NUnit トレーニング テキスト

C# or Visual Basic によるテストファースト・プログラミング

Rev.2004.8.25

履歴

◆ Rev.2004.8.2
NUnit2.1.4 への対応

目次

Session1　テスティング･フレームワークを利用する

Overview

Visual Studio.NET（以下 VS.NET）は、.NET アプリケーションを開発するための統合開発環境です。非常に強力で、Windows アプリケーションから Web アプリケーション、XML Web サービス、大規模な分散システムまで様々なアプリケーション開発をサポートしています。VS.NET に、単体テストを標準化し、効率的な開発を支援するテスティング・フレームワークである NUnit を連動させることで、快適な開発を行うことができます。今回対象としているシステムは、一度作ったら使い捨て的なものや個人の趣味で開発しているシステムを対象にしているのではなく、一定の期間において機能拡張やユーザ要求の対応などを行うシステムです。

この Session では、テスティング・フレームワークを利用するプロジェクトの作成手順や VS.NET のソリューション構成について、解説します。

この Session では、おもちゃ屋のシステムを例題として、作業を進めます。

Goal

- ❒ NUnit が利用できる
- ❒ テストファーストを体感する

Step 1　テスティングフレームワークをインストールする

①　NUnit を入手する

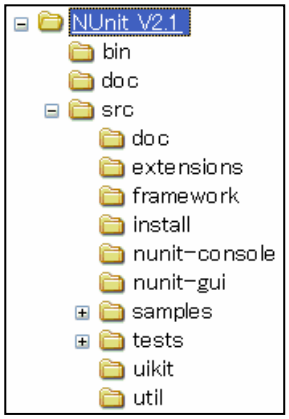
NUnit Ver2.1 をサイト、「http://sourceforge.net/projects/nunit/」からダウンロードします。
ダウンロードするファイルは、「NUnit-V2.1.4.msi」です。（2004 年 7 月時点では、フリーソフトです。また、ねんのためウィルスチェックを行うことをお勧めします）

②　インストールを実行する

ファイル名でお分かりだと思いますが、Windows のインストールパッケージになっています。WindowsXP であれば、クリックし、インストール・ウィザードの指示に従うだけで、インストールを行うことができます。インストールを行うと、実行モジュールと単体テスト用のフレームワーク、各種ドキュメント、ソースファイル、サンプルなどがコピーされます。

Hints

NUnit をデフォルトの設定でインストールを行うと、ＯＳをインストールしたドライブの「¥Program Files」ディレクトリに、右のようなディレクトリ構成でインストールが行われます。



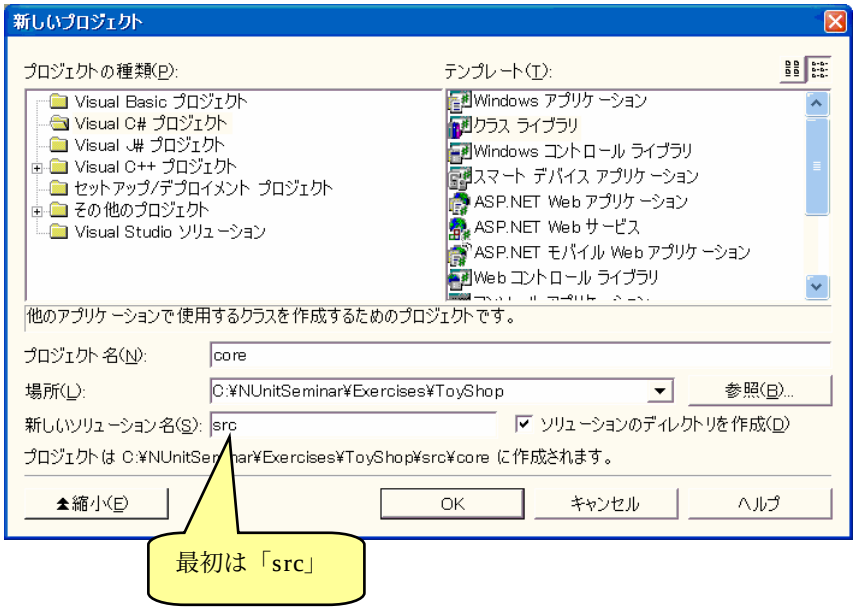
Step 2　新規プロジェクトを作成する

①　ルートディレクトリを作成する

「エクスプローラ」などを使用してルートディレクトリを作成します。具体的には、「¥NUnitSeminar¥Exercises」内に「ToyShop」作成します。

②　新規プロジェクトを作成する

「ファイル」メニューの「新規作成」から「プロジェクト」を選択して、新規プロジェクトの作成を開始します。「クラスライブラリ」テンプレートを使用して、次の設定でプロジェクトを作成します。



C#の場合

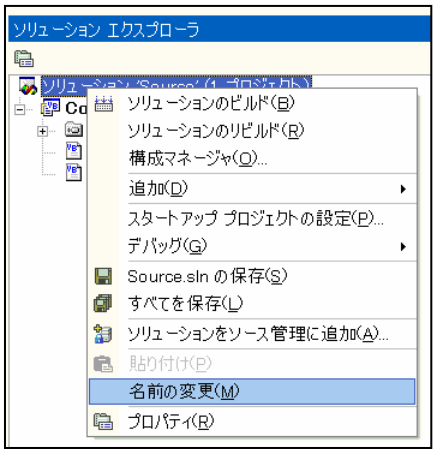
設定項目	設定内容
プロジェクトの種類	C# プロジェクト
テンプレート	クラスライブラリ
プロジェクト名	core
場所	¥NUnitSeminar¥Exercises¥ToyShop
新しいソリューション	src

VisualBasic の場合

設定項目	設定内容
プロジェクトの種類	Visual Basic プロジェクト
テンプレート	クラスライブラリ
プロジェクト名	core
場所	¥NUnitSeminar¥Exercises¥ToyShop
新しいソリューション	src

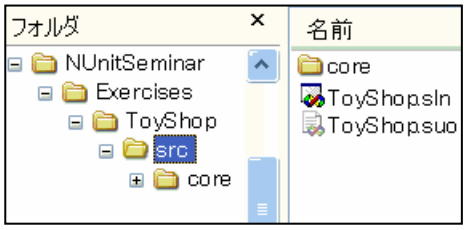
VS.NET では、最上位にソリューションがあります。その下には、複数のプロジェクトを管理することができます。プロジェクトには、1つのディレクトリや仮想ディレクトリ（Web アプリケーションの場合）に割り当てられます。また、新規作成で指定した、プロジェクトの種類やテンプレートは、作成した後に変更することはできません。

③ ソリューション名を変更する



「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンでコンテキストメニューを表示し、名前の変更を選択します。プロジェクト名を「ToyShop」に変更します。
この作業は、ソリューション名とディレクトリ名を異なったものにするためです。

④ ディレクトリを確認する



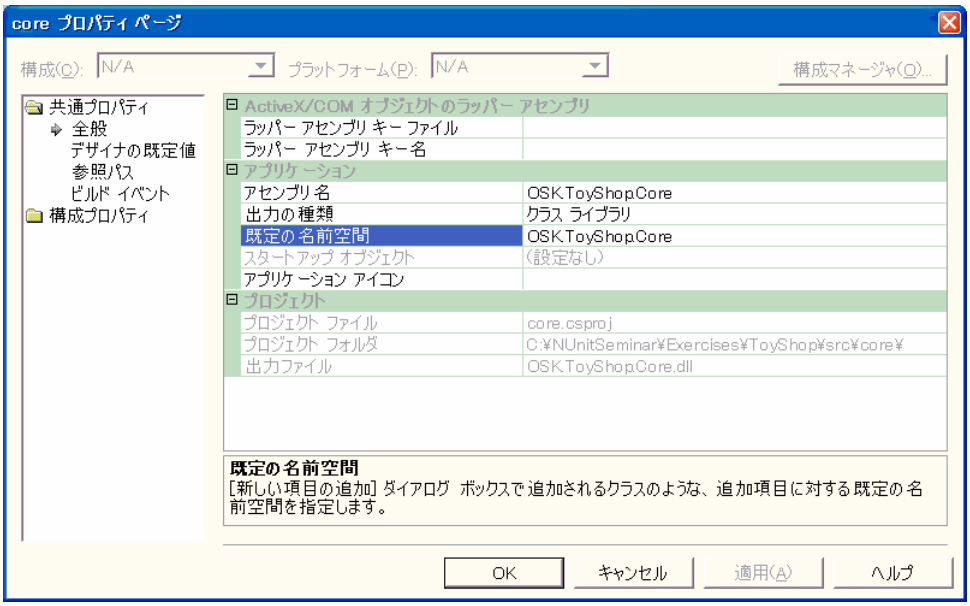
デフォルトで作成される Class1.cs(VB の場合は、Class1.vb)は削除します。その結果、ディレクトリ構造は右のようになります。Source ディレクトリ配下にある、「ToyShop.sln」ファイルがソリューションファイルです。また、core ディレクトリ配下にある「core.csproj」（VB の場合は、「core.vbproj」）ファイルがプロジェクトファイルです。

⑤ プロジェクトのプロパティを設定する

「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Core プロジェクトのコンテキストメニューを表示し、プロパティを選択します。以下の設定を変更します。

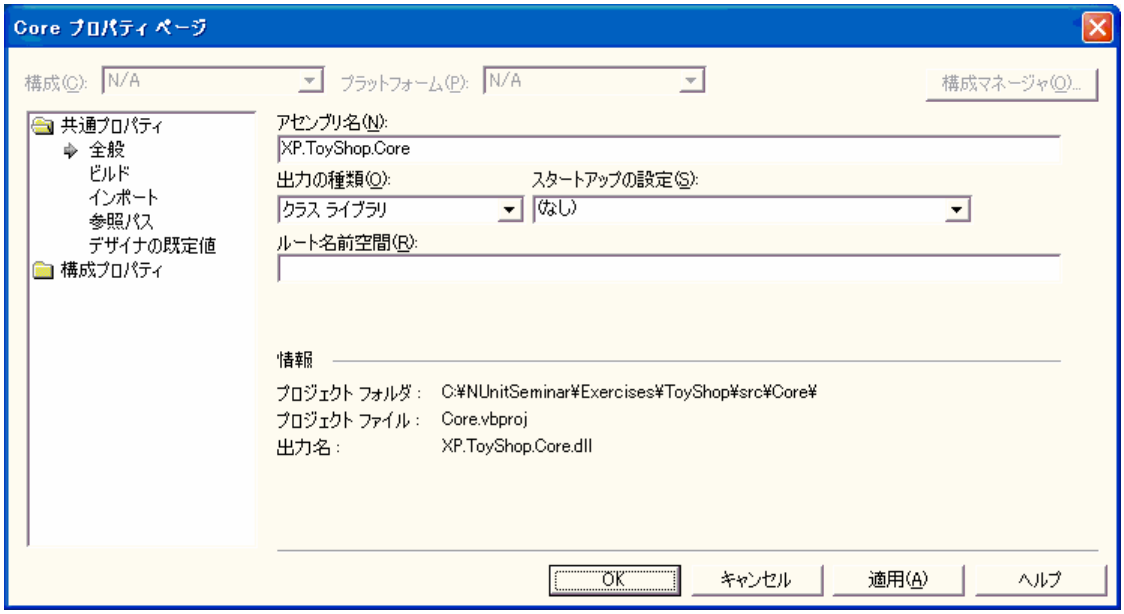
C#の場合

設定項目	設定内容
アセンブリ名	OSK.ToyShop.Core
既定の名前空間	OSK.ToyShop.Core



VisualBasic の場合

設定項目	設定内容
アセンブリ名	OSK.ToyShop.Core
ルート名前空間	[空白]



Hints

Visual Basic では、名前空間を意識しないで作業できるように、ルート名前空間を設定することができます。(C#の場合は、既定の名前空間で、まったく意味が異なります) しかし、複数の名前空間を使用する場合、かえって不整合が発生しますので、ルート名前空間は[空白]にして、名前空間を使用することをお勧めします。

Step 3　テストプロジェクトを追加する

①　テスト用のプロジェクトを追加する

「ファイル」メニューの「プロジェクトの追加」から「新規プロジェクト」を選択して、新規プロジェクトの作成を開始します。
「プロジェクト」の「クラスライブラリ」テンプレートを使用して、次の設定でプロジェクトを作成します。

C#の場合

設定項目	設定内容
プロジェクトの種類	C#プロジェクト
テンプレート	クラスライブラリ
プロジェクト名	tests
場所	¥NUnitSeminar¥Exercises¥ ToyShop¥src

デフォルトの Class1.cs は、削除します。

VisualBasic の場合

設定項目	設定内容
プロジェクトの種類	Visual Basic プロジェクト
テンプレート	クラスライブラリ
プロジェクト名	tests
場所	¥NUnitSeminar¥Exercises¥ ToyShop¥src

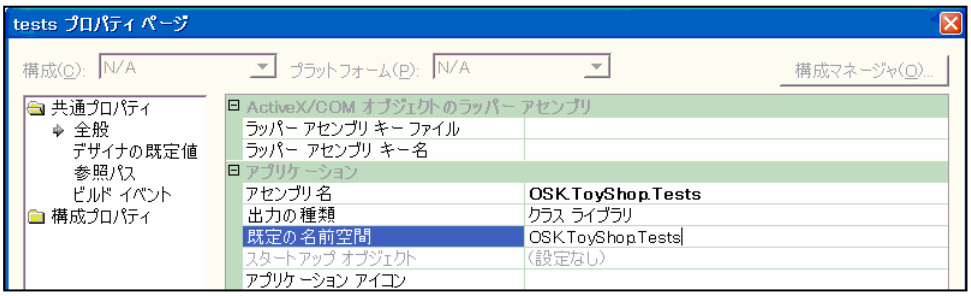
デフォルトの Class1.vb は、削除します。

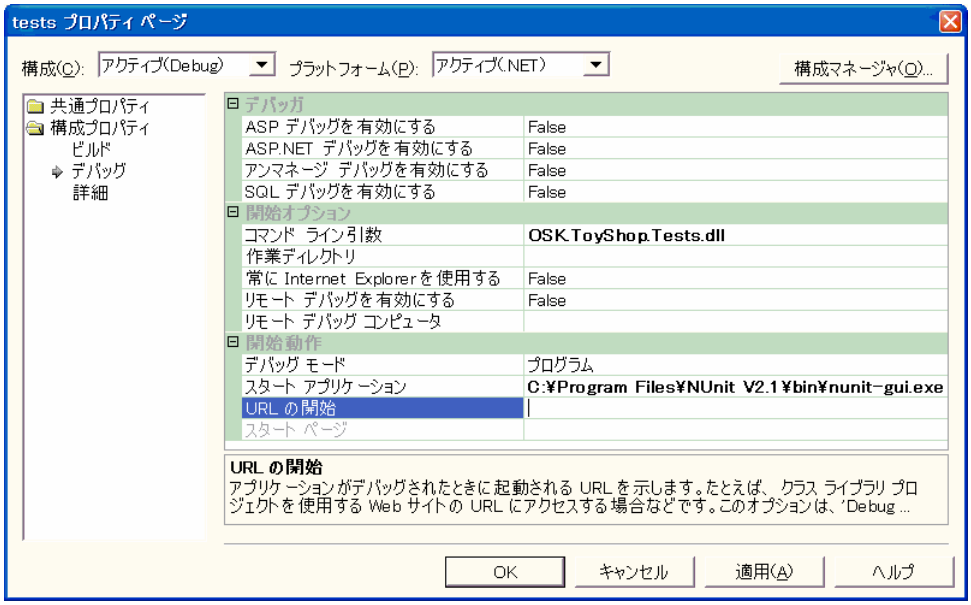
② プロジェクトのプロパティを設定する

NUnit を使用して、単体テストを行う場合は、NUnit が提供している EXE を使用します。ここでは、開発中に利用するために用意されている「nunit-gui.exe」を使用します。また、コマンドライン引数で、テストを行う DLL を指定する必要があります。
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Tests プロジェクトのコンテキストメニューを表示し、プロパティを選択します。以下の設定を変更します。

C#の場合

プロパティ種別	設定項目	設定内容
全般	アセンブリ名	OSK.ToyShop.Tests
全般	既定の名前空間	OSK.ToyShop.Tests
デバッグ	デバッグ モード	プログラム[1]
デバッグ	外部プログラムの開始	¥Program Files¥NUnit V2.1¥bin¥nunit-gui.exe
デバッグ	コマンド ライン引数	OSK.ToyShop.Tests.dll

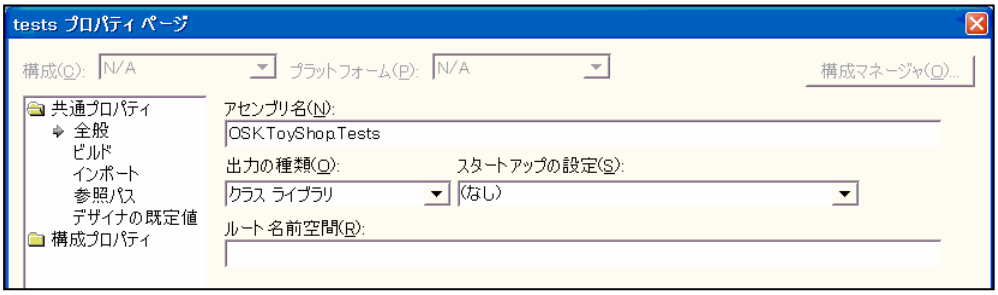


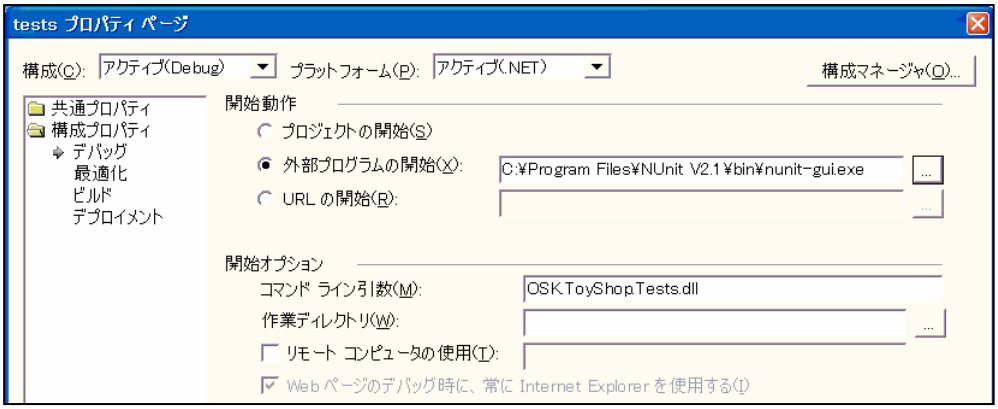


[1] デバッグモード変更後、適用ボタンをクリックする。

VisualBasic の場合

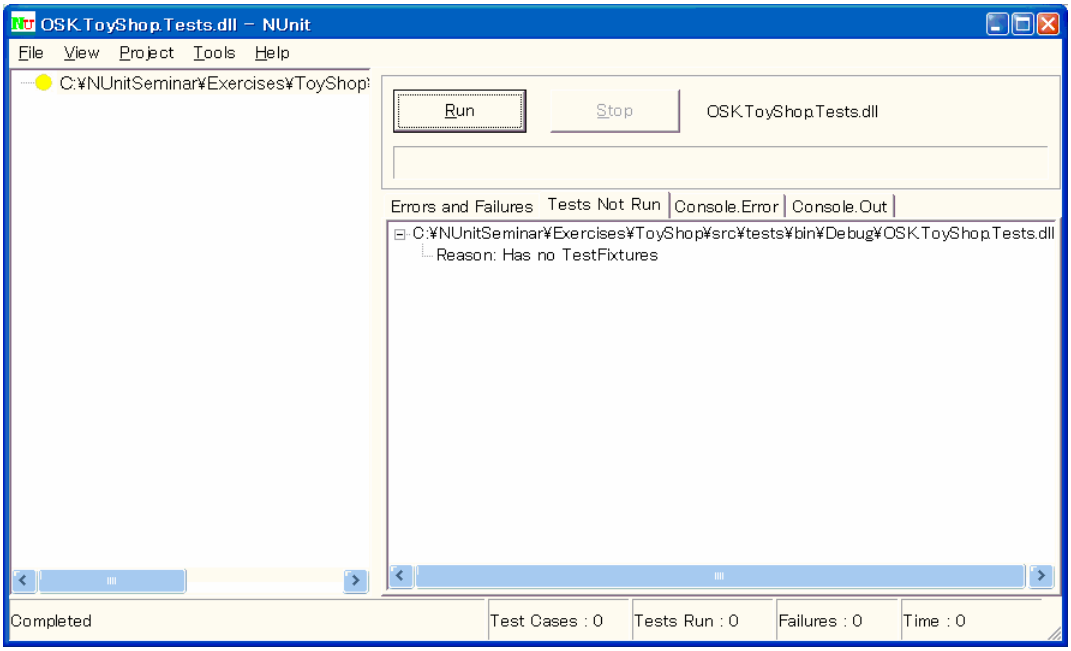
プロパティ種別	設定項目	設定内容
全般	アセンブリ名	OSK.ToyShop.Tests
全般	ルート名前空間	[空白]
デバッグ	外部プログラムの開始	¥Program Files¥NUnit V2.1¥bin¥nunit-gui.exe
デバッグ	コマンドライン引数	OSK.ToyShop.Tests.dll





③ ビルドと実行を行う

「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Tests プロジェクトのコンテキストメニューを表示し、「スタートアップ　プロジェクトに設定」を選択します。「ビルド」メニューの「ソリューションのビルド」を選択して、ビルドを行い、エラーが発生しないことを確認します。
「デバッグ」メニューの「開始」を選択して、テスト実行を行います。「nunit-gui.exe」のウィンドウの Run ボタンをクリックすると、以下ののような画面が表示されます。まだテストを作成していないのでテストが実行されない理由が表示されます。



Step 4　クラスとテストクラスを作成する

①　名前空間を決定する

.NET では、クラスを階層的にグループ化して管理を行っています。この階層のことを名前空間と呼びます。独自にクラスを作成する場合、独自の名前空間を用意する必要があります。名前空間によって、クラスが識別されますので、以下のような形式作成します。

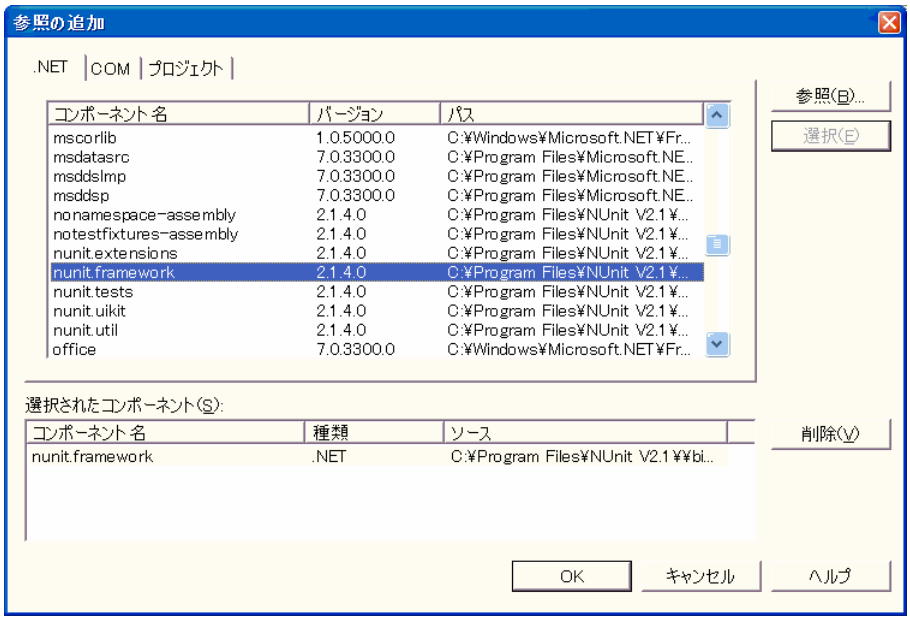
企業名（組織名）.テクノロジー名.機能名

この Session のサンプルでは、名前空間を「OSK.ToyShop.プロジェクト名」とします。

②　プロジェクト間の参照設定を行う

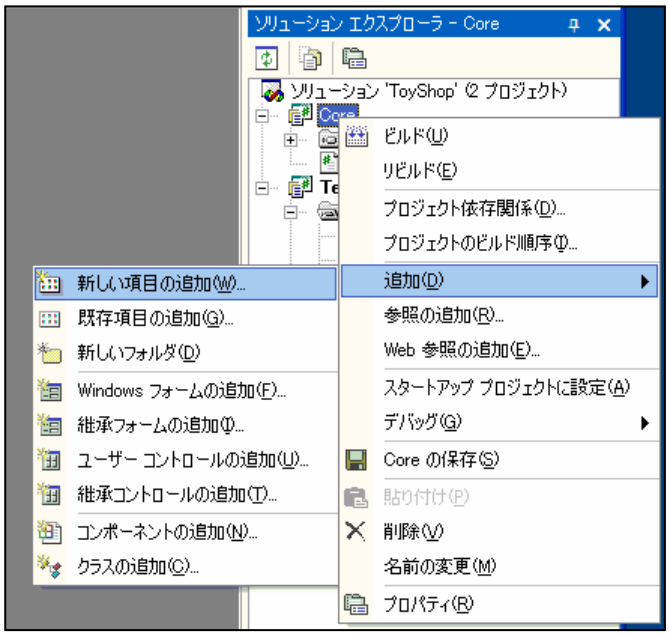
プロジェクト間の参照設定を行います。今回は、「tests」プロジェクトで「core」プロジェクトの参照設定を行います。
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Tests プロジェクトの参照設定コンテキストメニューを表示し、「参照」を選択します。

「参照の追加」ダイアログボックスの「プロジェクト」タグを選択して、選択ボタンを押します。選択されたコンポーネントのリストに追加されます。



③ クラスを作成する

おもちゃを Toy という名前で作成します。
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Core プロジェクトのコンテキストメニューを表示し、「追加」の「新しい項目の追加」を選択します。



表示された新しい項目の追加ダイアログボックスに以下の設定を行って、クラスを作成します。

設定項目	設定内容
カテゴリ	コード (選択しなくても、特に問題はない)
テンプレート	クラス
ファイル名	Toy.cs (VB の場合は、Toy.vb)

④ クラスの雛形を実装する

名前空間を指定して、Toy クラスのソースを作成します。クラスは、コンストラクタおよびプロパティを持っています。各実装はまだ行いません。

C#の場合

```
1:      using System;
2:      namespace OSK.ToyShop.Core
3:      {
4:              /// <summary>
5:              /// おもちゃクラス
6:              /// </summary>
7:              public class Toy
8:              {
9:                      public Toy( string name, int price )
10:                     {
11:                     }
12:                     public string Name
13:                     {
14:                             get { return ""; }
15:                     }
16:                     public int Price
17:                     {
18:                             get { return 0; }
19:                     }
20:             }
21:     }
```

VisualBasic の場合

```
22:    Imports System
23:    Namespace OSK.ToyShop.Core
24:        'おもちゃのクラス
25:        Public Class Toy
26:            Public Sub New(ByVal newName As String, ByVal newPrice As Integer)
27:
28:            End Sub
29:            Public ReadOnly Property Name() As String
30:                Get
31:
32:                End Get
33:            End Property
34:            Public ReadOnly Property Price() As Integer
35:                Get
36:
37:                End Get
38:            End Property
39:        End Class
40:    End Namespace
```

Hints

本来のテストファーストでは、クラスよりテストを先に作成しますが、VS.NET ではメソッドなどを空の状態で実装する方法も有効です。

⑤ テストクラスを作成する

Toy クラスをテストするクラスを ToyTestCase という名前で作成します。
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Tests プロジェクトのコンテキストメニューを表示し、「追加」の「新しい項目の追加」を選択します。表示された新しい項目の追加ダイアログボックスに以下の設定を行って、クラスを作成します。

設定項目	設定内容
カテゴリ	コード（選択しなくても、特に問題はない）
テンプレート	クラス
ファイル名	ToyTestCase.cs（VB の場合は、ToyTestCase.vb）

⑥ テストクラスを実装する

今回、ToyTestCase では、以下の項目について、テストを作成します。

- コンストラクタのテスト
  おもちゃの名前と値段を指定してコンストラクタ呼び出し、インスタンスが作成されたことを確認します。
- プロパティの設定と取得のテスト

クラスに属性として、TestFixture を付けます。同様に、テストメソッドにも属性として Test を付けます。最後に、テストプログラムを作成します。テストメソッドは、必ず以下の書式にする必要があります。

C#の場合

```
public void テストメソッド名（引数なし）
```

VisualBasic の場合

```
Public Sub テストメソッド名（引数なし）
```

以下のソースは、最終的な CircleTest のソースです。

C#の場合

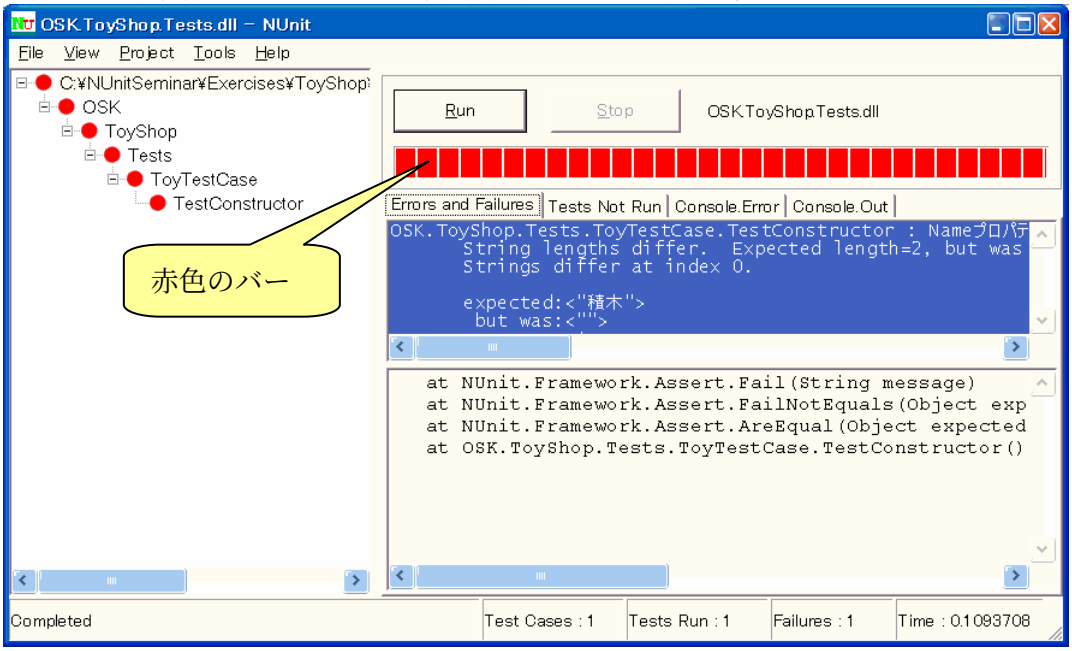
```
41:  using System;
42:  using OSK.ToyShop.Core;
43:  using NUnit.Framework;
44:  namespace OSK.ToyShop.Tests
45:  {
46:      /// <summary>
47:      /// Toy クラスのテストケースクラス
48:      /// </summary>
49:      [TestFixture]
50:      public class ToyTestCase
51:      {
52:          private Toy target;
53:          [SetUp]
54:          public void SetUp()
55:          {
56:              target = new Toy( "積木", 1000 );
57:          }
58:          [Test]
59:          public void TestConstructor()
60:          {
61:              Assert.IsNotNull(target,"インスタンスが作成されない");
62:              Assert.AreEqual("積木", target.Name, "Name プロパティ NG");
63:              Assert.AreEqual(1000, target.Price,"価格プロパティ NG");
64:          }
65:          [TearDown]
66:          public void TearDown()
67:          {
68:              target = null;
69:          }
70:      }
71:  }
```

VisualBasic の場合

```
72:  Imports System
73:  Imports OSK.ToyShop.Core
74:  Imports NUnit.Framework
75:  Namespace OSK.ToyShop.Tests
76:      <TestFixture()> _
77:      Public Class ToyTestCase
78:          Private target As Toy
79:          <SetUp()> _
80:          Public Sub SetUp()
81:              target = New Toy("積木", 1000)
82:          End Sub
83:          <Test()> _
84:          Public Sub TestConstructor()
85:              Assert.IsNotNull(target,"インスタンスが作成されない")
86:              Assert.AreEqual("積木", target.Name, "Name プロパティ NG")
87:              Assert.AreEqual(1000, target.Price,"価格プロパティ NG")
88:          End Sub
89:          <TearDown()> _
90:          Public Sub TearDown()
91:              target = Nothing
92:          End Sub
93:      End Class
94:  End Namespace
```

⑦ テストを実行する

テストプログラムを実装したら、テストを実行します。NUnit のウィンドウが表示されたら、Run ボタンを押して、テストを実行します。最初は、Toy クラスの何も実装されていないので、右のようにエラー（赤いバー）が表示されます。このエラーがなくなり、緑色のバーになるまで、実装を行うことになります。



最終的な Toy クラスのソースは、以下のようになります。

C#の場合

```
95:    using System;
96:    namespace OSK.ToyShop.Core
97:    {
98:            /// <summary>
99:            /// おもちゃクラス
100:           /// </summary>
101:           public class Toy
102:           {
103:                   public Toy( string name, int price )
104:                   {
105:                           this.name = name;
106:                           this.price = price;
107:                   }
108:                   private string name;
109:                   public string Name
110:                   {
111:                           get { return name; }
112:                   }
113:                   private int price;
114:                   public int Price
115:                   {
116:                           get { return price; }
117:                   }
118:           }
119:   }
```

VisualBasic の場合

```
120:  Imports System
121:  Namespace OSK.ToyShop.Core
122:      'おもちゃのクラス
123:      Public Class Toy
124:          Public Sub New(ByVal newName As String, ByVal newPrice As Integer)
125:              nameValue = newName
126:              priceValue = newPrice
127:          End Sub
128:          Private nameValue As String
129:          Public ReadOnly Property Name() As String
130:              Get
131:                  Return nameValue
132:              End Get
133:          End Property
134:          Private priceValue As Integer
135:          Public ReadOnly Property Price() As Integer
136:              Get
137:                  Return priceValue
138:              End Get
139:          End Property
140:      End Class
141:  End Namespace
```

Step 5　テストファーストで開発を行う

テストファーストで開発を行う場合、以下のようなリズムで開発を進めます。

- ◆ 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す。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（プログラムシナリオ）を明確にします。そして、正しく動作したことを想定される結果値あるいは結果と実際の動作を比較して判断する方法を決定します。
  良いテストが考えられない場合は、最初に判断方法から考えることで視点が変わりスムーズにテストを作成できる場合があります。

- ◆ クラスが公開するメソッドとプロパティを実装する

  決定した機能で必要になる、メソッドとプロパティの型を実装します。実際の処理は作成しません。

- ◆ テストを実装する

  クラスに、テストを実装します。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ことで行います。このテストを作成するときに、クラスが提供する機能の使いやすさなどのレビューを行います。処理の実装を行う前に、利用する側の視点で、クラスの機能を推敲することは、効果的です。

- ◆ 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します。クラスは、公開するメソッドとプロパティの型だけしか実装されていないので、テストは失敗します。

- ◆ クラスの実装を行う

  クラスの実装を行い、テストを実行します。テストが全てパスするまで実装を行います。テストが無事通過した時点が実装の終了となります。ペアプログラミングを行っている場合は、お菓子でささやかなお祝いをします。

① 新しい機能を実装する

「おもちゃ屋システム」に、レジの機能を追加します。レジのクラスは、Register という名前で作成します。作成する機能には、以下のものがあります。

- 複数のおもちゃの合計を計算する

② テストを考える

テストは、以下の手順で行います。

- 初期化を行う

  レジクラスのインスタンスを作成します。次に、正しく初期化されていることを確認します。
- 処理を行う

  おもちゃのインスタンスを数種類作成します。レジのクラスに、おもちゃクラスのインスタンスを渡して、金額を加算します。
- 動作を確認する

  想定される合計値と、レジクラスの合計値を比較します。
- 終了処理を行う

  レジクラスのインスタンスを解放します。

③ テストを実装する

テストを実装します。

C#の場合

```
142:  using System;
143:  using OSK.ToyShop.Core;
144:  using NUnit.Framework;
145:  namespace OSK.ToyShop.Tests
146:  {
147:          /// <summary>
148:          /// Register クラスのテストケースクラス
149:          /// </summary>
150:          [TestFixture]
151:          public class RegisterTestCase
152:          {
153:                  private Register target;
154:                  [SetUp]
155:                  public void SetUp()
156:                  {
157:                          target = new Register( );
158:                  }
159:                  [Test]
160:                  public void TestConstructor()
161:                  {
162:                          Assertion.AssertEquals( 0, target.Amount );
163:                  }
164:                  [Test]
165:                  public void TestAddToy()
166:                  {
167:                          Toy myToy = new Toy( "積木", 1000 );
168:                          Assertion.AssertEquals( 0, target.Amount );
169:                          target.Add( myToy );
170:                          Assertion.AssertEquals( 1000, target.Amount );
171:                          target.Add( myToy );
172:                          Assertion.AssertEquals( 2000, target.Amount );
173:                  }
174:                  [TearDown]
175:                  public void TearDown()
176:                  {
177:                          target = null;
178:                  }
179:          }
180:  }
```

VisualBasic の場合

```
181:  Imports System
182:  Imports OSK.ToyShop.Core
183:  Imports NUnit.Framework
184:  Namespace OSK.ToyShop.Tests
185:      <TestFixture()> _
186:      Public Class RegisterTestCase
187:          Private target As Register
188:          <SetUp()> _
189:          Public Sub SetUp()
190:              target = New Register()
191:          End Sub
192:          <Test()> _
193:          Public Sub TestConstructor()
194:              Assertion.AssertEquals(0, target.Amount)
195:          End Sub
196:          <Test()> _
197:          Public Sub TestAddToy()
198:              Dim myToy As New Toy("積木", 1000)
199:              Assertion.AssertEquals(0, target.Amount)
200:              target.Add(myToy)
201:              Assertion.AssertEquals(1000, target.Amount)
202:              target.Add(myToy)
203:              Assertion.AssertEquals(2000, target.Amount)
204:          End Sub
205:          <TearDown()> _
206:          Public Sub TearDown()
207:              target = Nothing
208:          End Sub
209:      End Class
210:  End Namespace
```

④ クラスを実装する

クラスを実装します。

C#の場合

```
211:  using System;
212:  namespace OSK.ToyShop.Core
213:  {
214:          /// <summary>
215:          /// Register のクラスです。
216:          /// </summary>
217:          public class Register
218:          {
219:                  public Register( )
220:                  {
221:                          amount = 0;
222:                  }
223:                  private int amount;
224:                  public int Amount   {
225:                          get { return amount; }
226:                  }
227:                  public void Add( Toy toy )
228:                  {
229:                          amount += toy.Price;
230:                  }
231:          }
232:  }
```

VisualBasic の場合

```
233:  Imports System
234:  Namespace OSK.ToyShop.Core
235:      Public Class Register
236:          Public Sub New()
237:              amountValue = 0
238:          End Sub
239:          Private amountValue As Integer
240:          Public ReadOnly Property Amount() As Integer
241:              Get
242:                  Return amountValue
243:              End Get
244:          End Property
245:          Public Sub Add(ByVal toyObject As Toy)
246:              amountValue += toyObject.Price
247:          End Sub
248:      End Class
249:  End Namespace
```

Session2　常時結合（Continuous Integration）を行う

Overview

システムを常に健康な状態に保つためには、正しい結合（インテグレーション）を行うことが有効です。正しい結合とは、正しいビルドを行い、正しく動作することを確認したものです。正しいビルドとは、正しい参照状態でエラーなくビルドされたものです。また、正しく動作することとは、想定した仕様通りに動作すること確認したことです。一言で言えば、「エラーなくコンパイルされて、すべてのテストで OK がでたもの」です。

さて、この正しい結合を一日に何度も行うことが、常時結合です。この常時結合を実現するために、開発されたビルドツールが NAnt です。また、VisualSourceSafe や IIS などを操作したい場合は、NAnt の拡張ライブラリである nantcontrib を使用することでさまざまは操作を行うことができます。

この Session では、Session1 のおもちゃ屋のシステムを例題として、作業を進めます。

Goal

- ❒ NAnt が利用できる
- ❒ 常時結合を体感する

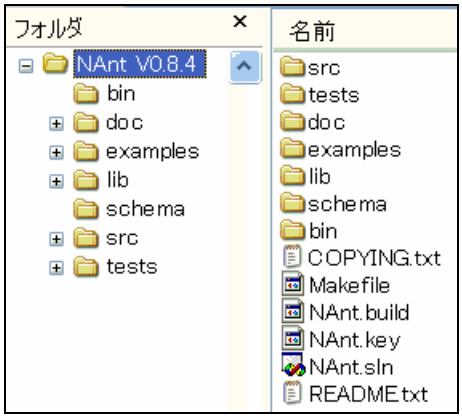
Step 1　ビルドツールをインストールする

①　NAnt を入手する

NAnt Ver0.8.4 をサイト、「http://nant.sourceforge.net/」からダウンロードします。
ダウンロードするファイルは、「nant-0.84.zip」です。（2004 年 8 月時点では、フリーソフトです。また、ねんのためウィルスチェックを行うことをお勧めします）

②　インストールを実行する

ファイルを解凍し、「Program Files」にコピーします。解凍したディレクトリには、実行モジュール、各種ドキュメント、ソースファイル、サンプルなどがコピーされます。



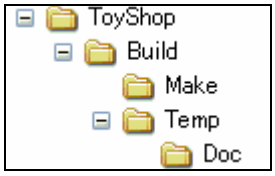
Step 2　ビルドディレクトリを作成する

①　ビルドに必要なディレクトリを決定する

ビルドに必要なディレクトリには、以下のものがあります。

- ビルド用定義ファイルのディレクトリ
- ビルド結果モジュール（EXE や DLL）ファイルのディレクトリ
- ビルドドキュメントファイルのディレクトリ
  ソールファイル中のコメントを抜き出して、作成される XML ファイルを保持するディレクトリ

今回は、以下のようなディレクトリを作成します。



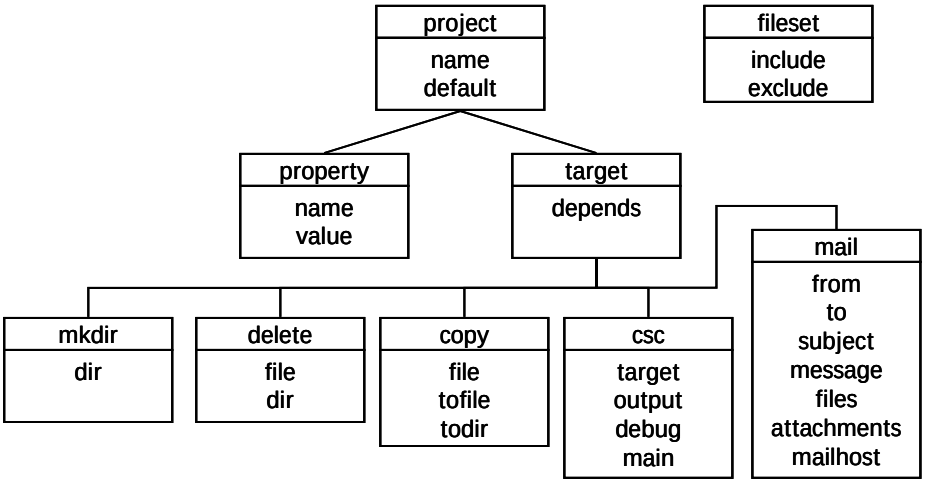
②　常時結合に必要なファイルを作成する

常時結合には、以下のファイルを作成する必要があります。

- NAnt 用のビルド定義ファイル
  今回は、「ToyShop.build」という名前で作成します。
- ビルドを実行するスプリクトファイル
  今回は、「ToyShop.continuous.build.wsf」という名前で、ウィンドウスプリクトファイルを作成します。

Step 3　ビルド定義ファイルを作成する

NAnt では、ビルド定義ファイルを XML 形式で作成します。この XML 定義ファイルは、以下のような構造を持ちます。



① 以前にビルドしたファイルをクリアする target を作成する

target の名前を clean で作成します。property として、ビルド結果のディレクトリとコメントドキュメントのディレクトリを用意します。この２つのディレクトリに入っているファイルをすべて削除するために delete タスクを定義します。
定義結果は、以下のようになります。

```
250:  <property name="build.dir" value="C:¥NUnitSeminar¥Exercises¥ToyShop¥Build¥Temp"/>
251:  <property name="build.doc.dir" value="C:¥NUnitSeminar¥Exercises¥ToyShop¥Build¥Temp¥Doc"/>
252:  <target name="clean" description="remove all build files">
253:      <delete failonerror="false">
254:          <fileset basedir="${build.dir}">
255:              <includes name="*.*"/>
256:          </fileset>
257:      </delete>
258:      <delete failonerror="false">
259:          <fileset basedir="${build.doc.dir}">
260:              <includes name="*.*"/>
261:          </fileset>
262:      </delete>
263:  </target>
```

② ビルドを行う target を作成する

target の名前を build で作成します。property として、ソールのディレクトリと各参照先のディレクトリを用意します。以下は、C#のコンパイルを行うタスクを定義します。
定義結果は、以下のようになります。

```
264:  <property name="nunit.dir"          value="C:¥Program Files¥NUnit V2.0¥bin"/>
265:  <property name="testresult.dir"     value="C:¥Program Files¥TestResult¥bin"/>
266:  <property name="source.dir"         value="C:¥NUnitSeminar¥Exercises¥ToyShop¥src"/>
267:  <target name="build" depends="clean" description="compiles core source code ">
268:      <csc output="${build.dir}¥OSK.ToyShop.Core.dll"
269:           target="library"
270:           debug="${debug}"
271:              doc="${build.doc.dir}¥OSK.ToyShop.Core.xml">
272:          <sources>
273:              <includes name="${source.dir}¥Core¥*.cs"/>
274:          </sources>
275:          <references>
276:              <includes name="System.dll"/>
277:              <includes name="System.Data.dll"/>
278:              <includes name="System.Drawing.dll"/>
279:              <includes name="System.Windows.Forms.dll"/>
280:              <includes name="System.XML.dll"/>
281:          </references>
282:      </csc>
283:      <csc output="${build.dir}¥OSK.ToyShop.Tests.dll"
284:           target="library"
285:           debug="${debug}"
286:              doc="${build.doc.dir}¥OSK.ToyShop.Tests.xml">
287:          <sources>
288:              <includes name="${source.dir}¥Tests¥*.cs"/>
289:          </sources>
290:          <references>
291:              <includes name="System.dll"/>
292:              <includes name="System.Data.dll"/>
293:              <includes name="System.Drawing.dll"/>
294:              <includes name="System.Windows.Forms.dll"/>
295:              <includes name="System.XML.dll"/>
296:              <includes name="${build.dir}¥OSK.ToyShop.Core.dll"/>
297:              <includes name="${nunit.dir}¥nunit.framework.dll"/>
298:          </references>
299:      </csc>
300:  </target>
```

③ テストを行う target を作成する

target の名前を test で作成します。ここでは、exec タスクでテストとテスト結果の成型プログラムを起動しています。

```
<target name="test" depends="build" description="run unit tests">
        <exec program="${nunit.dir}¥nunit-console.exe"
          commandline="/assembly:${build.dir}¥OSK.ToyShop.Tests.dll"
              failonerror="false"/>
        <exec program="${testresult.dir}¥TestResult.CUI.exe"
          commandline="${build.dir}¥TestResult.xml ${build.dir}¥TestResult.html"/>
</target>
```

④ 常時結合を行う continuous を作成する

target の名前を continuous で作成します。ここでは、call タスクで target の update と test を呼び出しています。

```
<target name="continuous" description="continuous build and test">
        <property name="debug" value="false"/>
        <echo message="Debug = ${debug}"/>
        <call target="update"/>
        <call target="test"/>
</target>
```

⑤ テストを実行するマクロファイルを作成する

常時結合を行うためのウィンドウマクロファイルを作成します。ファイル名を ToyShop.continuous.build.wsf で作成します。このマクロでは、NAnt を shell で起動して、その結果をメールしています。メールも NAnt のタスクで行うことができます。

```
<package>
<job>
<script language="VBScript">
'----+----+----+----+----+----+----+----+
' main
'--
Dim WshShell
Dim Result

Set WshShell = WScript.CreateObject( "WSCript.shell" )

Result = WshShell.Run( "cmd /C ""¥Program Files¥nant-0.8.01¥bin¥NAnt"" -buildfile:MobileCaster.build -verbose continuous > MobileCaster.continuous.build.log", 0, true )

'結果をメールする
If Result = 0 Then
        WshShell.Run "cmd /C ""¥Program Files¥nant-0.8.01¥bin¥NAnt"" -buildfile:mail.build -verbose resultMail", 0, true
Else
        WshShell.Run "cmd /C ""¥Program Files¥nant-0.8.01¥bin¥NAnt"" -buildfile:mail.build -verbose logMail", 0, true
End If

Set WshShell = Nothing

</script>
</job>
</package>
```

NUnit リファレンス

A) 属性

① TestFixture
テスト用のクラスであることを宣言する。

② Test
テストメソッドであることを宣言する。

③ SetUp
テストメソッド毎の初期化処理メソッドであることを宣言する。

④ TearDown
テストメソッド毎の終了処理メソッドであることを宣言する。

⑤ Ignore(“理由”)
一時的にテストクラスやテストメソッドを無効にすることを宣言する。
引数に、理由を記述する

⑥ ExpectedException(type)
テストメソッドから例外が発生されることを宣言する。発生しなかった場合はエラーとなる。

⑦ 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。

⑧ 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。

B) Assert クラス

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 IsFalse

条件が成り立たなかったら、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] condition

条件。

[入力] message

表示するメッセージ。

③ IsNull

オブジェクトがヌルだった、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

対象となるオブジェクト。

[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

対象となるオブジェクト。

[入力] message

表示するメッセージ。

⑤ AreSame

比較するオブジェクトが同じインスタンスだったら、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

比較対照となるオブジェクト。

[入力] object

比較対照となるオブジェクト。

[入力] message

表示するメッセージ。

⑥ 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] message

表示するメッセージ。

⑦ AreEqual

予測値と実測値の想定範囲内だったら、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] delta

範囲。

[入力] message

表示するメッセージ。

⑧ Fail

テストを失敗とする。

【引数】

[入力] message

表示するメッセージ。

C) Assertion クラス

2.1.4 以前との互換のためのクラス。

① Assert

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] message

表示するメッセージ。

[入力] condition

条件。

② AssertEquals

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] message

表示するメッセージ。

[入力] expected

予測値。

[入力] actual

実測値。

③ AssertEquals

予測値と実測値の想定範囲内だったら、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] message

表示するメッセージ。

[入力] expected

予測値。

[入力] actual

実測値。

[入力] delta

範囲。

④ AssertNotNull

オブジェクトがヌルではない場合に、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] message

表示するメッセージ。

[入力] anObject

オブジェクト。

⑤ AssertNull

オブジェクトがヌルの場合に、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] message

表示するメッセージ。

[入力] anObject

オブジェクト。

⑥ AssertSame

オブジェクトが等しい場合に、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] message

表示するメッセージ。

[入力] expected

予測オブジェクト。

[入力] actual

実オブジェクト。

⑦ Fail

テスト失敗とする。

【引数】

[入力] message

表示するメッセージ。