# NUnit Ver2 利用ガイド
# (NUnit Ver2 Users Guidebook)

株式会社 OSK

第2.1版 平成16年8月25日

# 履歴

## 1 第1．0版（平成13年4月20日）

このドキュメントは、弊社 OSK がVS.NET調査の一貫として行った、NUnit の調査結果を広く公開するものです。この公開は、.NET環境で開発を行うディベロッパーの開発効率と品質の向上の手助けになればと考えて行っています。ドキュメント内のサンプルソースについては、自由に利用及び再利用ください。

本ドキュメントの作成は、小井土、潮屋、渡邊が担当しています。

お問合せ等は、koido@kk-osk.co.jpまでお願いします。

## 2 第1．1版（平成13年11月22日）

VisualStudio Beta2 に対応した。

謝辞：本修正のために、青木さんから貴重な意見をいただいたことをここで感謝します。

## 3 第1．2版（平成14年5月17日）

VisualStudio 製品版に対応した。NUnit は、1.11 を使用。NUnit の入手は、http://nunit.sourceforge.net/からダウンロード可能。

変更点は以下の通り。

① **画面を 1.11 に変更**

② **コードをＣ＃の規約に合わせた**

③ **継続的インテグレーションの章を追加**

## 4 第2．0版（平成15年2月19日）

NUnit Ver2.0 に対応した。また、継続的インテグレーションの章は削除し、トレーニングテキスト（別ドキュメント）に移行した。

## 5 第2．1版（平成16年8月25日）

NUnit Ver2.1.4 に対応した。

# 目次

# 第 1 章 概要

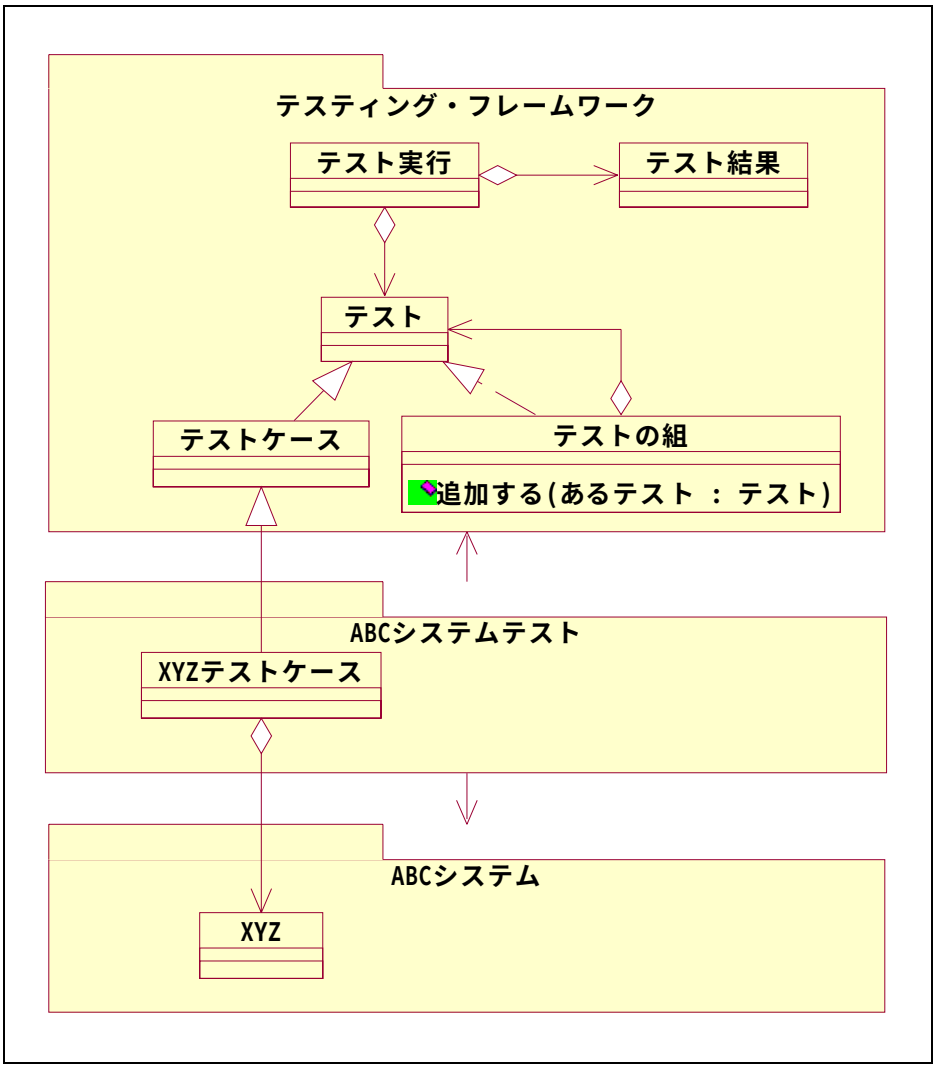
## 1 NUnit とは

### 1.1 テスティング･フレームワーク

現在、脚光を浴びているライトウェイトの開発手法である XP では、開発者が継続的にユニット･テストを行うことが実践項目として挙げられている。その継続的なテストを効率的に行う為に開発されたフレームワークがテスティング･フレームワークである。このテスティングフレームワークは、最初に XP 提唱者の一人である Kent Beck によって Smalltalk 版が開発された。後に JAVA 版 (JUnit[1]) が、Kent Beck とパターンの世界で非常に有名な Erich Gamma によって開発されて、フリーソフトとして提供されている。テスティングフレームワークは、自動テストを行うためのフレームワークである。このフレームワークは、自動テストを開発言語と同じ言語で記述することが特徴的である。この.Net に移植したものが NUnit である。NUnit は、C#で実装されている。

### 1.2 テスティング･フレームワークの基本構造

テスティング･フレームワークの基本構造は以下の様になっている。




[1] www.junit.orgから入手が可能である

### 1.3 NUnit とは

NUnit とは、.Net 環境で作成されたクラスを自動テストする為のフレームワークである。テストは、.Net 対応の全ての言語で記述することができる。テスト用プログラムの開発は非常に容易である。Ver2 から、.NET の特徴である属性ベースによるプログラミングを全面的に採用している。したがって、テストクラスやメソッドなどは、継承関係やメソッド名ではなく、属性によって識別が行われる。

本ドキュメントは、Philip Craig 氏を中心としたグループが開発した NUnit Ver2 について記述したものである。

## 2 NUnit を利用した開発手順

NUnit を利用してクラスを開発する場合の効率的な作業手順は以下のとおり。

① **テストを考える（クラスが外部に提供する機能を決定する）**
「テストを考える」とは、クラスを利用する側から見た、クラスが具体的にどのような機能を提供してほしいかを決定することである。具体的には、メソッドの呼出しやパラメータの設定/取得などの手順を明確にし、その機能によって想定される結果値あるいは結果を明確にすることである。

② **クラスが公開するメソッドとプロパティを実装する**
プロジェクトを作成し、公開するメソッドとプロパティの型を実装する。実際の処理は作成しない。

③ **テストを実装する**
プロジェクトを作成し、テストを行うクラスを実装する。テストは、想定した結果値と実際の処理を行った結果値を比較するものとなる。比較は、Assert を利用して行う。

④ **テストの失敗を確認する**
テストを実行し、失敗することを確認する。クラスは、公開するメソッドとプロパティの型だけを実装しているので、テストは失敗するはずである。

⑤ **クラスの実装を行う**
クラスの実装を行い、テストを実行する。テストが全て通過するまで実装を行う。テストが無事通過した時点が実装の終了となる。

以上のように、NUnit を利用した開発では、公開する機能（メソッドとプロパティ）を決定し、テストを書き、実装を行うといった手順で作業を行う。しかし、一度に全てのテストを書くのではなく、少しテストを書き、少し実装を行うといった手順が有効である。最初から複雑なテストを作成することは望ましくない。複雑なテストは、作成に工数が掛かってしまうこと、問題解決に時間が掛かってしまうことなどの問題がある。最初に単機能のテストから記述する。

## 3 NUnit のインストール

http://nunit.sourceforge.net/ から NUnit-V2.1.4.msi をダウンロードする。その後、エクスプローラのコンテキストメニューから「インストール」を選択し、インストールを行う。

## 4 NUnit の日本語化

NUnit Ver2.1.4 では、特に変更することなくそのまま日本語環境で利用することができる。

# 第2章 VisualStudio.NET 用テストサンプル

VisualStudio.NET を使用してシステムを開発する場合の NUnit を利用した開発手順について、簡易なプログラムを例に説明を行う。

## 1 例題のプログラム

販売店で顧客のポイントを管理するプログラムの一部を開発する。要求には、以下の項目がある。

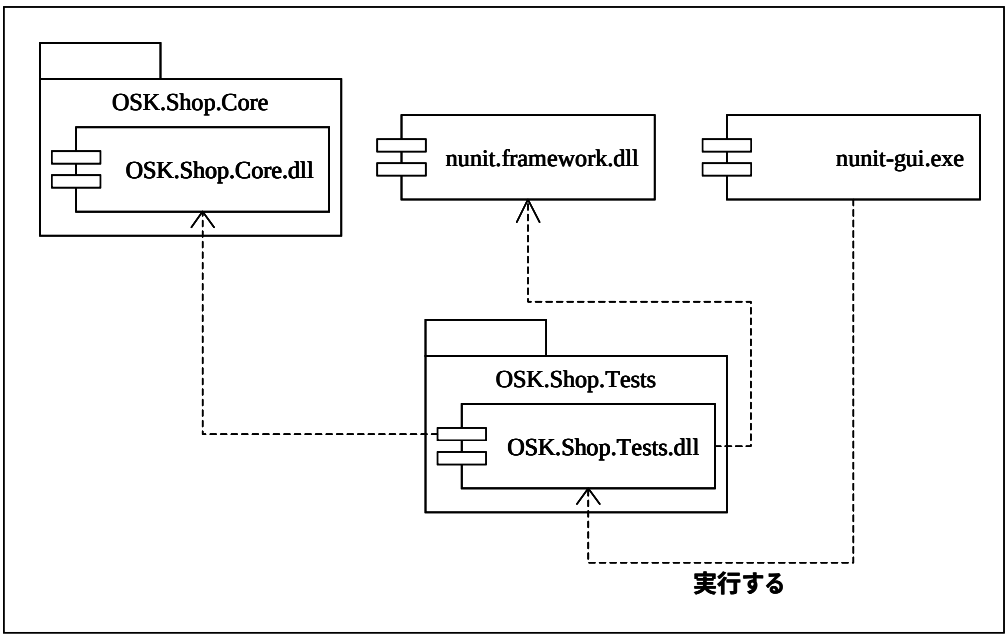
- ID と名前とポイントを管理する
- ポイントは基本的に価格の 5%であるが、10%のポイントが付く特別商品とポイントの付かない特価商品がある。

### 1.1 プロジェクト構成

本サンプルは、プロジェクトとして基本処理部分と単体テスト部分の構成になっている。プロジェクト及びモジュールは以下の図のような構成である。

① **OSK.Shop.Core**
システムの基本処理を行う

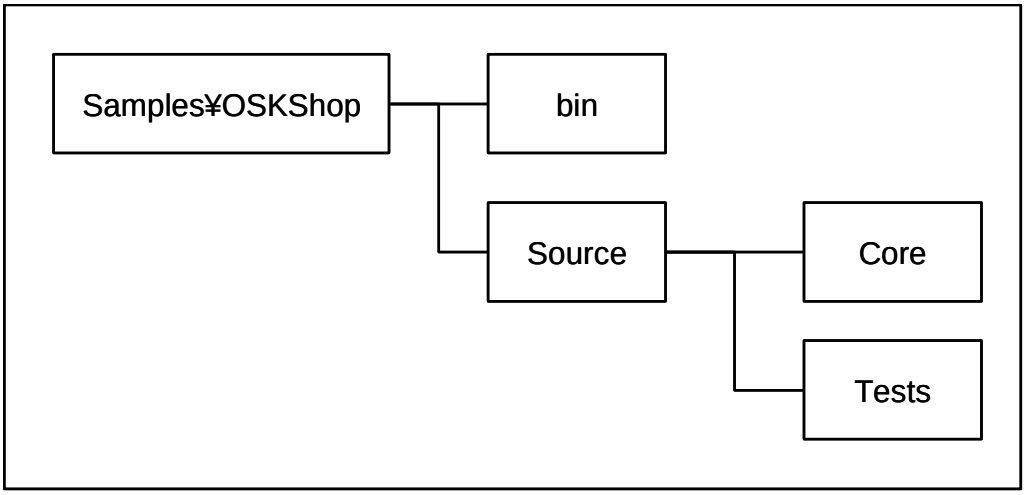
② **OSK.Shop.Tests**
単体テストを行う



# 2 テスト環境を整える

## 2.1 ルートディレクトリを作成する

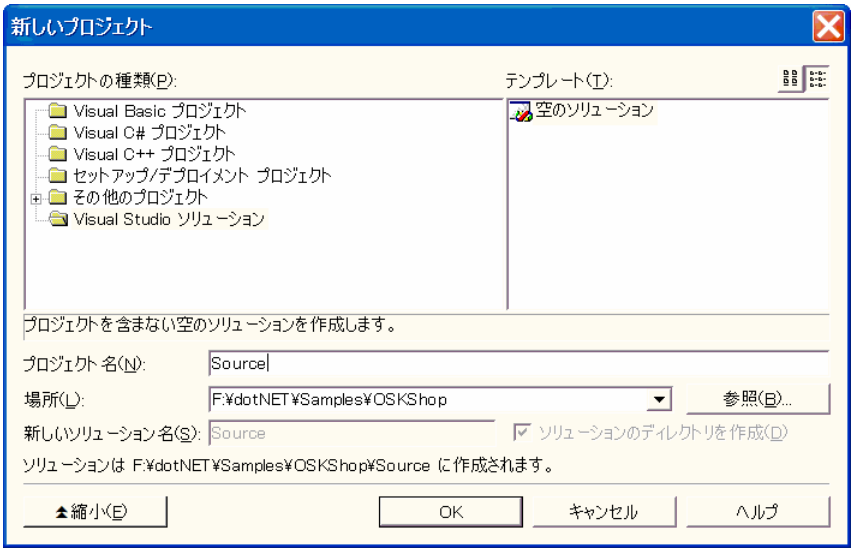
今回のサンプルでは、最終的に右の図のようなディレクトリ構成にする。ポイントは、完成したモジュール用のディレクトリと開発用のディレクトリの分離である。




まず、はじめに、ルートディレクトリと「bin」ディレクトリを作成する。その他のディレクトリは、VisualStudio.NET に作成を行わせる。

## 2.2 ソリューションを作成する

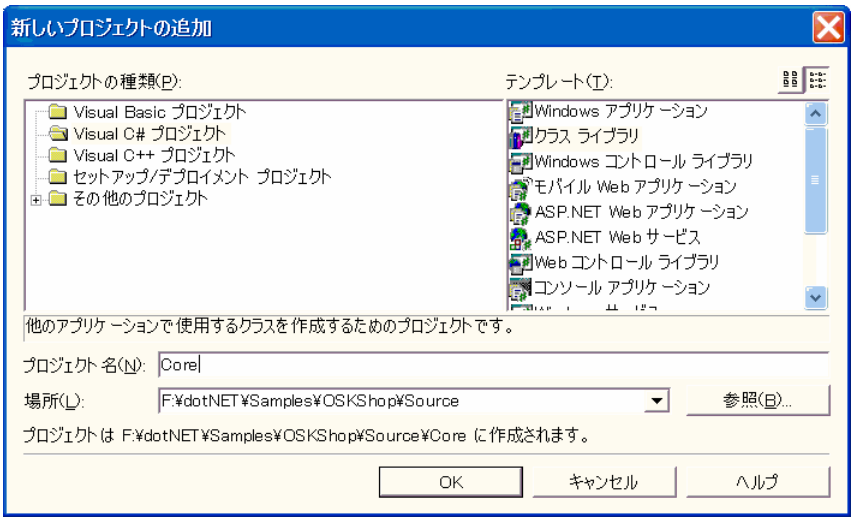
複数のプロジェクトをまとめるソリューションを作成する。まず、メニューから「ファイル」-「新規作成」-「空のソリューション」を選択する。「新しいプロジェクト」のダイアログが表示されたら、「VisualStudio ソリューション」を選択し、プロジェクト名「Source」と入力して実行する（これは、ソリューションファイルを Source ディレクトリに作成するためである）。その後、プロジェクト名を「Source」から「OSKShop」へ変更する。



## 2.3 基本部分のプロジェクトを作成する

今回のシステムの基本となるクラスライブラリのプロジェクトを作成する。ここでは、プロジェクト名を「Core」で作成する。まず、メニューから「ファイル」-「プロジェクトの追加」-「新しいプロジェクト」を選択する。「新しいプロジェクト」のダイアログが表示されたら、「Visual C# プロジェクト」と「クラス ライブラリ」を選択し、プロジェクト名を入力して実行する。



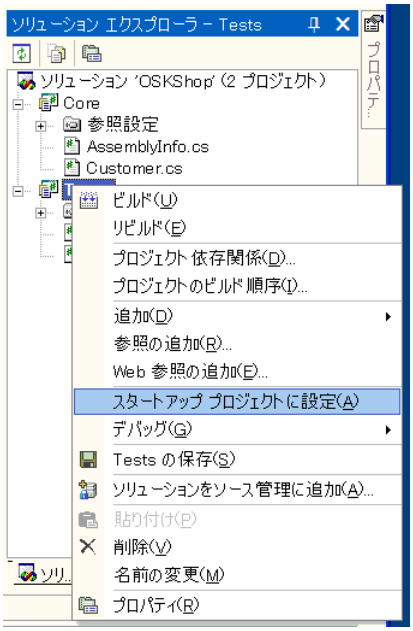

メニューの「プロジェクト」-「プロパティ」を選択して、プロジェクトのプロパティウィンドウを表示する。アセンブリ名を「OSK.Shop.Core」に変更する。

## 2.4 テストのプロジェクトを作成する

テストクラス用のプロジェクトを作成する。ここではプロジェクト名を「Tests」で作成する。まず、メニューから「ファイル」-「プロジェクトの追加」-「新しいプロジェクト」を選択する。「新しいプロジェクト」のダイアログが表示されたら、「Visual C# プロジェクト」と「クラス ライブラリ」を選択し、プロジェクト名を入力して実行する。作成されたプロジェクトのプロパティウィンドウで、アセンブリ名を「OSK.Shop.Tests」に変更する。

## 2.5 スタートアッププロジェクトの設定を行う

ソリューションを選択しプロパティウィンドウを表示して、「Tests」プロジェクトをスタートアッププロジェクトに設定する[2]。




[2] ソリューションエクスプローラのプロジェクト名上でマウスの右ボタンを押下して、メニューから指定することも可能。

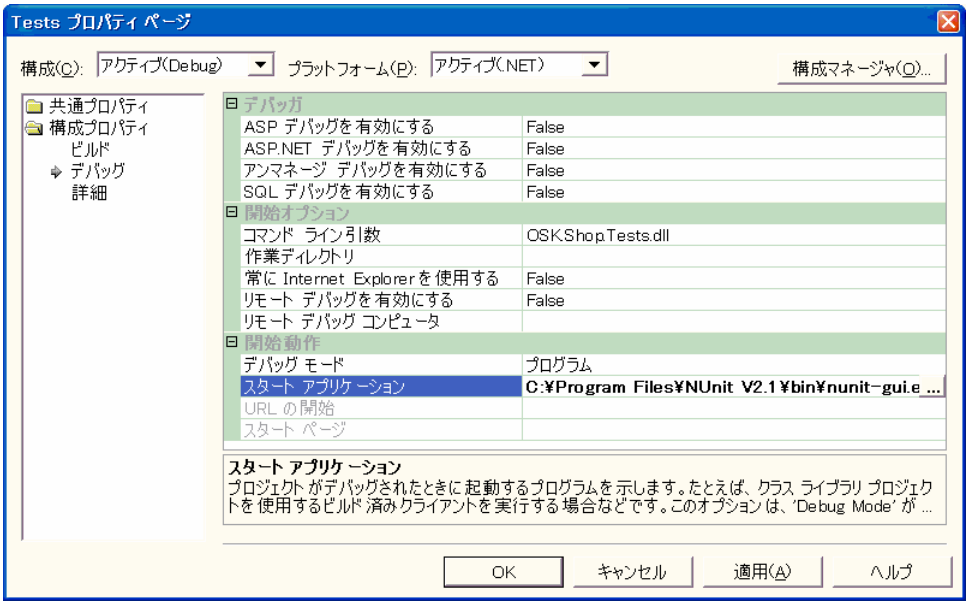
 

## 2.6 テストプロジェクトのプロパティを設定する

テスト用プロジェクトを選択し、メニューから「プロジェクト」-「プロパティ」を選択する。表示されたプロパティページで、デバッグ関連の指定を行う。デバッグモードをプログラムに設定し、スタートアプリケーションとして「nunit-gui.exe」を指定する。次に、コマンドライン引数としてテスト用のアセンブリ名を「DLL 名」の形式で指定する。

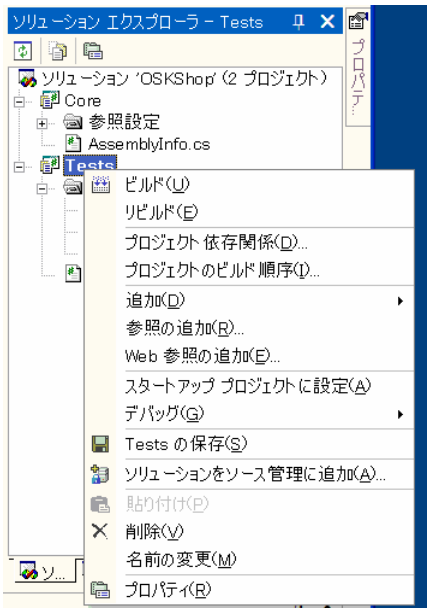
今回の例では、以下の様に設定する。

OSK.Shop.Tests.dll



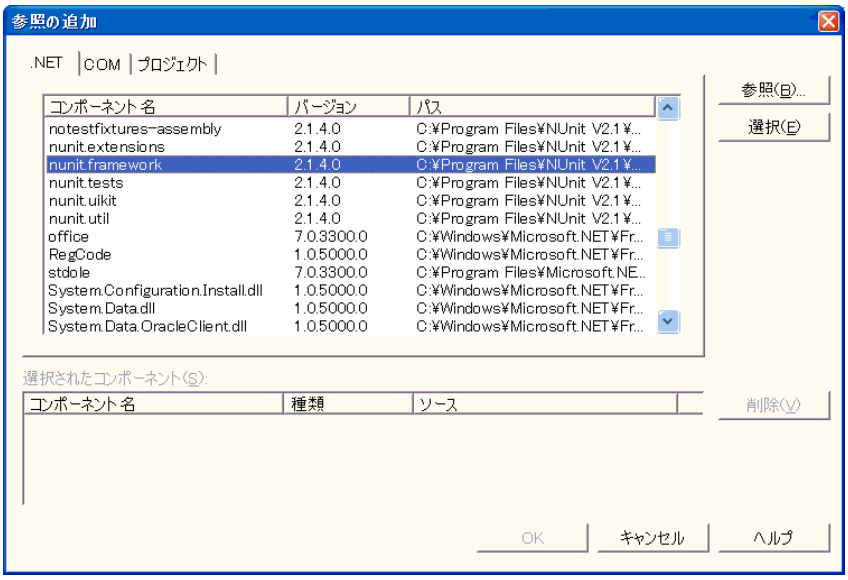

## 2.7 参照設定を行う

テストプロジェクトからは、NUnit とテスト対象のプロジェクトに対して参照設定を行う必要がある。まず、メニューから「プロジェクト」-「参照の追加」を選択する。

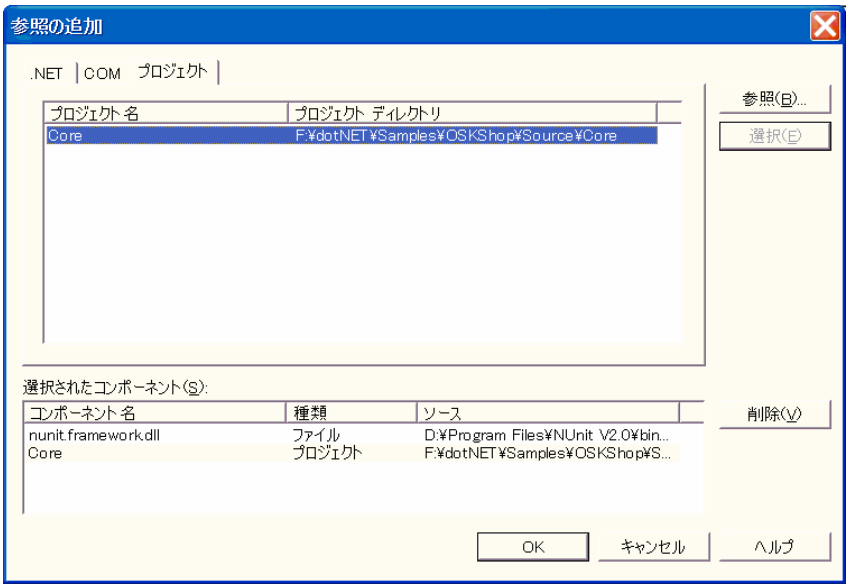


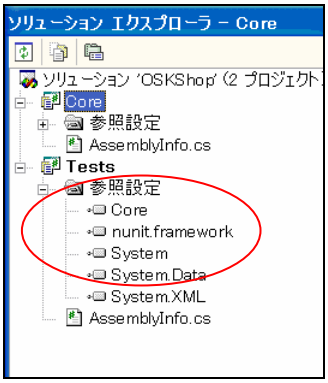
「参照の追加」ダイアログが表示されたら、「.NET」タグの「nunit.frameworkl」を選択し、選択ボタンを押す。




次に、「プロジェクト」タブを選択して「Core」のプロジェクトをダブルクリックして選択に追加する。




この作業によって、参照設定が正常に行われれば、ソリューション エクスプローラ ウィンドウが以下の様な表示になる。



## 2.8 クラスの作成を行う

顧客に関する機能を提供するクラスである Customer クラスを作成する。デフォルトで用意されているクラス「Class1.cs」のファイル名を「Customer.cs」に変更する。その後、クラス名も変更を行う。以下のソースは、変更直後のものである。

**Customer.cs**

```
 1:  using System;
 2:  namespace OSK.Shop.Core
 3:  {
 4:      /// <summary>
 5:      ///    顧客
 6:      /// </summary>
 7:      public class Customer
 8:      {
 9:          /// <summary>
10:          ///    コンストラクタ
11:          /// </summary>
12:          public Customer()
13:          {
14:          }
15:      }
16:  }
```

顧客 ID と名前およびポイントのプロパティを追加する。値を保持する変数の定義とプロパティを公開するためのプログラムを記述する。このとき、プロパティは取得のみとする。また、ID と名前を初期設定するコンストラクタを作成する[3]。

**Customer.cs**

```
17:  using System;
18:  namespace OSK.Shop.Core
19:  {
20:      /// <summary>
21:      ///    顧客
22:      /// </summary>
23:      public class Customer
24:      {
25:          protected int id;
26:          protected string name;
27:          protected int point = 0;
28:          /// <summary>
29:          ///    コンストラクタ
30:          /// </summary>
31:          public Customer(){}
32:
33:          /// <summary>
34:          ///    プロパティの初期化を行うコンストラクタ
35:          /// </summary>
36:          public Customer(int id, string name )
37:          {
38:                this.id = id;
39:                this.name = name;
40:          }
41:
42:          /// <summary>
43:          ///    プロパティ ID の取得
44:          /// </summary>
45:          public int ID
46:          {
47:                get { return id; }
48:          }
49:
50:          /// <summary>
51:          ///    プロパティ Name の取得
52:          /// </summary>
53:          public string Name
54:          {
55:                get { return name; }
56:          }
57:
58:          /// <summary>
59:          ///    プロパティ Point の取得
60:          /// </summary>
61:          public int Point
62:          {
63:                get { return point;}
64:          }
65:      }
66:  }
```

[3] ここの作業は、テストファーストではないが、プロパティの取得だけであることと、今後の作業が行いやすいのであえてプログラムを最初に実装している。

## 2.9 テストケースクラスの作成を行う

Customer クラスに関するテストを行うクラスである CustomerTestCase クラスを作成する。デフォルトで用意されているクラス「Class1.cs」のファイル名を「CustomerTestCase.cs」に変更する。その後、クラス名も変更を行う。テストケース用のクラスで最低限必要な以下の実装を行う。

### ① NUnit の利用を宣言する

### ② クラスに属性を付加する

クラスの属性は、[TestFixture]である。

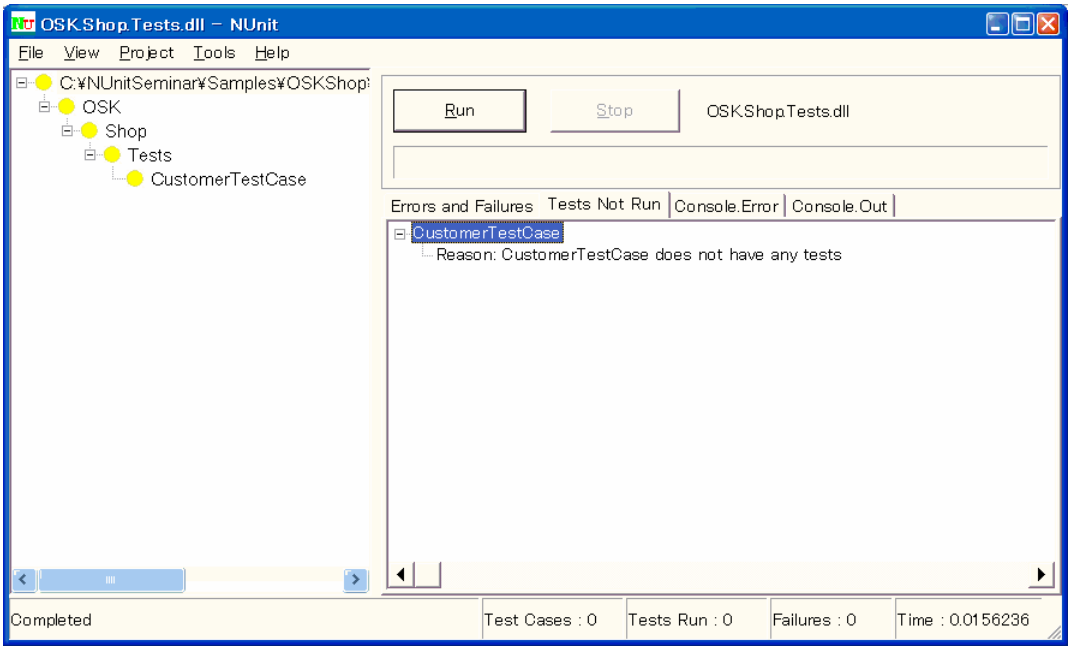
実装後のソースは、以下のとおり。

**CustomerTestCase.cs**

```
67:  using System;
68:
69:  using NUnit.Framework;
70:
71:  using OSK.Shop.Core;
72:
73:  namespace OSK.Shop.Tests
74:  {
75:      /// <summary>
76:      /// Customer クラスをテストするクラス
77:      /// </summary>
78:      [TestFixture]
79:      public class CustomerTestCase
80:      {
81:      }
82:  }
```

## 2.10 ビルド及び実行を行う

以上の設定によってビルドと実行が可能になる。まだ、テストプログラムを全く作成していないので、テストツリーと Tests Not Run のタグに以下の様なワーニングメッセージが表示される。



## 2.11 テストをコーディングする

テストの処理は、前処理、後処理と実テスト処理に分割される。各処理は、一般的に以下のような処理を行う。

**① 前処理 － SetUp**

[SetUp]属性を付加する。テストするクラスのインスタンスを作成して、確保を行う。

**② 後処理 － TearDown**

[TearDown]属性を付加する。テストするクラスのインスタンスを解放する。

**③ 実テスト処理 － Test**

[Test]属性を付加した Public メソッドを記述する。テストは、予測される値と実測値を AssertEquals メソッドで比較するのが最も一般的な方法である。当然、クラスの状態を判断して、Assert を発行することも可能である。この時、標準コンソールにメッセージを出力すると、テストの実行時の内容などを確認することができる。

**CustomerTestCase.cs**

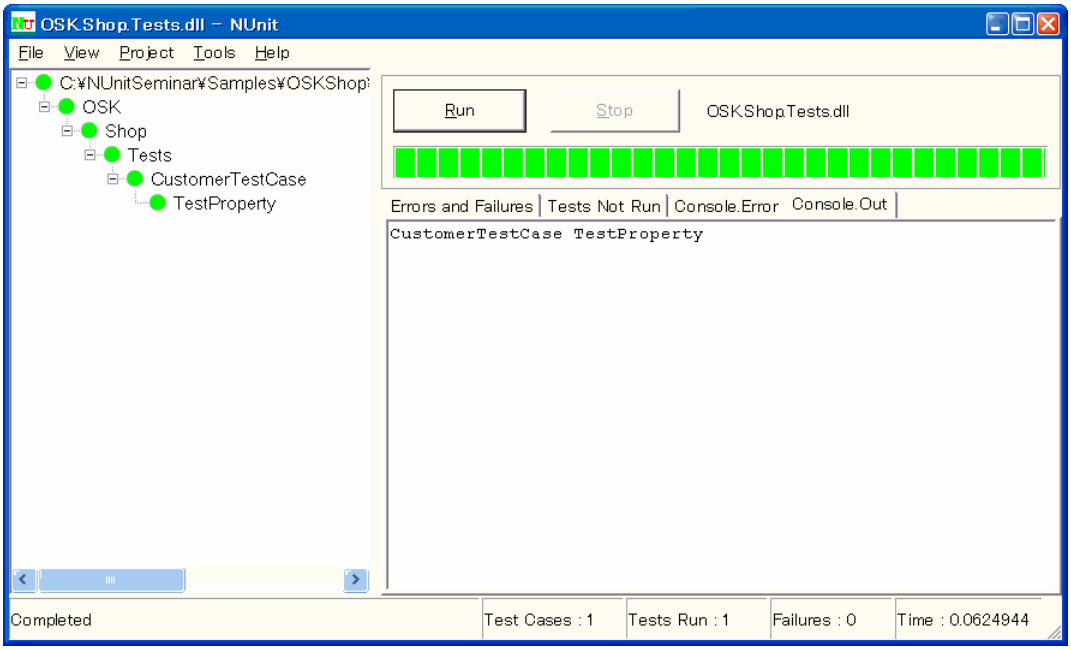
```
83:     [TestFixture]
84:     public class CustomerTestCase
85:     {
86:         /// <summary>
87:         ///    テストを行うクラスのインスタンス
88:         /// </summary>
89:         private Customer terget;
90:
91:         /// <summary>
92:         ///    テストの準備処理
93:         /// </summary>
94:         [SetUp]
95:         protected void SetUp()
96:         {
97:             terget = new Customer( 1, "OSK");
98:         }
99:
100:        /// <summary>
101:        ///    テストの終了処理
102:        /// </summary>
103:        [TearDown]
104:        protected void TearDown()
105:        {
106:            terget = null;
107:        }
108:
109:        /// <summary>
110:        ///    プロパティのテスト
111:        /// </summary>
112:        [Test]
113:        public void TestProperty()
114:        {
115:            Console.WriteLine("CustomerTestCase TestProperty");
116:            Assert.AreEqual( 1, terget.ID );
117:            Assert.AreEqual( "OSK", terget.Name );
118:            Assert.AreEqual( 0, terget.Point );
119:        }
120:    }
```

標準コンソールへの出力

### 2.12 テストの実行を行う

テストモジュールの実装が終了したら、Run ボタンでテストの実行を行う。正常にテストが終了すれば、以下のような緑のバーが表示される。また、標準出力に出力したメッセージは、「Console Out」タグに表示される。




## 3 テストファーストによるクラスの開発

ここからテストファーストによる、クラスの開発手順を説明する。

### 3.1 メソッドを定義する

最初にテストを行うクラスにメソッドを追加する。今回は、商品の種類と空のメソッドを実装する。

**Customer.cs**

```
121:  using System;
122:  namespace OSK.Shop.Core
123:  {
124:      /// <summary>
125:      ///     商品種類の定義
126:      /// </summary>
127:      public enum ProductType
128:      {
129:          Regular,Special,Inexpensive
130:      }
131:
132:      /// <summary>
133:      ///     顧客を管理する
134:      /// </summary>
135:      public class Customer
136:      {
中略
137:          /// <summary>
138:          ///     ポイントの追加
139:          /// </summary>
140:          public virtual void AddPoint( ProductType type, int price )
141:          {
142:          }
143:      }
```

空のメソッド

## 3.2 テストを実装する

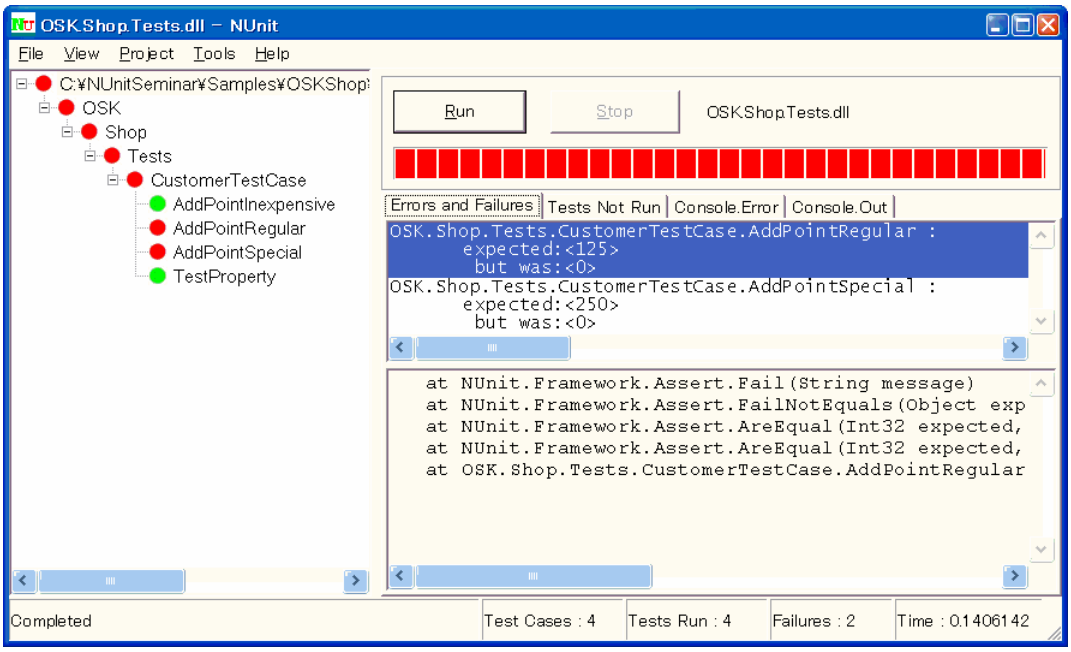
今回は、商品の種類ごとに、ポイントの追加をテストするメソッドを作成する。テストは、特定の価格のポイントを追加して、追加されるべき価格と、追加前の価格と追加後の価格の差額を比較する。

**CustomerTestCase.cs**

```
144:          /// <summary> ポイント追加テスト </summary>
145:          [Test]
146:          public void AddPointRegular()
147:          {
148:                Console.WriteLine("CustomerTestCase AddPointRegular");
149:                int oldPoint;
150:                int price = 2500;
151:                oldPoint = terget.Point;
152:                terget.AddPoint( ProductType.Regular, price);
153:                Assert.AreEqual( price * 5 / 100, terget.Point - oldPoint );
154:          }
155:
156:          /// <summary> ポイント追加テスト </summary>
157:          [Test]
158:          public void AddPointSpecial()
159:          {
160:                Console.WriteLine("CustomerTestCase AddPointSpecial");
161:                int oldPoint;
162:                int price = 2500;
163:                oldPoint = terget.Point;
164:                terget.AddPoint( ProductType.Special , price);
165:                Assert.AreEqual( price * 10 / 100, terget.Point - oldPoint );
166:          }
167:
168:          /// <summary> ポイント追加テスト </summary>
169:          [Test]
170:          public void AddPointInexpensive()
171:          {
172:                Console.WriteLine("CustomerTestCase AddPointInexpensive");
173:                int oldPoint;
174:                int price = 2500;
175:                oldPoint = terget.Point;
176:                terget.AddPoint( ProductType.Inexpensive , price);
177:                Assert.AreEqual( 0, terget.Point - oldPoint );
178:          }
179:    }
```

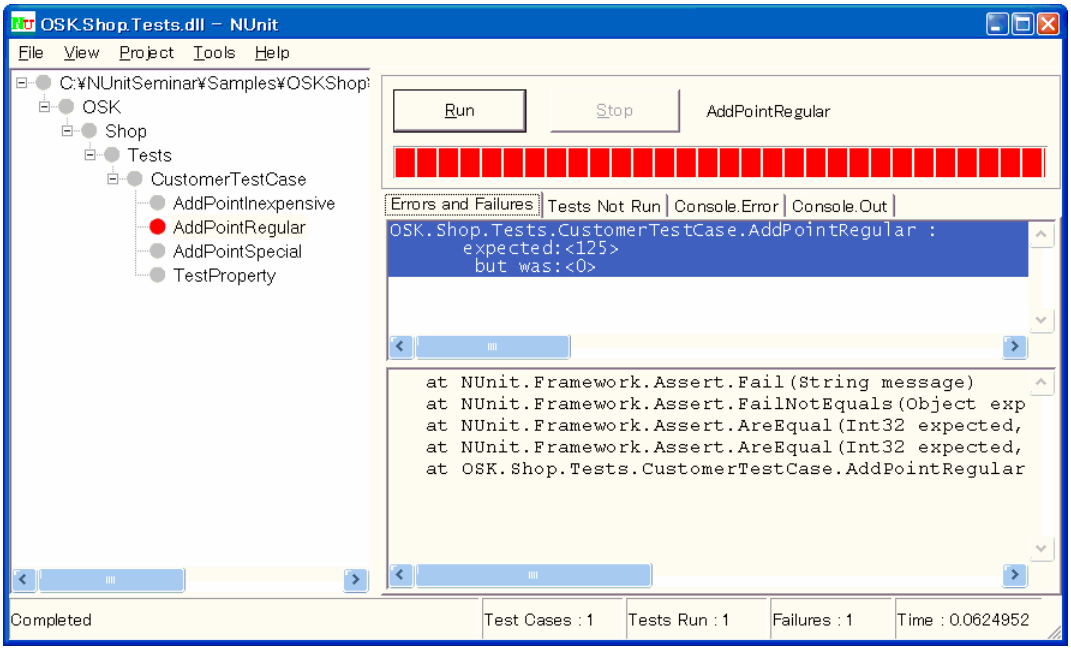
## 3.3 実行してエラーを確認する

テストの作成が終わったらプログラムを実行して、エラーが発生することを確認する。ここで面白いのは、特価商品はポイントが追加されない仕様であるため、空のメソッドであっても正常にテストが終了する。



また、NUnit では、テストのツリーリスト内のテストを選択し、Run ボタンを押すことで、任意のテストを実行することができる。




## 3.4 クラスを実装する

エラーが出なくなるまで、実装を行う。

## 3.5 作業を繰り返す

新たにメソッドの追加が必要になったり、新たなテストが必要になったら、上述の作業を繰り返す。例えば、ありえない価格の商品のエラーチェックなどが必要であれば、随時テストを追加する。

# 第3章 NUnitリファレンス

NUnitを利用する場合に、最低限必要な属性とクラスのリファレンスである。

## 1 属性

### 1.1 TestFixture

クラスがテストクラスであることを設定する属性である。

**【例】**

```
[TestFixture]
public class CustomerTestCase
```

### 1.2 Test

メソッドがテストメソッドであることを設定する属性である。

**【例】**

```
[Test]
public void AddPointRegular()
```

### 1.3 SetUp

メソッドがテストの前処理であることを設定する属性である。

**【例】**

```
[SetUp]
public void SetUp ()
```

### 1.4 TearDown

メソッドがテストの後処理であることを設定する属性である。

**【例】**

```
[TearDown]
public void TearDown ()
```

### 1.5 Suite

プロパティがテストのスイート(組)であることを設定する属性である。

**【例】**

```
[Suite]
public static TestSuite Suite
```

### 1.6 ExpectedException

テストメソッドが指定した例外を発行したらテストは成功する。

**【引数】**

***[入力] ExceptionType***
発行される例外クラス。

**【例】**

```
[ExpectedException(typeof(ArgumentException))]
public void AddPointRegular()
```

### 1.7 Ignore

クラスやメソッドを一時的に無効にすることを設定する属性である。

【引数】

*[入力] reason*
表示するメッセージ。（一般的には、無効にする理由。）

【例】

```
[Ignore("無効にする")]
public class CustomerTestCase
```

### 1.8 TestFixtureSetUp

テストクラスでクラス単位の初期化処理メソッドであることを宣言する。

【例】

```
[TestFixtureSetUp]
public void TestFixtureSetUp()
```

### 1.9 TestFixtureTearDown

テストクラスでクラス単位の終了処理メソッドであることを宣言する。

【例】

```
[TestFixtureTearDown]
public void TestFixtureTearDown()
```

## 2 クラス

### 2.1 Assert

#### 2.1.1 概要

診断を行うためのクラスである。

【宣言】

```
public class Assert
```

#### 2.1.2 プロパティ

なし

#### 2.1.3 メソッド

① IsTrue

条件が成り立ったら、テストは成功とする。失敗した場合は、エラー結果を記録する。

【引数】

*[入力] condition*
条件。

*[入力] message*
表示するメッセージ。

② **IsFail**

テストが必ず失敗する。失敗した場合は、エラー結果を記録する。

**【引数】**

***[入力] message***
表示するメッセージ。

③ **AreEqual**

予測値と実測値が同じだったら、テストは成功とする。失敗した場合は、エラー結果を記録する。

**【引数】**

***[入力] expected***
予測値。
***[入力] actual***
実測値。
***[入力] deltal***
実数の場合で比較したときに有効となる幅の値。
***[入力] message***
表示するメッセージ。

④ **IsNotNull**

オブジェクトがヌルでなかったら、テストは成功とする。失敗した場合は、エラー結果を記録する。

**【引数】**

***[入力] object***
オブジェクト。
***[入力] message***
表示するメッセージ。

⑤ **IsNull**

オブジェクトがヌルだったら、テストは成功とする。失敗した場合は、エラー結果を記録する。

**【引数】**

***[入力] object***
オブジェクト。
***[入力] message***
表示するメッセージ。

⑥ **AreSame**

予測したオブジェクトと実際のオブジェクトが同じだったら、テストは成功とする。失敗した場合は、エラー結果を記録する。

**【引数】**

***[入力] expected***
予測値。
***[入力] actual***
実測値。
***[入力] message***
表示するメッセージ。