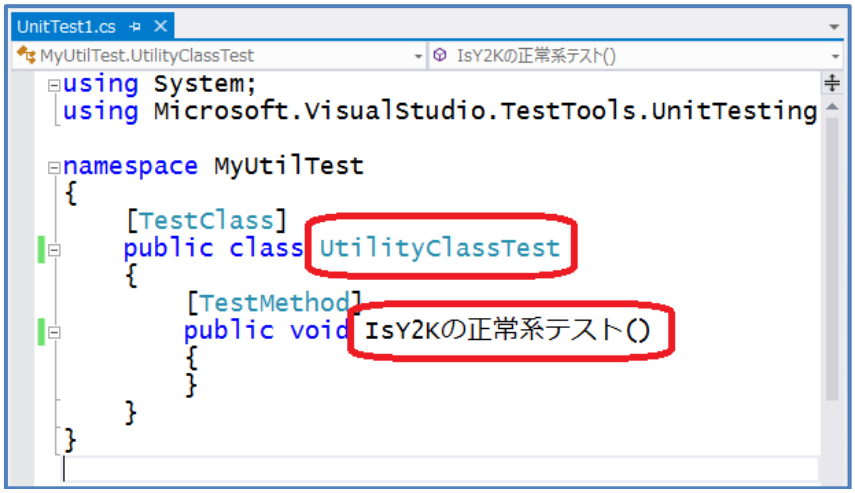


指定後、「参照マネージャー」にて、指定した DLL が表示されているので、チェックボックスが入っていることを確認し、「OK」を押します。

6.　　以下のように、参照設定に「MyUtilLibrary」が追加されていることを確認します。



7.　テンプレートから作成されたテストのクラス名を、”UtilityTestClass” に、またテストメソッドの名前を “IsY2Kの正常系テスト” に変更します。

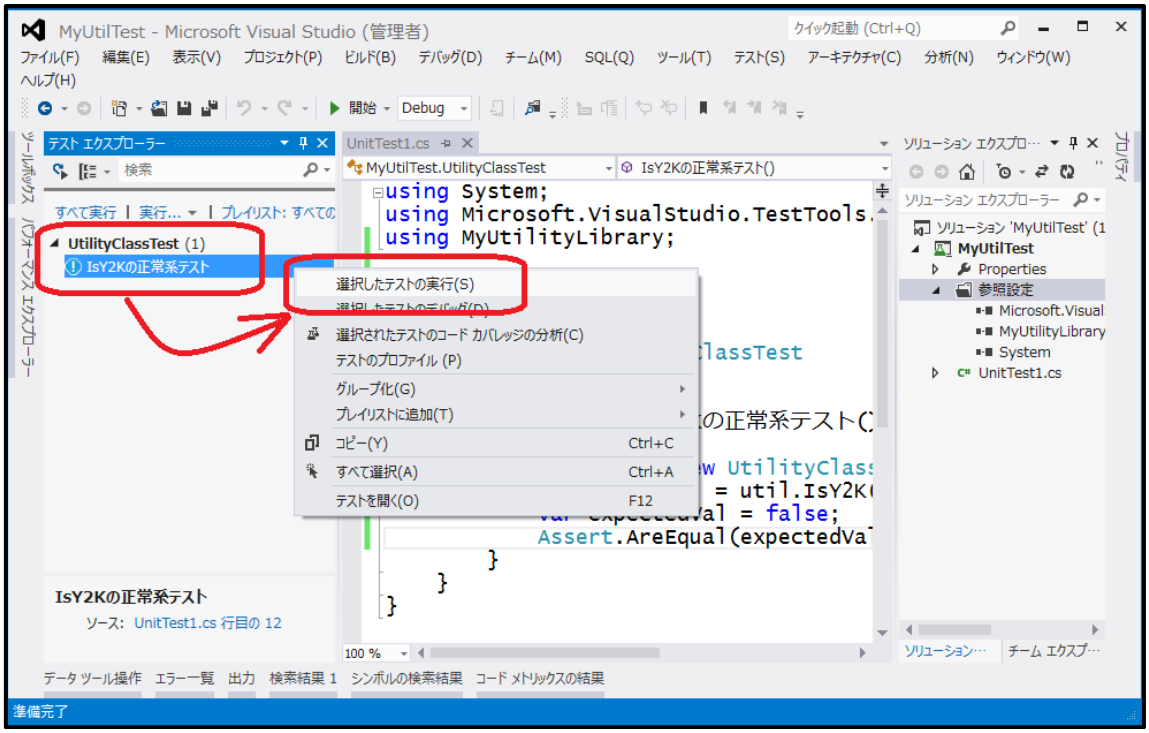


※　クラス名、メソッド名の変更は、Visual Studio のリファクタリング機能を利用します。一つ目のハンズオンの手順４を参照してください。

8.　テストメソッドとして、以下のコードを実装します。

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

namespace MyUtilTest
{
    [TestClass]
    public class UtilityClassTest
    {
        [TestMethod]
        public void IsY2Kの正常系テスト()
        {
            var util = new UtilityClass();
            var returnVal = util.IsY2K();
            var expectedVal = false;
            Assert.AreEqual(expectedVal, returnVal);
        }
    }
}
```

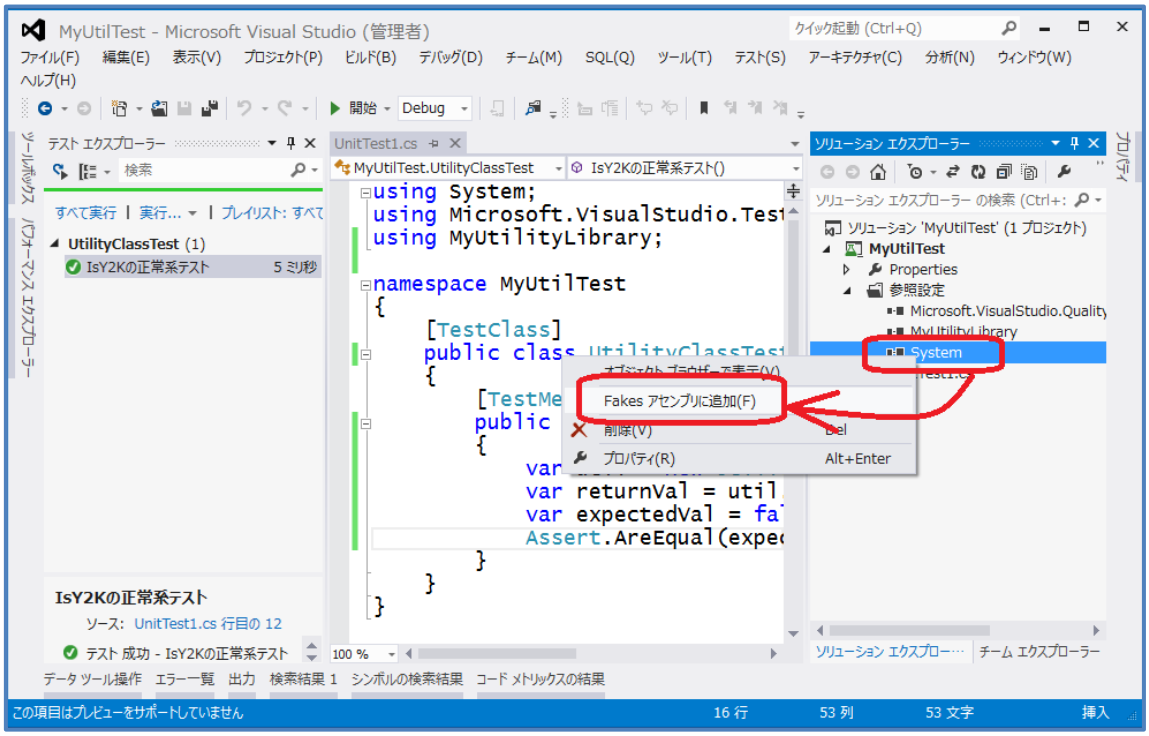
9.　　「ビルド」メニューから、「ソリューションのリビルド」を実行し、テストエクスプローラーに作成したテストメソッド(“IsY2Kの正常系テスト”)が表示されたら、右クリックでメニューを表示し、「選択したテストの実行」を行い、成功することを確認します。



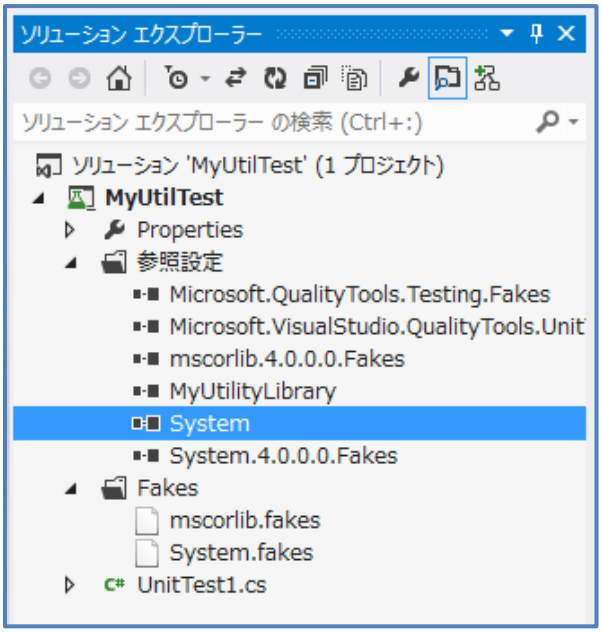
10.　　手順１のコードを再度確認すると、IsY2K メソッドは引数をとらないメソッドですが、外部の呼び出しとして System.DateTime.Now を呼び出しており、この呼び出し結果により振る舞いが変わる（ture を返すか、flase を返すか、という振る舞いが変わる）ことが認識できます。

一方で、IsY2K メソッドの単体テストとしては、System.DateTime.Now が、2000年1月1日を返した際に、true を返す、というシナリオも確認しなければ、十分な単体テストを実施している、とは言い難い状況です。このような場合に、Visual Studio 2012 から提供されている Fake Framework を使用すると、テストの実行時に、System.DateTime.Now の振る舞いを変更することが可能です。

ここでは、その第一の手順として、System 名前空間に含まれるクラス群の振る舞いを偽装(Fake)するために、fake アセンブリの追加を行います。
具体的には、ソリューションエクスプローラーにて、MyUtilTest プロジェクトが参照している System アセンブリを右クリックし、「Fakes アセンブリに追加」メニューを選択します。



11.　　ソリューション エクスプローラーにて、以下のように Fake のためのライブラリが作成、追加されたことを確認します。



12. MyUtilityClassTest クラスにおいて、次の2つのコードを追加します。

一つ目は、Fake Framework を使用するための using の追加です。

```
using Microsoft.QualityTools.Testing.Fakes;
```

2つ目が、あたらしいテストメソッドとして、2000年1月1日を期待するテストコードです（expectedVal として、true を期待するコードになっています）。

```
        [TestMethod]
        public void IsY2Kの2000年1月1日の正常系テスト()
        {
            var util = new UtilityClass();
            var returnVal = util.IsY2K();
            var expectedVal = true;
            Assert.AreEqual(expectedVal, returnVal);
        }
```

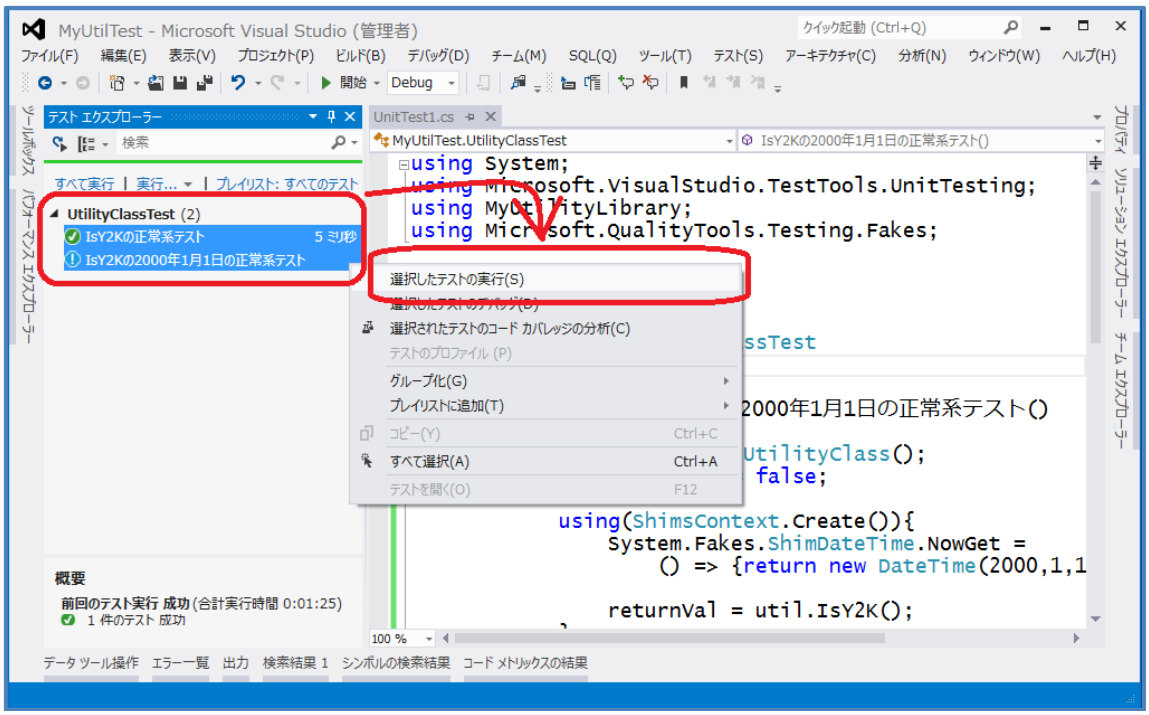
13. 次に、先ほど追加した “IsY2Kの2000年1月1日の正常系テスト” メソッドを、Fake Framework を使用して、System.DateTime.Now の振る舞いを偽装(Fake)するコードに書き換えます。

具体的には System.DateTime.Now の振る舞いを変えたい箇所（今回は、util.IsY2K() の呼び出し）を、ShimsContext によって囲むように書き換えます。また、ShimsContext において変更を行いたい振る舞いを追加します。

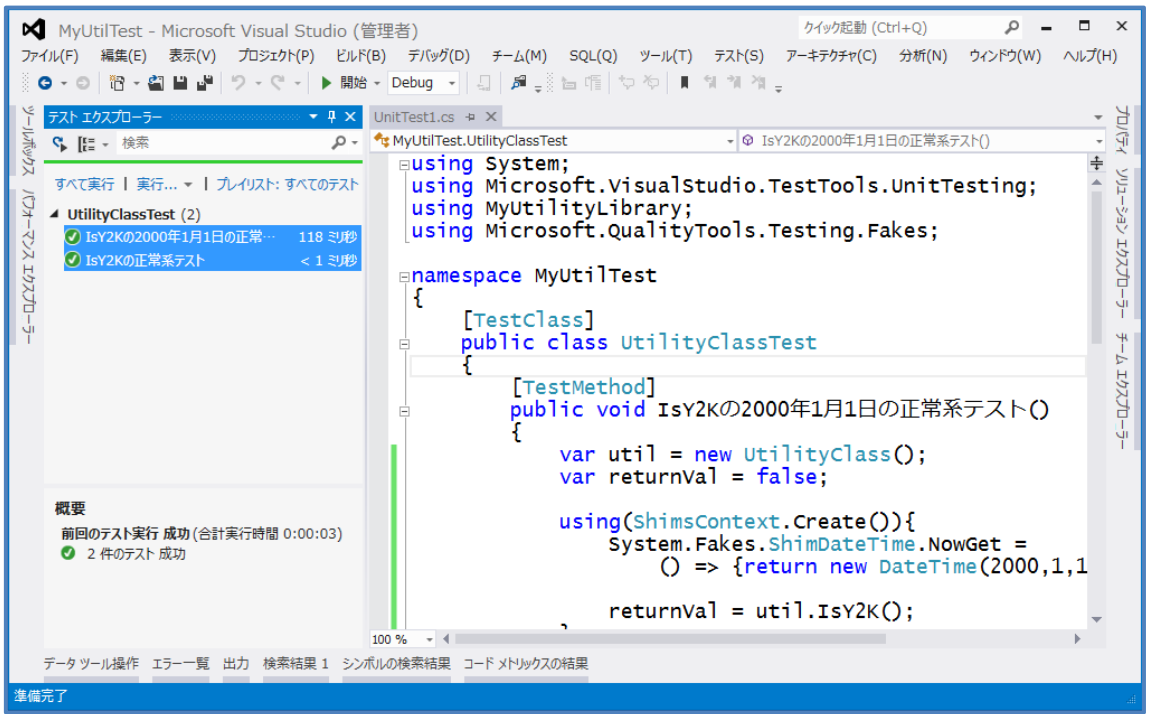
今回は、System.DateTime.Now を偽装するためのメソッド System.Fakes.ShimDateTime.NowGet メソッドの振る舞いを、2000年1月1日を指す DateTime インスタンスを返すように変更しています。

```
        [TestMethod]
        public void IsY2Kの2000年1月1日の正常系テスト()
        {
            var util = new UtilityClass();
            var returnVal = false;

            using(ShimsContext.Create()){
                System.Fakes.ShimDateTime.NowGet =
                    () => {return new DateTime(2000,1,1);};

                returnVal = util.IsY2K();
            }

            var expectedVal = true;
            Assert.AreEqual(expectedVal, returnVal);

        }
```

14.　　ソリューションのリビルドを行い、テストエクスプローラーに作成した “IsY2Kの2000年1月1日の正常系テスト” メソッドが表示されることを確認します。また、先に作成した ”IsY2Kの正常系テスト” と　“IsY2Kの2000年1月1日の正常系テスト” を選択し、「選択したテストの実行」を行います。



15.　　テスト エクスプローラーにて、二つのテストの実行結果が「成功」になったことを確認します。



以上で、3つ目のハンズオンは終了です。
3つ目のハンズオンでは、単体テストにおいて Fake Framework を使用し、より高度な単体テストを行うための方法として以下を確認しました。

(1) **過去に作成された DLL に対する単体テストの作成**
(2) **Fake Framework を利用した外部ライブラリの振る舞いの偽装**
    (ア) Fake アセンブリの追加
    (イ) ShimsContext を使用した外部ライブラリ呼び出し偽装の実装