.NET プログラミング技術

オブジェクト指向プログラミング言語として .NET

トレーニング　テキスト

Rev.2004.9.1

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。

.NET では、オブジェクト指向を全面的に作用しています。したがって、オブジェクト指向プログラミングをある程度行うことが要求されますし、オブジェクト指向プログラミングを活用することで、効率的なプログラム作成を行うことができます。この Session では、オブジェクト指向プログラミングの解説を具体的にプログラムを作成する方法で行います。

この Session では、積み木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実感する**
- ❒ **オブジェクト指向プログラミングを実感する**
  クラスの作成を行う、クラスの継承を行う

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

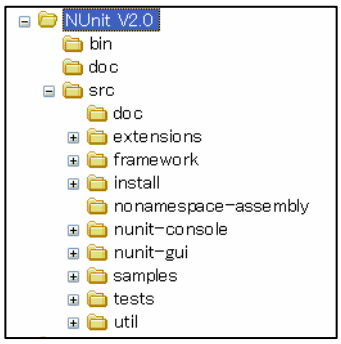
NUnit Ver2.1.4 をサイト、「http://sourceforge.net/projects/nunit/」からダウンロードします。
ダウンロードするファイルは、「NUnit-V2.1.4.msi」です。（2004 年 8 月時点では、フリーソフトです。また、ねんのためウィルスチェックを行うことをお勧めします）

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

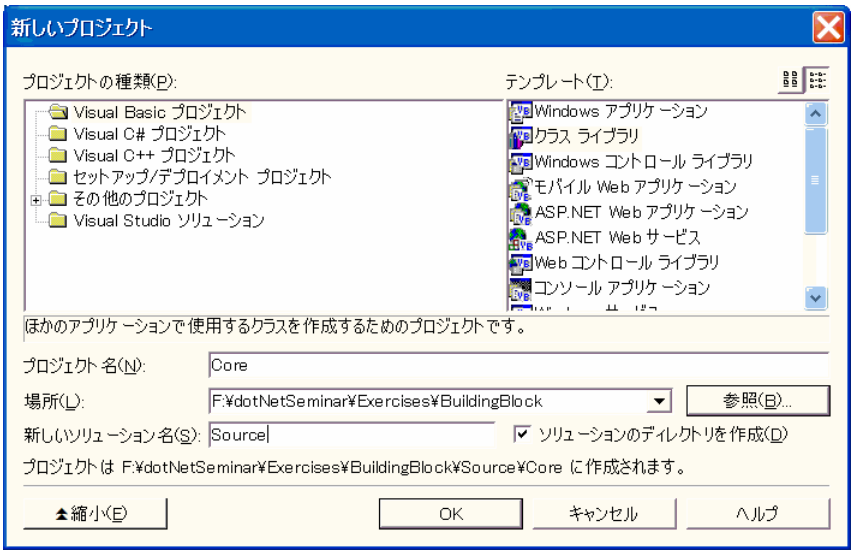
「エクスプローラ」などを使用してルートディレクトリを作成します。具体的には、「¥dotNetSeminar¥Exercises」内に「BuildingBlock」作成します。

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。

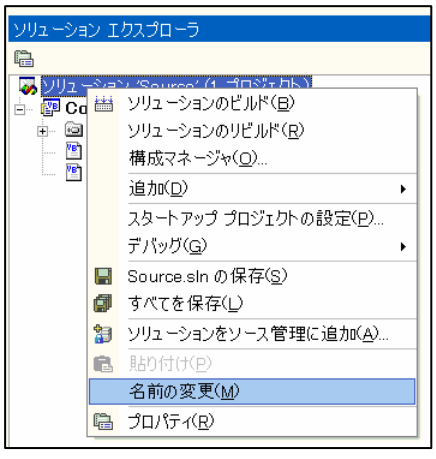
設定項目	設定内容
プロジェクトの種類	各言語のプロジェクト
テンプレート	クラスライブラリ
プロジェクト名	Core
場所	¥dotNetSeminar¥Exercises¥ BuildingBlock
新しいソリューション	Source

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。たとえば、Basic のつもりで作成したが、C#であったなどの場合は、作り直しが必要です。



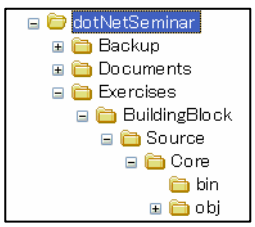
③　ソリューション名を変更する

「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「BuildingBlock」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、「BuildingBlock.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。

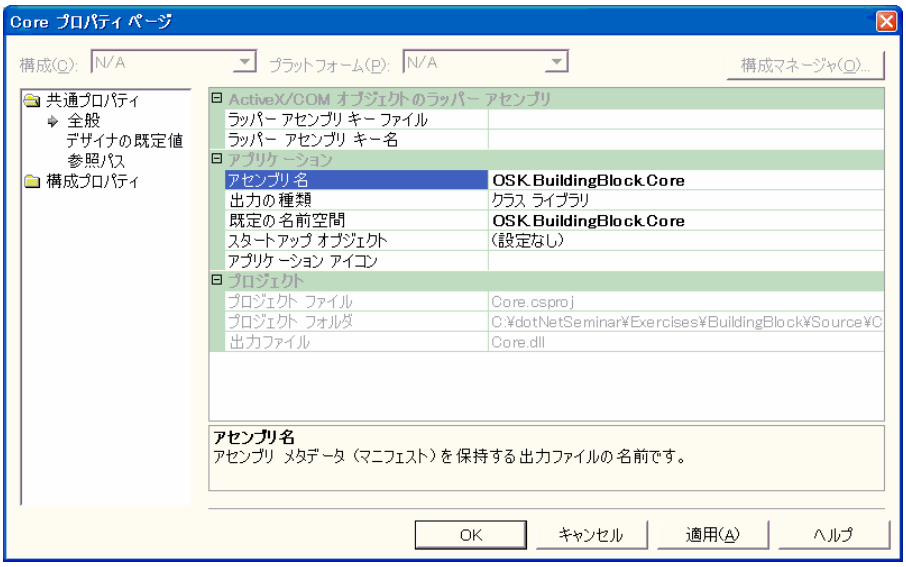


⑤ プロジェクトのプロパティを設定する

「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。

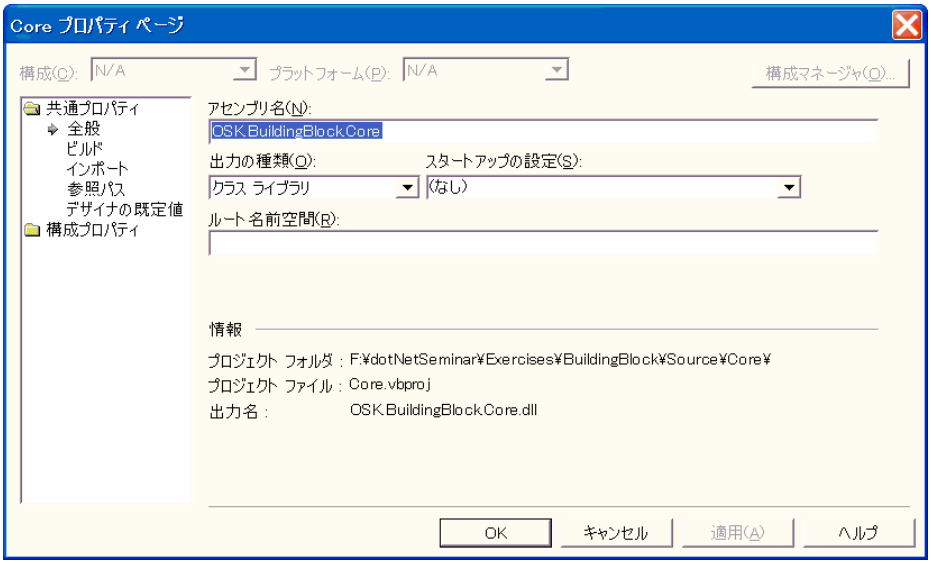
C#の場合

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



VB .NET の場合

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



Hints

ソリューションはできるだけ１つで作業することが効率的です。しかし、大規模分散システムなどでは、複数ソリューションとする必要があります。マイクロソフトの MSDN の「Visual Studio .NET と Visual SourceSafe を使用したチーム開発」の中の「第 3 章 ソリューションとプロジェクトの構造化」が参考になると思います。

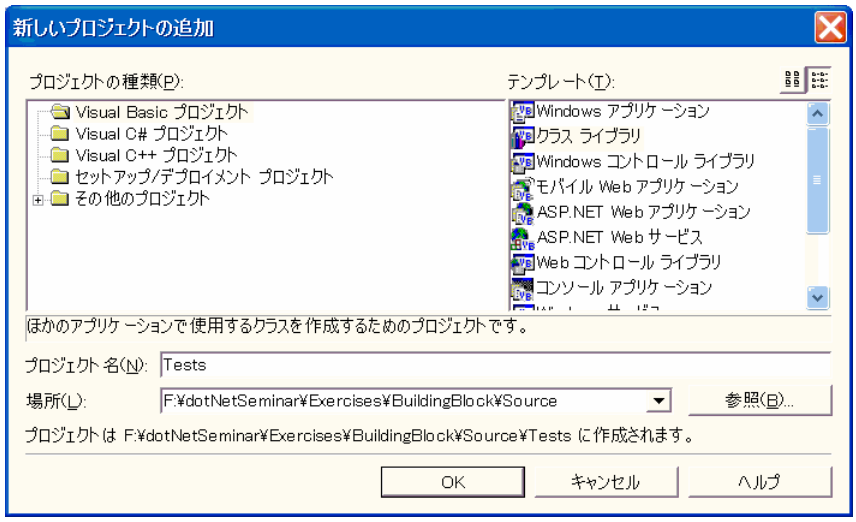
Step 3　テストプロジェクトを追加する

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。
「Visula Basic 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。

設定項目	設定内容
プロジェクトの種類	各言語のプロジェクト
テンプレート	クラスライブラリ
プロジェクト名	Tests
場所	¥dotNetSeminar¥Exercises¥ BuildingBlock¥Source

デフォルトの Class1.cs（VB の場合は Class1.vb）は、削除します。

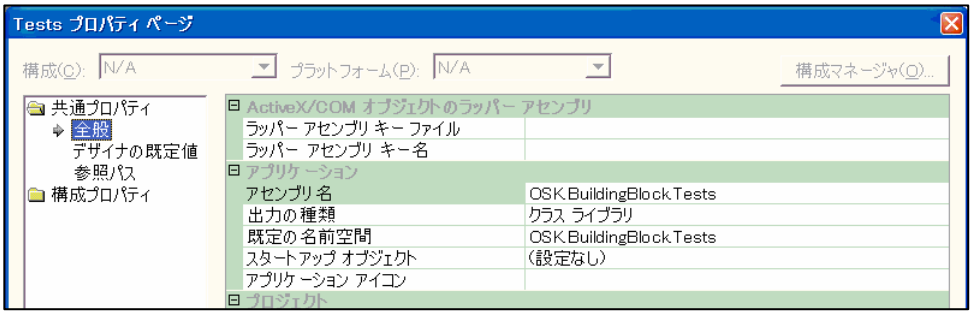


②　プロジェクトのプロパティを設定する

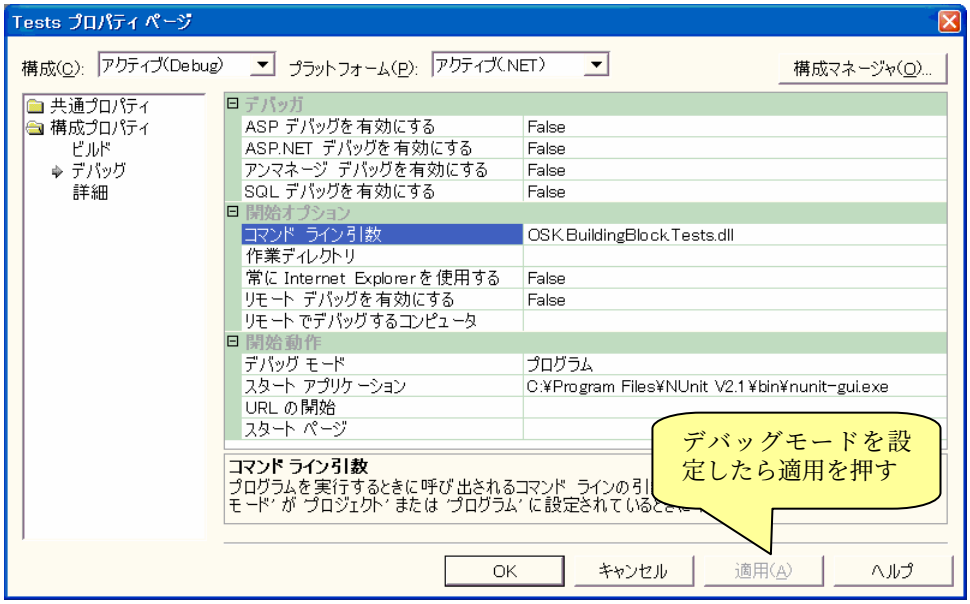
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Tests プロジェクトのコンテキストメニューを表示し、プロパティを選択します。以下の設定を変更します。
NUnit を使用して、単体テストを行う場合は、デバッグ用の外部プログラムとして NUnit が提供している EXE を使用します。ここでは、開発中に利用するために用意されている「nunit-gui.exe」を使用します。

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.BuildingBlock.Tests
全般	既定の名前空間	OSK.BuildingBlock.Tests
デバッグ	デバッグモード	プログラム
デバッグ	スタートアプリケーション	¥Program Files¥NUnit V2.1¥bin¥nunit-gui.exe
デバッグ	コマンドライン引数	OSK.BuildingBlock.Tests.dll

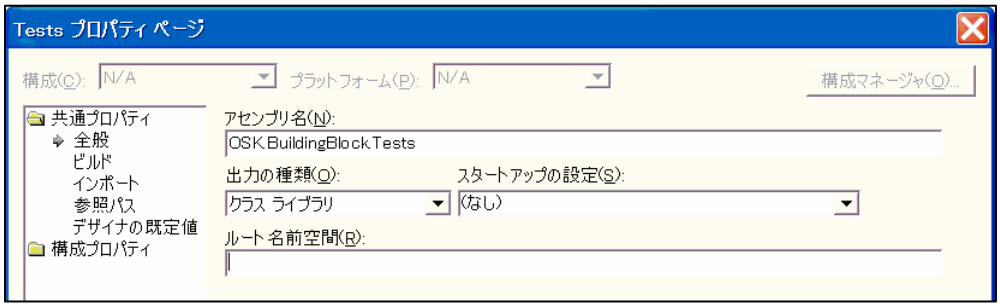


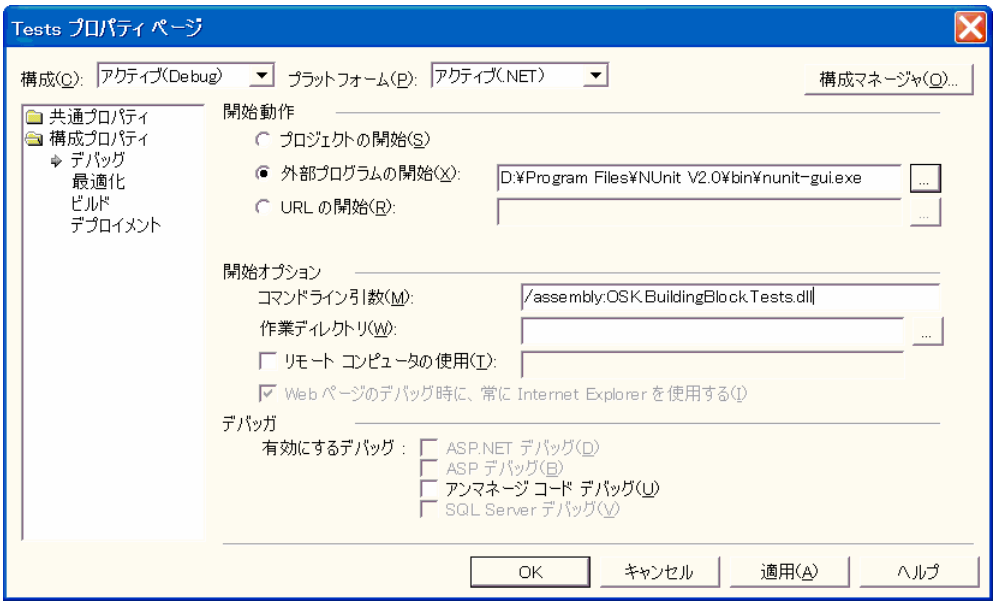
デバッグモードをプログラムに変更後、適用を押してから、スタートアプリケーションの設定を行います。



VB .NET の場合

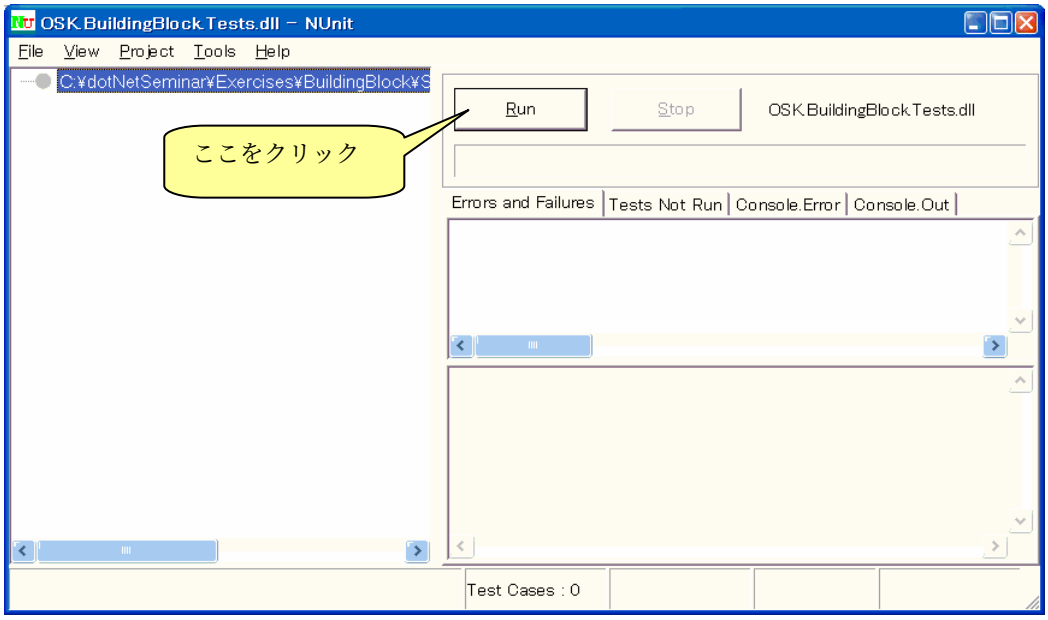
プロパティ種別	設定項目	設定内容
全般	アセンブリ名	OSK.BuildingBlock.Tests
全般	ルート名前空間	[空白]
デバッグ	外部プログラムの開始	¥Program Files¥NUnit V2.1¥bin¥nunit-gui.exe
デバッグ	コマンドライン引数	OSK.BuildingBlock.Tests.dll



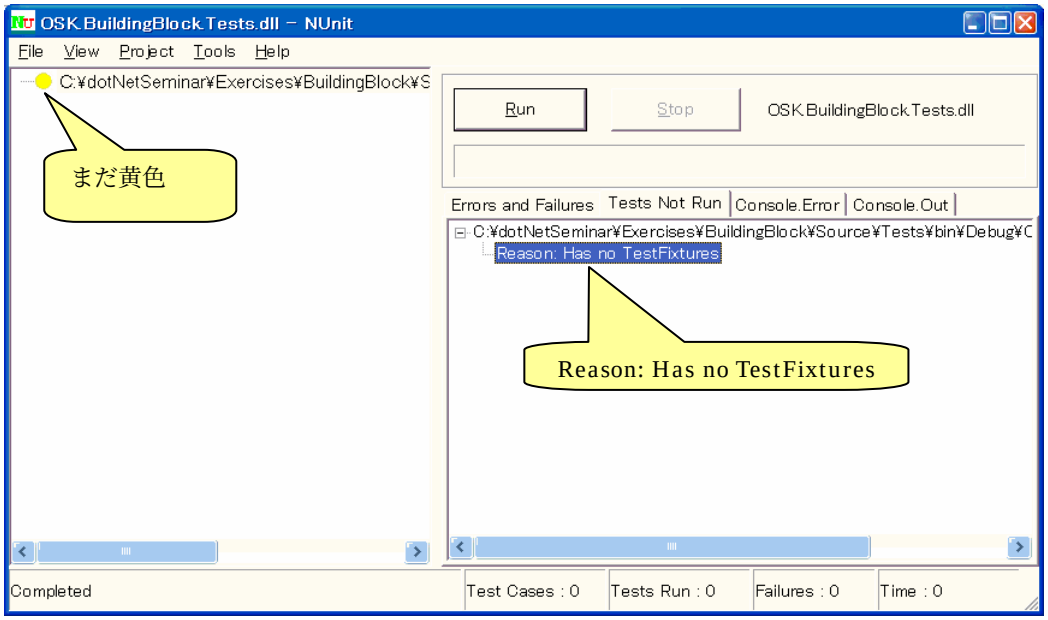


③ ビルドと実行を行う

「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。以下のようなウィンドウが表示されます。



その後、Run ボタンを押すと、テストリストの表示が黄色に変わります。また、Tests Not Run タグを選択すると、テストが無いというメッセージが表示されます。



Step 4　クラスとテストクラスを作成する

複数の機能から構築されるサービス（ここでは、一連の関連した機能によって提供される高度な機能をサービスと呼びます）を提供するためには、.NET では、クラスという概念を導入しています。クラスは、機能と情報をあわせ持っています。このことにより、クラスの行う処理が複雑な機能であっても、クラスを利用する側は複雑な処理を意識することなく、サービスを受けることができます。

①　名前空間を決定する

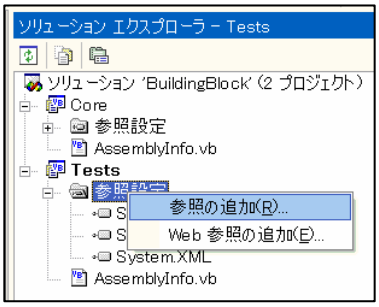
.NET では、クラスを階層的にグループ化して管理を行っています。この階層のことを名前空間と呼びます。独自にクラスを作成する場合、独自の名前空間を用意する必要があります。名前空間によって、クラスが識別されますので、以下のような形式作成します。

企業名（組織名）.テクノロジー名.機能名

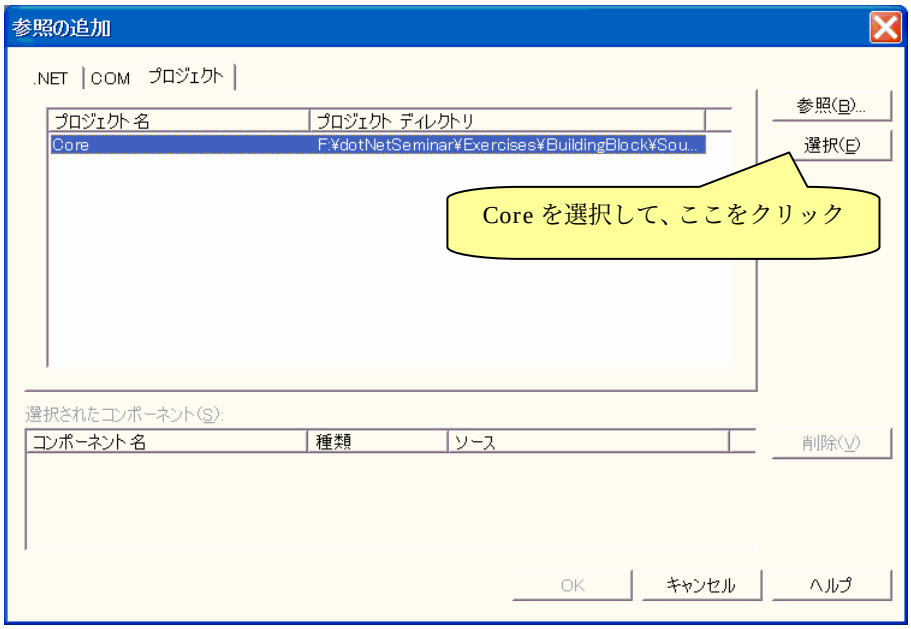
この Session のサンプルでは、名前空間を「OSK. BuildingBlock.プロジェクト名」とします。

②　プロジェクト間の参照設定を行う

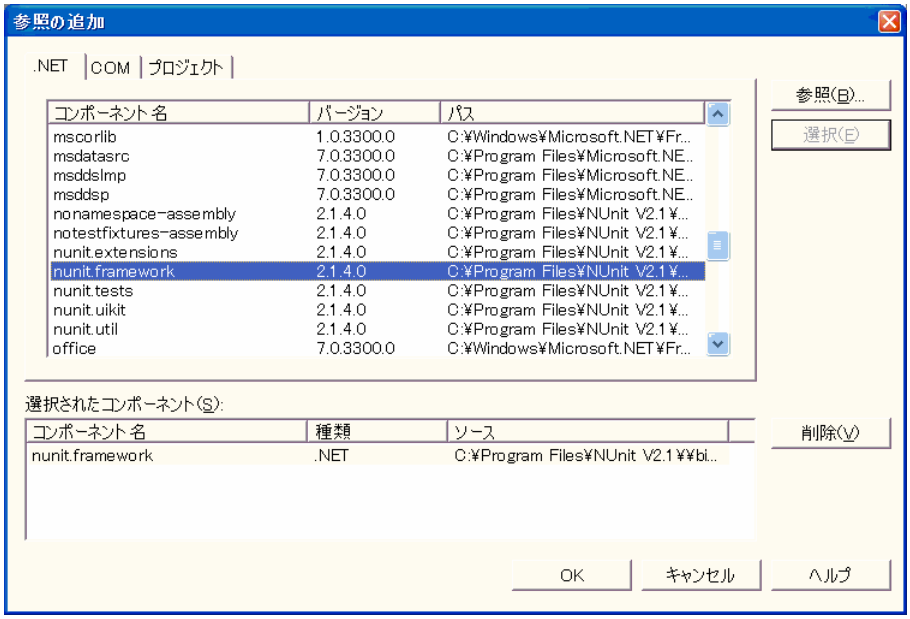
プロジェクト間の参照設定を行います。今回は、「Tests」プロジェクトで「Core」プロジェクトの参照設定を行います。
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Tests プロジェクトの参照設定コンテキストメニューを表示し、「参照」を選択します。



「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。



次に、.NET タグで NUnit のクラスライブラリ「nunit.framework」を参照します。

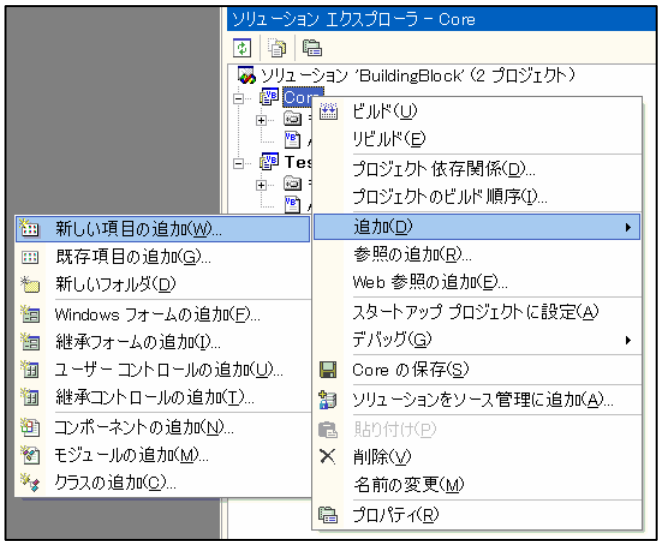


上記の方法以外に、参照ボタンを押して、DLL を直接していすることもできます。

	内容
ディレクトリ位置	¥Program Files¥NUnit V2.1¥bin
ファイル名	nunit.framework.dll

③ クラスを作成する

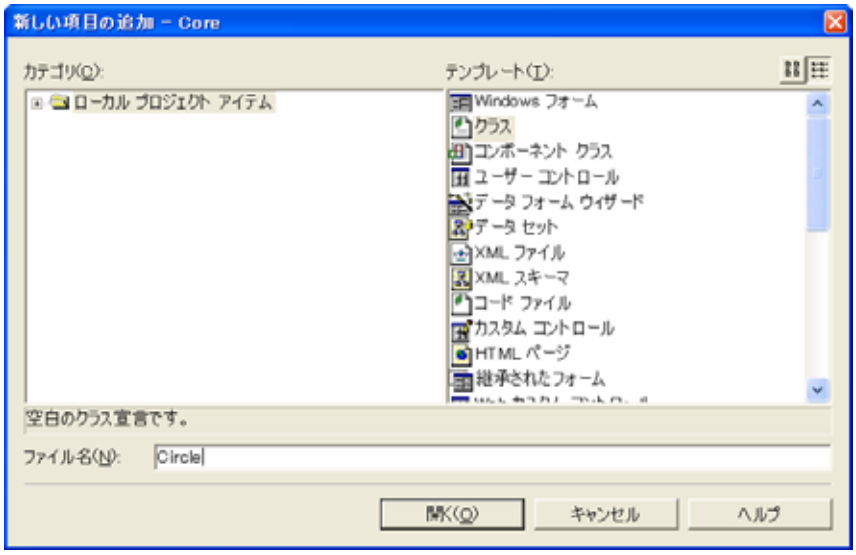
丸いブロックを Circle という名前で作成します。
「ソリューション　エクスプローラ」ウィンドウを表示して、マウスの右ボタンで Core プロジェクトのコンテキストメニューを表示し、「追加」の「新しい項目の追加」を選択します。



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	Circle.cs (VB **の場合は、**Circle.vb)

指定したファイル名がクラス名となって、ソースファイルが作成されます。
クラス名とファイル名が同じ必要はありませんが、わかりやすくするために、同じにすることをお勧めします。



④ クラスの雛形を実装する

名前空間を指定して、Circle 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、メソッドを持っています。各実装はまだ行いません。

C#の場合

```
1:      namespace OSK.BuildingBlock.Core
2:      {
3:              public class Circle
4:              {
5:                      public Circle(double diameter)
6:                      {
7:                      }
8:                      public double Radius
9:                      {
10:                             get
11:                             {
12:                                     return 0.0;
13:                             }
14:                     }
15:                     public string Nickname
16:                     {
17:                             get
18:                             {
19:                                     return "";
20:                             }
21:                             set
22:                             {
23:                             }
24:                     }
25:                     public double CalculateArea()
26:                     {
27:                             return 0.0;
28:                     }
29:             }
30:     }
```

VB .NET の場合

```
31:     Namespace OSK.BuildingBlock.Core
32:         Public Class Circle
33:             Public Sub New(ByVal diameter As Double)
34:             End Sub
35:             Public ReadOnly Property Radius() As Double
36:                 Get
37:                     Return 0.0
38:                 End Get
39:             End Property
40:             Public Property Nickname () As String
41:                 Get
42:                     Return “”
43:                 End Get
44:                 Set(ByVal Value As String)
45:                 End Set
46:             End Property
47:             Public Function CalculateArea() As Double
48:                 Return 0.0
49:             End Function
50:         End Class
51:     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。

Diameter は直径、Radius は半径。

⑤ テストクラスを作成する

Circle クラスをテストするクラスを CircleTestCase 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	CircleTestCase.cs（VB の場合は CircleTestCase.vb）

⑥ テストクラスを実装する

今回、CircleTestCase 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
  直径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。また、半径が正しく取得できることを確認します。
- プロパティの設定と取得のテスト
- 面積計算のテスト

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）
```

VB .NET 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

```
52:     using System;
53:     using NUnit.Framework;
54:     using OSK.BuildingBlock.Core;
55:     namespace OSK.BuildingBlock.Tests
56:     {
57:         [TestFixture]
58:         public class CircleTestCase
59:         {
60:             [Test]
61:             public void ConstructorTest()
62:             {
63:                 Circle circleObj;
64:                 circleObj = new Circle(10.0);
65:                 Assert.IsNotNull(circleObj,"インスタンスが作成された");
66:                 Assert.AreEqual(5.0, circleObj.Radius,"半径の確認で NG");
67:             }
68:             [Test]
69:             public void PropertyTest()
70:             {
71:                 Circle circleObj;
72:                 circleObj = new Circle(10.0);
73:                 circleObj.Nickname = "まる";
74:                 Assert.AreEqual("まる", circleObj.Nickname,"プロパティのチェック");
75:             }
76:             [Test]
77:             public void CalculateArea()
78:             {
79:                 Circle circleObj;
80:                 circleObj = new Circle(10.0);
81:                 Assert.AreEqual(5.0 * 5.0 * 3.14, circleObj.CalculateArea(),"面積計算");
82:             }
83:         }
84:     }
```

VB .NET の場合

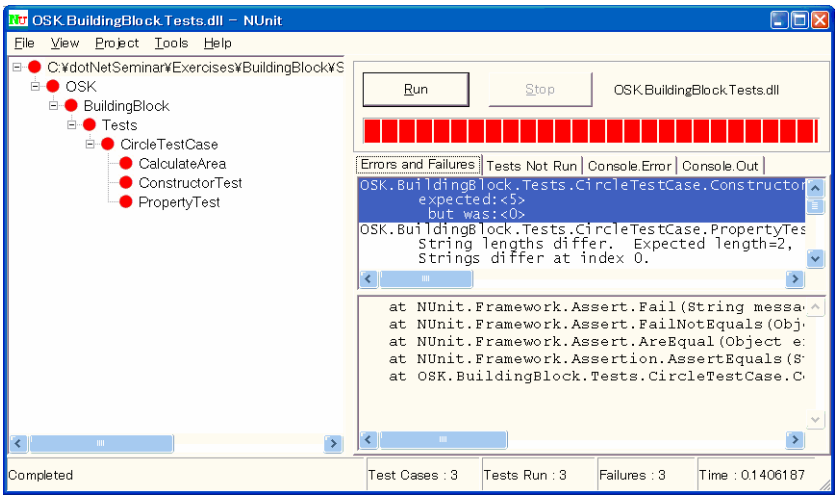
```
85:     Imports NUnit.Framework
86:     Imports OSK.BuildingBlock.Core
87:     Namespace OSK.BuildingBlock.Tests
88:         <TestFixture()> _
89:         Public Class CircleTestCase
90:             <Test()> _
91:             Public Sub ConstructorTest()
92:                 Dim circleObj As Circle
93:                 circleObj = New Circle(10.0)
94:                 Assert.IsNotNull(circleObj,"インスタンスが作成された")
95:                 Assert.AreEqual(5.0, circleObj.Radius,"半径の確認で NG")
96:             End Sub
97:             <Test()> _
98:             Public Sub PropertyTest()
99:                 Dim circleObj As Circle
100:                circleObj = New Circle(10.0)
101:                circleObj.Nickname = "まる"
102:                Assert.AreEqual("まる", circleObj.Nickname,"プロパティのチェック")
103:            End Sub
104:            <Test()> _
105:            Public Sub CalculateArea()
106:                Dim circleObj As Circle
107:                circleObj = New Circle(10.0)
108:                Assert.AreEqual(5.0 * 5.0 * 3.14, circleObj.CalculateArea(),"面積計算")
109:            End Sub
110:        End Class
111:    End Namespace
```

Hints

実際に作成するときは、１つテストを作成したら、その部分の実装を行います。また、１つテストを作成したら、その部分のテストを行います。このように、作業を進める方法が有効です。

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、Circle 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。



C#の場合

```
112:    using System;
113:    namespace OSK.BuildingBlock.Core
114:    {
115:        public class Circle
116:        {
117:            private const double PI = 3.14;
118:            private double diameterValue;
119:            public Circle(double diameter)
120:            {
121:                diameterValue = diameter;
122:            }
123:            public double Radius
124:            {
125:                get
126:                {
127:                    return diameterValue / 2.0;
128:                }
129:            }
130:            private string nicknameValue;
131:            public string Nickname
132:            {
133:                get
134:                {
135:                    return nicknameValue;
136:                }
137:                set
138:                {
139:                    nicknameValue = value;
140:                }
141:            }
142:            public double CalculateArea()
143:            {
144:                return Radius * Radius * PI;
145:            }
146:        }
147:    }
```

Hints

C#でプロパティを定義するためには、以下の形式でおこないます。

```
public 型名 プロパティ名
{
}
```

プロパティを設定する場合は、必ず value という変数が使用されます。

VB .NET の場合

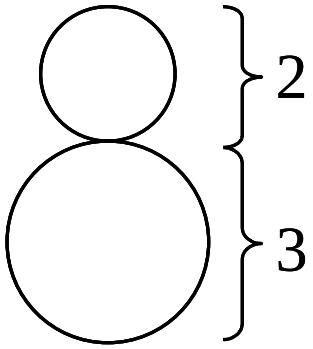
```
148:    Namespace OSK.BuildingBlock.Core
149:        Public Class Circle
150:            Private Const PI As Double = 3.14
151:            Private diameterValue As Double
152:            Public Sub New(ByVal diameter As Double)
153:                diameterValue = diameter
154:            End Sub
155:            Public ReadOnly Property Radius() As Double
156:                Get
157:                    Return diameterValue / 2.0
158:                End Get
159:            End Property
160:            Private nicknameValue As String
161:            Public Property Nickname() As String
162:                Get
163:                    Return nicknameValue
164:                End Get
165:                Set(ByVal Value As String)
166:                    nicknameValue = Value
167:                End Set
168:            End Property
169:            Public Function CalculateArea() As Double
170:                Return Radius * Radius * PI
171:            End Function
172:        End Class
173:    End Namespace
```

Step 5　他のクラスを利用して、新しいクラスを作成する

既に作成されたクラスを部品として利用し、新しいクラスを作成することができます。この様に、他のクラスの一部分となるクラスを部分クラスと呼びます。部分クラスは、全体クラスのインスタンスが作成されたときに、部分クラスのインスタンスも作成され、全体クラスのインスタンスが破棄されるときには、同時に破棄されるのが一般的です。したがって、部分クラスのインスタンスの生存期間は、全体クラスのインスタンスの生存期間に含まれます。

①　クラスを作成する

Circle クラスを２つ利用して、雪だるまクラスの Snowman クラスを新規作成します。
雪だるまは、高さを２対３にした２つの円を組み合わせたものとします。



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	Snowman.cs（VB の場合は、Snowman.vb）

Snowman クラスのコンストラクタには、雪だるまの高さを引数に渡します。また、プロパティとして Nickname、メソッドとして、面積計算を用意します。

C#の場合

```
174:    using System;
175:
176:    namespace OSK.BuildingBlock.Core
177:    {
178:        public class Snowman
179:        {
180:            public Snowman(double height)
181:            {
182:            }
183:            public string Nickname
184:            {
185:                get
186:                {
187:                    return "";
188:                }
189:                set
190:                {
191:                }
192:            }
193:            public double CalculateArea()
194:            {
195:                return 0.0;
196:            }
197:        }
198:    }
```

VB .NET の場合

```
199:    Namespace OSK.BuildingBlock.Core
200:        Public Class Snowman
201:            Public Sub New(ByVal height As Double)
202:            End Sub
203:            Public Property Nickname() As String
204:                Get
205:                    Return ""
206:                End Get
207:                Set(ByVal Value As String)
208:                End Set
209:            End Property
210:            Public Function CalculateArea() As Double
211:                Return 0.0
212:            End Function
213:        End Class
214:    End Namespace
```

② テストクラスに初期化処理と終了処理を作成する

Snowman クラスのテスト用クラスとして、SnowmanTestCase を新規作成します。

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	SnowmanTestCase.vb

テストクラスのテスト用のメソッドでは、テストを行うクラスの作成を行うことが良くあります。この処理をメソッド単位で行うと、プログラムが重複してしまうので、NUnit では、初期化処理と終了処理を SetUp 属性と TearDown 属性を指定することで、共通化することができます。この初期化処理と終了処理は、テストごとに呼び出されます。

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

- プロパティの設定と取得のテスト
- 面積計算のテスト

  高さを指定して作成したインスタンスの面積が正しく計算されることを確認します。

最終的なテストは、以下のようになります。

C#の場合

```
215:    using System;
216:    using NUnit.Framework;
217:    using OSK.BuildingBlock.Core;
218:
219:    namespace OSK.BuildingBlock.Tests
220:    {
221:        [TestFixture]
222:        public class SnowmanTestCase
223:        {
224:            private Snowman target;
225:            [SetUp]
226:            public void Setup()
227:            {
228:                target = new Snowman(20.0);
229:            }
230:            [TearDown]
231:            public void TearDown()
232:            {
233:                target = null;
234:            }
235:            [Test]
236:            public void PropertyTest()
237:            {
238:                target.Nickname = "雪だるま";
239:                Assert.AreEqual("雪だるま", target.Nickname,"プロパティのチェック");
240:            }
241:            [Test]
242:            public void CalculateArea()
243:            {
244:                Assert.AreEqual(4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14, target.CalculateArea(),"面積計算");
245:            }
246:        }
247:    }
```

VB .NET の場合

```
248:    Imports NUnit.Framework
249:    Imports OSK.BuildingBlock.Core
250:    Namespace OSK.BuildingBlock.Tests
251:        <TestFixture()> _
252:        Public Class SnowmanTestCase
253:            Private targetObj As Snowman
254:            <SetUp()> _
255:            Public Sub Setup()
256:                targetObj = New Snowman(20.0)
257:            End Sub
258:            <TearDown()> _
259:            Public Sub TearDown()
260:                targetObj = Nothing
261:            End Sub
262:            <Test()> _
263:            Public Sub PropertyTest()
264:                targetObj.Nickname = "雪だるま"
265:                Assert.AreEqual("雪だるま", targetObj.Nickname,"プロパティのチェック")
266:            End Sub
267:            <Test()> _
268:            Public Sub CalculateArea()
269:                Assert.AreEqual(4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14, targetObj.CalculateArea(), _
270:                                                "面積計算")
271:        End Class
272:    End Namespace
```

③ クラスを完成させる

テストが全て通るように Sonwman クラスを実装します。最終的なクラスは以下のようになります。

C#の場合

```
273:  using System;
274:
275:  namespace OSK.BuildingBlock.Core
276:  {
277:      public class Snowman
278:      {
279:          private Circle headObj;
280:          private Circle bodyObj;
281:          public Snowman(double height)
282:          {
283:              headObj = new Circle(height * 2 / 5);
284:              bodyObj = new Circle(height * 3 / 5);
285:          }
286:          private string nicknameValue;
287:          public string Nickname
288:          {
289:              get
290:              {
291:                  return nicknameValue;
292:              }
293:              set
294:              {
295:                  nicknameValue = value;
296:              }
297:          }
298:          public double CalculateArea()
299:          {
300:              return headObj.CalculateArea() + bodyObj.CalculateArea();
301:          }
302:      }
303:  }
```

VB .NET の場合

```
304:  Namespace OSK.BuildingBlock.Core
305:      Public Class Snowman
306:          Private headObj As Circle
307:          Private bodyObj As Circle
308:          Public Sub New(ByVal height As Double)
309:              headObj = New Circle(height * 2 / 5)
310:              bodyObj = New Circle(height * 3 / 5)
311:          End Sub
312:          Private nicknameValue As String
313:          Public Property Nickname() As String
314:              Get
315:                  Return nicknameValue
316:              End Get
317:              Set(ByVal Value As String)
318:                  nicknameValue = Value
319:              End Set
320:          End Property
321:          Public Function CalculateArea() As Double
322:              Return headObj.CalculateArea() + bodyObj.CalculateArea()
323:          End Function
324:      End Class
325:  End Namespace
```

Step 6　複数のクラスの共通部分をクラスにまとめる

クラスを複数作成すると、複数のクラス間で、まったく同じ部分を持っていることがあります。このまま同じプログラムを複数の場所に放置していると、同じ修正を何十回と繰り返すことになったり、またその作業でバグを混入させてしまったりと、保守性が著しく低下してしまいます。

この様な場合、オブジェクト指向プログラミングでは、共通部分を1つのクラスにして、各クラスで継承させる方法で、共通部分を1つにまとめることができます。この様な継承のことを汎化と呼びます。共通部分のクラスを親クラス、各クラスを子クラスと呼びます。親クラスの名前は、一般に抽象的な名前や総称的な名前を付けます。例えば、具象的な円ブロックや長方形ブロックといったもの、親クラスはブロックという総称的な名前にします。

①　クラスを作成する

様々なブロックの親クラスとなる、Block クラスを新規作成します。

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	Block.cs（VB の場合は Block.vb）

Block クラスは、プロパティとして Nickname を持ちます。このクラスは、他のクラスに継承してもらうことを前提にしていますので、クラスの種別として「abstract」（VB の場合は「MustInherit」）を指定します。
最終的な Block クラスは、以下のようになります。

C#の場合

```
326:    using System;
327:
328:    namespace OSK.BuildingBlock.Core
329:    {
330:        public abstract class Block
331:        {
332:            private string nicknameValue;
333:            public string Nickname
334:            {
335:                get
336:                {
337:                    return nicknameValue;
338:                }
339:                set
340:                {
341:                    nicknameValue = value;
342:                }
343:            }
344:        }
345:    }
```

VB .NET の場合

```
346:    Namespace OSK.BuildingBlock.Core
347:        Public MustInherit Class Block
348:            Private nicknameValue As String
349:            Public Property Nickname() As String
350:                Get
351:                    Return nicknameValue
352:                End Get
353:                Set(ByVal Value As String)
354:                    nicknameValue = Value
355:                End Set
356:            End Property
357:        End Class
358:    End Namespace
```

② 既存クラスを変更する

親クラスを作成したら、今まで作成した Circle クラスと Snowman クラスに継承を設定し、共通部分を削除します。

C#の場合

C#での継承は、クラス定義の後に、「:」を記述して、その後に親クラス名を記述します。その後、プロパティ Nickname に関する部分を削除します。ビルドを行ってエラーが発生しなければ、変更は終了です。

```
359:    namespace OSK.BuildingBlock.Core
360:    {
361:        public class Circle : Block
362:        {
```

VB .NET の場合

VB での継承は、クラス定義の後に、キーワード「Inherits」を記述して、その後に親クラス名を記述します。その後、プロパティ Nickname に関する部分を削除します。ビルドを行ってエラーが発生しなければ、変更は終了です。

```
363:    Namespace OSK.BuildingBlock.Core
364:        Public Class Circle
365:            Inherits Block
```

③ テストを行う

既に作成してある Circle のテストケースを動作し、クラスの構造を変えても、テストが正常に動作することを確認します。Snowman クラスも同様に作業を行います。

Points

この Step のように、自動テスト環境があることで、勇気を持ってクラスの構造を最適化することができます。もし、手作業で単体テストを行う環境では最適化は行えず、コードはどんどんスパゲッティ状態に退化してしまいます。

④ 親クラスを利用して、新しいクラスを作成する

親クラスを利用して、四角形クラスを作成します。このときも、今までの手順と同じです。

C#の場合

Rectangle クラスは以下のように作成します。

```
366:    using System;
367:
368:    namespace OSK.BuildingBlock.Core
369:    {
370:        public class Rectangle : Block
371:        {
372:            public Rectangle(double height, double width)
373:            {
374:            }
375:            public double CalculateArea()
376:            {
377:                return 0.0;
378:            }
379:        }
380:    }
```

RectanguleTestCase クラスは以下のようになります。

```
381:    using System;
382:    using NUnit.Framework;
383:    using OSK.BuildingBlock.Core;
384:
385:    namespace OSK.BuildingBlock.Tests
386:    {
387:        [TestFixture]
388:        public class RectanguleTestCase
389:        {
390:            private Rectangle target;
391:            [SetUp]
392:            public void Setup()
393:            {
394:                target = new Rectangle(5.0, 10.0);
395:            }
396:            [TearDown]
397:            public void TearDown()
398:            {
399:                target = null;
400:            }
401:            [Test]
402:            public void PropertyTest()
403:            {
404:                target.Nickname = "しかく";
405:                Assert.AreEqual("しかく", target.Nickname,"プロパティのチェック");
406:            }
407:            [Test]
408:            public void CalculateArea()
409:            {
410:                Assert.AreEqual( 5.0 * 10.0, target.CalculateArea(),"面積計算");
411:            }
412:        }
413:    }
```

テスト実行を行うと、一部のテストが正常に終了します。これは、親クラスが既に実装しているからです。
最終的な Rectangle クラスは以下のようになります。

```
414:    using System;
415:
416:    namespace OSK.BuildingBlock.Core
417:    {
418:        public class Rectangle : Block
419:        {
420:            private double heightValue;
421:            private double widthValue;
422:            public Rectangle(double height, double width)
423:            {
424:                heightValue = height;
425:                widthValue = width;
426:            }
427:            public double CalculateArea()
428:            {
429:                return heightValue * widthValue;
430:            }
431:        }
432:    }
```

VB .NET の場合

Rectangle クラスは以下のようになります。

```
433:    Namespace OSK.BuildingBlock.Core
434:        Public Class Rectangle
435:            Inherits Block
436:            Private heightValue As Double
437:            Private widthValue As Double
438:            Public Sub New(ByVal height As Double, _
439:                           ByVal width As Double)
440:                heightValue = height
441:                widthValue = width
442:            End Sub
443:            Public Function CalculateArea() As Double
444:                Return heightValue * widthValue
445:            End Function
446:        End Class
447:    End Namespace
```

RectanguleTestCase クラスは以下のようになります。

```
448:    Imports NUnit.Framework
449:    Imports OSK.BuildingBlock.Core
450:    Namespace OSK.BuildingBlock.Tests
451:        <TestFixture()> _
452:        Public Class RectanguleTestCase
453:            Private targetObj As Rectangle
454:            <SetUp()> _
455:            Public Sub Setup()
456:                targetObj = New Rectangle(5.0, 10.0)
457:            End Sub
458:            <TearDown()> _
459:            Public Sub TearDown()
460:                targetObj = Nothing
461:            End Sub
462:            <Test()> _
463:            Public Sub PropertyTest()
464:                targetObj.Nickname = "しかく"
465:                Assert.AreEqual("しかく", targetObj.Nickname, _
466:                                "プロパティのチェック")
467:            End Sub
468:            <Test()> _
469:            Public Sub CalculateArea()
470:                Assert.AreEqual(5.0 * 10.0, targetObj.CalculateArea(), _
471:                                "面積計算")
472:            End Sub
473:        End Class
474:    End Namespace
```

Step 7　親クラスで雛型メソッドを作成する

Step6 で、クラスの共通部分を親クラスとしてまとめましたが、これだけでは今までの共通モジュールとそれほど変わりません。例えて言えば、完成品のパソコンのようなものです。これでは、必ずしも、自分にあったパソコンではないことも多く、適応範囲が限定されてしまいます。しかし、すべてを自分で組み立てるのも大変です。そこで登場するのが、カスタムメイドのパソコンです。個別に変更可能な部分をあらかじめ限定して用意することで、適応範囲を広げることができます。

.NET では、あらかじめ変更部分を親クラスでメソッドなどとして用意することができます。用意したメソッドとには、C#では abstract (VB の場合は MustOverride) を宣言します。これで、このメソッドは、必ず子クラスで定義することを義務付けることになります。

①　カスタマイズポイントとなるメソッドを定義する

カスタマイズポイントとなるメソッドとは、実際のプログラムは子クラスで実装し、型だけを親クラスで定義するものです。今回のシステムでは、ブロックの面積の 10 倍を重さとします。したがって、計算式は、重さ＝面積 * 10 となります。しかし、面積の計算方法は各ブロックの形によって異なります。そこで、メソッドの型だけを親クラスで定義します。

C#の場合

```
475:        public abstract double CalculateArea();
```

VB .NET の場合

```
476:        Public MustOverride Function CalculateArea() As Double
```

②　各クラスのカスタマイズポイントを差し替えるメソッドを定義する

カスタマイズポイントとなるメソッドを各子クラスで実装します。今回は、既に実装されていますので、キーワード「Overrides」をメソッドに付けます。そして、ビルドを行い、テストを実行します。

C#の場合

```
477:        public override double CalculateArea()
```

VB .NET の場合

```
478:        Public Overrides Function CalculateArea() As Double
```

③ 変更部分を含んだメソッドを定義する

Block クラスに、重さを計算するメソッドを定義します。その後、各クラスに重さをテストするテストメソッドを追加し、テストが成功することを確認します。

C#の場合

```
479:  using System;
480:
481:  namespace OSK.BuildingBlock.Core
482:  {
483:      public abstract class Block
484:      {
485:          private string nicknameValue;
486:          public string Nickname
487:          {
488:              get
489:              {
490:                  return nicknameValue;
491:              }
492:              set
493:              {
494:                  nicknameValue = value;
495:              }
496:          }
497:          public abstract double CalculateArea();
498:          public double CalculateWeight()
499:          {
500:              return CalculateArea() * 10.0;
501:          }
502:      }
503:  }
```

VB .NET の場合

```
504:  Namespace OSK.BuildingBlock.Core
505:      Public MustInherit Class Block
506:          Private nicknameValue As String
507:          Public Property Nickname() As String
508:              Get
509:                  Return nicknameValue
510:              End Get
511:              Set(ByVal Value As String)
512:                  nicknameValue = Value
513:              End Set
514:          End Property
515:          Public MustOverride Function CalculateArea() As Double
516:          Public Function CalculateWeight () As Double
517:              Return CalculateArea() * 10.0
518:          End Function
519:      End Class
520:  End Namespace
```

Session2　オブジェクト指向プログラミングを行う

Overview

.NET では、効率的な開発を行うために、オブジェクト指向技術を全面的に採用しています。このセッションでは、オブジェクト指向技術で、難しいと言われているポリモーフィズムなどに焦点を当てて説明を行います。この Session でも、オブジェクト指向プログラミングの解説を具体的にプログラムを作成する方法で行います。

この Session でも、引き続き、積み木システムを例題として、作業を進めます。

Goal

- ❒ **オブジェクト指向のポリモーフィズムを実感する**
- ❒ **インターフェイスベースのプログラミングを実感する**
- ❒ **委譲を実感する**

Step 1　ポリモーフィズムを利用してブロックセット作成する

複数のブロックをブロックセットとして、まとめて管理できるクラスを作成する場合、セットへいろいろなブロックを追加するメソッドを用意する必要があります。今までの VB プログラムであれば、AddCirlce や AddSnowman などのように複数のメソッドを用意することがありました。各ブロックをブロックという抽象レベルで扱うことで、Add(ブロック)というひとつのメソッドで複数のクラスのインスタンスを同じように操作することができるようになります。

① クラスを作成する

ブロックセットの BlockSet 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	BlockSet.cs (VBでは BlockSet.vb)

今回このクラスには、ブロックを追加するメソッドを用意します。また、追加されたブロックの面積と重さの合計を計算するメソッドを用意します。更に、プロパティとして Nickname を用意します。実は、ブロックのセットも、ブロックとして扱うことができます。したがって、BlockSet も Block を継承させることにします。

クラスの最初のソースは、以下のようになります。

C#の場合

```
521:    using System;
522:
523:    namespace OSK.BuildingBlock.Core
524:    {
525:        public class BlockSet : Block
526:        {
527:            public void Add( Block newBlock )
528:            {
529:            }
530:            public override double CalculateArea()
531:            {
532:                return 0.0;
533:            }
534:        }
535:    }
```

VB .NET の場合

```
536:    Namespace OSK.BuildingBlock.Core
537:        Public Class BlockSet
538:            Inherits Block
539:            Public Sub Add(ByRef newBlock As Block)
540:            End Sub
541:            Public Overrides Function CalculateArea() As Double
542:                Return 0.0
543:            End Function
544:        End Class
545:    End Namespace
```

② テストクラスを作成する

BlockSet クラスのテスト用クラスとして、BlockSetTestCase を作成します。
ブロックを追加して、総面積と総重量を計算するテストを行います。また、ブロックがひとつもない場合のテストも用意します。

C#の場合

```
546:    using System;
547:    using NUnit.Framework;
548:    using OSK.BuildingBlock.Core;
549:    namespace OSK.BuildingBlock.Tests
550:    {
551:        [TestFixture]
552:        public class BlockSetTestCase
553:        {
554:            private BlockSet target;
555:            [SetUp]
556:            public void Setup()
557:            {
558:                target = new BlockSet();
559:            }
560:            [TearDown]
561:            public void TearDown()
562:            {
563:                target = null;
564:            }
565:            [Test]
566:            public void PropertyTest()
567:            {
568:                target.Nickname = "ブロックセット";
569:                Assert.AreEqual("ブロックセット", target.Nickname, "プロパティのチェック");
570:            }
571:            [Test]
572:            public void NoBlockTest()
573:            {
574:                Assert.AreEqual(0, target.CalculateArea(),"面積計算");
575:                Assert.AreEqual(0, target.CalculateWeight(),"重さ計算");
576:            }
577:            [Test]
578:            public void OneBlockTest()
579:            {
580:                double setArea;
581:                target.Add( new Circle(10.0) );
582:                setArea = 5.0 * 5.0 * 3.14;
583:                Assert.AreEqual(setArea, target.CalculateArea(),"面積計算");
584:                Assert.AreEqual(setArea * 10.0, target.CalculateWeight(),"重さ計算");
585:            }
```

```
586:            [Test]
587:            public void AnyBlockTest()
588:            {
589:                double setArea;
590:                target.Add( new Circle(10.0) );
591:                setArea = 5.0 * 5.0 * 3.14;
592:                target.Add( new Snowman(20.0) );
593:                setArea += 4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14;
594:                target.Add( new Rectangle(5.0, 10.0) );
595:                setArea += 5.0 * 10.0;
596:                Assert.AreEqual(setArea, target.CalculateArea(),"面積計算");
597:                Assert.AreEqual(setArea * 10.0, target.CalculateWeight(),"重さ計算");
598:            }
599:        }
600:    }
```

VB .NET の場合

```
601:    Imports NUnit.Framework
602:    Imports OSK.BuildingBlock.Core
603:    Namespace OSK.BuildingBlock.Tests
604:        <TestFixture()> _
605:        Public Class BlockSetTestCase
606:            Private targetObj As BlockSet
607:            <SetUp()> _
608:            Public Sub Setup()
609:                targetObj = New BlockSet()
610:            End Sub
611:            <TearDown()> _
612:            Public Sub TearDown()
613:                targetObj = Nothing
614:            End Sub
615:            <Test()> _
616:            Public Sub PropertyTest()
617:                targetObj.Nickname = "ブロックセット"
618:                Assert.AreEqual("ブロックセット", targetObj.Nickname, "プロパティのチェック")
619:            End Sub
620:            <Test()> _
621:            Public Sub NoBlock()
622:                Assert.AreEqual(0, targetObj.CalculateArea(),"面積計算")
623:                Assert.AreEqual(0, targetObj.CalculateWeight (),"重さ計算")
624:            End Sub
625:            <Test()> _
626:            Public Sub OneBlock()
627:                Dim setArea As Double
628:                targetObj.Add(New Circle(10.0))
629:                setArea = 5.0 * 5.0 * 3.14
630:                Assert.AreEqual(setArea, targetObj.CalculateArea(),"面積計算")
631:                Assert.AreEqual(setArea * 10.0, targetObj.CalculateWeight (),"重さ計算")
632:            End Sub
633:            <Test()> _
634:            Public Sub AnyBlock()
635:                Dim setArea As Double
636:                targetObj.Add(New Circle(10.0))
637:                setArea = 5.0 * 5.0 * 3.14
638:                targetObj.Add(New Snowman(20.0))
639:                setArea += 4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14
640:                targetObj.Add(New Rectangle(5.0, 10.0))
641:                setArea += 5.0 * 10.0
642:                Assert.AreEqual(setArea, targetObj.CalculateArea(),"面積計算")
643:                Assert.AreEqual(setArea * 10.0, targetObj.CalculateWeight (),"重さ計算")
644:            End Sub
645:        End Class
646:    End Namespace
```

③ 配列クラスを利用する

.NET Framework には、複数のインスタンスを扱うためのコレクションクラスが用意されています。以下のクラスが代表的なクラスです。

- ArrayList
  サイズを動的に変更できる配列クラスです。
- Queue
  先入れ先出しのコレクションクラスです。
- Stack
  後入れ先出しのコレクションクラスです。
- Hashtable
  キーのハッシュコードによって編成されたキーと値を組み合わせたコレクションです。
- SortedList
  キーによって並べ替えられた、キーと値を組み合わせたコレクションです。

今回、ArrayList を使用して、ブロックを管理します。

C#の場合

```
647:    using System;
648:    using System.Collections;
649:
650:    namespace OSK.BuildingBlock.Core
651:    {
652:        public class BlockSet : Block
653:        {
654:            private ArrayList blockSetObj;
655:            public BlockSet()
656:            {
657:                blockSetObj = new ArrayList();
658:            }
659:            public void Add( Block newBlock )
660:            {
661:                blockSetObj.Add(newBlock);
662:            }
663:            public override double CalculateArea()
664:            {
665:                double result = 0.0;
666:                Block blockObj;
667:                int i;
668:                for( i = 0; i < blockSetObj.Count; ++i )
669:                {
670:                    blockObj = (Block)blockSetObj[i];
671:                    result += blockObj.CalculateArea();
672:                }
673:                return result;
674:            }
675:        }
676:    }
```

VB .NET の場合

```
677:    Namespace OSK.BuildingBlock.Core
678:        Public Class BlockSet
679:            Inherits Block
680:            Private blockSetObj As ArrayList
681:            Public Sub New()
682:                blockSetObj = New ArrayList()
683:            End Sub
684:            Public Sub Add(ByRef newBlock As Block)
685:                blockSetObj.Add(newBlock)
686:            End Sub
```

```
687:            Public Overrides Function CalculateArea() As Double
688:                Dim result As Double
689:                Dim blockOoj As Block
690:                Dim i As Integer
691:                For i = 0 To (blockSetObj.Count - 1)
692:                    blockOoj = blockSetObj.Item(i)
693:                    result += blockOoj.CalculateArea()
694:                Next
695:                Return result
696:            End Function
697:        End Class
698:    End Namespace
```

Points

.NET Framework では、すべてのクラスが１つの親クラスである Object クラスを継承しています。.NET では、Object クラスの継承の記述は省略することができます。当然、提供されているクラスライブラリのクラスもすべて、Object クラスを継承しています。したがって、.NET で現れるすべてのインスタンスは、Object クラスのインスタンスとして扱うことができます。このことは、すべてのインスタンスをポルモーフィックに扱うことができることを意味しています。この性質を利用して、コレクションクラスは実装されています。

④ ブロックセットをブロックとして処理するテストを追加する

BlockSet がブロックとして扱えることをテストで確認します。

C#の場合

```
699:            [Test]
700:            public void BlockSetIsBlock()
701:            {
702:                BlockSet newBlockSet;
703:                double setArea;
704:                target.Add( new Circle(10.0) );
705:                setArea = 5.0 * 5.0 * 3.14;
706:                target.Add( new Snowman(20.0) );
707:                setArea += 4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14;
708:                target.Add( new Rectangle(5.0, 10.0) );
709:                setArea += 5.0 * 10.0;
710:                newBlockSet = new BlockSet();
711:                newBlockSet.Add(new Circle(10.0));
712:                newBlockSet.Add(new Snowman(20.0));
713:                newBlockSet.Add(new Rectangle(5.0, 10.0));
714:                target.Add(newBlockSet);
715:                setArea = setArea * 2.0;
716:                Assert.AreEqual(setArea, target.CalculateArea(),
717:                    "面積計算");
718:                Assert.AreEqual(setArea * 10.0, target.CalculateWeight(),
719:                    "重さ計算");
720:            }
```

VB .NET の場合

```
721:            <Test()> _
722:            Public Sub BlockSetIsBlock()
723:                Dim newBlockSet As BlockSet
724:                Dim setArea As Double
725:                targetObj.Add(New Circle(10.0))
726:                setArea = 5.0 * 5.0 * 3.14
727:                targetObj.Add(New Snowman(20.0))
728:                setArea += 4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14
729:                targetObj.Add(New Rectangle(5.0, 10.0))
730:                setArea += 5.0 * 10.0
```

```
731:                newBlockSet = New BlockSet()
732:                newBlockSet.Add(New Circle(10.0))
733:                newBlockSet.Add(New Snowman(20.0))
734:                newBlockSet.Add(New Rectangle(5.0, 10.0))
735:                targetObj.Add(newBlockSet)
736:                setArea = setArea * 2.0
737:                Assert.AreEqual(setArea, targetObj.CalculateArea(), _
738:                                        "面積計算")
739:                Assert.AreEqual(setArea * 10.0,targetObj.CalculateWeight (), _
740:                                         "重さ計算")
741:            End Sub
```

Step 2　インスタンスの作成を利用から切り離す

ポルモーフィズムを利用するときの問題として、インスタンスの作成があります。これは、親クラスだけですべての処理ができたとしても、インスタンスを作成には、具体的な子クラスが必要になるためです。ここまでの例で言い換えれば、Block という抽象的なクラスのインスタンスを作ることはできなくて、できるのは具体的な Circle クラスなどのブロックということです。この解決策として、インスタンスを作成する場所を一ヶ所に限定する方法があります。

①　クラスを作成する

ブロックを作成する BlockFactory クラスを作成します。このクラスは、作成できるブロックのリストを提供します。また、指定されたリストのブロックを作ります。

C#の場合

```
742:    using System;
743:
744:    namespace OSK.BuildingBlock.Core
745:    {
746:        public class BlockFactory
747:        {
748:            private string[] BlockList;
749:            public BlockFactory()
750:            {
751:                BlockList = new string[5];
752:                BlockList[0] = "直径 10 センチの丸形";
753:                BlockList[1] = "高さ 20 センチの雪だるま";
754:                BlockList[2] = "縦 5 センチ横 10 センチの四角形";
755:                BlockList[3] = "5 センチの正方形";
756:                BlockList[4] = "入門セット";
757:            }
758:            public string[] GetBlockList()
759:            {
760:                return BlockList;
761:            }
762:            public Block CreateBlock( int blockIndex )
763:            {
764:                return null;
765:            }
766:        }
767:    }
```

VB .NET の場合

```
768:  Namespace OSK.BuildingBlock.Core
769:      Public Class BlockFactory
770:          Private BlockList(4) As String
771:          Public Sub New()
772:              BlockList(0) = "直径 10 センチの丸形"
773:              BlockList(1) = "高さ 20 センチの雪だるま"
774:              BlockList(2) = "縦 5 センチ横 10 センチの四角形"
775:              BlockList(3) = "5 センチの正方形"
776:              BlockList(4) = "入門セット"
777:          End Sub
778:          Public Function GetBlockList() As String()
779:              Return BlockList
780:          End Function
781:          Public Function CreateBlock(ByVal blockIndex As Integer) As Block
782:              Return Nothing
783:          End Function
784:      End Class
785:  End Namespace
```

② テストクラスを作成する

テスト用クラスとして、BlockFactoryTestCase を作成します。
ブロックの作成を依頼して、作成されたブロックのクラスのチェックと面積のテストを行います。

C#の場合

```
786:  using System;
787:  using NUnit.Framework;
788:  using OSK.BuildingBlock.Core;
789:
790:  namespace OSK.BuildingBlock.Tests
791:  {
792:      [TestFixture]
793:      public class BlockFactoryTestCase
794:      {
795:          private BlockFactory target;
796:          [SetUp]
797:          public void Setup()
798:          {
799:              target = new BlockFactory();
800:          }
801:          [TearDown]
802:          public void TearDown()
803:          {
804:              target = null;
805:          }
806:          [Test]
807:          public void CreateBlockTest()
808:          {
809:              Block newBlock;
810:              double AllArea;
811:              newBlock = target.CreateBlock(0);
812:              Assert.IsNotNull(newBlock,"ブロックが作成されない");
813:              Assert.IsTrue(typeof(Circle) == newBlock.GetType(),"クラス違い");
814:              Assert.AreEqual(5.0 * 5.0 * 3.14, newBlock.CalculateArea(),"面積計算" );
815:              newBlock = target.CreateBlock(1);
816:              Assert.IsNotNull(newBlock,"ブロックが作成されない");
817:              Assert.IsTrue(typeof(Snowman) == newBlock.GetType(),"クラス違い" );
818:              Assert.AreEqual(4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14, newBlock.CalculateArea(),"面積計算" );
819:              newBlock = target.CreateBlock(2);
820:              Assert.IsNotNull(newBlock, "ブロックが作成されない");
821:              Assert.IsTrue(typeof(Rectangle) == newBlock.GetType() , "クラス違い");
822:              Assert.AreEqual(5.0 * 10.0, newBlock.CalculateArea() , "面積計算");
823:              newBlock = target.CreateBlock(3);
824:              Assert.IsNotNull(newBlock, "ブロックが作成されない");
825:              Assert.IsTrue(typeof(Rectangle) == newBlock.GetType() , "クラス違い");
826:              Assert.AreEqual(5.0 * 5.0, newBlock.CalculateArea(),"面積計算", );
```

```
827:                    newBlock = target.CreateBlock(4);
828:                    Assert.IsNotNull(newBlock,"ブロックが作成されない");
829:                    Assert.IsTrue( typeof(BlockSet) == newBlock.GetType() , "クラス違い");
830:                    AllArea = 5.0 * 5.0 * 3.14 + (4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14)
831:                                + 5.0 * 10.0 + 5.0 * 5.0;
832:                    Assert.AreEqual(AllArea, newBlock.CalculateArea() , "面積計算");
833:                }
834:            }
835:        }
```

VB .NET の場合

```
836:    Imports NUnit.Framework
837:    Imports OSK.BuildingBlock.Core
838:    Namespace OSK.BuildingBlock.Tests
839:        <TestFixture()> _
840:        Public Class BlockFactoryTestCase
841:            Private targetObj As BlockFactory
842:            <SetUp()> _
843:            Public Sub Setup()
844:                targetObj = New BlockFactory()
845:            End Sub
846:            <TearDown()> _
847:            Public Sub TearDown()
848:                targetObj = Nothing
849:            End Sub
850:            <Test()> _
851:            Public Sub CreateBlockTest()
852:                Dim newBlock As Block
853:                Dim AllArea As Double
854:                newBlock = targetObj.CreateBlock(0)
855:                Assert.IsNotNull(newBlock, "ブロックが作成されない")
856:                Assert.IsTrue((TypeOf newBlock Is Circle), "クラス違い")
857:                Assert.AreEqual(5.0 * 5.0 * 3.14, newBlock.CalculateArea() _
858:                                                "面積計算")
859:                newBlock = targetObj.CreateBlock(1)
860:                Assert.IsTrue((TypeOf newBlock Is Snowman), "クラス違い")
861:                Assert.AreEqual(4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14, newBlock.CalculateArea(), _
862:                                                "面積計算")
863:                newBlock = targetObj.CreateBlock(2)
864:                Assert.IsTrue( (TypeOf newBlock Is Rectangle), "クラス違い")
865:                Assert.AreEqual(5.0 * 10.0, newBlock.CalculateArea() _
866:                                                "面積計算")
867:                newBlock = targetObj.CreateBlock(3)
868:                Assert.IsTrue( (TypeOf newBlock Is Rectangle), "クラス違い")
869:                Assert.AreEqual(5.0 * 5.0, newBlock.CalculateArea()_
870:                                                "面積計算")
871:                newBlock = targetObj.CreateBlock(4)
872:                Assert.IsTrue( (TypeOf newBlock Is BlockSet), "クラス違い")
873:                AllArea = 5.0 * 5.0 * 3.14 + (4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14) _
874:                            + 5.0 * 10.0 + 5.0 * 5.0
875:                Assert.AreEqual(AllArea, newBlock.CalculateArea() _
876:                                                "面積計算")
877:            End Sub
878:        End Class
879:    End Namespace
```

③ クラスを完成させる

C#の場合

```
880:    using System;
881:
882:    namespace OSK.BuildingBlock.Core
883:    {
```

```
884:        public class BlockFactory
885:        {
886:            private string[] BlockList;
887:            public BlockFactory()
888:            {
889:                BlockList = new string[5];
890:                BlockList[0] = "直径 10 センチの丸形";
891:                BlockList[1] = "高さ 20 センチの雪だるま";
892:                BlockList[2] = "縦 5 センチ横 10 センチの四角形";
893:                BlockList[3] = "5 センチの正方形";
894:                BlockList[4] = "入門セット";
895:            }
896:            public string[] GetBlockList()
897:            {
898:                return BlockList;
899:            }
900:            public Block CreateBlock( int blockIndex )
901:            {
902:                Block newBlock = null;
903:
904:                switch( blockIndex )
905:                {
906:                    case 0:
907:                        newBlock = new Circle(10.0);
908:                        break;
909:                    case 1:
910:                        newBlock = new Snowman(20.0);
911:                        break;
912:                    case 2:
913:                        newBlock = new Rectangle(5.0, 10.0);
914:                        break;
915:                    case 3:
916:                        newBlock = new Rectangle(5.0, 5.0);
917:                        break;
918:                    case 4:
919:                        BlockSet newBlockSet;
920:                        newBlockSet = new BlockSet();
921:                        newBlockSet.Add( new Circle(10.0) );
922:                        newBlockSet.Add( new Snowman(20.0) );
923:                        newBlockSet.Add( new Rectangle(5.0,10.0) );
924:                        newBlockSet.Add( new Rectangle(5.0,5.0) );
925:                        newBlock = newBlockSet;
926:                        break;
927:                    default:
928:                        break;
929:                }
930:                return newBlock;
931:            }
932:        }
933:    }
```

VB .NET の場合

```
934:    Namespace OSK.BuildingBlock.Core
935:        Public Class BlockFactory
936:            Private BlockList(4) As String
937:            Public Sub New()
938:                BlockList(0) = "直径 10 センチの丸形"
939:                BlockList(1) = "高さ 20 センチの雪だるま"
940:                BlockList(2) = "縦 5 センチ横 10 センチの四角形"
941:                BlockList(3) = "5 センチの正方形"
942:                BlockList(4) = "入門セット"
943:            End Sub
944:            Public Function GetBlockList() As String()
945:                Return BlockList
946:            End Function
```

```
947:            Public Function CreateBlock(ByVal blockIndex As Integer) As Block
948:                Dim newBlock As Block
949:                Select Case blockIndex
950:                    Case 0
951:                        newBlock = New Circle(10.0)
952:                    Case 1
953:                        newBlock = New Snowman(20.0)
954:                    Case 2
955:                        newBlock = New Rectangle(5.0, 10.0)
956:                    Case 3
957:                        newBlock = New Rectangle(5.0, 5.0)
958:                    Case 4
959:                        Dim newBlockSet As BlockSet
960:                        newBlockSet = New BlockSet()
961:                        newBlockSet.Add(New Circle(10.0))
962:                        newBlockSet.Add(New Snowman(20.0))
963:                        newBlockSet.Add(New Rectangle(5.0, 10.0))
964:                        newBlockSet.Add(New Rectangle(5.0, 5.0))
965:                        newBlock = newBlockSet
966:                End Select
967:                Return newBlock
968:            End Function
969:        End Class
970:    End Namespace
```

Step 3　インターフェイスを導入する

通常、クラスが他のクラスを利用する場合、具体的なクラスを参照します。しかし、利用される側のクラスに変更があった場合、必ず利用する側のクラスも再コンパイルする必要があります。このように利用される側と利用する側の作成順序に制約が発生します。しかし、パソコンの基盤と CPU やハードディスクのように、インタフェースが標準的に定義されていて、その定義を厳守していれば、利用する側と利用される側の作成される順序に制約がなくなります。また、利用される側の機能アップしても、インターフェイスに変更がなければ、差し替えも可能です。インターフェイスは、オブジェクト指向の概念のカプセル化を最も忠実に実現したものです。

VB.NET では、インターフェイスを定義することができるようになっています。このインターフェイスも、パソコンなどの実世界のインターフェイスと同じで、定義だけで、実装は一切含まれていません。インターフェイスには、メソッド、プロパティ、イベントを定義することができます。インターフェイスのイベント定義の書式は、以下のようになります。

C#の場合

```
public Interface インターフェイス名
{
    ‘メソッドやプロパティ、イベントの定義
}
```

VB .NET の場合

```
Public Interface インターフェイス名

    ‘メソッドやプロパティ、イベントの定義

End Interface
```

① インターフェイスを定義する

インターフェイスは、クラスの特殊な形態ですから、クラスを作成する手順と同じです。一般的に、インターフェイスは、先頭に I を付加します。

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	IBlock.cs（VB の場合は IBlock.vb）

インターフェイスは、型を定義するだけですので、すべてのメソッドやプロパティは必ず継承したクラスが実装する必要があります。したがって、クラスには、public（VB の場合は Public）などの定義や abstract （VB の場合は MustInherit）のキーワードは付加しません。ここで定義するインターフェイスは、プロパティとしてニックネームとメソッドとして面積の計算と重さの計算を用意しています。

C#の場合

```
971:    using System;
972:
973:    namespace OSK.BuildingBlock.Core
974:    {
975:        public interface IBlock
976:        {
977:            string Nickname
978:            {
979:                get;
980:                set;
981:            }
982:            double CalculateArea();
983:            double CalculateWeight();
984:        }
985:    }
```

VB .NET の場合

```
986:    Namespace OSK.BuildingBlock.Core
987:        Public Interface IBlock
988:            Property Nickname() As String
989:            Function CalculateArea() As Double
990:            Function CalculateWeight () As Double
991:        End Interface
992:    End Namespace
```

② インターフェイスを継承させる

インターフェイスは、クラスと同様に継承させることができます。クラスの継承と異なり、インターフェイスは複数のインターフェイスを継承させることができます。（クラスの継承は、一つだけです）
インターフェイスの継承は、クラス継承と同じ「:」に続け、インターフェイス名を記述します。（VB の場合はキーワード Implements を使用します。）Block クラスに IBlock を継承させます。
インターフェイスのメソッドやプロパティを実装する場合も、クラスの実装であることを宣言します。

C#の場合

```
993:    using System;
994:
995:    namespace OSK.BuildingBlock.Core
996:    {
997:        public abstract class Block : IBlock
998:        {
```

```
999:            private string nicknameValue;
1000:           public string Nickname
1001:           {
1002:               get
1003:               {
1004:                   return nicknameValue;
1005:               }
1006:               set
1007:               {
1008:                   nicknameValue = value;
1009:               }
1010:           }
1011:           public abstract double CalculateArea();
1012:           public double CalculateWeight()
1013:           {
1014:               return CalculateArea() * 10.0;
1015:           }
1016:       }
1017:   }
```

VB .NET の場合

```
1018:   Namespace OSK.BuildingBlock.Core
1019:       Public MustInherit Class Block
1020:           Implements IBlock
1021:           Private nicknameValue As String
1022:           Public Property Nickname() As String _
1023:               Implements OSK.BuildingBlock.Core.IBlock.Nickname
1024:               Get
1025:                   Return nicknameValue
1026:               End Get
1027:               Set(ByVal Value As String)
1028:                   nicknameValue = Value
1029:               End Set
1030:           End Property
1031:           Public MustOverride Function CalculateArea() As Double _
1032:               Implements OSK.BuildingBlock.Core.IBlock.CalculateArea
1033:           Public Function CalculateWeight () As Double _
1034:               Implements OSK.BuildingBlock.Core.IBlock.CalculateWeight
1035:               Return CalculateArea() * 10.0
1036:           End Function
1037:       End Class
1038:   End Namespace
```

Points

インターフェイスのメソッドやプロパティを実装するときに個別に宣言を行うのは、複数のインターフェイスを実装したときに、まったく名前や型が同じメソッドがあった場合でも、実装できるようにするためです。

③ インターフェイスを利用する

BlockFactory クラスの CreateBlock メソッドの返り値を IBlock に変更しましょう。BlockFactoryTestCase のテストも変更します。

Step 4　委譲を実装する

既に定義されているクラス利用して、新しいクラスを作成したいと考えたときでも、必ずしも継承ができるわけではありません。例えば、利用できる機能が一部分だけであったり、利用したいクラスが継承を禁止していたり、既に他のクラスを継承する必要がある場合などです。このようなときは、継承ではなく、利用したいクラスを内部に持つことで、解決することができます。このようなとき、外部からの処理依頼の一部をこの内部クラスに依頼することになります。このようなクラスの利用方法を委譲と呼びます。

①　クラスを作成する

四角形のクラス Rectangule を委譲で使用して、正方形クラスの Square クラスを新規作成します。（今回の場合、継承を使用しても作成することができます）

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	Square.cs（VB の場合は Square.vb）

クラスの初期のコードは以下のようになります。

C#の場合

```
1039:  using System;
1040:
1041:  namespace OSK.BuildingBlock.Core
1042:  {
1043:      public class Square : Block
1044:      {
1045:          public Square(double side)
1046:          {
1047:          }
1048:          public override double CalculateArea()
1049:          {
1050:              return 0.0;
1051:          }
1052:      }
1053:  }
```

VB .NET の場合

```
1054:  Namespace OSK.BuildingBlock.Core
1055:      Public Class Square
1056:          Inherits Block
1057:          Public Sub New(ByVal side As Double)
1058:          End Sub
1059:          Public Overrides Function CalculateArea() As Double
1060:              Return 0.0
1061:          End Function
1062:      End Class
1063:  End Namespace
```

② テストクラスを作成する

Square クラスのテスト用クラスとして、SquareTestCase を作成します。

C#の場合

```
1064:  using System;
1065:  using NUnit.Framework;
1066:  using OSK.BuildingBlock.Core;
1067:
1068:  namespace OSK.BuildingBlock.Tests
1069:  {
1070:      [TestFixture]
1071:      public class SquareTestCase
1072:      {
1073:          private Square target;
1074:          [SetUp]
1075:          public void Setup()
1076:          {
1077:              target = new Square(5.0);
1078:          }
1079:          [TearDown]
1080:          public void TearDown()
1081:          {
1082:              target = null;
1083:          }
1084:          [Test]
1085:          public void CalculateAreaTest()
1086:          {
1087:              Assert.AreEqual(5.0 * 5.0, target.CalculateArea(),"面積計算");
1088:          }
1089:          [Test]
1090:          public void CalculateWeightTest()
1091:          {
1092:              Assert.AreEqual(5.0 * 5.0 * 10.0, target.CalculateWeight(),"重さ計算");
1093:          }
1094:      }
1095:  }
```

VB .NET の場合

```
1096:  Imports NUnit.Framework
1097:  Imports OSK.BuildingBlock.Core
1098:  Namespace OSK.BuildingBlock.Tests
1099:      <TestFixture()> _
1100:      Public Class SquareTestCase
1101:          Private targetObj As Square
1102:          <SetUp()> _
1103:          Public Sub Setup()
1104:              targetObj = New Square(5.0)
1105:          End Sub
1106:          <TearDown()> _
1107:          Public Sub TearDown()
1108:              targetObj = Nothing
1109:          End Sub
1110:          <Test()> _
1111:          Public Sub CalculateArea()
1112:              Assert.AreEqual(5.0 * 5.0, targetObj.CalculateArea(), _
1113:                                          "面積計算")
1114:          End Sub
1115:          <Test()> _
1116:          Public Sub CalculateWeight ()
1117:              Assert.AreEqual(5.0 * 5.0 * 10.0, targetObj.CalculateWeight (),"重さ計算")
1118:          End Sub
1119:      End Class
1120:  End Namespace
```

③ クラスを完成させる

処理はすべて、内部に持っている Rectangle のインスタンスに依頼します。

C#の場合

```
1121:  using System;
1122:
1123:  namespace OSK.BuildingBlock.Core
1124:  {
1125:      public class Square : Block
1126:      {
1127:          Rectangle rectanguleObject;
1128:          public Square(double side)
1129:          {
1130:              rectanguleObject = new Rectangle( side,side );
1131:          }
1132:          public override double CalculateArea()
1133:          {
1134:              return rectanguleObject.CalculateArea();
1135:          }
1136:      }
1137:  }
```

VB .NET の場合

```
1138:  Namespace OSK.BuildingBlock.Core
1139:      Public Class Square
1140:          Inherits Block
1141:          Private rectanguleObject As Rectangle
1142:          Public Sub New(ByVal side As Double)
1143:              rectanguleObject = New Rectangle(side, side)
1144:          End Sub
1145:          Public Overrides Function CalculateArea() As Double
1146:              Return rectanguleObject.CalculateArea()
1147:          End Function
1148:      End Class
1149:  End Namespace
```

④ クラスを利用する

BlockFactory クラスの CreateBlock メソッド内の正方形を作るのに Rectangle を利用していた部分 Square に変更します。
まず、はじめにテストを以下のように修正します。

C#の場合

```
1150:      [TestFixture]
1151:      public class BlockFactoryTestCase
1152:      {
1153:  //中略
1154:              newBlock = target.CreateBlock(3);
1155:              Assert.IsNotNull(newBlock, "ブロックが作成されない");
1156:              Assert.IsTrue(typeof(Square) == newBlock.GetType(),"クラス違い" );
```

ここを Square になる

VB .NET の場合

```
1157:      <TestFixture()> _
1158:      Public Class BlockFactoryTestCase
1159:  '中略
1160:              newBlock = targetObj.CreateBlock(3)
1161:              Assert.IsTrue((TypeOf newBlock Is Square), "クラス違い")
1162:              Assert.AreEqual(5.0 * 5.0, newBlock.CalculateArea(),_
1163:                                  "面積計算")
```

テスト結果が緑になるようにクラスを変更する。

Session3　Windows アプリケーションを作成する

. Overview

.NET を使用して Windows アプリケーションを作成します。.NET でも、VB6 と同様の手順で Widnows アプリケーションを作成することができます。しかし、Windows アプリケーションで問題となるのがテストです。ユーザインターフェイスをテストする場合、手作業で行うことがほとんどでした。特に、VB6 のプログラムでは、コントロールのイベントをハンドルするメソッドの中にすべての処理を記述することが多かったため、自動がすることは非常に困難でした。テストツールを使って、操作をトレースしても、結果は目で確認しなければならなかったため、自動化は行われていないことがほとんどでした。手作業によるテストは、非常に時間が掛かり、しかも退屈であるため、納期が近づくと部分的にしか行われないことになりがちでした。結果として、リリース後に不具合が発生することも、まれではありませんでした。

.NET でも、特に意識しないでプログラムを作成するとまったく同じ状態に陥ることになります。これを回避するために、必要になるのが、Form クラスに記述されていた処理を別のクラスに分離することです。また、この分離したクラスは、EXE を作成するプロジェクトではなく、クラスライブラリ（DLL 形式）のプロジェクトに作成する必要があります。これは EXE を作成するプロジェクトのクラスは、外部から呼び出すことができないからです。呼び出せないのでは、自動テストを行うことはできません。この Session では、.NET で強化された Windows アプリケーションに関する新機能を紹介するとともに、ユーザインターフェイスのテストについて説明を行います。

この Session でも、引き続き、積み木システムを例題として、作業を進めます。

Goal

- ❒ Windows アプリケーションを作成する
- ❒ ユーザインターフェイスをテストする

Step 1　新規プロジェクトを作成する

①　Windows アプリケーション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。「各言語プロジェクト」の「Windows アプリケーション」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。

設定項目	設定内容
プロジェクトの種類	各言語プロジェクト
テンプレート	Windows アプリケーション
プロジェクト名	GUI
場所	¥dotNetSeminar¥Exercises¥ BuildingBlock¥Source

②　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 GUI 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.BuildingBlock.GUI
既定の名前空間	OSK.BuildingBlock.GUI

VB .NET 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.BuildingBlock.GUI
全般	ルート名前空間	[空白]

③ ユーザインターフェイス補助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。「各言語プロジェクト」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。

設定項目	設定内容
プロジェクトの種類	各言語プロジェクト
テンプレート	クラスライブラリ
プロジェクト名	UIState
場所	¥dotNetSeminar¥Exercises¥ BuildingBlock¥Source

Class1.cs（VB では.vb)は削除します。

④ 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 UIState 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

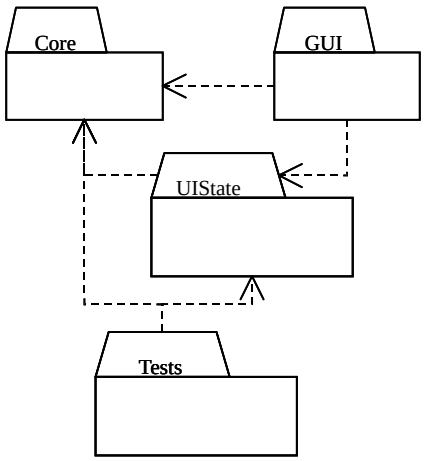
設定項目	設定内容
アセンブリ名	OSK.BuildingBlock.UIState
既定の名前空間	OSK.BuildingBlock.UIState

VB .NET 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.BuildingBlock.UIState
全般	ルート名前空間	[空白]

⑤ 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。まず、「GUI」プロジェクトで「Core」プロジェクトと「UIState」プロジェクトの参照設定を行います。次に、「UIState」プロジェクトで「Core」プロジェクトの参照設定を行います。最後に、「Tests」プロジェクトに「UIState」プロジェクトの参照設定を行います。依存関係は、以下の図のようになります。



Step 2　フォームを作成する

①　フォームのコードを変更する

「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 Form1 のコンテキストメニューを表示し、「コード表示」を選択します。以下のようにコードを変更します。変更箇所は、名前空間とクラス名です。

C#の場合

```
1164:  namespace OSK.OSK.BuildingBlock.GUI
1165:  {
1166:          public class BlockSuiteForm : System.Windows.Forms.Form
1167:
1168:  // 中略
1169:
1170:                  [STAThread]
1171:                  static void Main()
1172:                  {
1173:                          Application.Run(new BlockSuiteForm());
1174:                  }
1175:          }
1176:  }
```

VB .NET の場合

```
1177:  Namespace OSK.BuildingBlock.GUI
1178:      Public Class BlockSuiteForm
1179:          Inherits System.Windows.Forms.Form
1180:  #Region " Windows フォーム デザイナで生成されたコード "
1181:
1182:  #End Region
1183:      End Class
1184:  End Namespace
```

②　ファイル名を変更する

「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 Form1 の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。ファイル名を BlockSuiteForm に変更します。

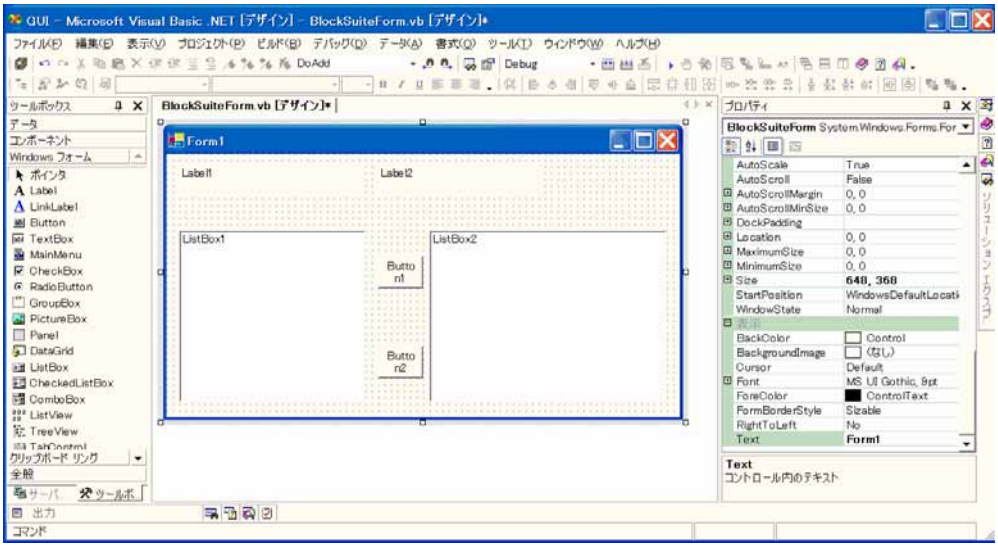
③　プロジェクトの属性を変更する

Form1 のクラス名を変更したので、関連する以下のプロジェクトの属性を設定します。

プロパティ種別	設定項目	設定内容
全般	スタートアップの設定	OSK.BuildingBlock.GUI.BlockSuiteForm

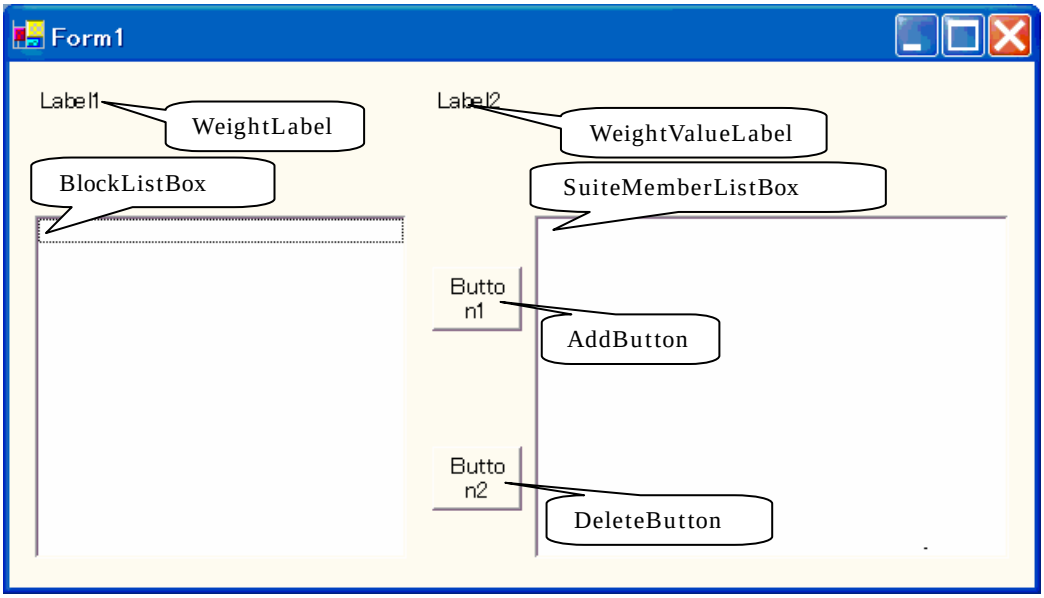
④ フォームにコントロールを配置する

ツールボックスからコントロールをドラッグ・アンド・ドロップして、以下のようなウィンドウを作成します。



⑤ コントロール名を変更する

プログラムの作成をスムーズに進めるために、コントロール名を意味のあるものに変更します。また、各コントロールのプロパティを以下のように変更します。



BlockSuiteForm について、以下の設定を行う

種別	項目	設定内容
表示	Text	ブロックセット重量計算
配置	MinimumSize	400,350

WeightLabel について、以下の設定を行う

種別	項目	設定内容
表示	Text	重量トータル
表示	Font	サイズを 18

WeightValueLabel について、以下の設定を行う

種別	項目	設定内容
表示	Text	[空白]
表示	Font	サイズを 18
表示	BorderStyle	Fixed3D

AddButton について、以下の設定を行う

種別	項目	設定内容
表示	Text	追加

DeleteButton について、以下の設定を行う

種別	項目	設定内容
表示	Text	削除

⑥ アンカーリングの設定を行う

アンカーリングとは、ウィンドウのサイズを変更したときに、そのサイズに合わせて、コントロールの位置を自動的に変更する機能です。

WeightLabel について

種別	項目	設定内容
位置	Anchor	Top,Left

WeightValueLabel について

種別	項目	設定内容
位置	Anchor	Top,Left,Right

AddButton について

種別	項目	設定内容
位置	Anchor	Top,Left

DeleteButton について、以下の設定を行う

種別	項目	設定内容
位置	Anchor	Bottm,Left

BlockListBox について

種別	項目	設定内容
位置	Anchor	Top, Bottom,Left

SuiteMemberListBox について、以下の設定を行う

種別	項目	設定内容
位置	Anchor	Top, Bottom, Left,Right

⑦ ウィンドウを表示させる

プロジェクト GUI を一時的に「スタートアッププロジェクトに設定」を行い、ウィンドウを表示させてみる。

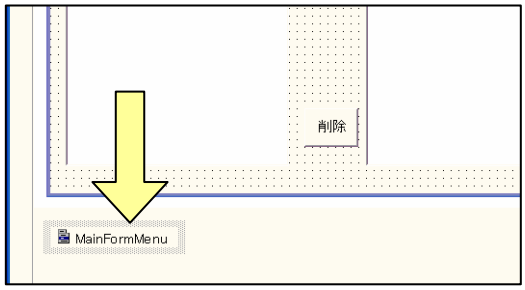
Topic

VB.NET になって、フォームに必ず最前列に表示するためのプロパティ「TopMost」や半透明にするプロパティ「Opacity」、更には、透過する色を設定するプロパティ「TransparencyKey」などが追加されている。

Step 3　メニューを作成する

①　フォームに、メニューコントロールをドロップする

「ツールボックス」ウィンドウから「MainMenu」コントロールをフォームに、ドラッグ&ドロップします。コントロール名を「MainFormMenu」に変更します。
その後、ウィンドウ上のメニューに直接「終了(&X)」と入力します。
BlockSuiteForm のプロパティの Menu を MainFormMenu に設定します。



②　メニュー「終了(&X)」のコードを作成する

ウィンドウの「終了(&X)」メニューをダブルクリックすると、処理を行うコードが生成されます。そこに、以下のように実装を行います。

C#の場合

```
1185:       private void MenuItem1_Click(object sender, System.EventArgs e)
1186:       {
1187:           Application.Exit();
1188:       }
```

VB .NET の場合

```
1189:       Private Sub MenuItem1_Click _
1190:           (ByVal sender As System.Object, ByVal e As System.EventArgs) _
1191:           Handles MenuItem1.Click
1192:           Application.Exit()
1193:       End Sub
```

Step 4　ユーザインターフェイスをテストする

今回のウィンドウでは、以下のようなクライアント部分のロジックを用意します。

- 初期表示のチェック
  ブロックのリストに正しく、ブロックが表示されている
- 追加処理
  指定されたブロックがリストに追加される
- 削除処理
  指定されたブロックがリストから削除される
- 重量トータル表示
  重量トータルは、カンマ編集されて少数点以下 1 桁で「xxxx.x グラム」と表示される
  初期値、追加、削除によって、重量トータル表示が変更される

①　クラスを作成する

UIState プロジェクトに、フォーム BlockSuiteForm クラスの処理を制御するクラスとして BlockSuiteFormAction 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	BlockSuiteFormAction.vs(VB の場合は vb)

クラスを名前空間で囲み、利用するクラスライブラリ OSK.BuildingBlock.Core を定義します。

初期表示時にブロックのリストを表示するために必要なメソッド 2 つを用意します。

C#の場合

```
1194:  using System;
1195:  using OSK.BuildingBlock.Core;
1196:
1197:  namespace OSK.BuildingBlock.UIState
1198:  {
1199:      public class BlockSuiteFormAction
1200:      {
1201:          public int GetBlockListCount()
1202:          {
1203:              return 0;
1204:          }
1205:          public string GetBlockName( int index )
1206:          {
1207:              return "";
1208:          }
1209:      }
1210:  }
```

VB .NET の場合

```
1211:  Imports OSK.BuildingBlock.Core
1212:  Namespace OSK.BuildingBlock.UIState
1213:      Public Class BlockSuiteFormAction
1214:          Public Function GetBlockListCount() As Integer
1215:              Return 0
1216:          End Function
1217:          Public Function GetBlockName(ByVal index As Integer) As String
1218:              Return ""
1219:          End Function
1220:      End Class
1221:  End Namespace
```

② テストクラスを作成する

BlockSuiteForm クラスをテストするクラスを BlockSuiteFormActionTestCase という名前で作成します。作成するプロジェクトは、Tests プロジェクトです。

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	BlockSuiteFormActionTestCase.cs(VB の場合は.vb)

C#の場合

```
1222:  using System;
1223:  using NUnit.Framework;
1224:  using OSK.BuildingBlock.Core;
1225:  using OSK.BuildingBlock.UIState;
1226:
1227:  namespace OSK.BuildingBlock.Tests
1228:  {
1229:      [TestFixture]
1230:      public class BlockSuiteFormActionTestCase
1231:      {
1232:          private BlockSuiteFormAction target;
1233:          [SetUp]
1234:          public void Setup()
1235:          {
1236:              target = new BlockSuiteFormAction();
1237:          }
```

```
1238:             [TearDown]
1239:             public void TearDown()
1240:             {
1241:                 target = null;
1242:             }
1243:         }
1244:     }
```

VB .NET の場合

```
1245:   Imports NUnit.Framework
1246:   Imports OSK.BuildingBlock.Core
1247:   Imports OSK.BuildingBlock.UIState
1248:   Namespace OSK.BuildingBlock.Tests
1249:       <TestFixture()> _
1250:       Public Class BlockSuiteFormActionTestCase
1251:           Private targetObj As BlockSuiteFormAction
1252:           <SetUp()> _
1253:           Public Sub Setup()
1254:               targetObj = New BlockSuiteFormAction()
1255:           End Sub
1256:           <TearDown()> _
1257:           Public Sub TearDown()
1258:               targetObj = Nothing
1259:           End Sub
1260:       End Class
1261:   End Namespace
```

③ 初期化テストを作成する

初期状態をチェックするテストを作成します。テストは、ブロックの数、そのブロックの名前をチェックします。

C#の場合

```
1262:             [Test]
1263:             public void InitTest()
1264:             {
1265:                 Assert.AreEqual(5,target.GetBlockListCount(),"ブロックの数が合わない" );
1266:                 Assert.AreEqual("直径 10 センチの丸形",              target.GetBlockName(0) );
1267:                 Assert.AreEqual("高さ 20 センチの雪だるま",          target.GetBlockName(1) );
1268:                 Assert.AreEqual("縦 5 センチ横 10 センチの四角形", target.GetBlockName(2) );
1269:                 Assert.AreEqual("5 センチの正方形",                  target.GetBlockName(3) );
1270:                 Assert.AreEqual("入門セット",                        target.GetBlockName(4) );
1271:             }
```

VB .NET の場合

```
1272:           <Test()> _
1273:           Public Sub InitTest()
1274:               Assert.AreEqual(5, targetObj.GetBlockListCount (), _
1275:                                   "ブロックの数が合わない")
1276:               Assert.AreEqual("直径 10 センチの丸形", _
1277:                                   targetObj.GetBlockName(0))
1278:               Assert.AreEqual("高さ 20 センチの雪だるま", _
1279:                                   targetObj.GetBlockName(1))
1280:               Assert.AreEqual("縦 5 センチ横 10 センチの四角形", _
1281:                                   targetObj.GetBlockName(2))
1282:               Assert.AreEqual("5 センチの正方形", _
1283:                                   targetObj.GetBlockName(3))
1284:               Assert.AreEqual("入門セット", _
1285:                                   targetObj.GetBlockName(4))
1286:           End Sub
```

テストが通るように、クラスの実装を行います。

④ 追加と削除を実装する

追加処理と削除処理を実装します。BlockSuiteFormAction クラスを追加処理と削除処理を行うためにメソッド 4 つを用意します。

C#の場合

```
1287:       public void AddBlock( int index )
1288:       {
1289:       }
1290:       public void DeleteBlock( int index )
1291:       {
1292:       }
1293:       public int GetSuiteMemberListCount()
1294:       {
1295:           return 0;
1296:       }
1297:       public double GetTotalWeight()
1298:       {
1299:           return 0.0;
1300:       }
```

VB .NET の場合

```
1301:       Public Sub AddBlock(ByVal index As Integer)
1302:       End Sub
1303:       Public Sub DeleteBlock(ByVal index As Integer)
1304:       End Sub
1305:       Public Function GetSuiteMemberListCount() As Integer
1306:           Return 0
1307:       End Function
1308:       Public Function GetTotalWeight() As Double
1309:           Return 0.0
1310:       End Function
```

追加処理と削除処理をチェックするテストを作成します。テストは、追加および削除処理を行い、ブロックの数、合計の重さをチェックします。

C#の場合

```
1311:       private const double Weight0 = 5.0 * 5.0 * 3.14 * 10.0;
1312:       private const double Weight1 = (4.0 * 4.0 * 3.14 + 6.0 * 6.0 * 3.14) * 10.0;
1313:       private const double Weight2 = 5.0 * 10.0 * 10.0;
1314:       private const double Weight3 = 5.0 * 5.0 * 10.0;
1315:       private const double Weight4 = Weight0 + Weight1 + Weight2 + Weight3;
1316:       [Test]
1317:       public void AddDeleteTest()
1318:       {
1319:           double totalWeight;
1320:
1321:           Assert.AreEqual(0, target.GetSuiteMemberListCount(),"初期エラー");
1322:           Assert.AreEqual(0.0, target.GetTotalWeight(),"初期エラー");
1323:           target.AddBlock(4);
1324:           totalWeight = Weight4;
1325:           target.AddBlock(3);
1326:           totalWeight += Weight3;
1327:           target.AddBlock(0);
1328:           totalWeight += Weight0;
1329:           Assert.AreEqual(3, target.GetSuiteMemberListCount(),"追加エラー");
1330:           Assert.AreEqual(totalWeight, target.GetTotalWeight(),"追加エラー");
1331:           target.DeleteBlock(1);
1332:           totalWeight -= Weight3;
1333:           Assert.AreEqual(2, target.GetSuiteMemberListCount(),"削除エラー");
1334:           Assert.AreEqual(totalWeight, target.GetTotalWeight(),"削除エラー");
1335:       }
```

VB .NET の場合

```
1336:       <Test()> _
1337:       Public Sub AddDeleteTest()
1338:           Dim totalWeight As Double
1339:           Assert.AreEqual(0, targetObj.GetSuiteMemberListCount(),"初期エラー")
1340:           Assert.AreEqual(0.0, targetObj.GetTotalWeight(),"初期エラー")
1341:           targetObj.AddBlock(4)
1342:           totalWeight += Weight4
1343:           targetObj.AddBlock(3)
1344:           totalWeight += Weight3
1345:           targetObj.AddBlock(0)
1346:           totalWeight += Weight0
1347:           Assert.AreEqual(3, targetObj.GetSuiteMemberListCount(),"追加エラー")
1348:           Assert.AreEqual(totalWeight, targetObj.GetTotalWeight(),"追加エラー")
1349:           targetObj.DeleteBlock(1)
1350:           totalWeight -= Weight3
1351:           Assert.AreEqual(2, targetObj.GetSuiteMemberListCount(),"削除エラー")
1352:           Assert.AreEqual(totalWeight, targetObj.GetTotalWeight()",削除エラー")
1353:       End Sub
```

⑤ 重量トータルの文字列を実装する

重量トータルの文字列を実装します。BlockSuiteFormAction クラスを重量トータルの文字列を行うためにメソッドを用意します。

C#の場合

```
1354:       public string GetTotalWeightString()
1355:       {
1356:           return "";
1357:       }
```

VB .NET の場合

```
1358:       Public Function GetTotalWeightString() As String
1359:           Return ""
1360:       End Function
```

重量トータルの文字列をチェックするテストを作成します。テストは、追加および削除処理のテストに追加を行います。

C#の場合

```
1361:           Assert.AreEqual("0.0 グラム", target.GetTotalWeightString(),"初期エラー");
1362:   // 省略
1363:           Assert.AreEqual("4,202.8 グラム", target.GetTotalWeightString(),"追加エラー");
1364:   // 省略
1365:           Assert.AreEqual("3,952.8 グラム", target.GetTotalWeightString(),"削除エラー");
```

VB .NET の場合

```
1366:           Assert.AreEqual("0.0 グラム", targetObj.GetTotalWeightString(), _
1367:                                         "初期エラー")
1368:   '省略
1369:           Assert.AreEqual("4,202.8 グラム", targetObj.GetTotalWeightString(),_
1370:                                         "追加エラー")
1371:   '省略
1372:           Assert.AreEqual("3,952.8 グラム", targetObj.GetTotalWeightString(), _
1373:                                         "削除エラー")
```

また、3 桁の表示のカンマ編集については、以下のテストを行います。

C#の場合

```
1374:        [Test]
1375:        public void TotalWeightStringTest()
1376:        {
1377:            target.AddBlock(3);
1378:            Assert.AreEqual("250.0 グラム", target.GetTotalWeightString(),"追加エラー");
1379:        }
```

VB .NET の場合

```
1380:        <Test()> _
1381:        Public Sub TotalWeightStringTest()
1382:            targetObj.AddBlock(3)
1383:            Assert.AreEqual("250.0 グラム", targetObj.GetTotalWeightString(), _
1384:                                        "追加エラー",)
1385:        End Sub
```

⑥ 通知イベントを宣言する

重量トータルの変更をイベントとして実装します。
VB6 では、コマンドボタンなどのコントロールから通知されるイベントを使用したプログラムが行われました。.NET Framework が提供するイベントの機構は、デリゲートと呼ばれます。コントロールだけではなく、クラスでもイベントを利用したプログラムを作成することができます。特徴として、非常に簡単に利用できることと、イベントを発信するクラスは、イベントを受信する側のクラスに関する情報をまったく必要としないことです。
まず、はじめに、イベントをクラスの外で定義します。

C#の場合

```
1386:        public delegate void WeightUpdateEventHandler();
```

VB .NET の場合

```
1387:    Public Delegate Sub WeightUpdateEventHandler()
```

次に、発行するイベントをクラス内で宣言します。

C#の場合

```
1388:        public event WeightUpdateEventHandler WeightUpdateEvent;
```

VB .NET の場合

```
1389:        Public Event WeightUpdateEvent As WeightUpdateEventHandler
```

最後に、初期化、追加、削除時にイベントを発行します。また、初期化メソッドも追加します。

C#の場合

```
1390:        WeightUpdateEvent();
```

VB .NET の場合

```
1391:        RaiseEvent WeightUpdateEvent()
```

イベント発行のテストを行います。

C#の場合

初期化時に、イベントハンドラを追加します。ハンドラを new し、処理を行うメソッドと関連付けます。

```
1392:        target.WeightUpdateEvent += new WeightUpdateEventHandler( WeightUpdateEventHandler );
```

VB .NET の場合

テストクラス内の BlockSuiteFormAction の宣言文に WithEvents を追加します。

```
1393:        Private WithEvents targetObj As BlockSuiteFormAction
```

初期化処理を追加します。

C#の場合

```
1394:        public void Initialize()
1395:        {
1396:        }
```

VB .NET の場合

```
1397:        Public Sub Initialize()
1398:        End Sub
```

次に、テストの作成です。初期化および追加処理の前にフラグを False に設定し、イベントが発行されたら True になるようにしてテストを行います。
イベント処理をするメソッドの定義は、VB6 のコントロールのイベント処理を追加する場合と同じ方法で作成することができます。具体的な操作は、エディタ上部の左のリストボックスから変数を選択して、右のリストボックスからイベントを選択します。

C#の場合

```
1399:        private bool eventFlag;
1400:        private void WeightUpdateEventHandler()
1401:        {
1402:            eventFlag = true;
1403:        }
1404:        [Test]
1405:        public void InitializeWeightUpdateEventTest()
1406:        {
1407:            eventFlag = false;
1408:            target.Initialize();
1409:            Assert.IsTrue(eventFlag,"Event 発行エラー");
1410:        }
1411:        [Test]
1412:        public void WeightUpdateEventTest()
1413:        {
1414:            eventFlag = false;
1415:            target.AddBlock(3);
1416:            Assert.IsTrue(eventFlag,"Event 発行エラー");
1417:        }
```

VB .NET の場合

```
1418:        Private eventFlag As Boolean
1419:        Private Sub targetObj_WeightUpdateEvent() Handles targetObj.WeightUpdateEvent
1420:            eventFlag = True
1421:        End Sub
1422:        <Test()> _
1423:        Public Sub InitializeWeightUpdateEventTest()
1424:            eventFlag = False
1425:            targetObj.Initialize()
1426:            Assert.IsTrue(eventFlag,"Event 発行エラー")
1427:        End Sub
1428:        <Test()> _
1429:        Public Sub WeightUpdateEventTest()
1430:            eventFlag = False
1431:            targetObj.AddBlock(3)
1432:            Assert.IsTrue(eventFlag,"Event 発行エラー")
1433:        End Sub
```

最終的な BlockSuiteFormAction クラスは以下のようになります。

C#の場合

```
1434:  using System;
1435:  using System.Collections;
1436:  using OSK.BuildingBlock.Core;
1437:  namespace OSK.BuildingBlock.UIState
1438:  {
1439:      public delegate void WeightUpdateEventHandler();
1440:      public class BlockSuiteFormAction
1441:      {
1442:          private BlockFactory blockFactoryObject;
1443:          private string[] blockListString;
1444:          private ArrayList suiteMemberListObject;
1445:          public event WeightUpdateEventHandler WeightUpdateEvent;
1446:          public BlockSuiteFormAction()
1447:          {
1448:              blockFactoryObject = new BlockFactory();
1449:              suiteMemberListObject    =new ArrayList();
1450:              blockListString = blockFactoryObject.GetBlockList();
1451:          }
1452:          public void Initialize()
1453:          {
1454:              WeightUpdateEvent();
1455:          }
1456:          public int GetBlockListCount()
1457:          {
1458:              return blockListString.Length;
1459:          }
1460:          public string GetBlockName( int index )
1461:          {
1462:              return blockListString[index];
1463:          }
1464:          public void AddBlock( int index )
1465:          {
1466:              IBlock newBlock;
1467:              newBlock = blockFactoryObject.CreateBlock(index);
1468:              suiteMemberListObject.Add(newBlock);
1469:              WeightUpdateEvent();
1470:          }
1471:          public void DeleteBlock( int index )
1472:          {
1473:              suiteMemberListObject.RemoveAt(index);
1474:              WeightUpdateEvent();
1475:          }
1476:          public int GetSuiteMemberListCount()
1477:          {
1478:              return suiteMemberListObject.Count;
1479:          }
1480:          public double GetTotalWeight()
1481:          {
1482:              int i;
1483:              double totalWeight = 0.0;
1484:              IBlock blockObject;
1485:              for( i = 0; i < GetSuiteMemberListCount(); ++i )
1486:              {
1487:                  blockObject = (IBlock)suiteMemberListObject[i];
1488:                  totalWeight += blockObject.CalculateWeight();
1489:              }
1490:              return totalWeight;
1491:          }
1492:          public string GetTotalWeightString()
1493:          {
1494:              string totalWeightString;
1495:              totalWeightString = string.Format("{0:#,##0.0}グラム", GetTotalWeight());
1496:              return totalWeightString;
1497:          }
1498:      }
1499:  }
```

VB .NET の場合

```
1500:  Imports OSK.BuildingBlock.Core
1501:  Namespace OSK.BuildingBlock.UIState
1502:      Public Delegate Sub WeightUpdateEventHandler()
1503:      Public Class BlockSuiteFormAction
1504:          Public Event WeightUpdateEvent As WeightUpdateEventHandler
1505:          Private blockFactoryObject As BlockFactory
1506:          Private blockListString() As String
1507:          Private suiteMemberListObject As ArrayList
1508:          Public Sub New()
1509:              blockFactoryObject = New BlockFactory()
1510:              blockListString = blockFactoryObject.GetBlockList()
1511:              suiteMemberListObject = New ArrayList()
1512:          End Sub
1513:          Public Sub Initialize()
1514:              RaiseEvent WeightUpdateEvent()
1515:          End Sub
1516:          Public Function GetBlockListCount() As Integer
1517:              Return blockListString.Length
1518:          End Function
1519:          Public Function GetBlockName(ByVal index As Integer) As String
1520:              Return blockListString(index)
1521:          End Function
1522:          Public Sub AddBlock(ByVal index As Integer)
1523:              Dim newBlock As IBlock
1524:              newBlock = blockFactoryObject.CreateBlock(index)
1525:              suiteMemberListObject.Add(newBlock)
1526:              RaiseEvent WeightUpdateEvent()
1527:          End Sub
1528:          Public Sub DeleteBlock(ByVal index As Integer)
1529:              suiteMemberListObject.RemoveAt(index)
1530:              RaiseEvent WeightUpdateEvent()
1531:          End Sub
1532:          Public Function GetSuiteMemberListCount() As Integer
1533:              Return suiteMemberListObject.Count
1534:          End Function
1535:          Public Function GetTotalWeight() As Double
1536:              Dim i As Integer
1537:              Dim totalWeight As Double
1538:              Dim blockObject As IBlock
1539:              For i = 0 To (GetSuiteMemberListCount() - 1)
1540:                  blockObject = suiteMemberListObject.Item(i)
1541:                  totalWeight += blockObject.CalculateWeight()
1542:              Next
1543:              Return totalWeight
1544:          End Function
1545:          Public Function GetTotalWeightString() As String
1546:              Dim totalWeightString As String
1547:              totalWeightString = String.Format("{0:#,##0.0}グラム", GetTotalWeight())
1548:              Return totalWeightString
1549:          End Function
1550:      End Class
1551:  End Namespace
```

Step 5　フォームウィンドウの処理を実装する

①　フォームウィンドウにテスト済みにクラスを実装する

BlockSuiteForm のコードをひらき、利用するクラスライブラリの宣言を行います。次に、BlockSuiteFormAction クラスのインスタンスをメンバ変数として宣言します。

C#の場合

初期化時に、イベントハンドラを追加します。

```
1552:  using OSK.BuildingBlock.Core;
1553:  using OSK.BuildingBlock.UIState;
1554:  namespace OSK.OSK.BuildingBlock.GUI
1555:  {
1556:      public class BlockSuiteForm : System.Windows.Forms.Form
1557:      {
1558:          internal System.Windows.Forms.MainMenu MainFormMenu;
1559:          internal System.Windows.Forms.MenuItem MenuItem1;
1560:          internal System.Windows.Forms.Label SuiteMemberListLabel;
1561:          internal System.Windows.Forms.Label BlockListLabel;
1562:          internal System.Windows.Forms.Button DeleteButton;
1563:          internal System.Windows.Forms.Button AddButton;
1564:          internal System.Windows.Forms.ListBox SuiteMemberListBox;
1565:          internal System.Windows.Forms.ListBox BlockListBox;
1566:          internal System.Windows.Forms.Label WeightValueLabel;
1567:          internal System.Windows.Forms.Label WeightLabel;
1568:          private System.ComponentModel.Container components = null;
1569:
1570:          private BlockSuiteFormAction actionObject;
1571:          public BlockSuiteForm()
1572:          {
1573:              // Windows フォーム デザイナ サポートに必要です。
1574:              InitializeComponent();
1575:              actionObject = new BlockSuiteFormAction();
1576:              actionObject.WeightUpdateEvent += new WeightUpdateEventHandler( WeightUpdateEventHandler );
1577:           }
```

ここがイベント追加

VB .NET の場合

このとき、WithEvents をつけてください。コンストラクタの中で、インスタンスの作成を行います。

```
1578:  Imports OSK.BuildingBlock.Core
1579:  Imports OSK.BuildingBlock.UIState
1580:
1581:  Namespace OSK.BuildingBlock.GUI
1582:      Public Class BlockSuiteForm
1583:          Inherits System.Windows.Forms.Form
1584:
1585:          Private WithEvents actionObject As BlockSuiteFormAction
1586:  #Region " Windows フォーム デザイナで生成されたコード "
1587:          Public Sub New()
1588:              MyBase.New()
1589:              ' この呼び出しは Windows フォーム デザイナで必要です。
1590:              InitializeComponent()
1591:              ' InitializeComponent() 呼び出しの後に初期化を追加します。
1592:              actionObject = New BlockSuiteFormAction()
1593:          End Sub
```

② フォームのロード時に初期化を行う

フォームのロード処理で、コントロールの初期化と BlockSuiteFormAction の初期化を呼び出します。
このときに、BlockSuiteFormAction から作成できるブロック名を読み出して、リストボックスに追加します。

C#の場合

```
1594:   private void BlockSuiteForm_Load(object sender, System.EventArgs e)
1595:   {
1596:       InitializeBlockListBox();
1597:       actionObject.Initialize();
1598:   }
1599:
1600:   private void InitializeBlockListBox()
1601:   {
1602:       int i;
1603:       for( i = 0; i < actionObject.GetBlockListCount(); ++i )
1604:       {
1605:           BlockListBox.Items.Add(actionObject.GetBlockName(i));
1606:       }
1607:   }
```

リストへ文字列を追加

VB .NET の場合

```
1608:   Private Sub BlockSuiteForm_Load _
1609:       (ByVal sender As System.Object, ByVal e As System.EventArgs) _
1610:       Handles MyBase.Load
1611:       InitializeBlockListBox()
1612:       actionObject.Initialize()
1613:   End Sub
1614:   Private Sub InitializeBlockListBox()
1615:       Dim i As Integer
1616:       For i = 0 To (actionObject.GetBlockListCount() - 1)
1617:           BlockListBox.Items.Add(actionObject.GetBlockName(i))
1618:       Next
1619:   End Sub
```

③ 追加ボタンの処理を実装する

追加ボタンの処理として、もしリストボックスが選択されていたら、リストと BlockSuiteFormAction に選択されているブロックを追加する。

C#の場合

```
1620:   private void AddButton_Click(object sender, System.EventArgs e)
1621:   {
1622:       if( IsBlockSelected() )
1623:       {
1624:           int index;
1625:           index = BlockListBox.SelectedIndex;
1626:           actionObject.AddBlock(index);
1627:           SuiteMemberListBox.Items.Add(actionObject.GetBlockName(index));
1628:       }
1629:   }
1630:   private bool IsBlockSelected()
1631:   {
1632:       if( BlockListBox.SelectedIndex >= 0 )
1633:       {
1634:           return true;
1635:       }
1636:       else
1637:       {
1638:           return false;
1639:       }
1640:   }
```

ブロックの選択状態を得る

VB .NET の場合

```
1641:       Private Sub AddButton_Click _
1642:           (ByVal sender As System.Object, ByVal e As System.EventArgs) _
1643:           Handles AddButton.Click
1644:           If IsBlockSelected() Then
1645:               Dim index As Integer
1646:               index = BlockListBox.SelectedIndex
1647:               actionObject.AddBlock(index)
1648:               SuiteMemberListBox.Items.Add(actionObject.GetBlockName(index))
1649:           End If
1650:       End Sub
1651:
1652:       Private Function IsBlockSelected() As Boolean
1653:           If BlockListBox.SelectedIndex >= 0 Then
1654:               Return True
1655:           Else
1656:               Return False
1657:           End If
1658:       End Function
```

④ 削除ボタンの処理を実装する

削除ボタンの処理として、もしリストボックスが選択されていたら、リストと BlockSuiteFormAction に選択されているブロックを削除する。

C#の場合

```
1659:       private void DeleteButton_Click(object sender, System.EventArgs e)
1660:       {
1661:           if( IsSuiteMemberSelected() )
1662:           {
1663:               int index;
1664:               index = SuiteMemberListBox.SelectedIndex;
1665:               actionObject.DeleteBlock(index);
1666:               SuiteMemberListBox.Items.RemoveAt(index);
1667:           }
1668:       }
1669:       private bool IsSuiteMemberSelected()
1670:       {
1671:           if( SuiteMemberListBox.SelectedIndex >= 0 )
1672:           {
1673:               return true;
1674:           }
1675:           else
1676:           {
1677:               return false;
1678:           }
1679:       }
```

VB .NET の場合

```
1680:       Private Sub DeleteButton_Click _
1681:           (ByVal sender As System.Object, ByVal e As System.EventArgs) _
1682:           Handles DeleteButton.Click
1683:           If IsSuiteMemberSelected() Then
1684:               Dim index As Integer
1685:               index = SuiteMemberListBox.SelectedIndex
1686:               actionObject.DeleteBlock(index)
1687:               SuiteMemberListBox.Items.RemoveAt(index)
1688:           End If
1689:       End Sub
```

```
1690:       Private Function IsSuiteMemberSelected() As Boolean
1691:           If SuiteMemberListBox.SelectedIndex >= 0 Then
1692:               Return True
1693:           Else
1694:               Return False
1695:           End If
1696:       End Function
```

⑤ 更新イベント処理を実装する

C#の場合

①で追加したイベントを処理するメソッドを実装します。

```
1697:       private void WeightUpdateEventHandler()
1698:       {
1699:           WeightValueLabel.Text = actionObject.GetTotalWeightString();
1700:       }
```

VB .NET の場合

エディタ上部の左のドロップダウン・リストボックスから actionObject を選択し、右のドロップダウン・リストボックスから、WeightUpdateEvent を選択します。イベントを処理するメソッドの雛型が作成されるので、処理を実装します。

```
1701:       Private Sub actionObject_WeightUpdateEvent() _
1702:           Handles actionObject.WeightUpdateEvent
1703:           WeightValueLabel.Text = actionObject.GetTotalWeightString()
1704:       End Sub
```

⑥ 実行する

プロジェクト「GUI」をスタートアッププロジェクトに設定し、動作させます。最終的な動作を確認して、処理の作成は終了です。

Session4　Web アプリケーションを作成する

Overview

.NET を使用して Web アプリケーションを作成します。VS.NET では、Windows アプリケーションと同様の手順で、Web アプリケーションを作成することができます。しかし、Web アプリケーションで問題となるのが状態の管理です。多くのクライアントの処理を行わなければならない Web サーバ上でクライアントの状態を管理すると、リソースなどに問題となります。そこで、VS.NET では、.NET が提供するコントロールの状態を不可視コントロールに保持する機構が組み込まれています。このことによって、フォームはそのつど作り直されますが、コントロールは状態を維持しているように扱うことができます。しかし、作り直されることを意識しなくて良いわけではありません。

この Session でも、引き続き、積み木システムを例題として、作業を進めます。

Goal

- ❒ Web アプリケーションを作成する
- ❒ 状態管理を行う

Step 1　仮想フォルダを作成する

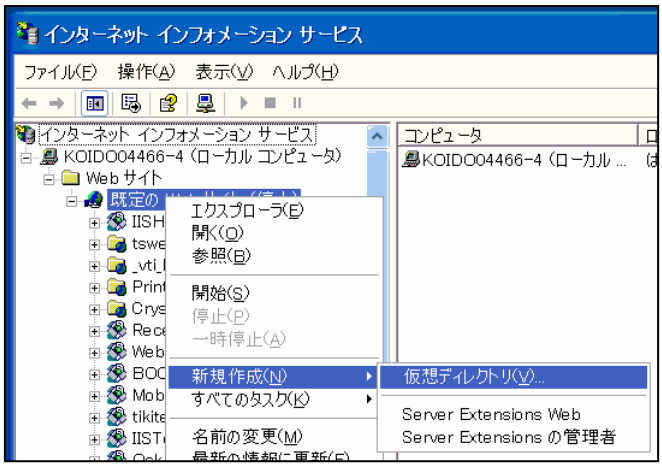
①　フォルダを作成する

エクスプローラなどを使用して、「BuildingBlock」のディレクトリ配下の「Source」配下に、「WebUI」というディレクトリを作成します。

②　IIS 上に、仮想フォルダを作成する

コントロールパネルの「管理ツール」内の「インターネット　インフォメーションサービス」を立ち上げます。その後、コンテキストメニューから「新規作成」の「仮想ディレクトリ」を選択して、仮想フォルダを作成します。

仮想フォルダ名（エイリアス名）は、「BuildingBlock」とします。また、ディレクトリは、①で作成したフォルダを指定します。その他の設定は、デフォルトのままです。

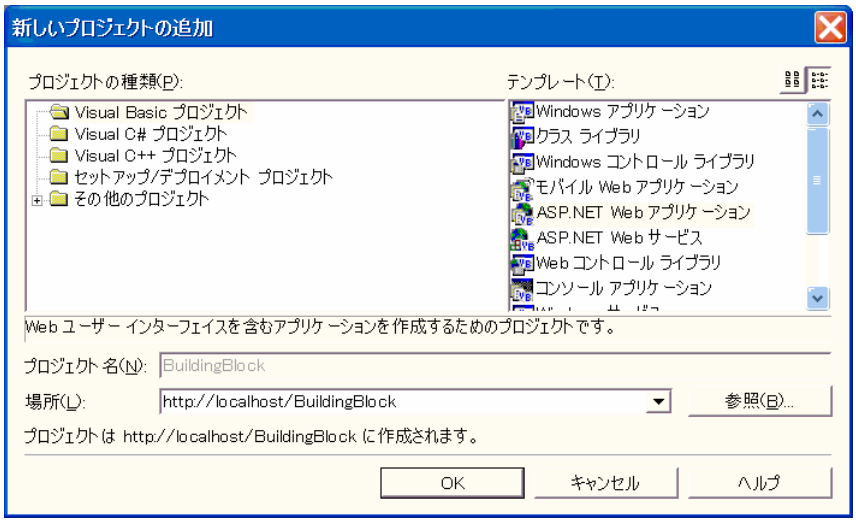


Step 2　プロジェクトを作成する

①　Web アプリケーション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。「各言語プロジェクト」の「Web アプリケーション」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。

設定項目	設定内容
プロジェクトの種類	各言語プロジェクト
テンプレート	ASP.NET Web アプリケーション
場所	http://localhost/BuildingBlock



②　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 BuildingBlock 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.BuildingBlock.WebUI
既定の名前空間	OSK.BuildingBlock.WebUI

VB .NET 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.BuildingBlock.WebUI
全般	ルート名前空間	[空白]

③

④ Global.asax の名前空間を設定する

Global.asax を選択して、コンテキストメニューから「コードの表示」を選択します。その後、以下のように名前空間を設定します。

C#の場合

```
1705:  namespace OSK.BuildingBlock.WebUI
1706:  {
1707:      public class Global : System.Web.HttpApplication
1708:      {
1709:          public Global()
1710:          {
1711:              InitializeComponent();
1712:          }
```

VB .NET の場合

```
1713:  Namespace OSK.BuildingBlock.WebUI
1714:      Public Class Global
1715:          Inherits System.Web.HttpApplication
1716:  '省略
1717:      End Class
1718:  End Namespace
```

Topic

「Global.asax」とは、以下の処理をするための特別なページおよびモジュールです。

① アプリケーション全体で使われるリソースの初期化および終了時のクリーンアップ処理

② ユーザセッションが開始されたときの処理とセッションがクローズされたときの処理

⑤ フォームのコードを変更する

「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 WebForm1 のコンテキストメニューを表示し、「コード表示」を選択します。以下のようにコードを変更します。

VB .NET の場合

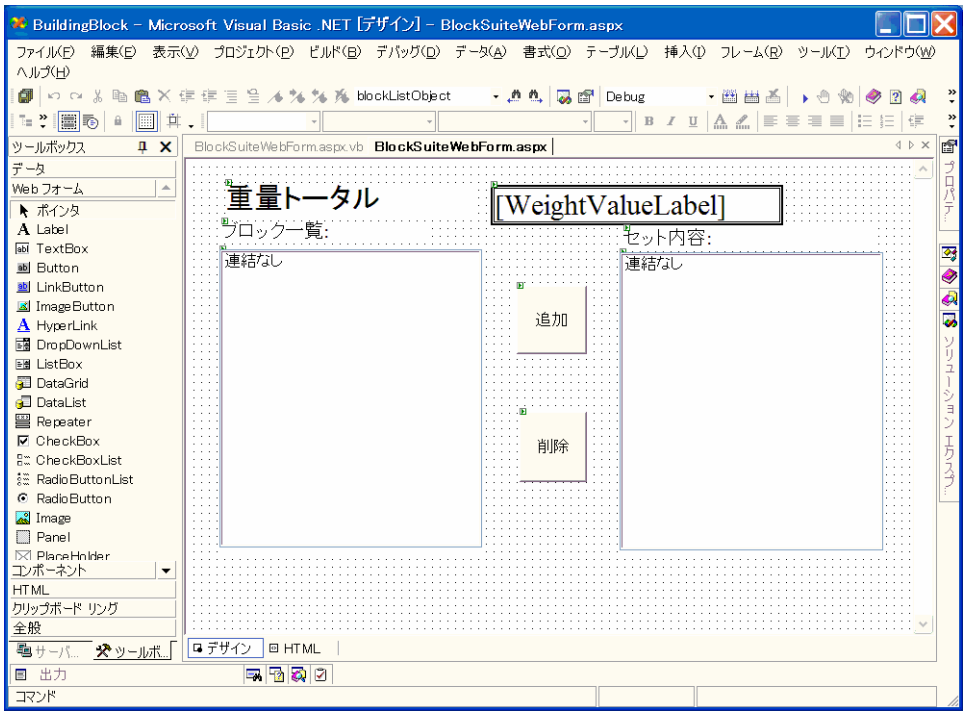
```
1719:  Namespace OSK.BuildingBlock.WebUI
1720:
1721:      Public Class BlockSuiteWebForm
1722:          Inherits System.Web.UI.Page
1723:  '省略
1724:      End Class
1725:  End Namespace
```

⑥ ファイル名を変更する

「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 WebForm1 の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。ファイル名を BlockSuiteWebForm に変更します。

⑦ フォームにコントロールを配置する

ツールボックスからコントロールをドラッグ・アンド・ドロップして、以下のようなフォームを作成します。



⑧ コントロール名を変更する

プログラムの作成をスムーズに進めるために、コントロール名を Windows アプリケーションのフォームと同じものに変更します。また、各コントロールのプロパティを以下のように変更します。

DOCUMENT(BlockSuiteWebForm)について、以下の設定を行う

種別	項目	設定内容
表示	titile	ブロックセット重量計算

WeightLabel について、以下の設定を行う

種別	項目	設定内容
表示	Text	重量トータル
表示	Font	Large

WeightValueLabel について、以下の設定を行う

種別	項目	設定内容
表示	Text	[空白]
表示	Font	Large
表示	BorderStyle	Double

AddButton について、以下の設定を行う

種別	項目	設定内容
表示	Text	追加

DeleteButton について、以下の設定を行う

種別	項目	設定内容
表示	Text	削除

BlockListBox と SuiteMemberListBox は、名前以外は特に変更しない。

Step 3　Web フォームの処理を実装する

①　フォームウィンドウに入力補助クラスを実装する

BlockSuiteWebForm のコードをひらき、利用するクラスライブラリの宣言を行います。次に、BlockSuiteFormAction クラスのインスタンスをメンバ変数として宣言します。このとき、WithEvents をつけてください。コンストラクタの中で、インスタンスの作成を行います。

C#の場合

```
1726:  using OSK.BuildingBlock.Core;
1727:  using OSK.BuildingBlock.UIState;
1728:
1729:  namespace OSK.BuildingBlock.WebUI
1730:  {
1731:          public class BlockSuiteWebForm : System.Web.UI.Page
1732:          {
1733:            protected System.Web.UI.WebControls.Label SuiteMemberListBoxLabel;
1734:            protected System.Web.UI.WebControls.Label BlockListBoxLabel;
1735:            protected System.Web.UI.WebControls.ListBox SuiteMemberListBox;
1736:            protected System.Web.UI.WebControls.Button DeleteButton;
1737:            protected System.Web.UI.WebControls.Button AddButton;
1738:            protected System.Web.UI.WebControls.ListBox BlockListBox;
1739:            protected System.Web.UI.WebControls.Label WeightValueLabel;
1740:            protected System.Web.UI.HtmlControls.HtmlForm Form1;
1741:            protected System.Web.UI.WebControls.Label WeightLabel;
1742:
1743:            private BlockSuiteFormAction actionObject;
```

VB .NET の場合

このとき、WithEvents をつけてください。コンストラクタの中で、インスタンスの作成を行います。

```
1744:  Imports OSK.BuildingBlock.Core
1745:  Imports OSK.BuildingBlock.UIState
1746:
1747:  Namespace OSK.BuildingBlock.WebUI
1748:
1749:      Public Class BlockSuiteWebForm
1750:          Inherits System.Web.UI.Page
1751:          Protected WithEvents WeightLabel As System.Web.UI.WebControls.Label
1752:          Protected WithEvents WeightValueLabel As System.Web.UI.WebControls.Label
1753:          Protected WithEvents BlockListBox As System.Web.UI.WebControls.ListBox
1754:          Protected WithEvents AddButton As System.Web.UI.WebControls.Button
1755:          Protected WithEvents DeleteButton As System.Web.UI.WebControls.Button
1756:          Protected WithEvents SuiteMemberListBox As System.Web.UI.WebControls.ListBox
1757:          Protected WithEvents BlockListBoxLabel As System.Web.UI.WebControls.Label
1758:          Protected WithEvents SuiteMemberListBoxLabel As System.Web.UI.WebControls.Label
1759:
1760:          Private WithEvents actionObject As BlockSuiteFormAction
```

②　ページの初期化時に初期化を行う

ページの初期化時に、コントロールの初期化と BlockSuiteFormAction の初期化を呼び出します。
このときに、BlockSuiteFormAction から作成できるブロック名を読み出して、リストボックスに追加します。

C#の場合


```
1761:          private void WeightUpdateEventHandler()
1762:          {
1763:          }
```

```
#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
    //
    // CODEGEN: この呼び出しは、ASP.NET Web フォーム デザイナで必要です。
    //
    InitializeComponent();
    base.OnInit(e);

    actionObject = new BlockSuiteFormAction();
    actionObject.WeightUpdateEvent += new WeightUpdateEventHandler(WeightUpdateEventHandler);
    InitializeBlockListBox();
    actionObject.Initialize();
}
```

初期化処理

VB .NET の場合

```
Private Sub Page_Init _
    (ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles MyBase.Init
    ' CODEGEN: このメソッド呼び出しは Web フォーム デザイナで必要です。
    ' コード エディタを使って変更しないでください。
    InitializeComponent()
    actionObject = New BlockSuiteFormAction()
    InitializeBlockListBox()
    actionObject.Initialize()
End Sub
Private Sub InitializeBlockListBox()
    Dim i As Integer
    For i = 0 To (actionObject.GetBlockListCount() - 1)
        BlockListBox.Items.Add(actionObject.GetBlockName(i))
    Next
End Sub
```

③ 追加ボタンの処理を実装する

追加ボタンの処理として、もしリストボックスが選択されていたら、リストと BlockSuiteFormAction に選択されているブロックを追加する。

C#の場合

```
private void AddButton_Click(object sender, System.EventArgs e)
{
    if( IsBlockSelected() )
    {
        int index;
        index = BlockListBox.SelectedIndex;
        actionObject.AddBlock(index);
        SuiteMemberListBox.Items.Add(actionObject.GetBlockName(index));
    }
}
private bool IsBlockSelected()
{
    if( BlockListBox.SelectedIndex >= 0 )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

VB .NET の場合

```
1815:       Private Sub AddButton_Click _
1816:           (ByVal sender As System.Object, ByVal e As System.EventArgs) _
1817:           Handles AddButton.Click
1818:           If IsBlockSelected() Then
1819:               Dim index As Integer
1820:               index = BlockListBox.SelectedIndex
1821:               actionObject.AddBlock(index)
1822:               SuiteMemberListBox.Items.Add(actionObject.GetBlockName(index))
1823:           End If
1824:       End Sub
1825:
1826:       Private Function IsBlockSelected() As Boolean
1827:           If BlockListBox.SelectedIndex >= 0 Then
1828:               Return True
1829:           Else
1830:               Return False
1831:           End If
1832:       End Function
```

④ 削除ボタンの処理を実装する

削除ボタンの処理として、もしリストボックスが選択されていたら、リストと BlockSuiteFormAction に選択されているブロックを削除する。

C#の場合

```
1833:       private void DeleteButton_Click(object sender, System.EventArgs e)
1834:       {
1835:           if( IsSuiteMemberSelected() )
1836:           {
1837:               int index;
1838:               index = SuiteMemberListBox.SelectedIndex;
1839:               actionObject.DeleteBlock(index);
1840:               SuiteMemberListBox.Items.RemoveAt(index);
1841:           }
1842:       }
1843:       private bool IsSuiteMemberSelected()
1844:       {
1845:           if( SuiteMemberListBox.SelectedIndex >= 0 )
1846:           {
1847:               return true;
1848:           }
1849:           else
1850:           {
1851:               return false;
1852:           }
1853:       }
```

VB .NET の場合

```
1854:       Private Sub DeleteButton_Click _
1855:           (ByVal sender As System.Object, ByVal e As System.EventArgs) _
1856:           Handles DeleteButton.Click
1857:           If IsSuiteMemberSelected() Then
1858:               Dim index As Integer
1859:               index = SuiteMemberListBox.SelectedIndex
1860:               actionObject.DeleteBlock(index)
1861:               SuiteMemberListBox.Items.RemoveAt(index)
1862:           End If
1863:       End Sub
```

```
1864:       Private Function IsSuiteMemberSelected() As Boolean
1865:           If SuiteMemberListBox.SelectedIndex >= 0 Then
1866:               Return True
1867:           Else
1868:               Return False
1869:           End If
1870:       End Function
```

⑤ 更新イベント処理を実装する

エディタ上部の左のドロップダウン・リストボックスから actionObject を選択し、右のドロップダウン・リストボックスから、WeightUpdateEvent を選択します。イベントを処理するメソッドの雛型が作成されるので、処理を実装します。

C#の場合

```
1871:       private void WeightUpdateEventHandler()
1872:       {
1873:           WeightValueLabel.Text = actionObject.GetTotalWeightString();
1874:       }
```

VB .NET の場合

```
1875:       Private Sub actionObject_WeightUpdateEvent() _
1876:           Handles actionObject.WeightUpdateEvent
1877:           WeightValueLabel.Text = actionObject.GetTotalWeightString()
1878:       End Sub
```

⑥ 実行する

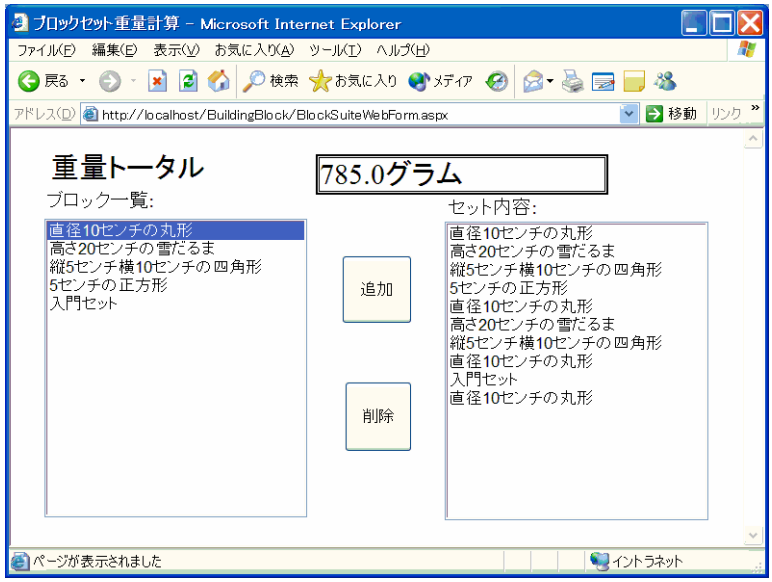
プロジェクト「BuildingBlock」をスタートアッププロジェクトに設定します。また、「BlockSuiteWebForm.aspx」をスタートページに設定します。実行を行います。

一見問題なく動作しているようですが、2 つの問題があります。

一つは、重量トータルの値が正しくないことです。もう一つ、削除ボタンを押すとエラーが発生することです。

なぜ、このようなエラーが発生するのでしょうか？これは、BlockSuiteFormAction のインスタンスが処理ごとに再作成されて、しかも状態が保持されないからです。

そこで、今回は、状態の保持されているセット内容のリストボックスから、BlockSuiteFormAction を再構築することにします。



Step 4　既存クラスに機能拡張する

①　必要とするメソッドを決定する

リストボックスから BlockSuiteFormAction を再構築するためには、文字列によってセットリストにブロックを追加する必要があります。メソッドは、以下のようになります。

C#の場合

```
1879:    public void AddBlock( string blockName)
1880:    {
1881:    }
```

VB .NET の場合

```
1882:    Public Sub AddBlock(ByVal blockName As String)
1883:    End Sub
```

Topic

.NET では、引数が異なれば、同じ名前のメソッドを作成することができます。このような概念をオーバーロードと呼びます。

例えば、VB6 では、追加を行うメソッドを用意するときに、引数の型別に、AddInteger や AddLong、更には AddString などを用意する必要がありました。しかし、これでは、メソッド呼び出す側に負担をかけることになります。VB.NET では、Add という名前メソッドで、Overloads キーワード（省略可能です）を指定することで引数が異なったものを複数用意することができます。また同様に VB6 では、メソッドに引数を追加して機能拡張する場合に、メソッド名に Ex を付加するように場合がありました。.NET では、引数を追加して、同じ名前で作成することができます。ここでは、AddBlock というメソッドを 2 つ作成しています。

②　テストメソッドを決定する

新規に追加する AddBlock をテストするためのメソッドを BlockSuiteFormActionTestCase クラスに追加します。テストでは、ブロックの名前が必要になるため、定数として定義します。また、テストは、追加を行い、トータルの重さによって、結果の判別を行っています。

C#の場合

```
1884:    [Test]
1885:    public void AddNameTest()
1886:    {
1887:        double totalWeight = 0.0;
1888:        target.AddBlock(BlockName4);
1889:        totalWeight += Weight4;
1890:        target.AddBlock(BlockName3);
1891:        totalWeight += Weight3;
1892:        target.AddBlock(BlockName0);
1893:        totalWeight += Weight0;
1894:        Assert.AreEqual(3, target.GetSuiteMemberListCount(),"追加エラー");
1895:        Assert.AreEqual(totalWeight, target.GetTotalWeight(),"追加エラー");
1896:        Assert.AreEqual("4,202.8 グラム", target.GetTotalWeightString(),"追加エラー");
1897:    }
```

VB .NET の場合

```
1898:    Private Const BlockName0 As String = "直径 10 センチの丸形"
1899:    Private Const BlockName1 As String = "高さ 20 センチの雪だるま"
1900:    Private Const BlockName2 As String = "縦 5 センチ横 10 センチの四角形"
1901:    Private Const BlockName3 As String = "5 センチの正方形"
1902:    Private Const BlockName4 As String = "入門セット"
```

```
1903:       <Test()> _
1904:       Public Sub AddNameTest()
1905:           Dim totalWeight As Double
1906:           targetObj.AddBlock(BlockName4)
1907:           totalWeight += Weight4
1908:           targetObj.AddBlock(BlockName3)
1909:           totalWeight += Weight3
1910:           targetObj.AddBlock(BlockName0)
1911:           totalWeight += Weight0
1912:           Assert.AreEqual(3, targetObj.GetSuiteMemberListCount(),"追加エラー")
1913:           Assert.AreEqual(totalWeight, targetObj.GetTotalWeight(),"追加エラー")
1914:           Assert.AreEqual("4,202.8 グラム", targetObj.GetTotalWeightString(), _
1915:                                           "追加エラー")
1916:       End Sub
```

③ メソッドを実装する

メソッド AddBlock は以下のようになります。

C#の場合

```
1917:       public void AddBlock( string blockName)
1918:       {
1919:           int index;
1920:           IBlock newBlock;
1921:           index = GetIndex(blockName);
1922:           newBlock = blockFactoryObject.CreateBlock(index);
1923:           suiteMemberListObject.Add(newBlock);
1924:           WeightUpdateEvent();
1925:       }
1926:       private int GetIndex( string blockName )
1927:       {
1928:           int i;
1929:           for( i = 0; i < GetBlockListCount(); ++i )
1930:           {
1931:               if( GetBlockName(i) == blockName )
1932:               {
1933:                   break;
1934:               }
1935:           }
1936:           return i;
1937:       }
```

VB .NET の場合

```
1938:       Public Sub AddBlock(ByVal blockName As String)
1939:           Dim index As Integer
1940:           Dim newBlock As IBlock
1941:           index = GetIndex(blockName)
1942:           newBlock = blockFactoryObject.CreateBlock(index)
1943:           suiteMemberListObject.Add(newBlock)
1944:           RaiseEvent WeightUpdateEvent()
1945:       End Sub
1946:
1947:       Private Function GetIndex(ByVal blockName As String) As Integer
1948:           Dim i As Integer
1949:           For i = 0 To (GetBlockListCount() - 1)
1950:               If GetBlockName(i) = blockName Then
1951:                   Exit For
1952:               End If
1953:           Next
1954:           Return i
1955:       End Function
```

Step 5　オブジェクトを再構築する

①　ページのロード時に、オブジェクトを再構築する

セット内容のリストボックスからブロック名を取得して、actionObject のブロックを追加するメソッドを呼び出します。これで、状態の再構築は終了です。

C#の場合

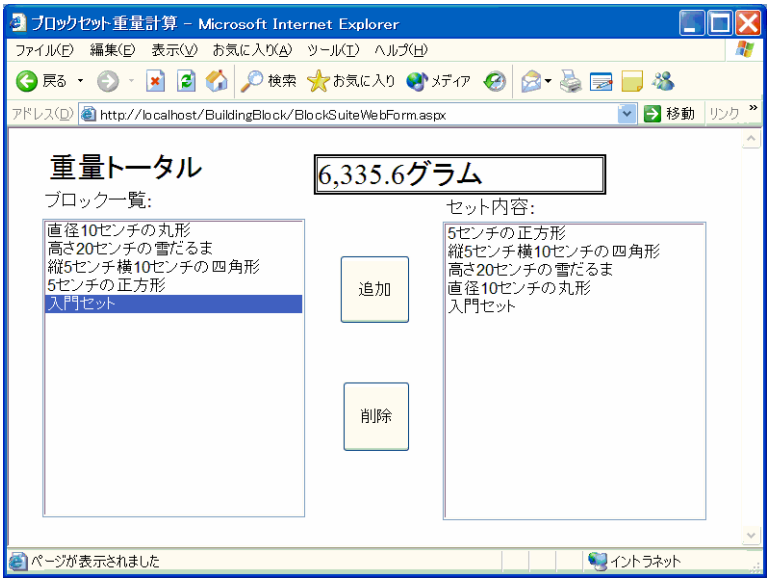
```
1956:        private void Page_Load(object sender, System.EventArgs e)
1957:                {
1958:            InitializeActionObject();
1959:                }
1960:        private void InitializeActionObject()
1961:        {
1962:            int i;
1963:
1964:            for( i = 0 ; i < SuiteMemberListBox.Items.Count; ++i )
1965:            {
1966:                actionObject.AddBlock(SuiteMemberListBox.Items[i].ToString());
1967:            }
1968:        }
```

VB .NET の場合

```
1969:    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) _
1970:            Handles MyBase.Load
1971:            ' ページを初期化する ユーザー コードをここに挿入します。
1972:            InitializeActionObject()
1973:
1974:        End Sub
1975:
1976:        Private Sub InitializeActionObject()
1977:            Dim i As Integer
1978:            For i = 0 To (SuiteMemberListBox.Items.Count - 1)
1979:                actionObject.AddBlock(SuiteMemberListBox.Items(i).ToString())
1980:            Next
1981:        End Sub
```

②　動作を確認する

実行を行って動作を確認します。



付録 1　NUnit リファレンス

A) 属性

③ TestFixture
テスト用のクラスであることを宣言する。

④ Test
テストメソッドであることを宣言する。

⑤ 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。

⑥ 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。

⑦ 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

⑧ 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。

⑨ 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。

⑩ 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

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。