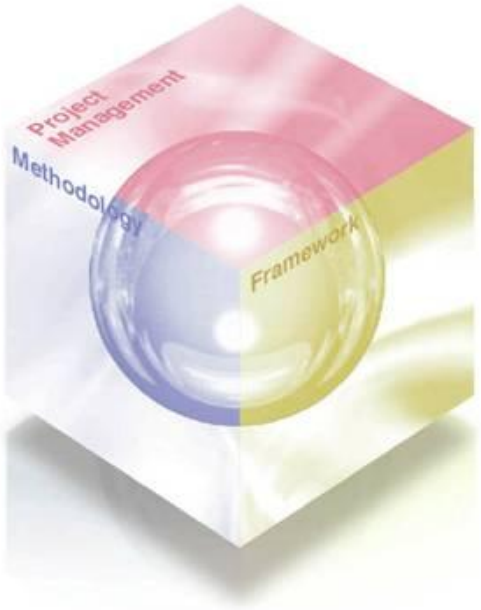
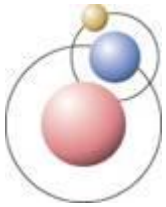
TERASOLUNA® Framework for .NET (Rich版)
第3.0.0.0版
チュートリアル



株式会社NTTデータ

TERASOLUNA®
BEST SOLUTION FOR SYSTEM DEVELOPMENT

NTT DaTa



はじめに

- 本ドキュメントは、簡単な足し算アプリケーションを題材に、TERASOLUNA Server/Client Framework for .NETの機能概要およびフレームワークを使用したアプリケーションの開発手順を大まかに理解することを目的としている

サンプルアプリケーション概要

画面一覧

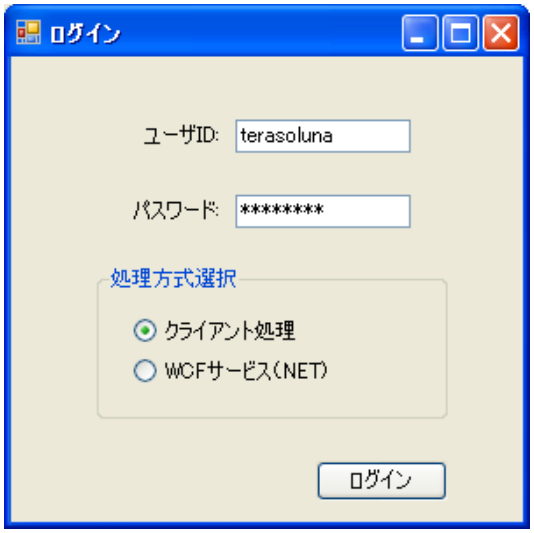
画面名	説明
ログイン画面	ユーザ認証を行う画面
メニュー画面	メニュー選択する画面
計算画面	サーバAPで足し算を実施し結果を出力する画面

処理一覧

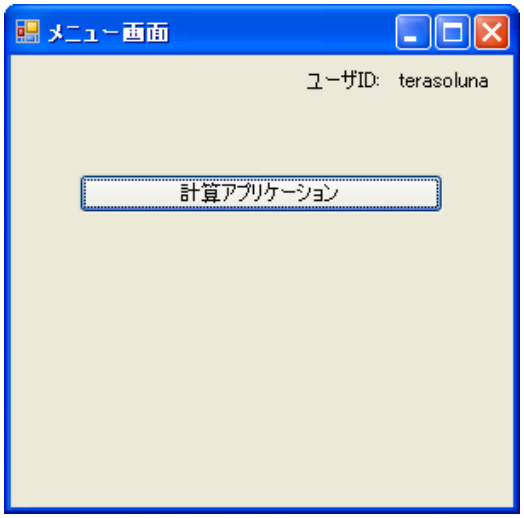
画面名	ボタン	説明
ログイン画面	ログイン	「処理方式選択」のラジオボタンで選択した方式に従い、ユーザIDとパスワードが正しいかを確認する
計算画面	計算	「処理方式選択」のラジオボタンで選択した方式に従い、数値1と数値2の値を足し算し計算結果を表示する

サンプルアプリケーション概要

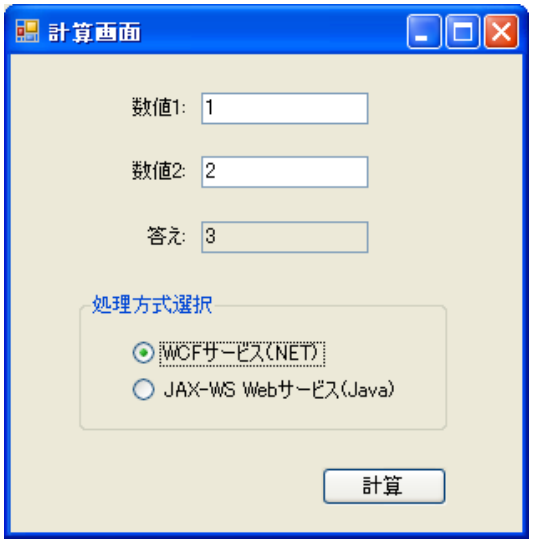
画面遷移図



ログイン画面
(LoginView.cs)



メニュー画面
(MenuView.cs)



計算画面
(CalcView.cs)

（参考）チュートリアル対象外業務

- 本チュートリアルの手順としては対象外ではあるが、完成版のチュートリアルAPIには以下の業務も実装しているので、参考するとよい

画面一覧

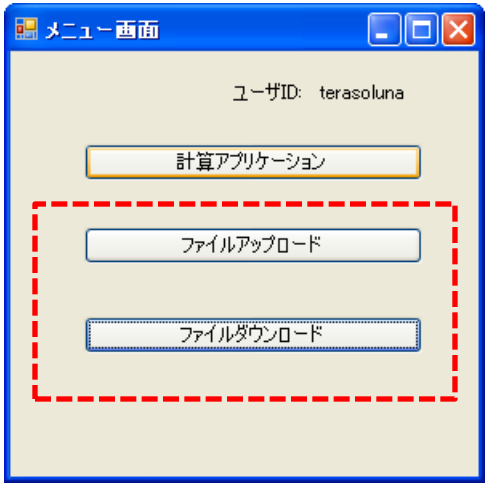
画面名	説明
ファイルアップロード画面	ユーザ認証を行う画面
ファイルダウンロード画面	メニュー選択する画面

処理一覧

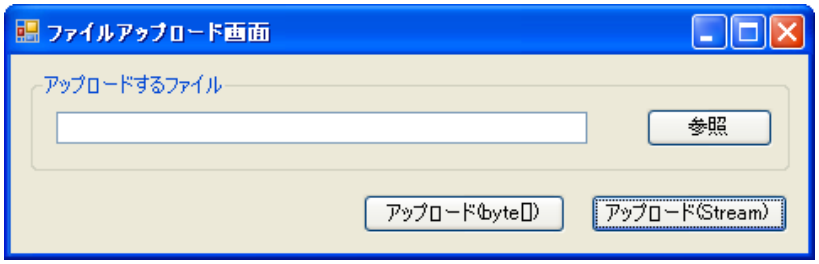
画面名	ボタン	説明
ファイル アップロード画面	アップロード(byte[]) アップロード(Stream)	指定したファイルをWCFサービスへアップロードする
ファイル ダウンロード画面	ダウンロード(byte[]) ダウンロード(Stream)	指定した形式のファイルをWCFサービスからダウンロードする

（参考）チュートリアル対象外業務

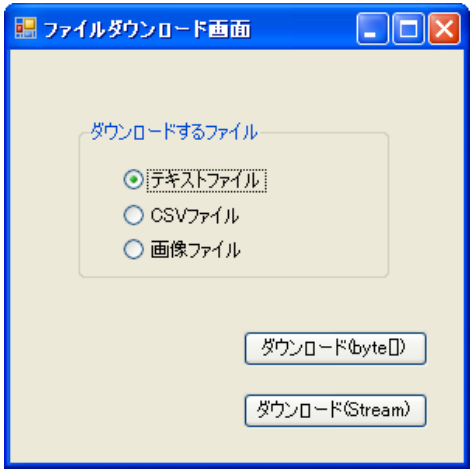
- 画面遷移図



メニュー画面
(MenuView.cs)



ファイルアップロード画面
(FileUploadView.cs)



ファイルダウンロード画面
(FileDownloadView.cs)

（参考）単体テストプロジェクトについて

- チュートリアルAPの完成版には、単体テストプロジェクトつきで用意しているので、単体テストの実施方法理解のための参考にするとよい
  - CalcSampleApp.sln（またはCalcSampleAppForTeraDebug.sln ）
    - クライアント/サーバAPおよび単体テストのプロジェクトを含む
  - CalcSampleApp.Client.sln
    - クライアントAPおよび単体テストプロジェクトのみ含む
  - CalcSampleApp.Server.sln
    - サーバAPおよび単体テストプロジェクトのみ含む

※各ソリューションの説明については、完成版チュートリアルAPの「README.txt」を参照ください

- Express Edition　Standard Editionの場合
  - ソリューションを開く際、単体テストプロジェクトの読み込みに失敗しますが、アプリケーションを起動することは可能です

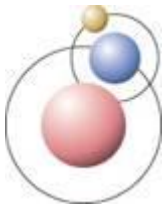
チュートリアルの学習方法

- .NETクライアントに閉じた処理を作成したい場合
  - 「ソリューションの作成」、「.NETクライアントAPプロジェクトの作成」、「ログイン処理の作成①」、「メニュー業務作成」の順に実施してください
- .NETクライアント-.NETサーバ接続を作成したい場合
  - 「.NETクライアントAPプロジェクトの作成」、「ログイン処理の作成①」、「メニュー業務作成」、「計算処理の作成①」、「ログイン処理の作成②」の順に実施してください
- .NETクライアント-JAVAサーバ接続を作成したい場合
  - .NETの開発環境のほかに、Java側の開発環境の準備が必要です
  - JavaのサーバAPについては作成済みのJavaプロジェクト(calcsample)を使用します
  - 「.NETクライアントAPプロジェクトの作成」、「ログイン処理の作成①」、「メニュー業務作成」、「計算処理の作成②」の順に実施してください
    - .NETサーバ接続の実施が一部混じっていますが、不要ならその部分の実装手順は無視して進めて下さい
- Click Onceのデモを試したい場合
  - IISの環境設定が必要です
  - 上記の手順でアプリケーションを作成後、「Click Onceの実行」を実施してください

開発環境整備

チュートリアルの実行には、以下の開発環境が必要です

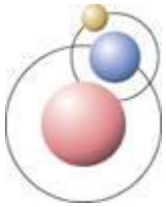
- .NET環境
  - .NET Framework 3.5SP1およびVisual Studio 2008
  - SQLServer 2005または2008 Express
  - TERASOLUNA Framework for .NET ver3.0.0.0
    - 同梱されたインストーラ「TERASOLUNA-3.0.0.0.msi」を実行してください
  - カスタムテンプレート・スニペット
    - 同梱されたインストーラ「TemplatesInstaller.msi」を実行してください
  - IIS(Internet Information Service)
    - Click Onceのデモを実施する場合のみ必要です
- Java環境(Javaサーバ APを動作させる場合に必要です)
  - JDK6
  - EclipseまたはAnt
  - Tomcat 5.5または6.0
  - Metro 1.5また2.0



（参考）Metroのインストール

- Metroのダウンロード
  - https://metro.dev.java.net/
- 圧縮ファイルを解凍、metroフォルダが展開される
  - Metro1.5の場合、「java -jar metro-1_5.jar」を実行
  - Metro2.0の場合、metro-2_0.zipを解凍
- TomcatにMetroのjarをインストール
  - 「ant -Dtomcat.home=<TOMCAT_INSTALL_DIR> -f <METRO_INSTALL_DIR>/metro-on-tomcat.xml install 」を実行
    Tomcatの shared/lib ディレクトリ等にMetroのjarがコピーされる
    - webservices-XXX.jar
  - EclipseのWTPの設定で、Metroのインストールされたtomcatを追加し利用する
  - https://metro.dev.java.net/1.5/docs/install.html
  - https://metro.dev.java.net/2.0/

TERASOLUNA



（参考）Metro1.5の場合に必要な設定

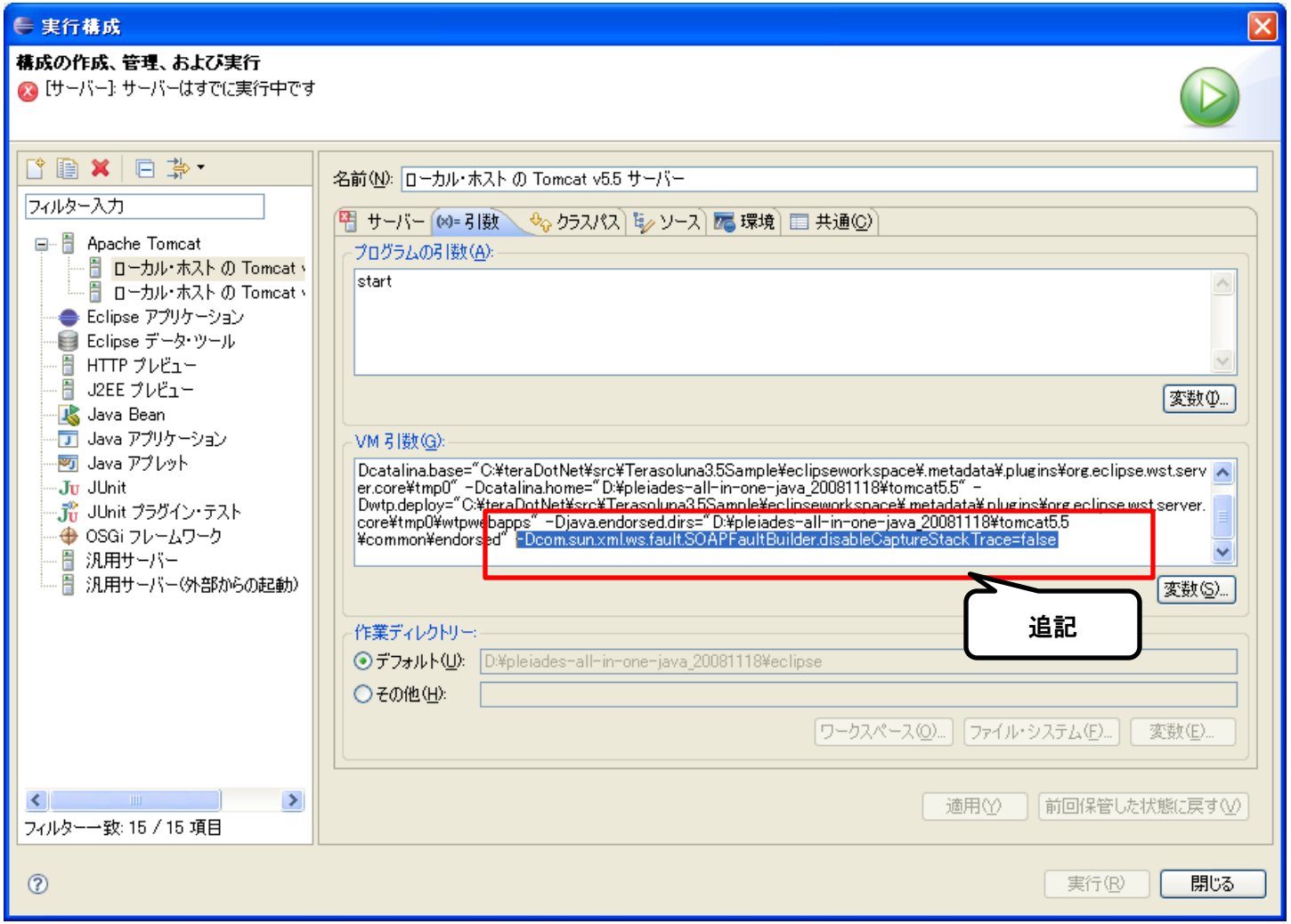
- Metro1.5の場合は、Tomcatの起動オプションに以下を追加し、クライアントAPへスタックトレースを送信しないようにする

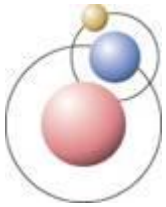
```
-Dcom.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace=false
```

- Metro2.0の場合は、上記の設定不要である。

TERASOLUNA

（参考）Metro1.5の場合に必要な設定



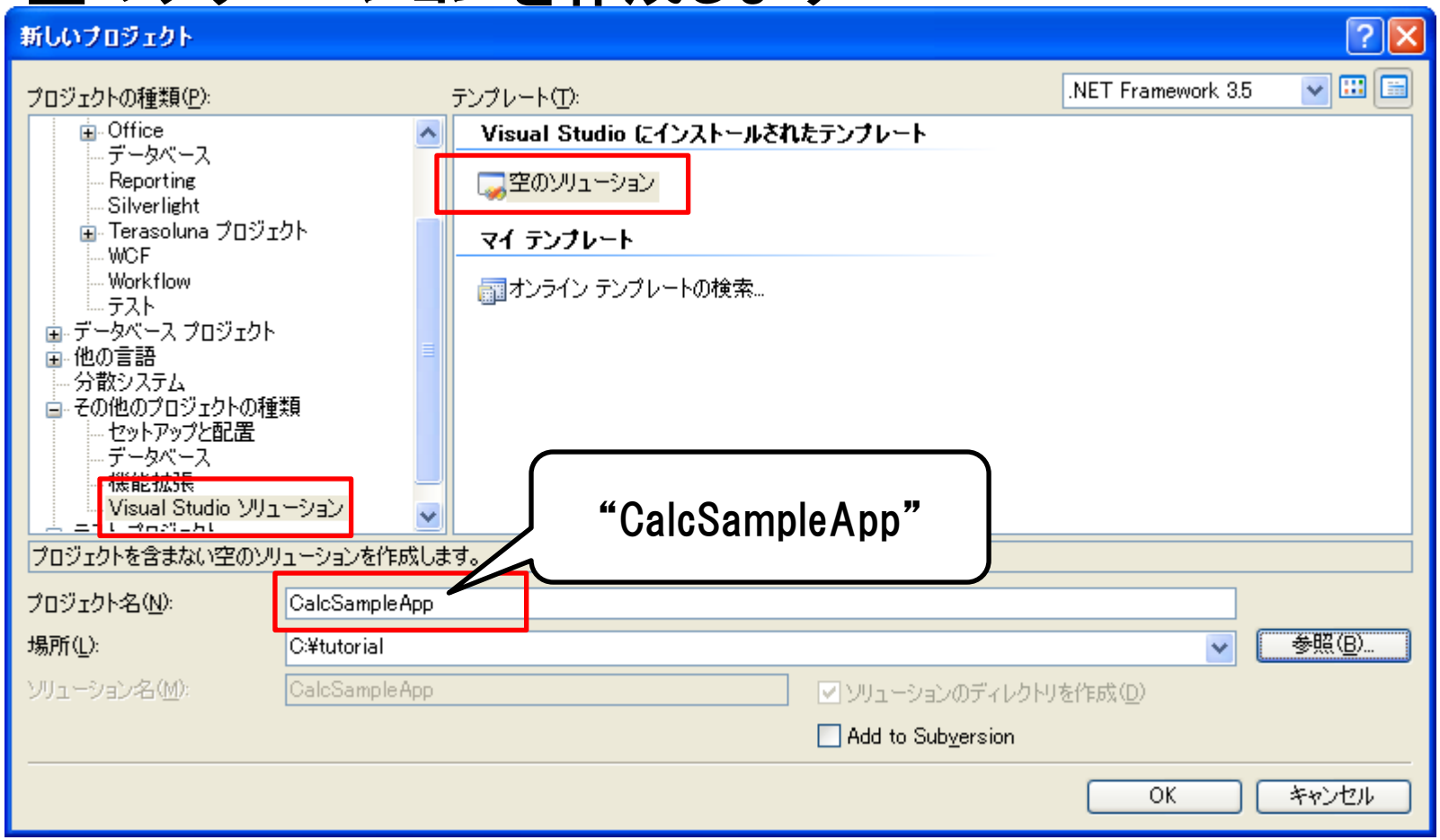


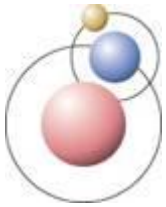
ソリューションの作成

TERASOLUNA

ソリューションの作成

- 空のソリューションを作成します



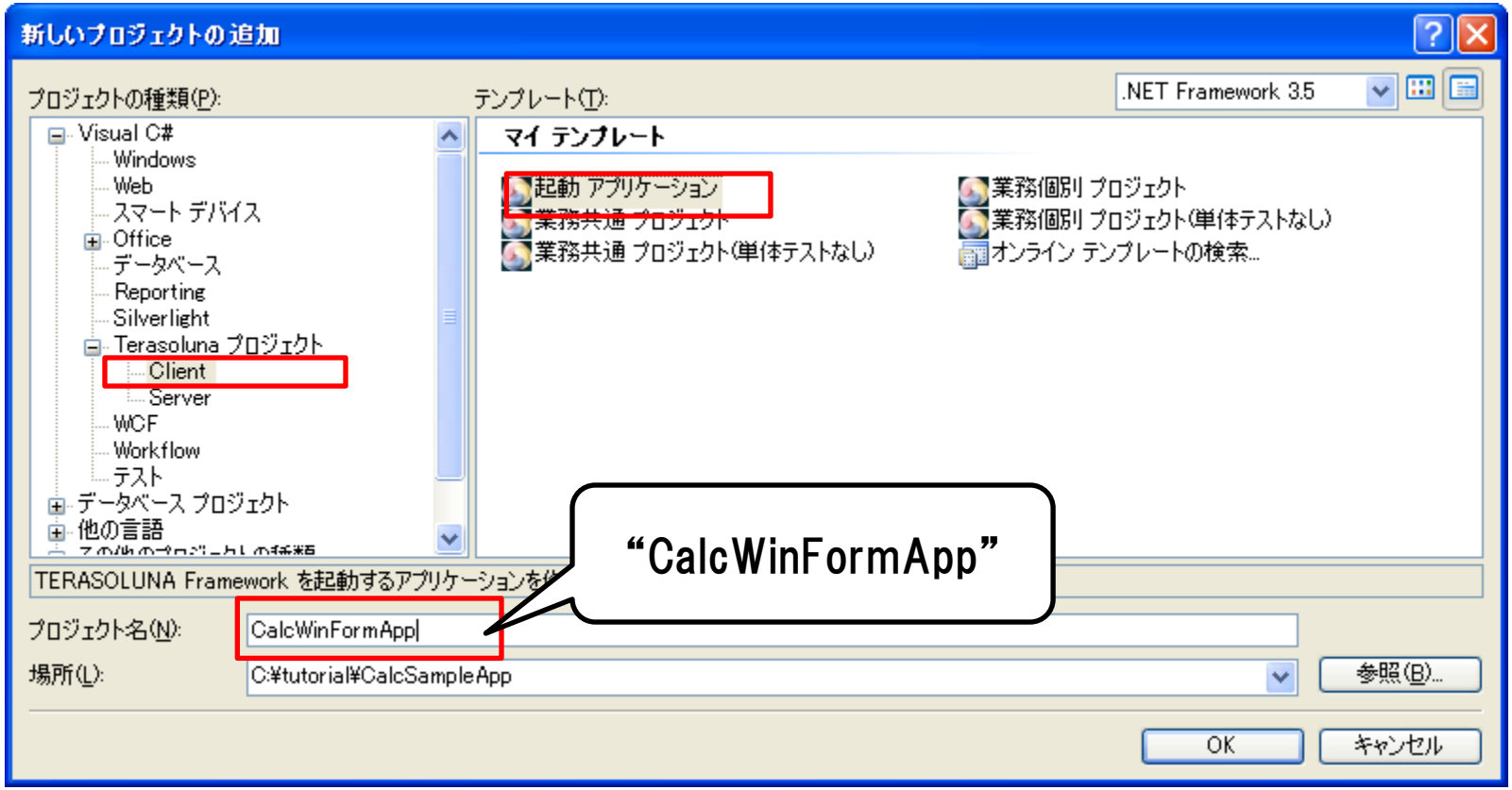


.NETクライアントプロジェクトの作成

TERASOLUNA

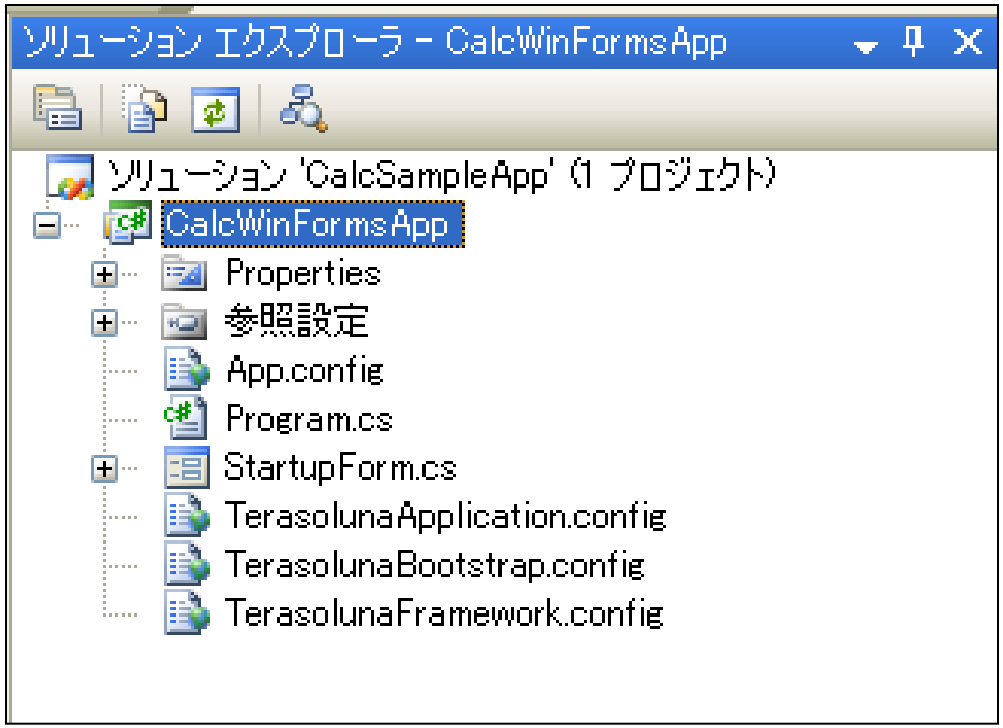
起動アプリケーションプロジェクトの作成

- 「TERASOLUNAプロジェクト」-「Client」-「起動アプリケーション」テンプレートで作成します



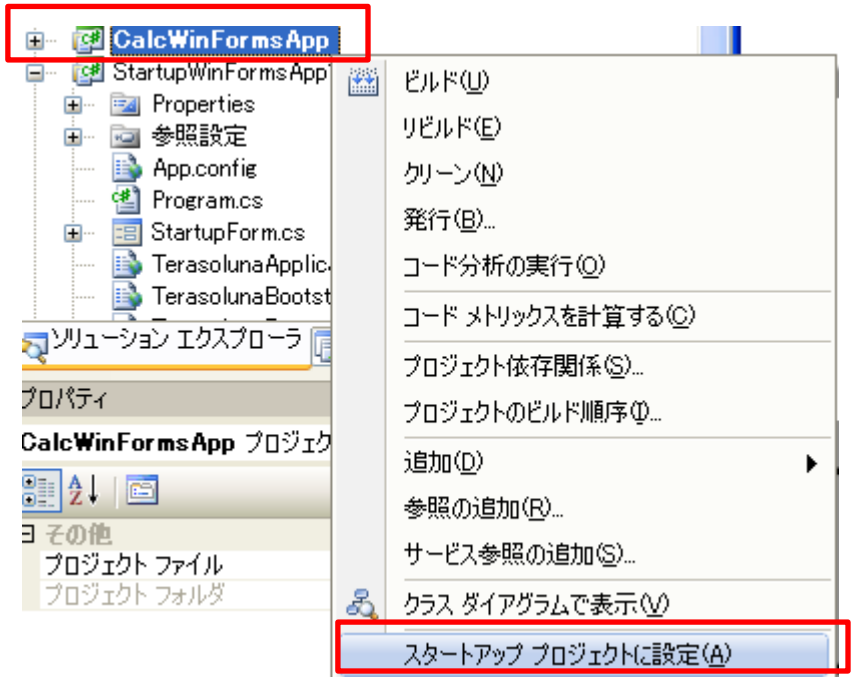
起動アプリケーションプロジェクトの作成

- 起動アプリケーションプロジェクトのひな型が作成されます



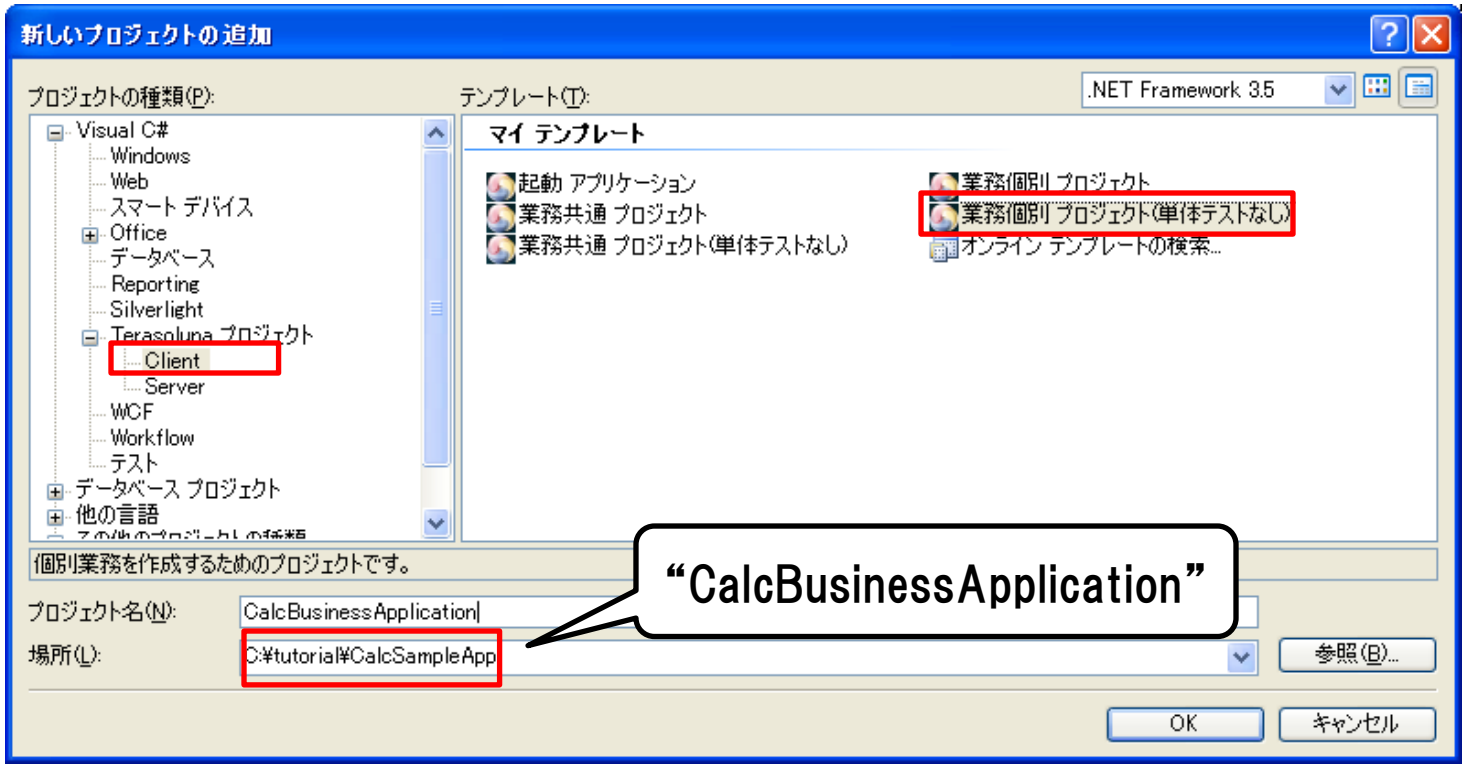
スタートアッププロジェクトの設定

- 起動アプリケーションプロジェクト(CalcWinFormsApp)をスタートアッププロジェクトに設定します
  - 「CalcWinFormsApp」プロジェクトを右クリックし、「スタートアッププロジェクトに設定」



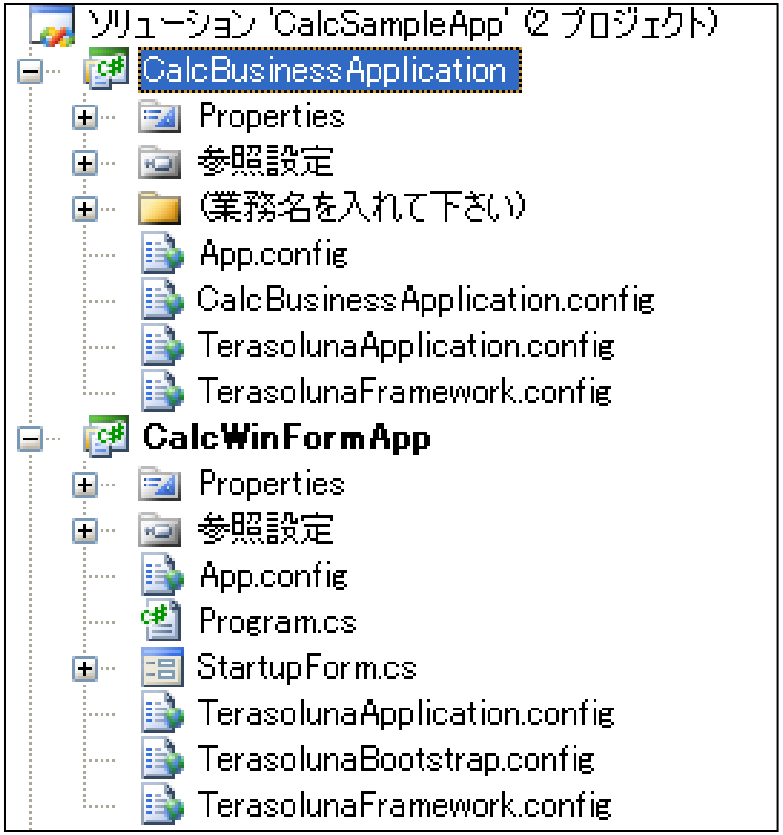
業務個別プロジェクトの作成

- 「TERASOLUNAプロジェクト」-「Client」-「業務個別プロジェクト(単体テストなし)」テンプレートで作成します
  - 通常の業務開発では、UTプロジェクトを同時に生成 する「業務個別プロジェクト」テンプレートを利用するのが推奨パターン



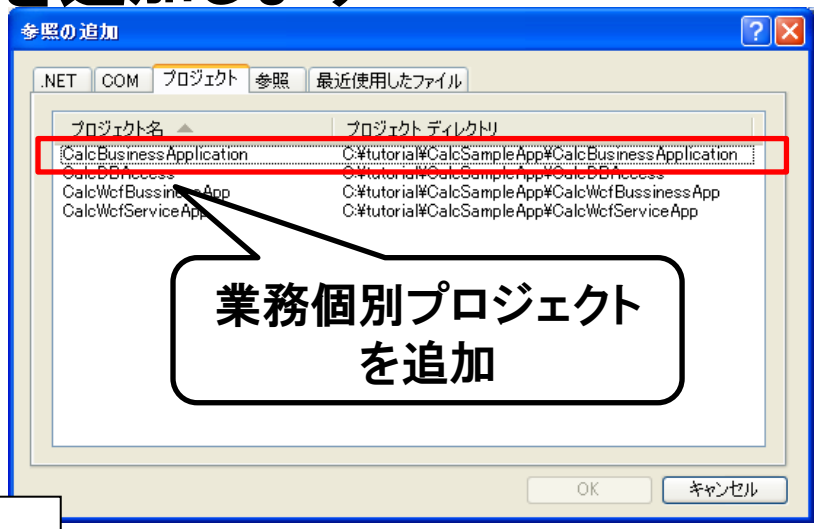
業務個別プロジェクトの作成

- 業務個別プロジェクトの作成のひな型が作成されます



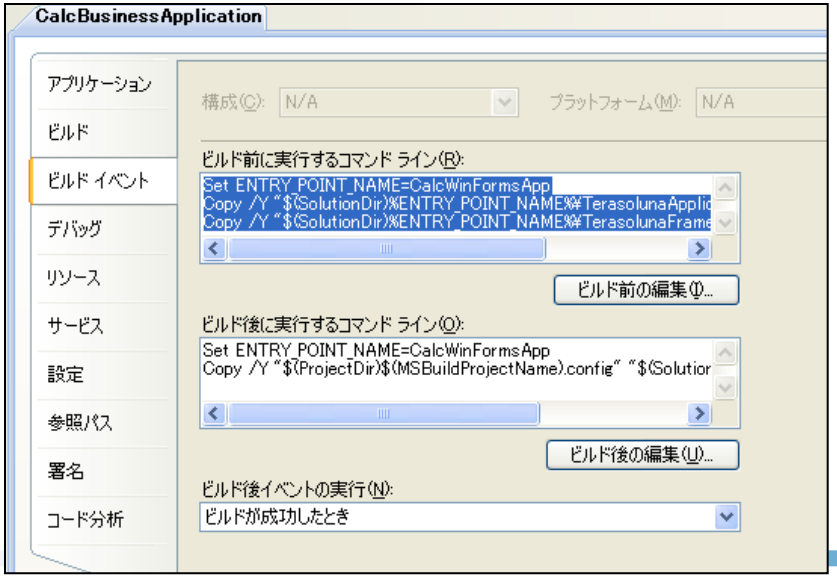
プロジェクトの参照設定の追加

- 起動アプリケーションプロジェクト(CalcWinFormsApp)で「参照の追加」を実施し、業務個別プロジェクト(CalcBusinessApplication)を追加します



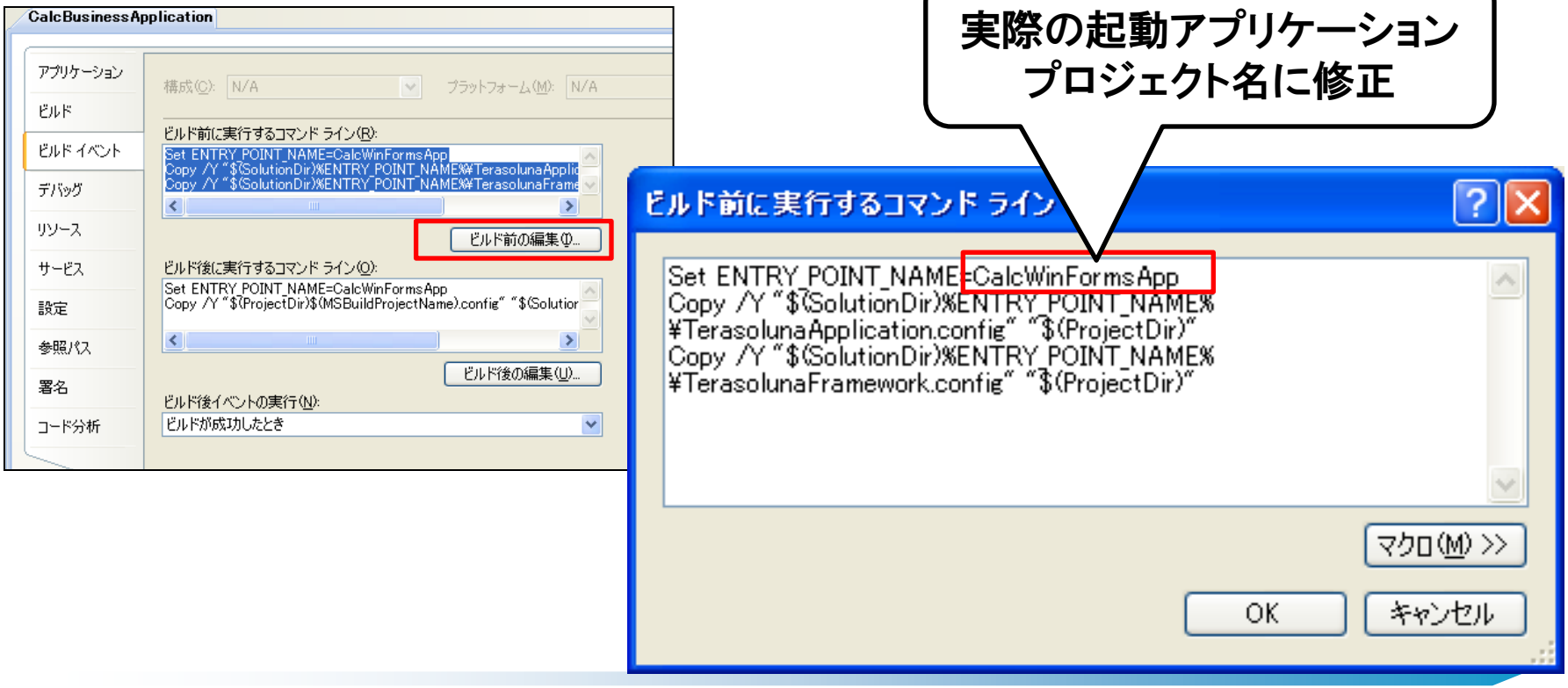
業務個別プロジェクトのビルドイベントの修正

- 業務個別プロジェクト(CalcBusinessApplication)のビルドイベントが正しく動作するように修正します
  - 対象プロジェクトを選択し右クリックメニューで「プロパティ」、「ビルドイベント」タブを選択
    - ビルドイベントには、構成ファイルをプロジェクト間でコピーするスクリプトが設定されています



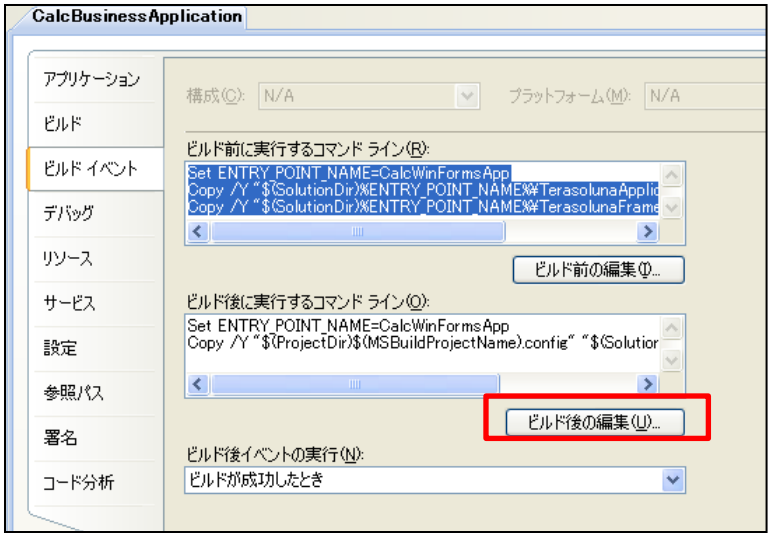
業務個別プロジェクトのビルドイベントの修正

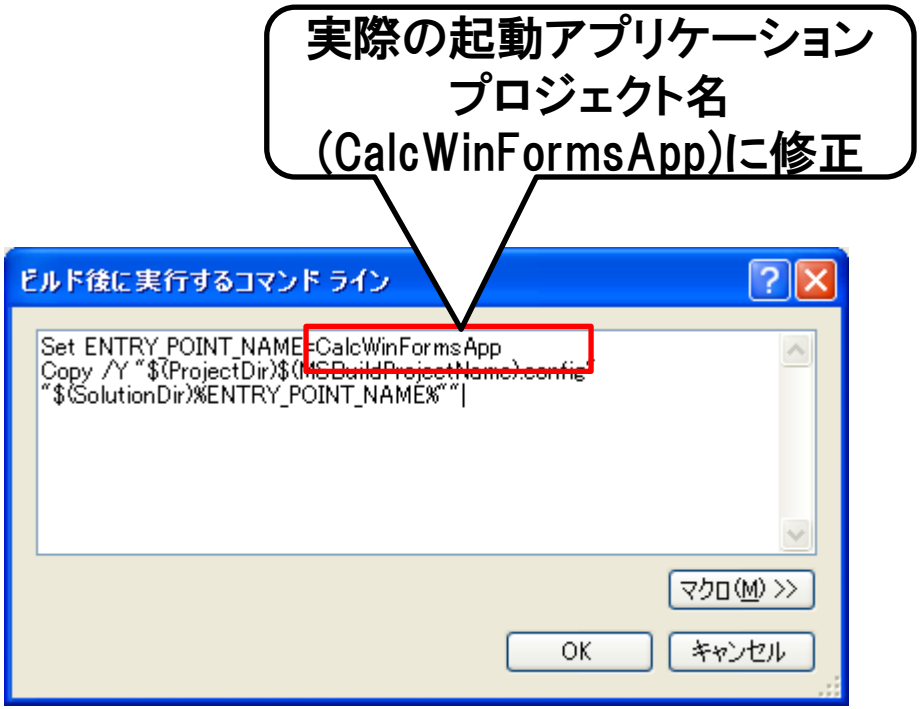
- 「ビルド前の編集」ボタンを押下し、コマンドの1行目の「Set ENTERY_POINT_NAME = StartupWinFormsApp」を、実際の起動アプリケーションプロジェクト名(CalcWinFormsApp)に修正します



業務個別プロジェクトのビルドイベントの修正

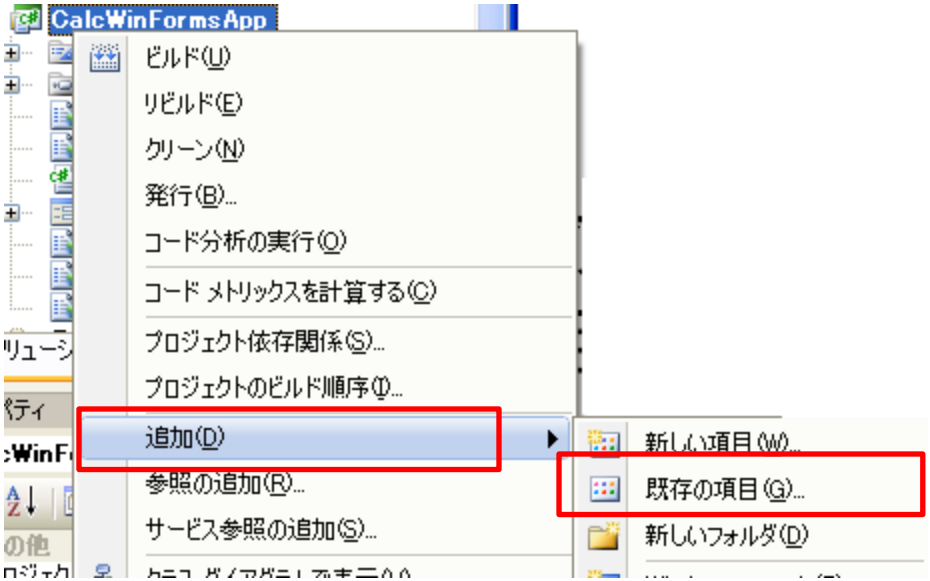
- 同様に、「ビルド後の編集」ボタンを押下し、コマンドの1行目の「Set ENTERY_POINT_NAME = StartupWinFormsApp」を実際の起動アプリケーションプロジェクト名(CalcWinFormsApp)に修正します





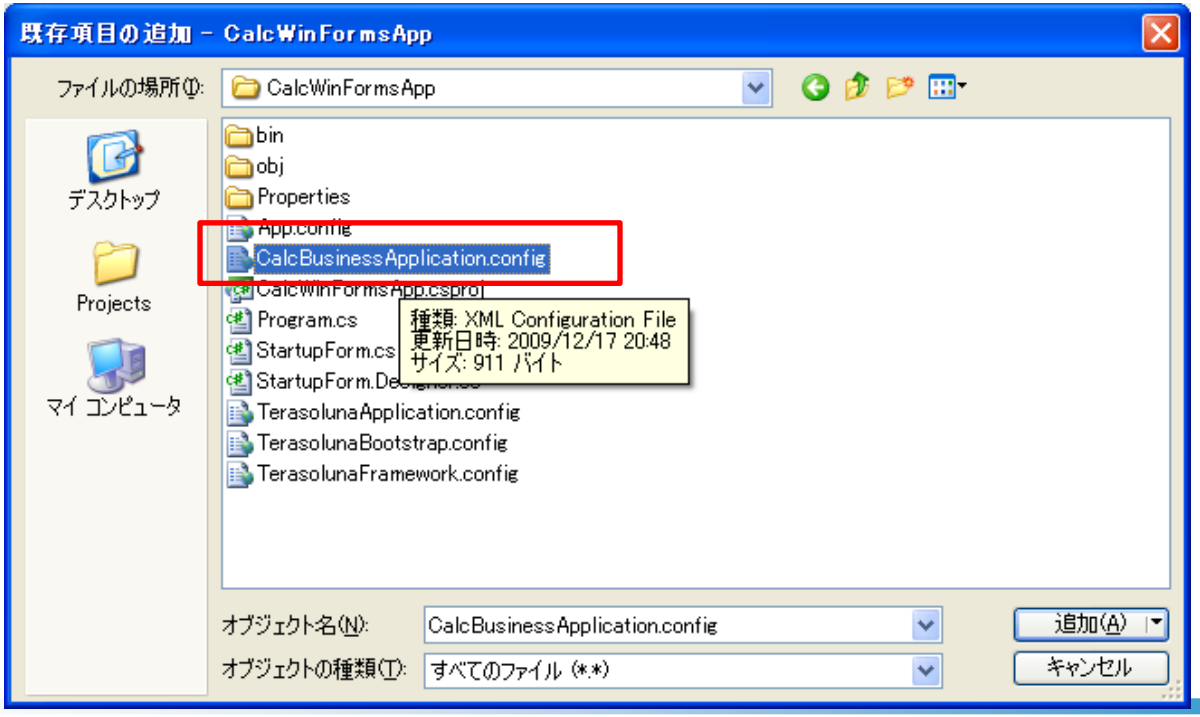
業務構成ファイルの追加

- ソリューションのビルドを実行します
  - 前述のビルドイベントを動作させるためです
- 起動アプリケーションプロジェクト(CalcWinFormsApp)の右クリックメニューで「追加」-「既存の項目」を選択します



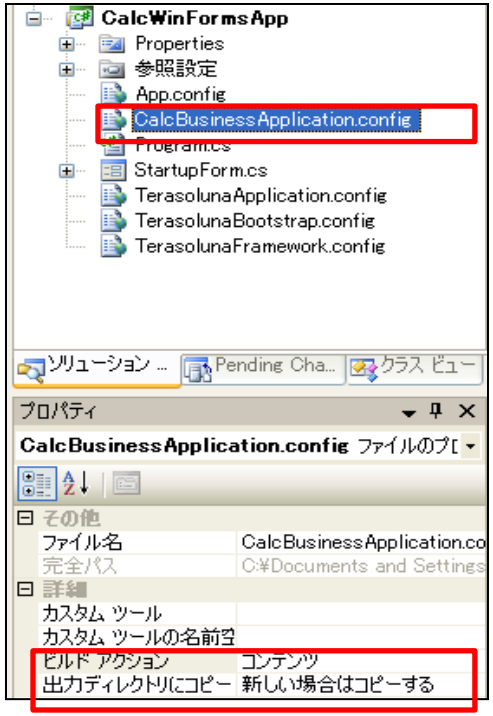
業務構成ファイルの追加

- 業務個別プロジェクトのビルドイベントにより、起動アプリケーションプロジェクト（CalcWinFormsApp）のフォルダ直下に、自動的にコピーされた「CalcBusinessApplication.config」を手動でプロジェクトに追加します（「追加」-「既存の追加」）。
  - CalcBusinessApplication.configが存在しない場合は、一度ソリューションのビルドを実行してみてください。それでも存在しない場合は、業務個別プロジェクト（CalcBusinessApplication）のビルドイベントに誤りがないか再度確認して下さい



業務構成ファイルの設定

- 先ほど追加したCalcBusinessApplication.config について、「ビルドアクション」と「出力ディレクトリにコピー」の値を設定します。
  - 「ビルドアクション」を「コンテンツ」に設定
    - ClickOnceを実施した場合に構成ファイルも配信されるようにするための設定です
  - 「出力ディレクトリにコピー」を「新しい場合はコピーする」に設定
    - 実行ファイルのフォルダに構成ファイルを出力させるための設定です

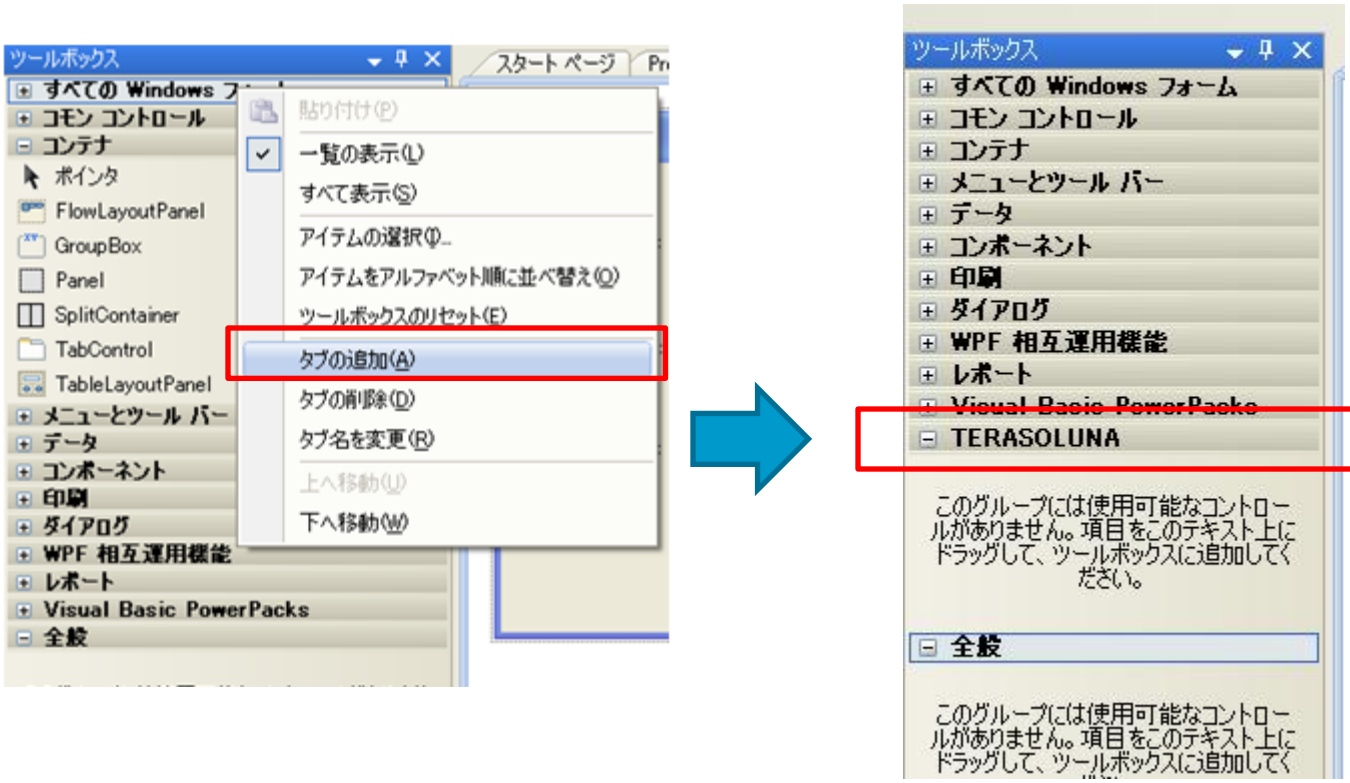


TERASOLUNAフレームワーク提供部品の追加

- TERASOLUNAフレームワークを初めて利用する場合、Visual Studioの「ツールボックス」にフレームワーク提供の画面部品を追加する必要があります
  - Terasoluna.Windows.Forms.dllによる提供部品
    - EventProcessWorkerクラス
    - FormForwarderクラス
    - SelectableValuesProviderクラス
    - ContentPlaceHolderクラス

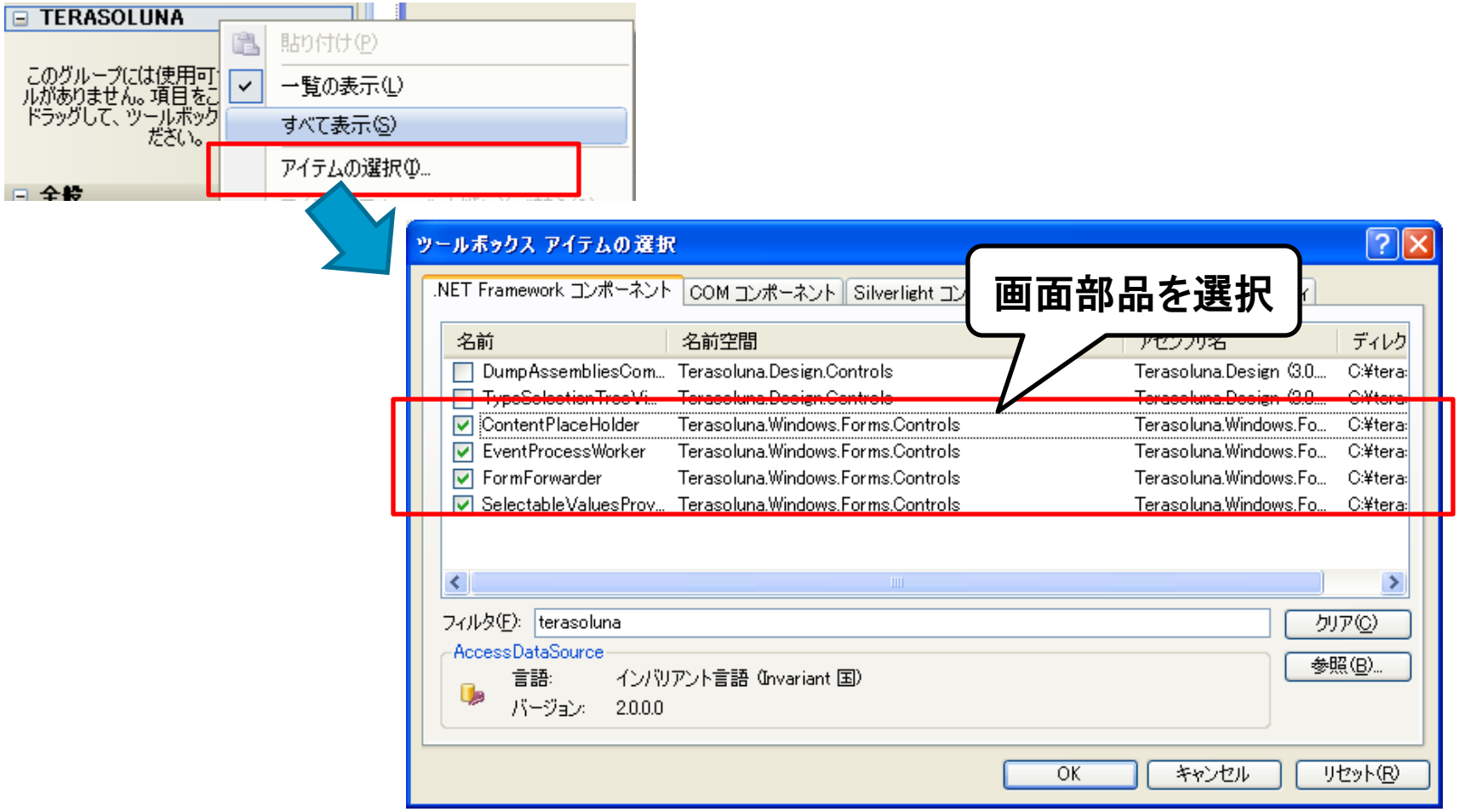
TERASOLUNAフレームワーク提供部品の追加

- 「タブの追加」でツールボックスにタブを追加します
  - ツールボックスの名前は任意の文字列で指定します
    - ここでは「TERASOLUNA」としています



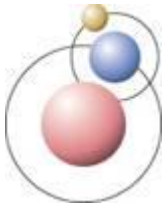
TERASOLUNAフレームワーク提供部品の追加

- 「アイテムの選択」でフレームワーク提供部品を追加します



TERASOLUNAフレームワーク提供部品の追加

- ツールボックスに部品が追加されます



.NETサーバプロジェクトの作成

TERASOLUNA

WCFサービスアプリケーションプロジェクトの作成

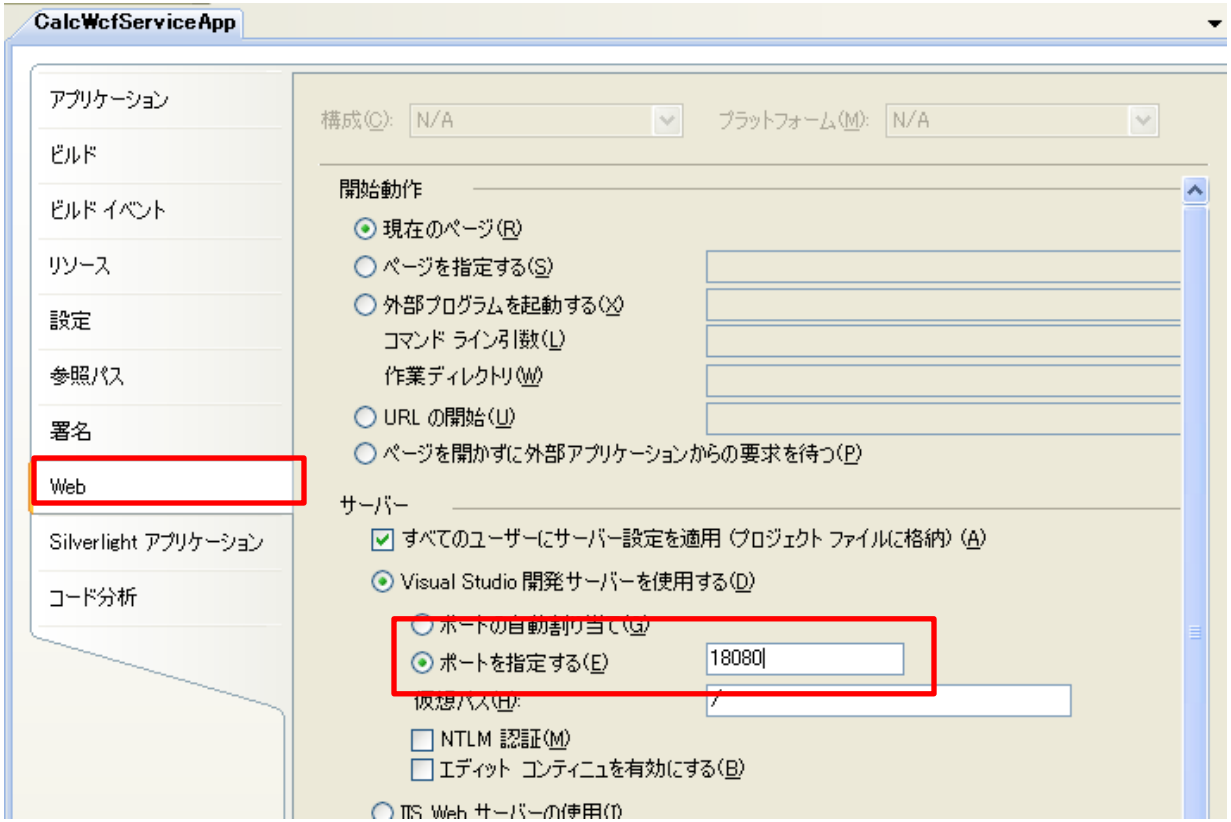
- 「TERASOLUNAプロジェクト」-「Server」-「WCFサービスアプリケーション」テンプレートで作成します

WCFサービスアプリケーションプロジェクトの作成

- 「WCFサービスアプリケーション」プロジェクトのひな型が作成されます

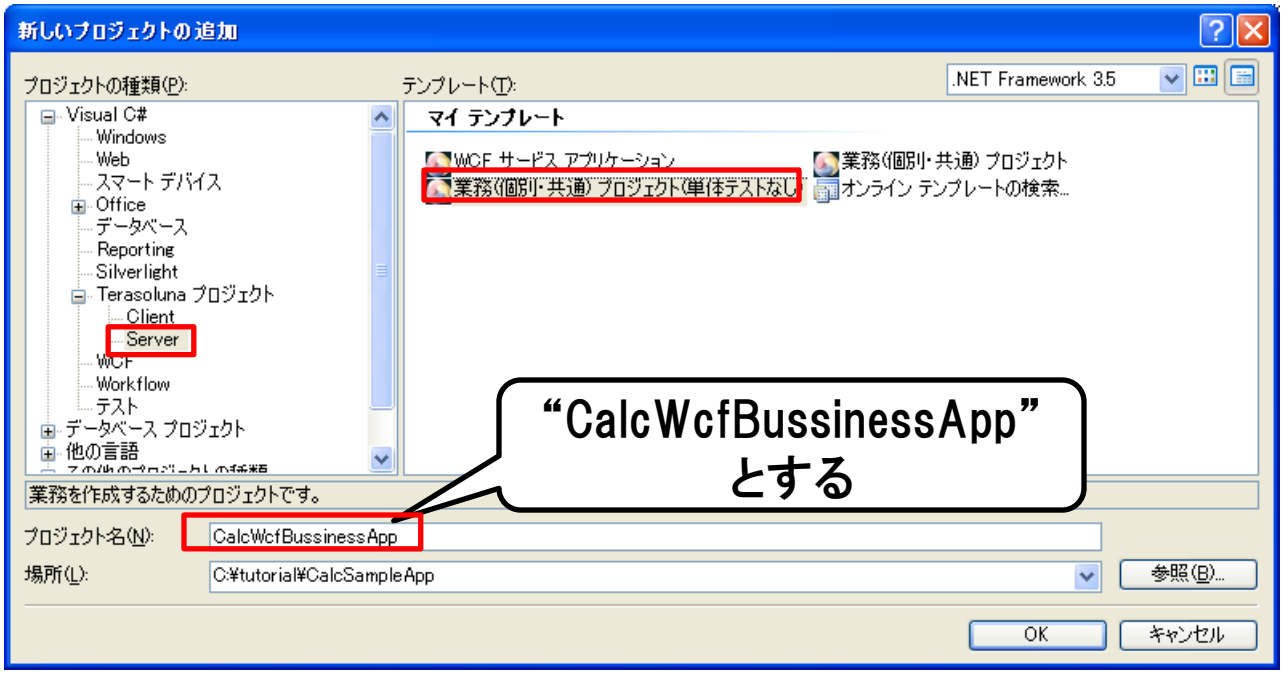
WCFサービスアプリケーションプロジェクトの作成

- WCFサービスアプリケーションプロジェクトの「プロパティ」-「Web」タブで、サーバのポート番号を「18080」固定に設定します
  - 18080番が使用済みの場合は、別のポート番号に置き換えて進めてください



業務個別プロジェクトの作成

- 「TERASOLUNAプロジェクト」-「Server」-「業務(個別・共通)プロジェクト(単体テストなし)」テンプレートで作成します(※)
  - 通常の業務開発では、UTプロジェクトを同時に生成 する「業務(個別・共通)プロジェクト」テンプレートを利用するのが推奨パターン



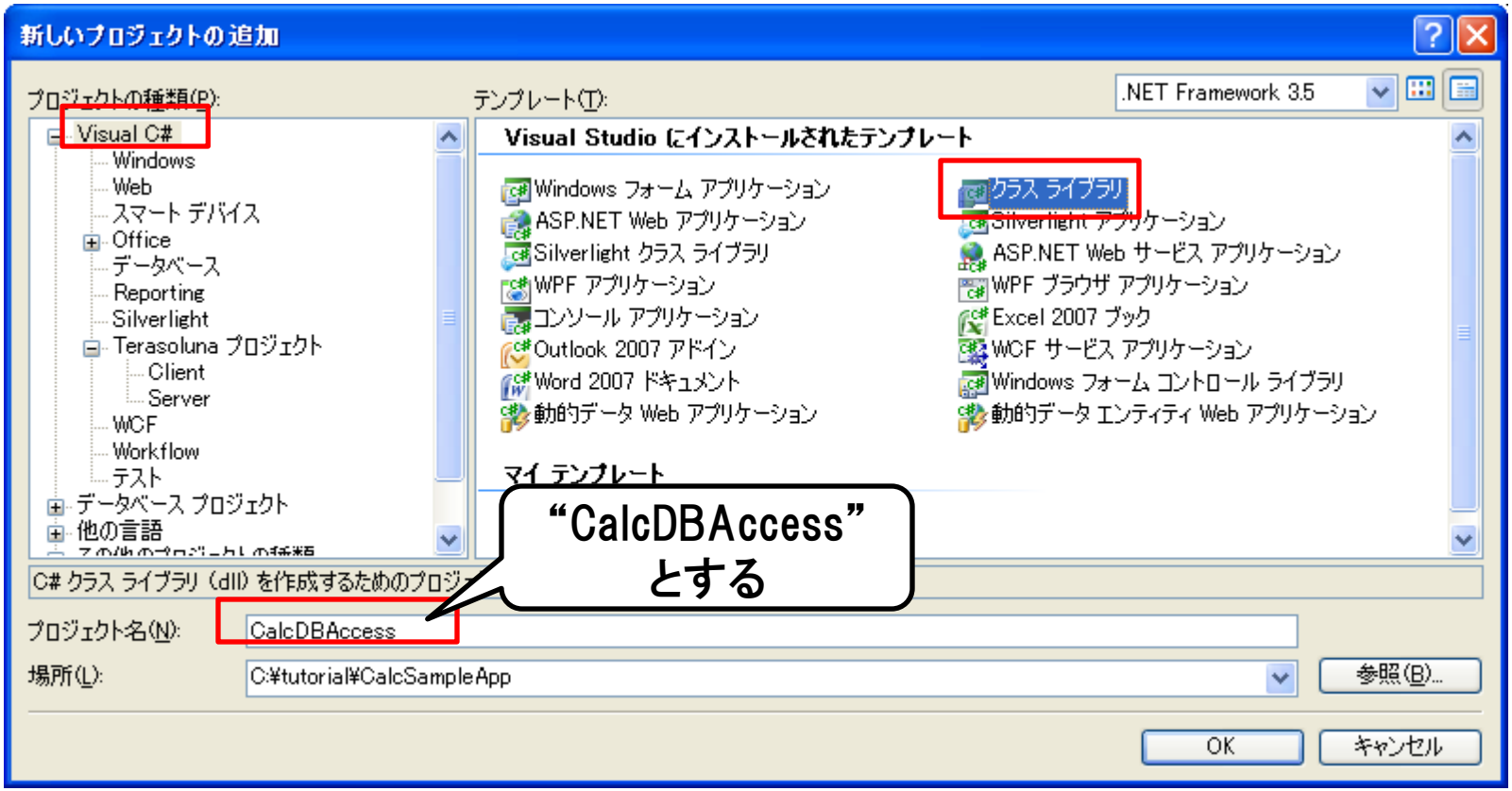
※　Visual Web Developer 2008 Express editionは「WCFサービスライブラリ」プロジェクトに対応していないため、それを基とした当プロジェクトテンプレートは利用できない。「クラスライブラリ」プロジェクトテンプレートで代用する

業務個別プロジェクトの作成

- 業務個別プロジェクトのひな型が作成されます

データアクセスプロジェクトの作成

- .NETの「クラスライブラリ」プロジェクトテンプレートで
  データアクセス用のプロジェクトを作成します

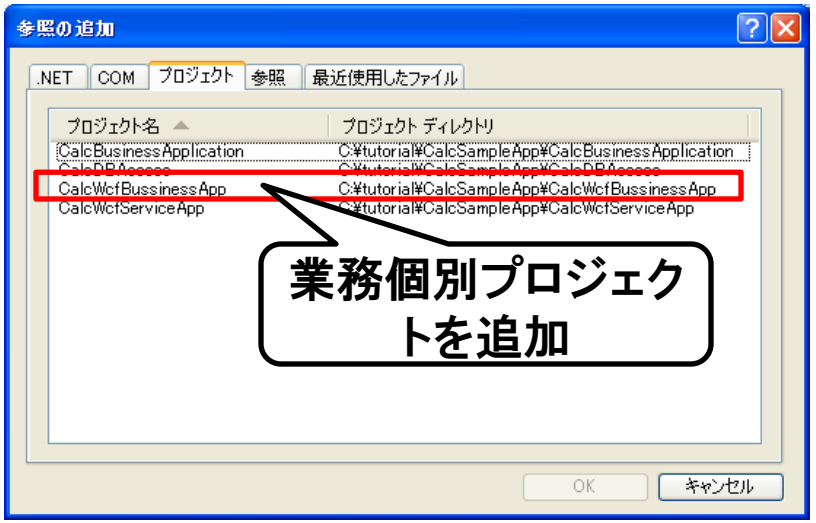


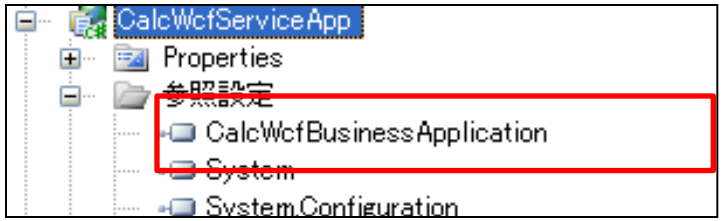
データアクセスプロジェクトの作成

- データアクセスプロジェクトのひな型が作成されます

プロジェクトの参照設定の追加

- WCFサービスアプリケーションプロジェクト(CalcWcfServiceApp)で「参照の追加」を実施し、業務個別プロジェクト(CalcWcfBusinessApplication)を追加します

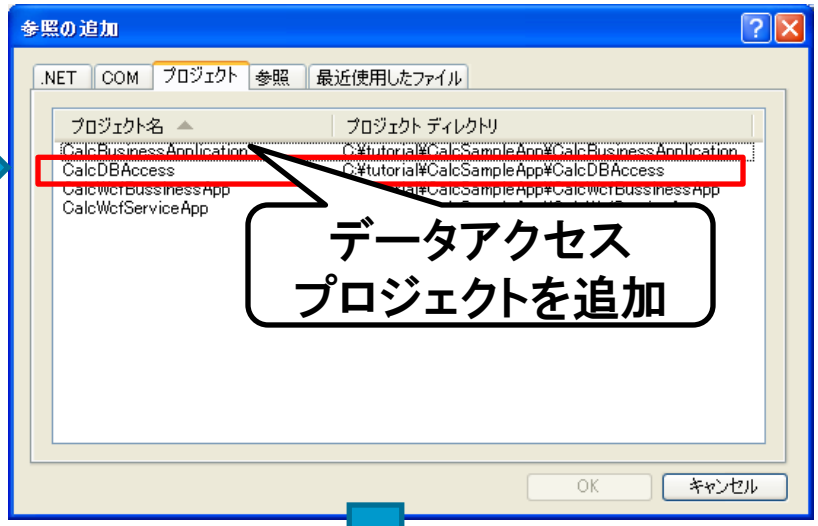


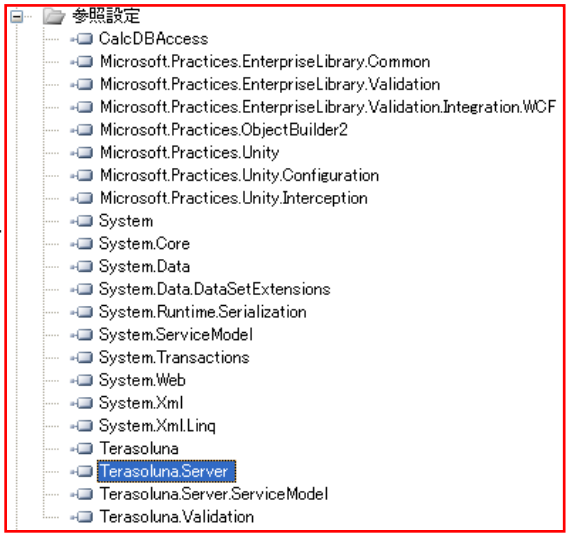


参照設定に追加される

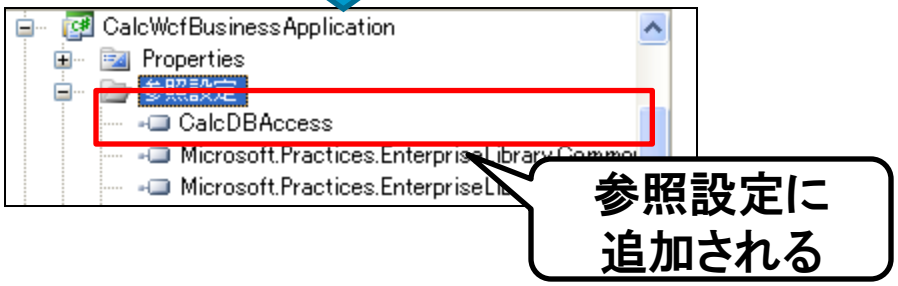
プロジェクトの参照設定の追加

- 個別業務プロジェクト(CalcWcfBusinessApplication)で「参照の追加」を実施し、データアクセスプロジェクト(CalcDBAccess)を追加します(※)





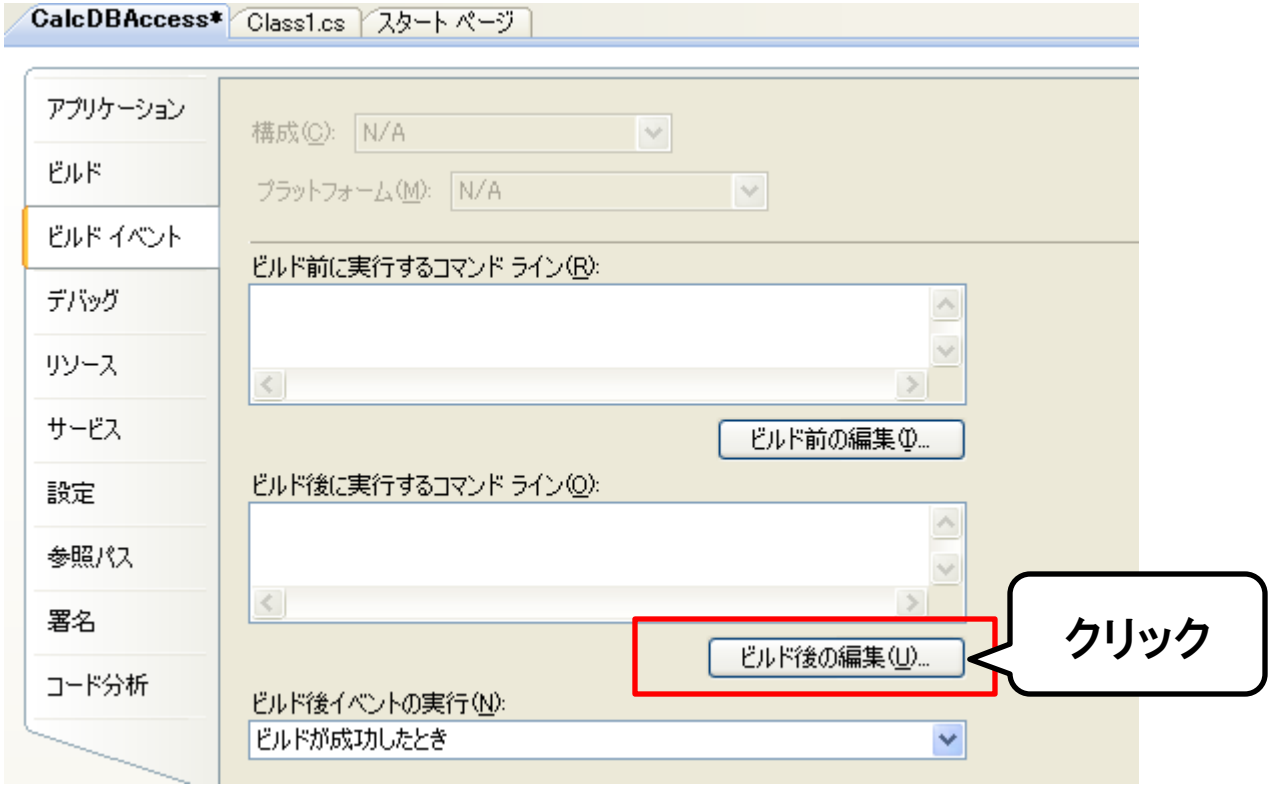
参照設定の全内容
（ Visual Web Developer 2008 Express editionの場合、手動で設定要）



※　Visual Web Developer 2008 Express editionで、「クラスライブラリ」プロジェクトテンプレートをベースにしている場合、上記設定の他、「参照設定の全内容」にある「参照」のうち足りないものを手動で追加して下さい

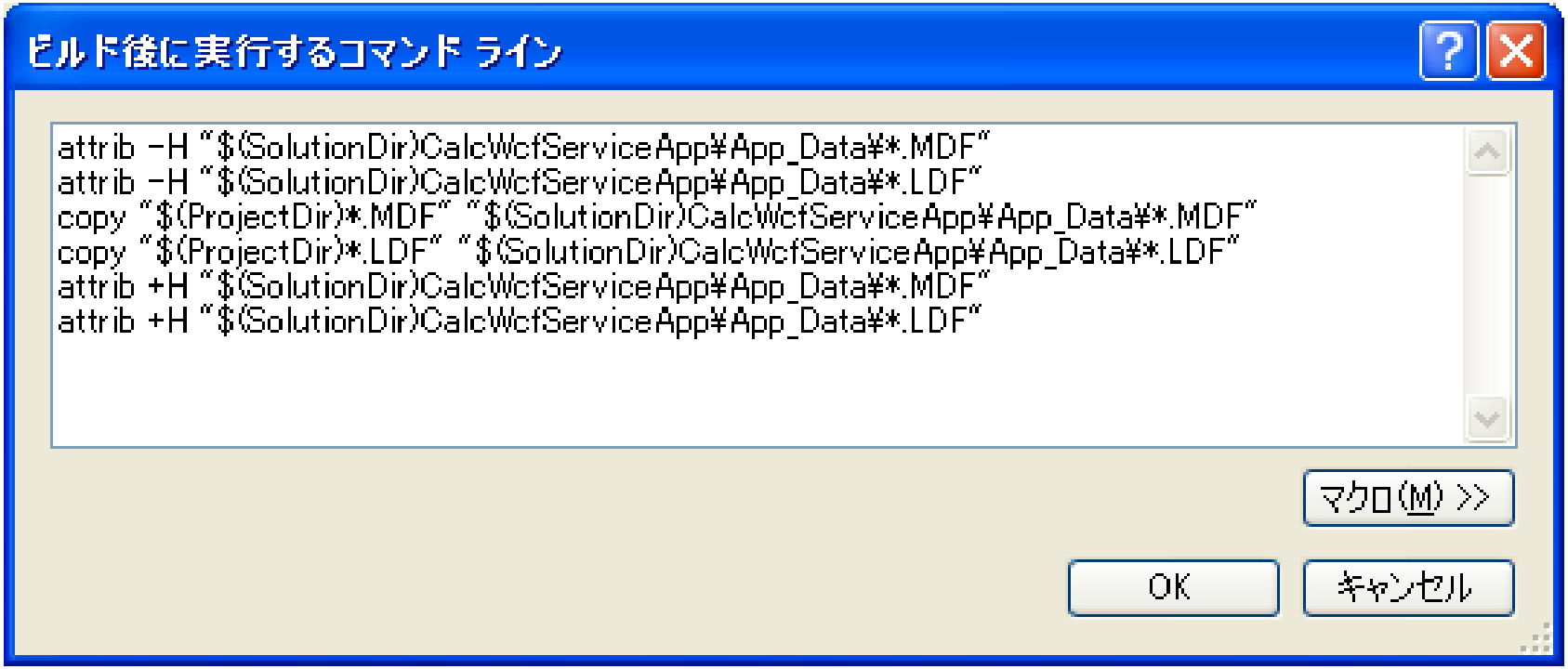
データアクセスプロジェクトのビルドイベントの修正

- ビルド時に、WCFサービスアプリケーションのApp_Dataにmdfファイルとldfファイルをコピーするようにビルドイベントを記述します



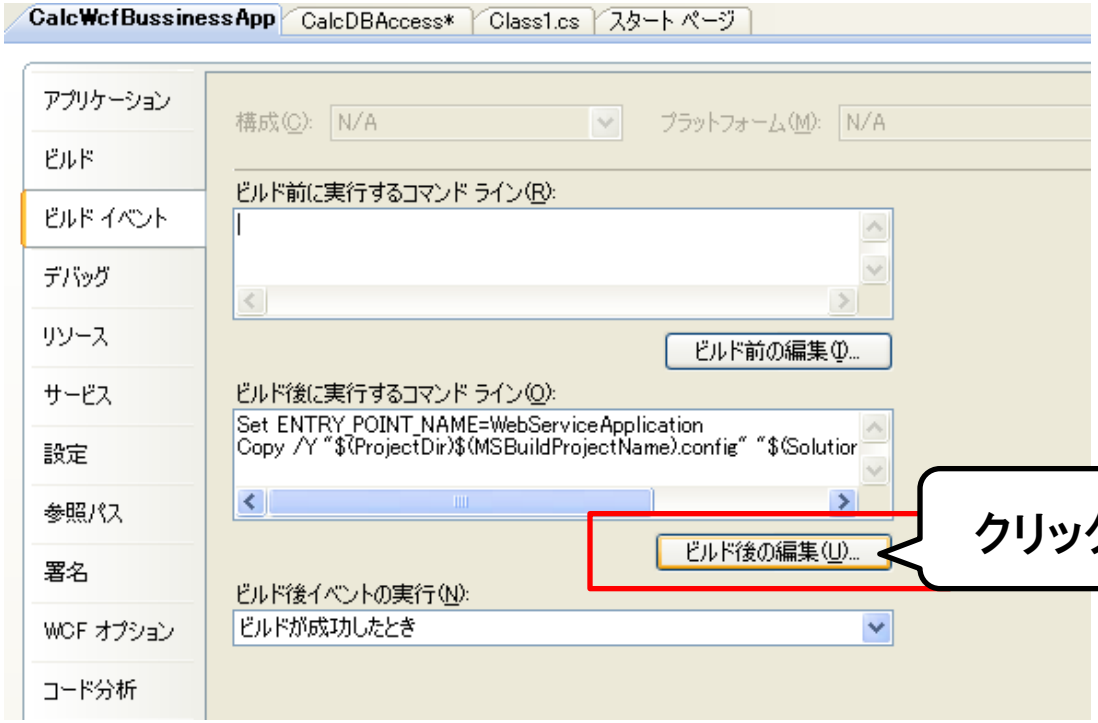
データアクセスプロジェクトのビルドイベントの修正

- 以下のビルドイベントを記述します



業務個別プロジェクトのビルドイベントの修正

- 業務個別プロジェクト(CalcWcfBusinessApplication)のビルドイベントが正しく動作するように修正します(※)
  - ビルドイベントには、WCFサービスアプリケーションプロジェクトに業務構成ファイルをコピーするスクリプトが設定されています



【設定内容】
Set ENTRY_POINT_NAME=CalcWcfServiceApp
Copy /Y "$(ProjectDir)$(MSBuildProjectName).config"
"$(SolutionDir)%ENTRY_POINT_NAME%"

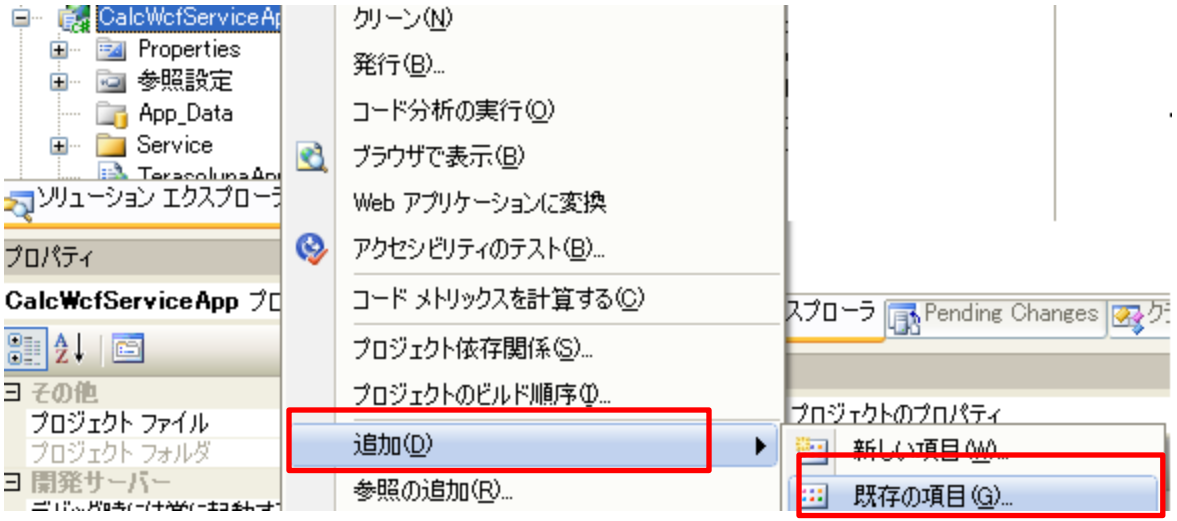
※ Visual Web Developer 2008 Express editionで、「クラスライブラリ」プロジェクトテンプレートをベースにしている場合、ビルドイベントに【設定内容】の内容を貼りつけて下さい。

業務個別プロジェクトのビルドイベントの修正

- 「ビルド後の編集」ボタンを押下し、コマンドの1行目の「Set ENTERY_POINT_NAME=WebServiceApplication」となっている部分を実際のWCFサービスアプリケーションプロジェクト名(CalcWcfServiceApp)に修正します

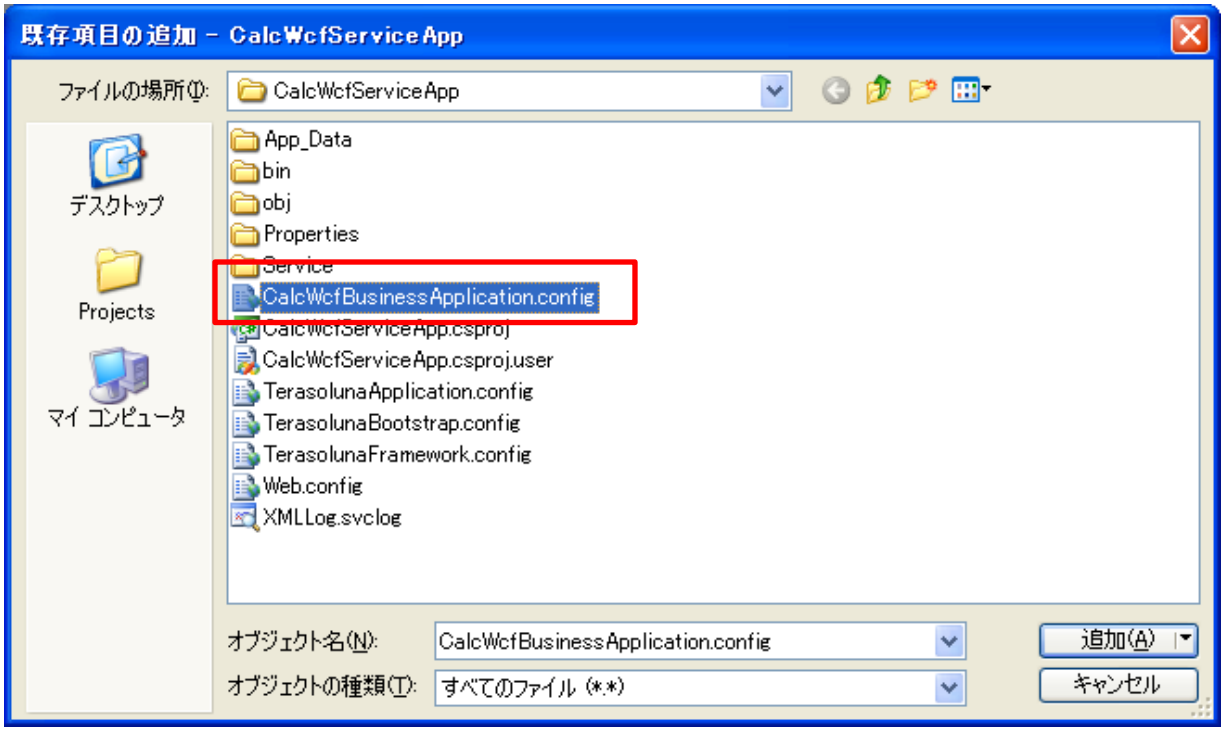
業務構成ファイルの追加

- ソリューションのビルドを実行します
  - 前述のビルドイベントを動作させるためです
- WCFサービスアプリケーションプロジェクト(CalcWcfServiceApp)の右クリックメニューで「追加」-「既存の項目」を選択します



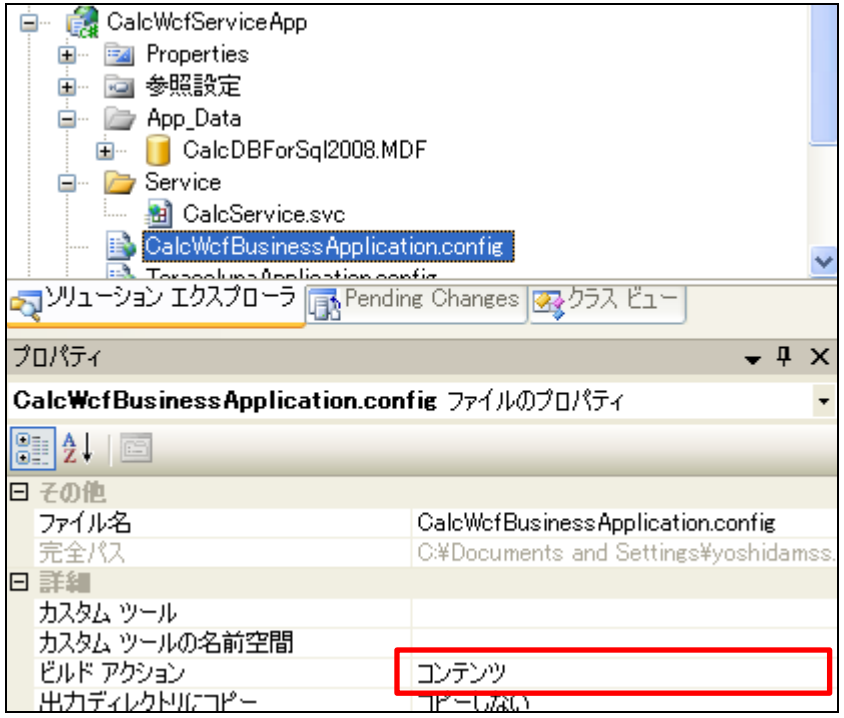
業務構成ファイルの追加

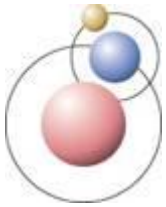
- 業務個別プロジェクトのビルドイベントにより、WCFサービスアプリケーションプロジェクト(CalcWcfServiceApp)のフォルダ直下に自動的にコピーされた「CalcWcfBusinessApplication.config」をプロジェクトに追加します。
  - CalcWcfBusinessApplication.configが存在しない場合は、一度ソリューションのビルドを実行してみてください。それでも存在しない場合は、業務個別プロジェクト(CalcWcfBusinessApplication)のビルドイベントに誤りがないか再度確認して下さい



業務構成ファイルの追加

- 「ビルドアクション」の値を「コンテンツ」に設定
  - 構成ファイルを出力させるための設定です



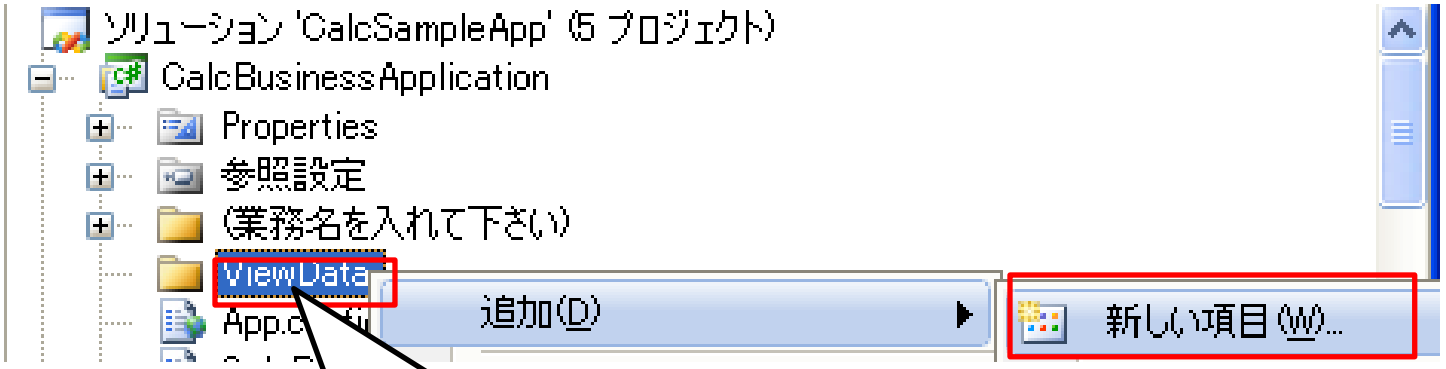


ログイン処理の作成①
（.NETクライアントに閉じた処理）

TERASOLUNA

ログイン画面用の画面データ作成

- 業務個別プロジェクトにログイン画面用の画面データを作成します
  - なお、「(業務名を入れてください)」という名前のフォルダが生成されますが今回は使用しません。
  - 通常の業務開発プロジェクトでは、このフォルダを利用して、サブ業務ごとに名前空間を分割し開発することを推奨

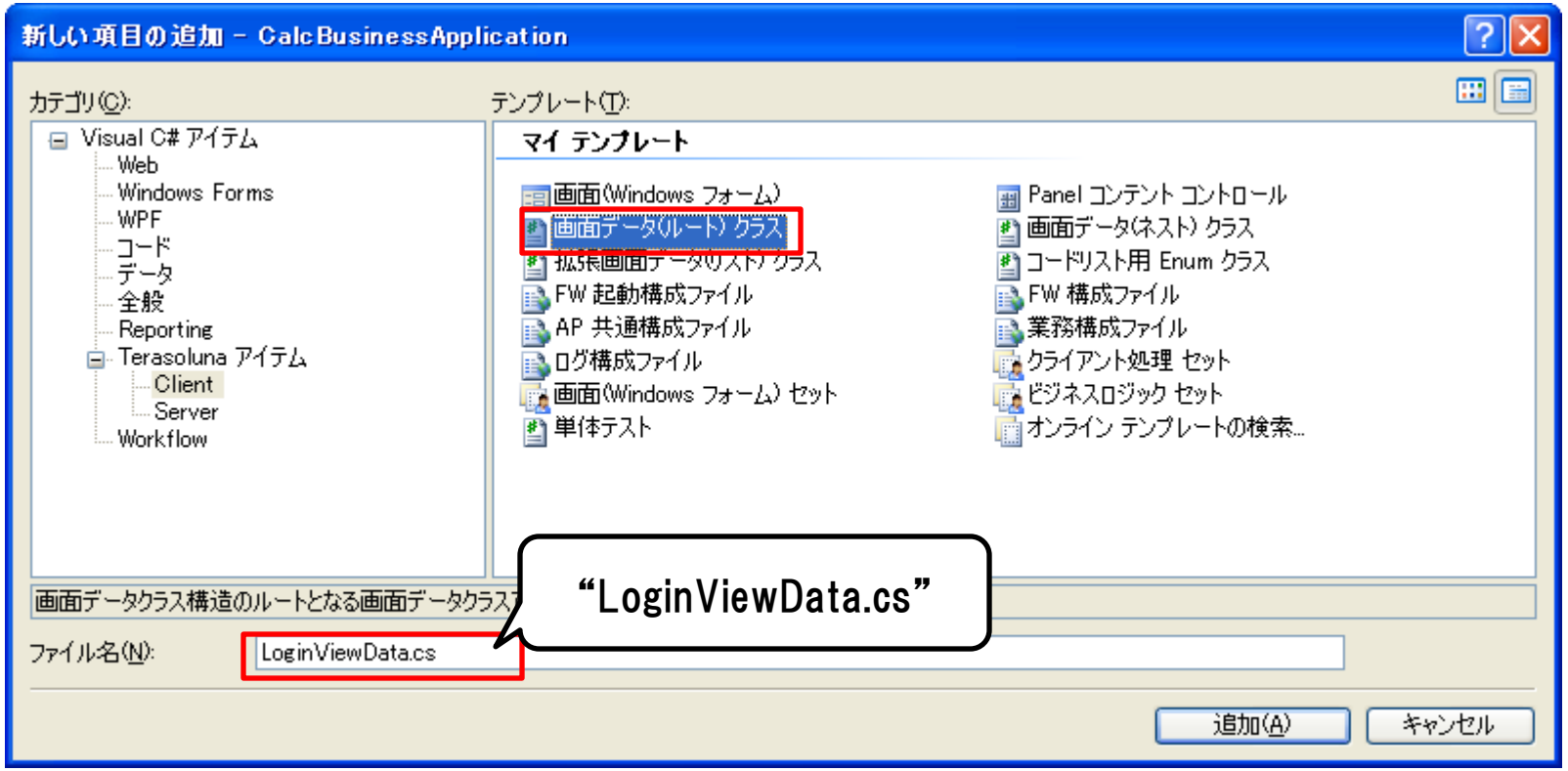


「ViewData」フォルダを作成

ViewDataフォルダで新しい項目を追加

ログイン画面用の画面データの作成

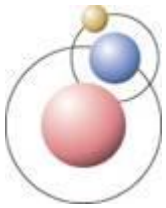
- 画面データ(ルート)クラスを作成します
  - TERASOLUNAのアイテムテンプレートを使用します



ログイン画面用の画面データの作成

- 画面データクラス(ルート)のひな形が作成されます。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using Terasoluna.Validation.Validators;
using Terasoluna.Windows.ViewModel.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;

namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("")]
        public virtual string Prop1 { get; set; }
    }
}
```



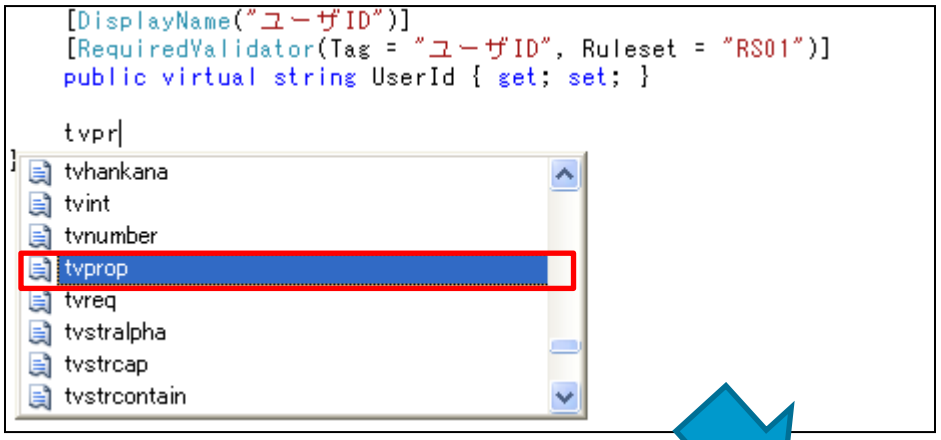
ログイン画面用の画面データの作成

- ひな型を以下のように修正し、UserIdというプロパティを作成します

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        public virtual string UserId { get; set; }
    }
}
```

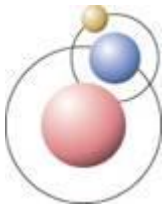
ログイン画面用の画面データの作成

- プロパティを追加する場合は、スニペット「tvprop」で追加します
  - TERASOLUNAフレームワークのカスタムスニペットは全て「tv」から始まる文字列になります



プロパティの
ひな形が追加される

```
[DisplayName("ユーザID")]
[RequiredValidator(Tag = "ユーザID", Ruleset = "RS01")]
public virtual string UserId { get; set; }

[DisplayName("表示名")]
public virtual string MyProperty { get; set; }
```

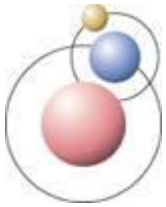


ログイン画面用の画面データの作成

- ひな型を以下のように修正しPasswordというプロパティを作成します

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        public virtual string UserId { get; set; }

        [DisplayName("パスワード")]
        public virtual string Password { get; set; }
    }
}
```


TERASOLUNA



ログイン画面用の画面データの作成

- 画面の入力値検証処理を実装します
  - 単項目チェックの実装は、フレームワークが提供するバリデータのカスタム属性を付与します
  - カスタム属性ごとにスニペットが用意されており、通常スニペットでカスタム属性を追加します

ログイン画面用の画面データの作成

- UserIDプロパティに必須入力チェック(RequiredValidator属性)を定義します
  - スニペットは「tvreq」を使います
  - TERASOLUNAフレームワークのカスタムスニペットは全て「tv」から始まる文字列になります

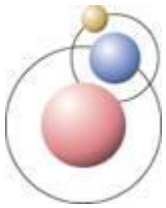
```
[DefaultRuleset("RS01")]
// [RulesetMapping("RS01", "", "")]
public class LoginViewData : ValidatableRootViewData
{
    [DisplayName("ユーザID")]
    tvre
    public virtual string UserId { get; set; }
```

tvint
tvnumber
tvprop
tvreq
tvstralpha
tvstrcap
tvstrcontain

set; }

```
[DefaultRuleset("RS01")]
// [RulesetMapping("RS01", "", "")]
public class LoginViewData : ValidatableRootViewData
{
    [DisplayName("ユーザID")]
    [RequiredValidator(Tag = "項目名", Ruleset = "RS01")]
    public virtual string UserId { get; set; }
```

バリデータの
ひな形が追加される



ログイン画面用の画面データの作成

- ひな型のTagプロパティを修正します
  - Tagプロパティは、エラーメッセージの置換文字列に相当
    - エラーメッセージ「”{2}”は入力必須項目です。」の{2}

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        [RequiredValidator(Tag = "ユーザID", Ruleset = "RS01")]
        public virtual string UserId { get; set; }

        [DisplayName("パスワード")]
        public virtual string Password { get; set; }
    }
}
```

ログイン画面用の画面データの作成

- スニペットにより生成されたRulesetプロパティを確認
  - Ruleset(ルールセット)名は、どのバリデータの組み合わせを入力チェックとして実行するかを識別する名前です
  - ここでは、そのままの値(”RS01”)にしておきます

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        [RequiredValidator(Tag = "ユーザID", Ruleset = "RS01")]
        public virtual string UserId { get; set; }

        [DisplayName("パスワード")]
        public virtual string Password { get; set; }
    }
}
```

ログイン画面用の画面データの作成

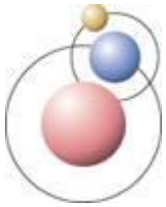
- テンプレートにより生成されたDefaultRuleset属性を確認します
  - DefaultRuleset属性は、即値チェック(ロストフォーカス時の入力チェック)として実行したいルールセット名を表します。
  - ここでは、そのままの値(“RS01”)にしておきます

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        [RequiredValidator(Tag = "ユーザID", Ruleset = "RS01")]
        public virtual string UserId { get; set; }

        [DisplayName("パスワード")]
        public virtual string Password { get; set; }
    }
}
```

ログイン画面用の画面データの作成

- 同様の手順でPasswordプロパティにも必須入力チェックを定義します。

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class LoginViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        [RequiredValidator(Tag = "ユーザID", Ruleset = "RS01")]
        public virtual string UserId { get; set; }

        [DisplayName("パスワード")]
        [RequiredValidator(Tag = "パスワード", Ruleset = "RS01")]
        public virtual string Password { get; set; }
    }
}
```

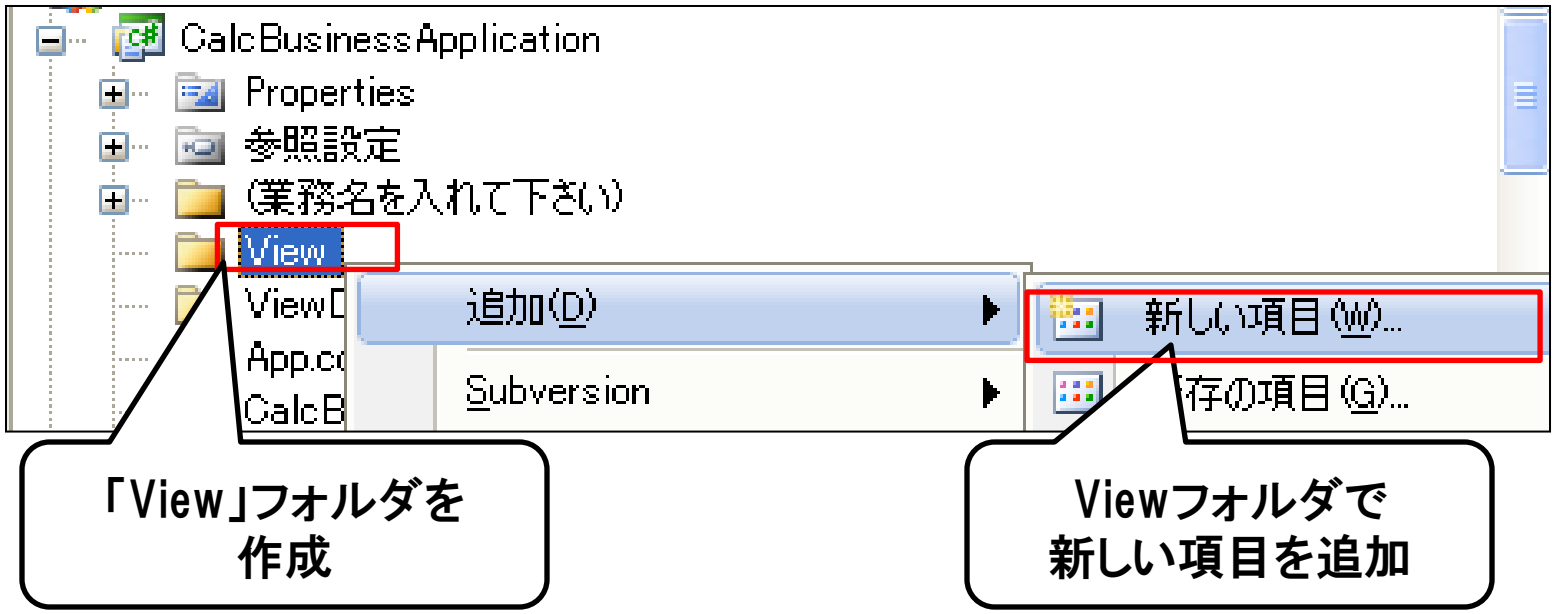


ログイン画面用の画面データの作成

- ビルドして正常終了することを確認します
  - 次の作業で、「データソースウィザード」を使って本画面データクラスを追加するためには、あらかじめビルドをしておく必要があります

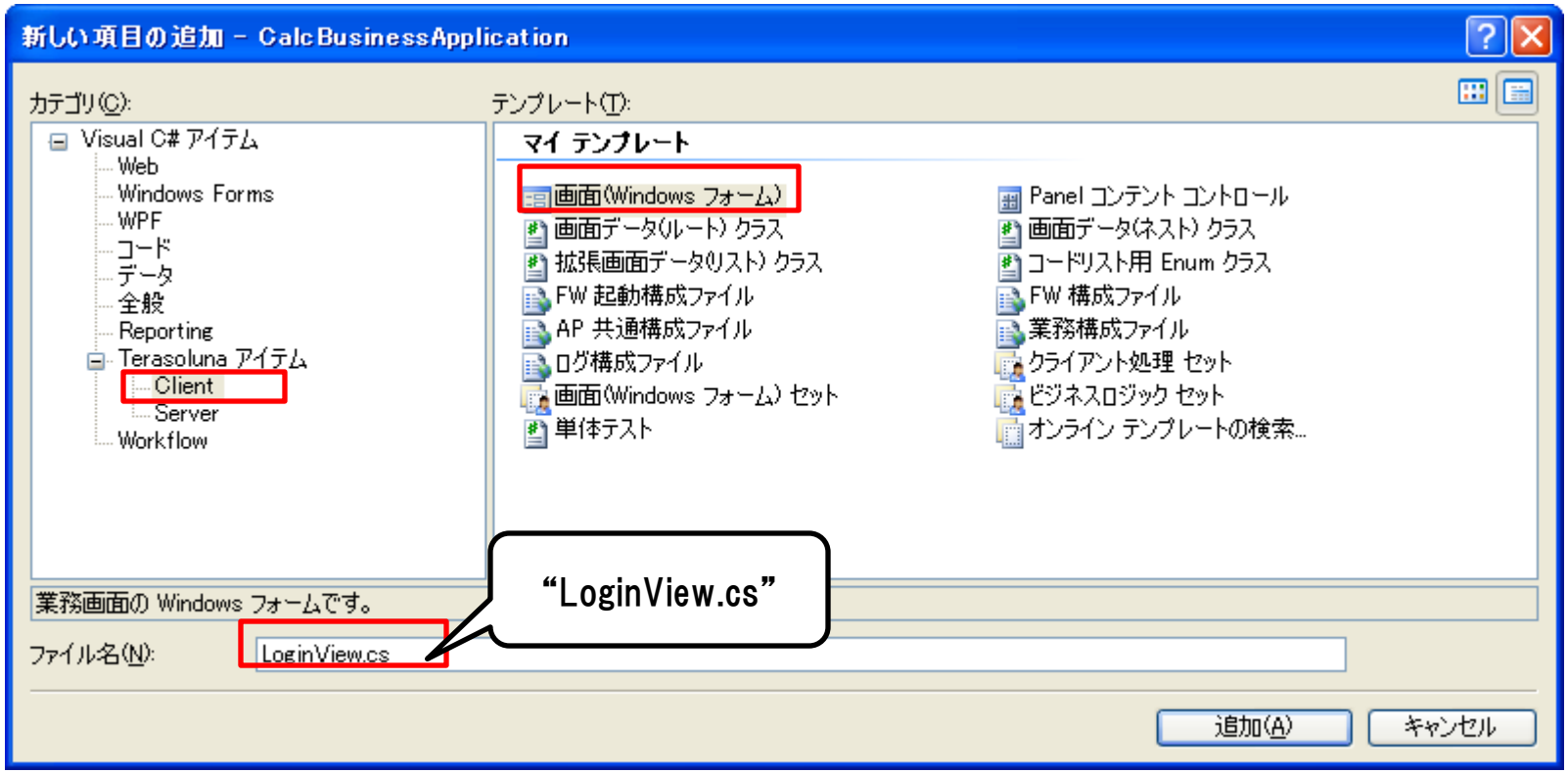
ログイン画面の作成

- 業務個別プロジェクトにログイン画面を作成します



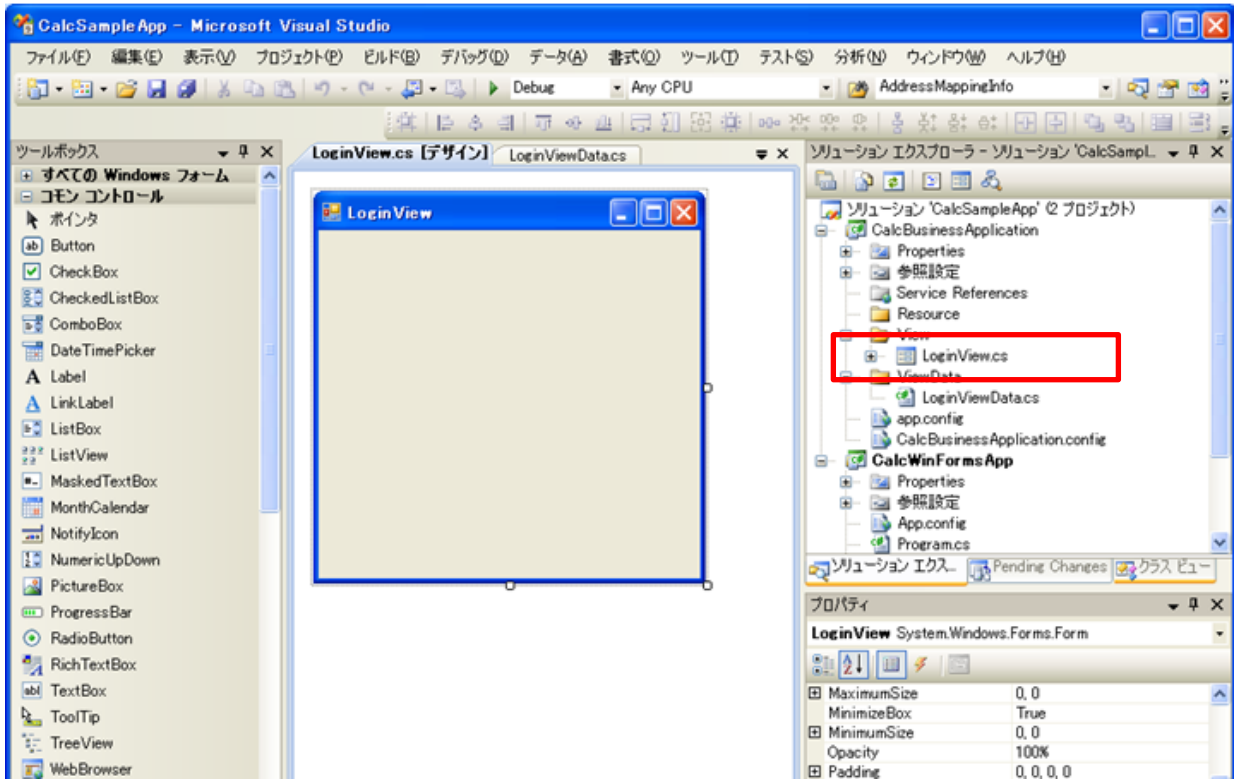
ログイン画面の作成

- 画面クラスを作成します
  - TERASOLUNAのアイテムテンプレートを使用します



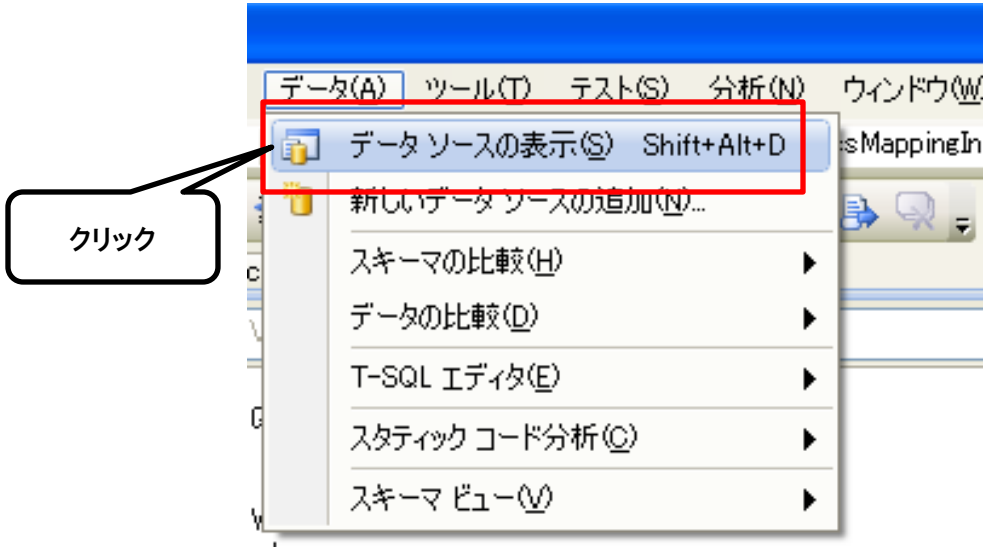
ログイン画面の作成

- 画面が生成され、デザイナが起動します
- 画面のStartPositionプロパティを「CenterScreen」にします



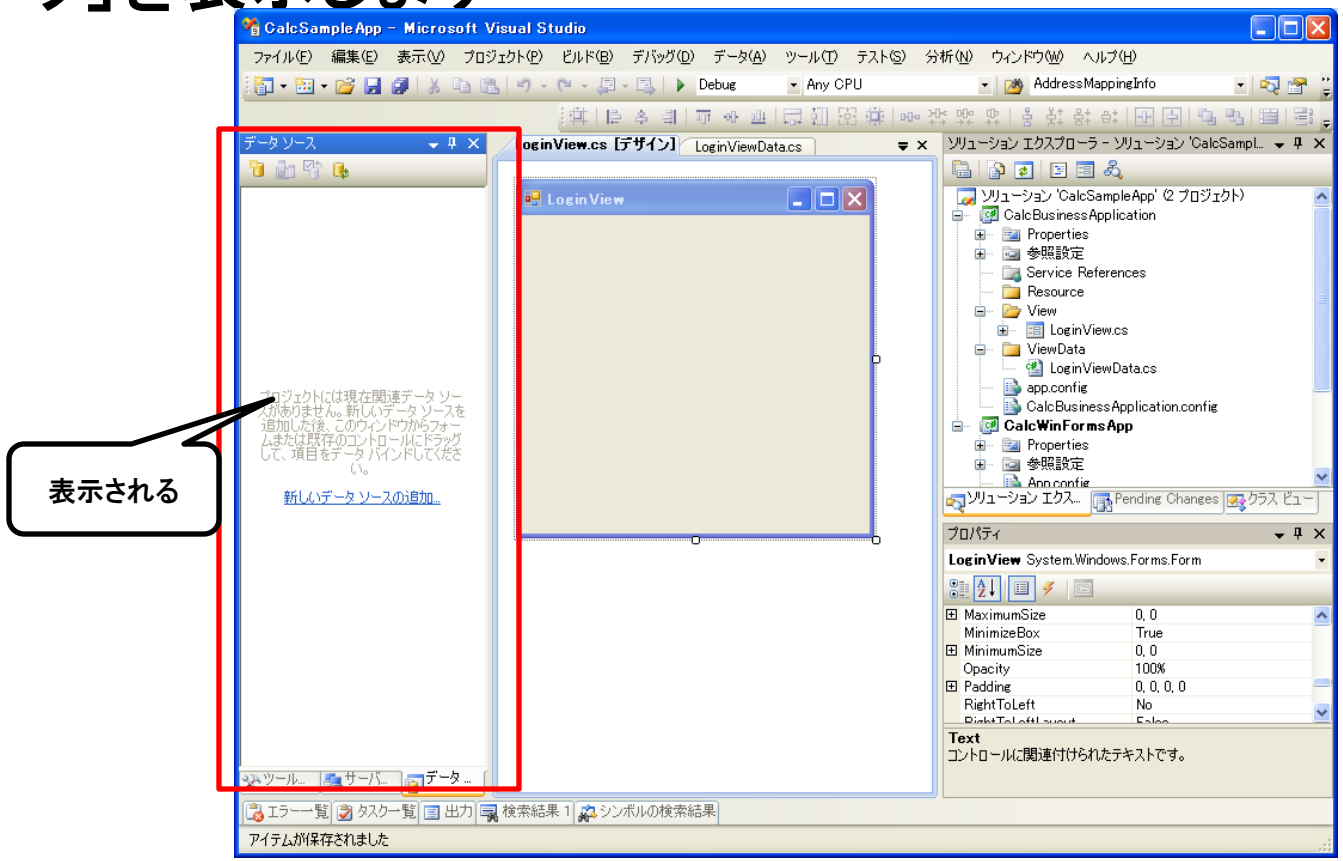
データソースウィンドウの表示

- 「データソースの表示」を選択し「データソースウィンドウ」を表示します



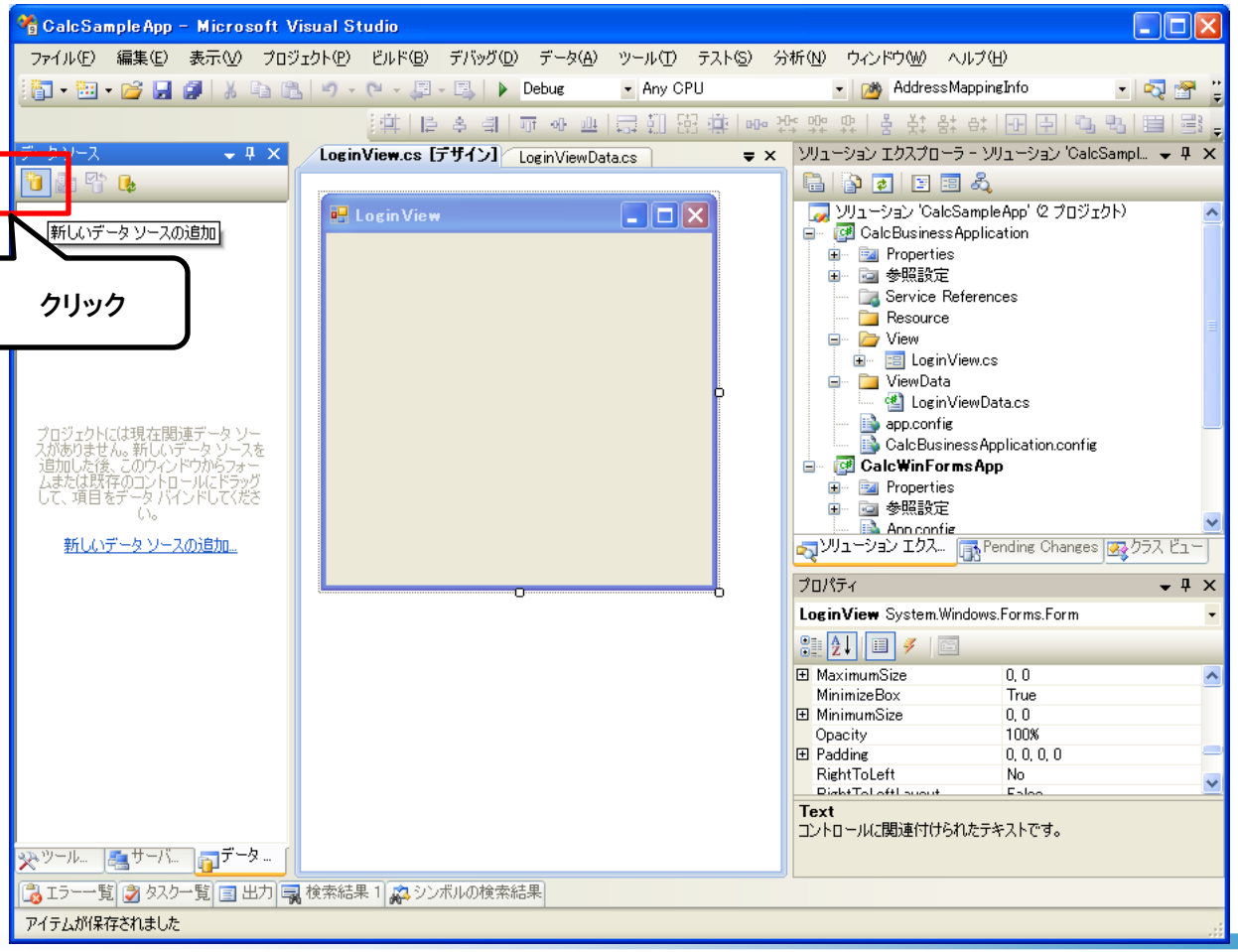
データソースウィンドウの表示

- 「データソースの表示」を選択し「データソースウィンドウ」を表示します



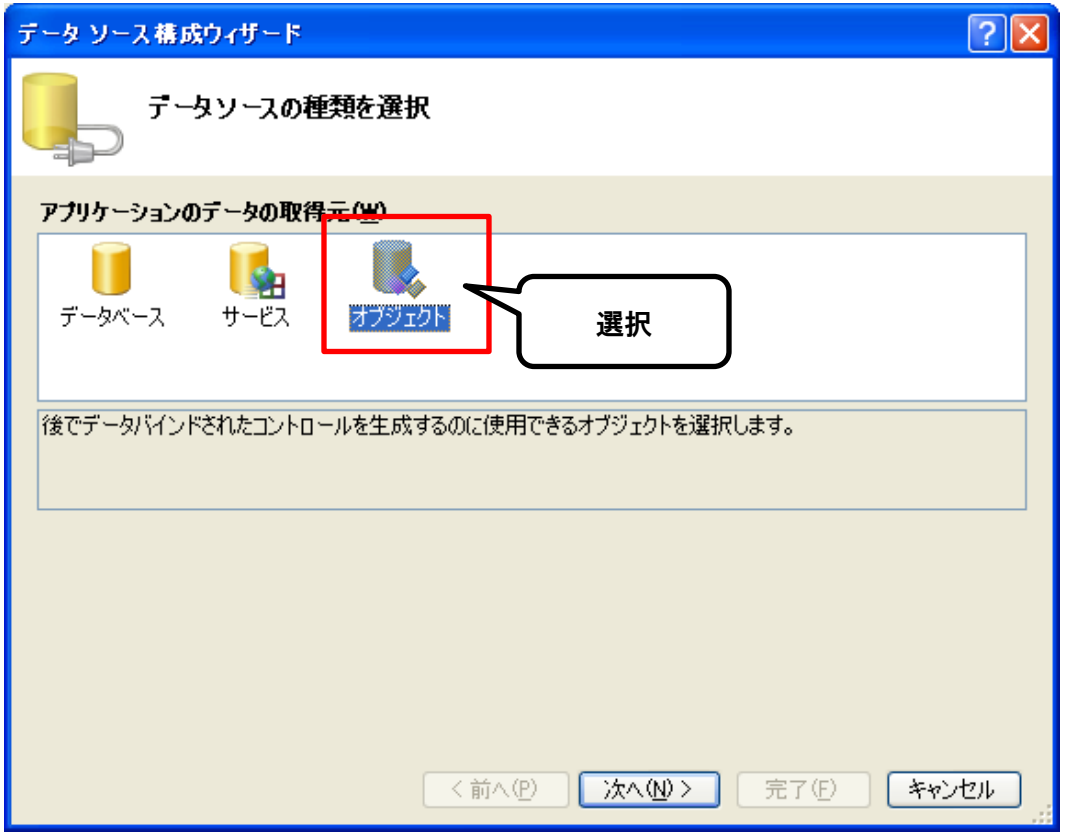
データソース構成ウィザード

- データソース構成ウィザードを起動します



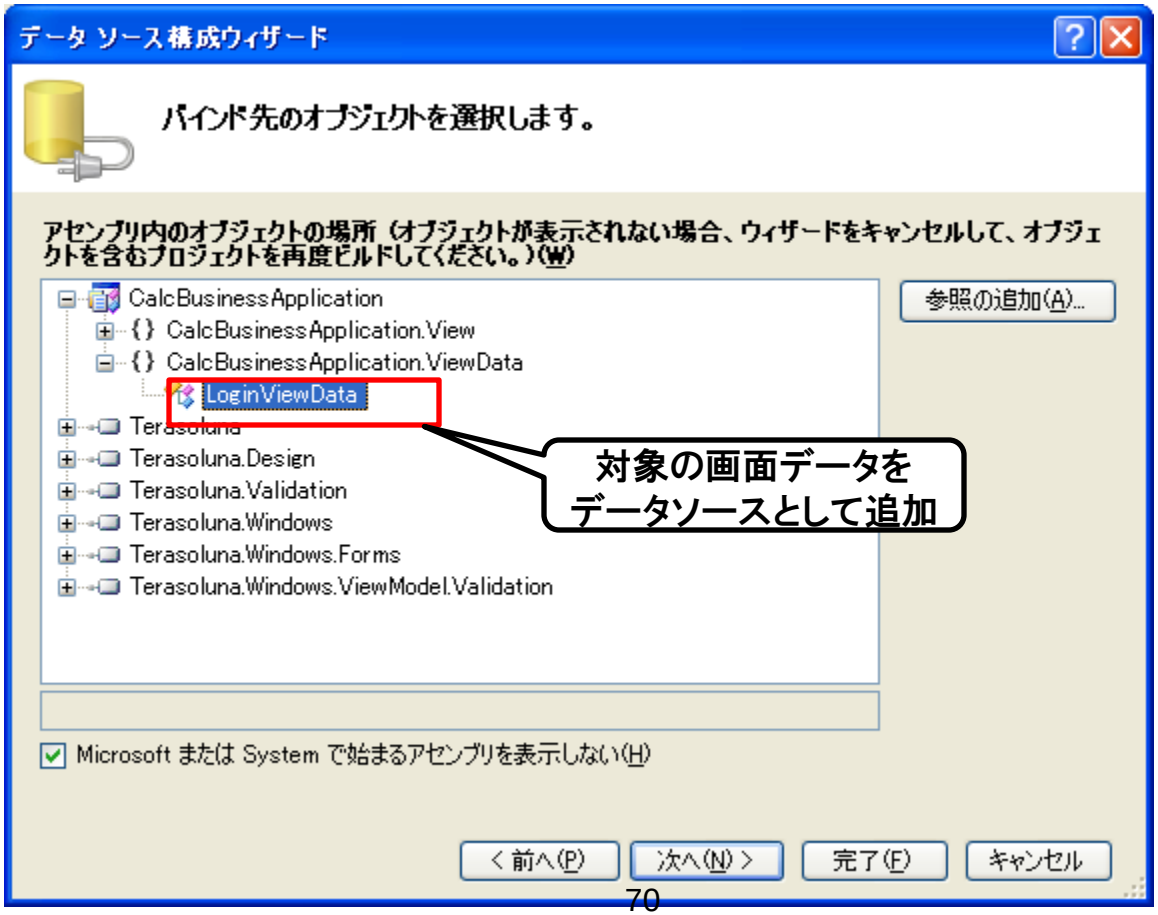
データソース構成ウィザード

- データソースの種類として「オブジェクト」を選択します



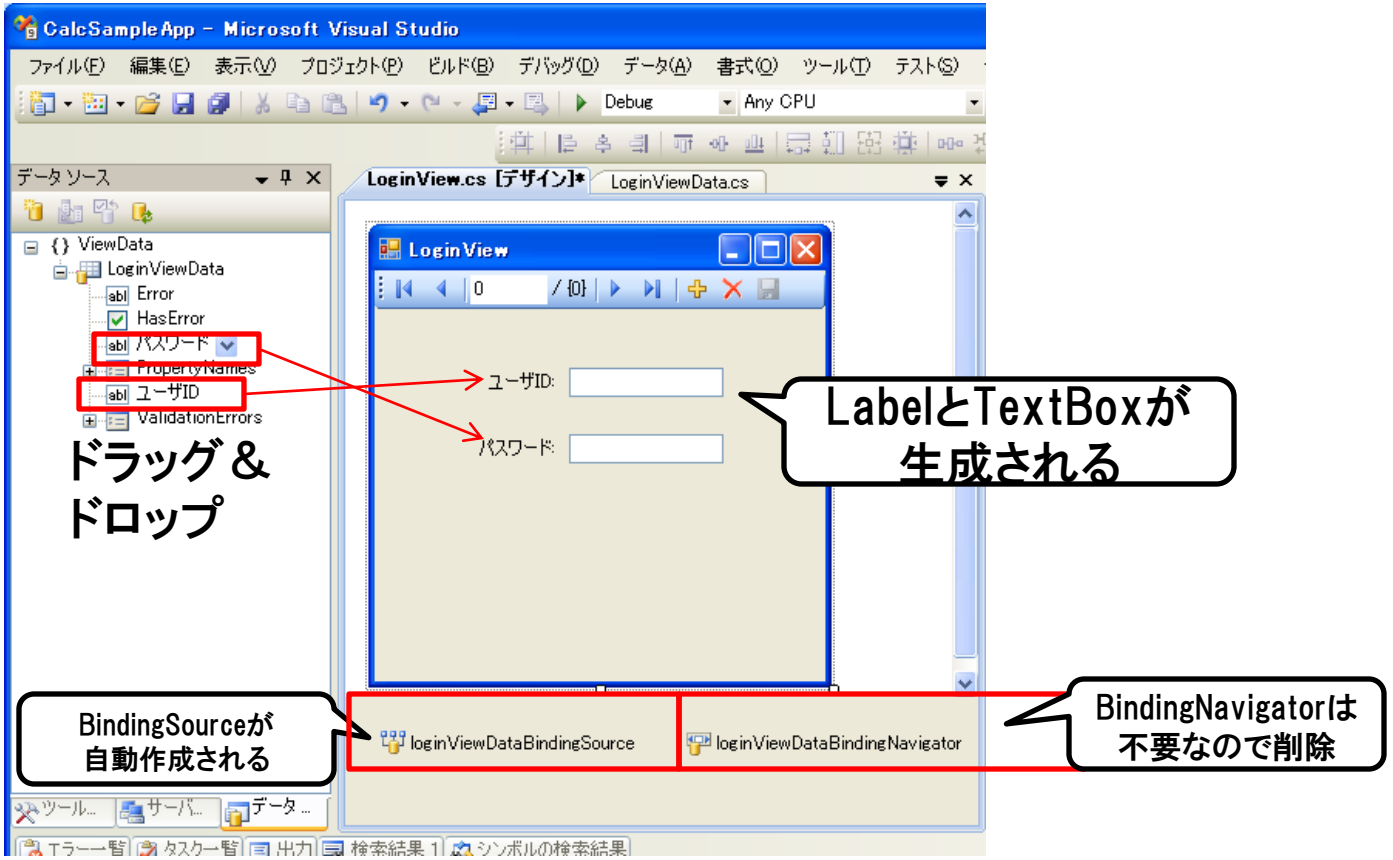
データソース構成ウィザード

- 「LoginViewData」クラスを選択します



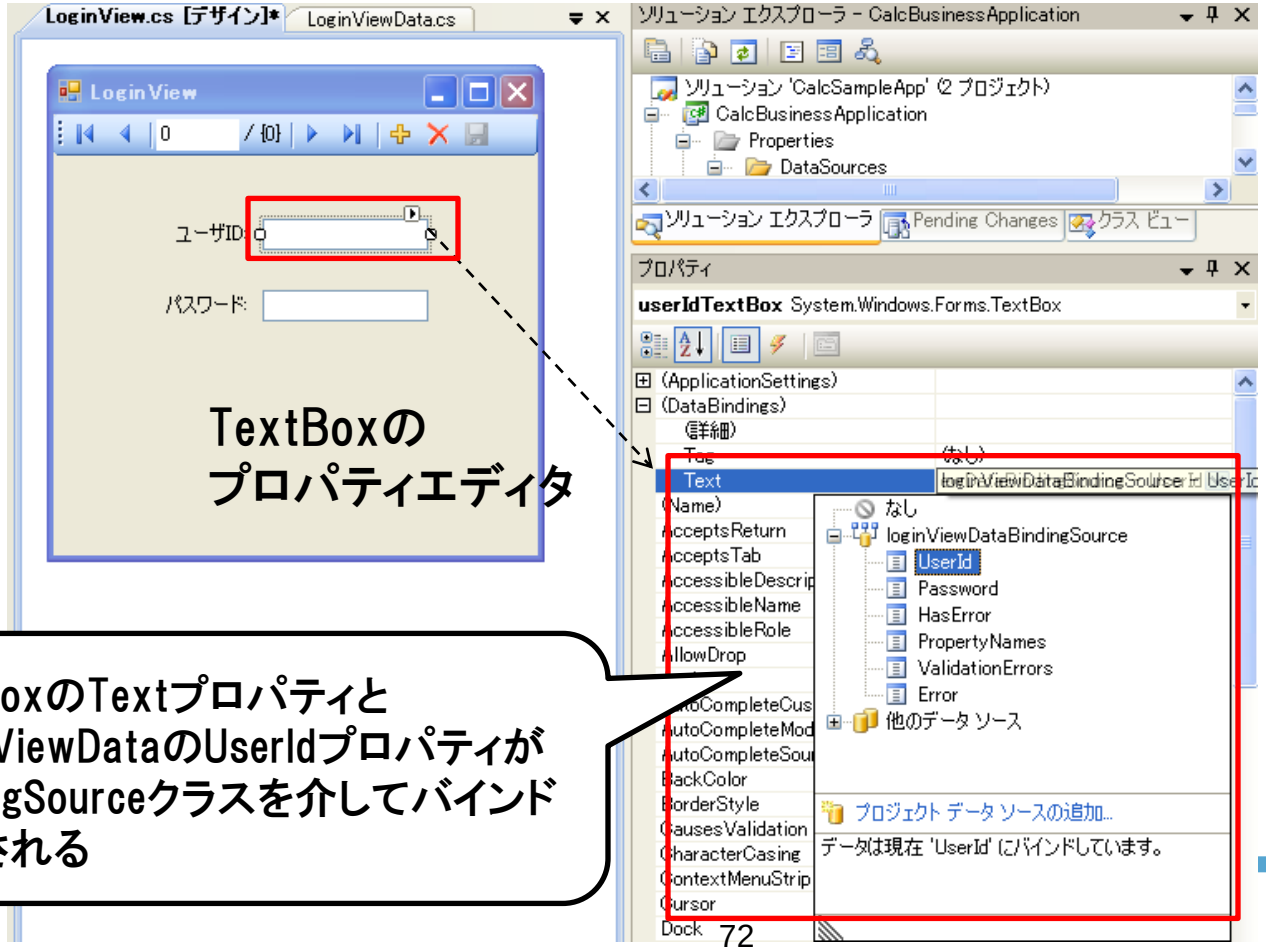
コントロールの貼り付け

- データソースウィンドウより、画面データの各項目をドラッグ＆ドロップして貼り付けます



ドラッグ＆ドロップ後の状態

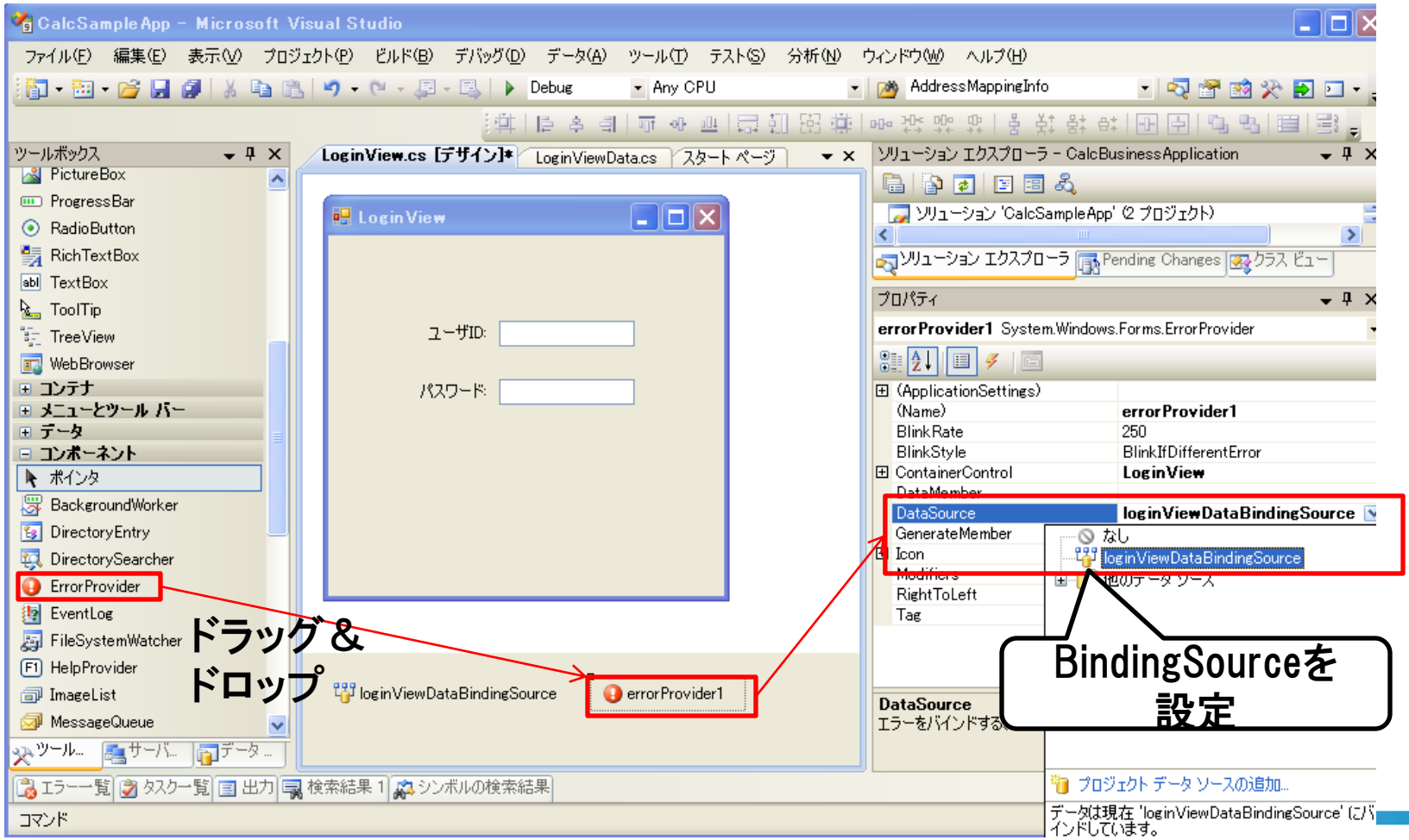
- UIコントロールと画面データのプロパティが双方向バインドされているのが確認できます



TextBoxのTextプロパティと
LoginViewDataのUserIdプロパティが
BindingSourceクラスを介してバインド
設定される

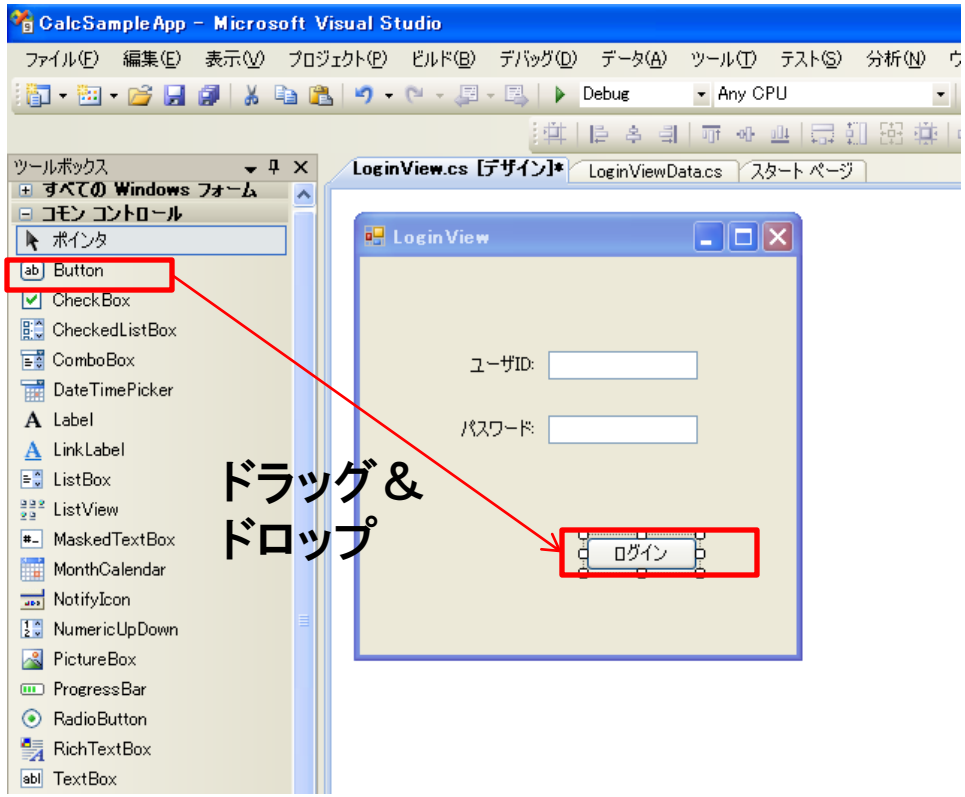
ErrorProviderの貼り付け

- ErrorProviderを張り付けDataSourceプロパティに自動作成されたBindingSource(loginViewDataBindingSource)を設定します



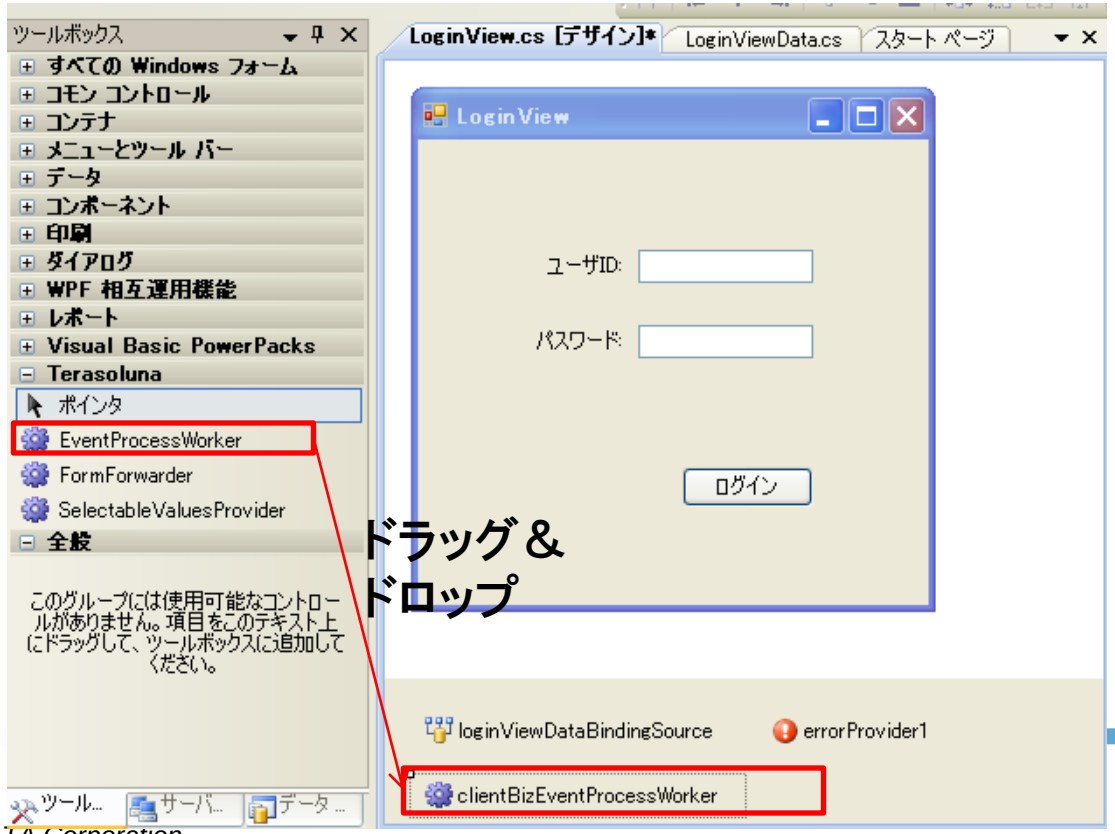
ボタンの貼り付け

- ボタンを貼り付けプロパティを変更します
  - nameを「loginButton」
  - Textプロパティを「ログイン」



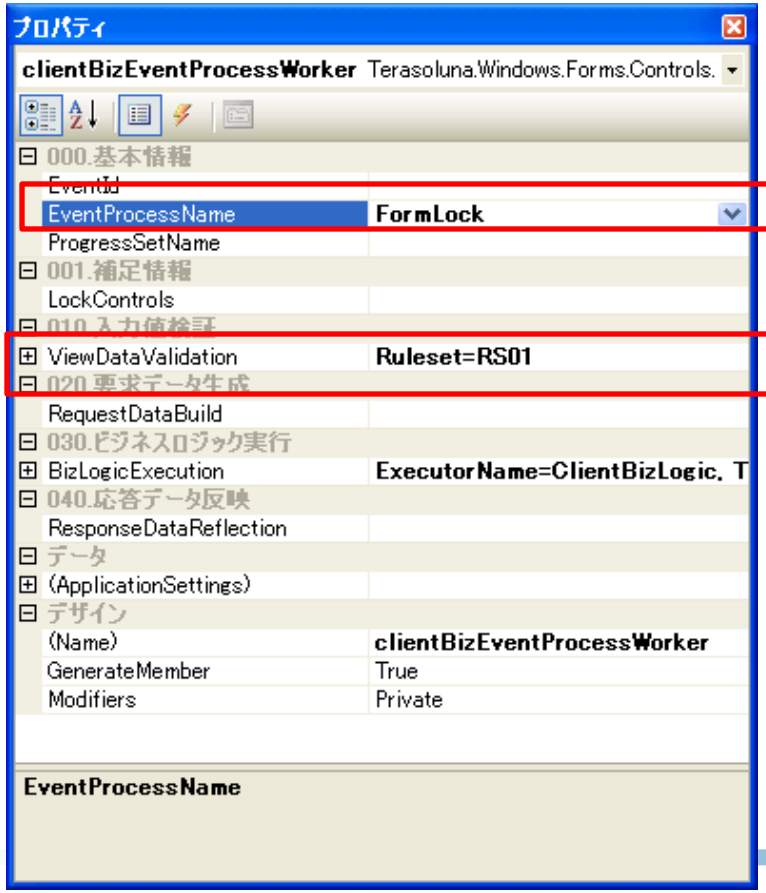
EventProcessWorkerの貼り付け

- ツールボックスよりEventProcessWorkerを張り付けます
  - EventProcessWorkerを使用するには、前述の手順でツールボックスにTERASOLUNA提供部品を追加しておく必要があります
  - nameを「clientBizEventProcessWorker」に変更します



EventProcessWorkerの設定

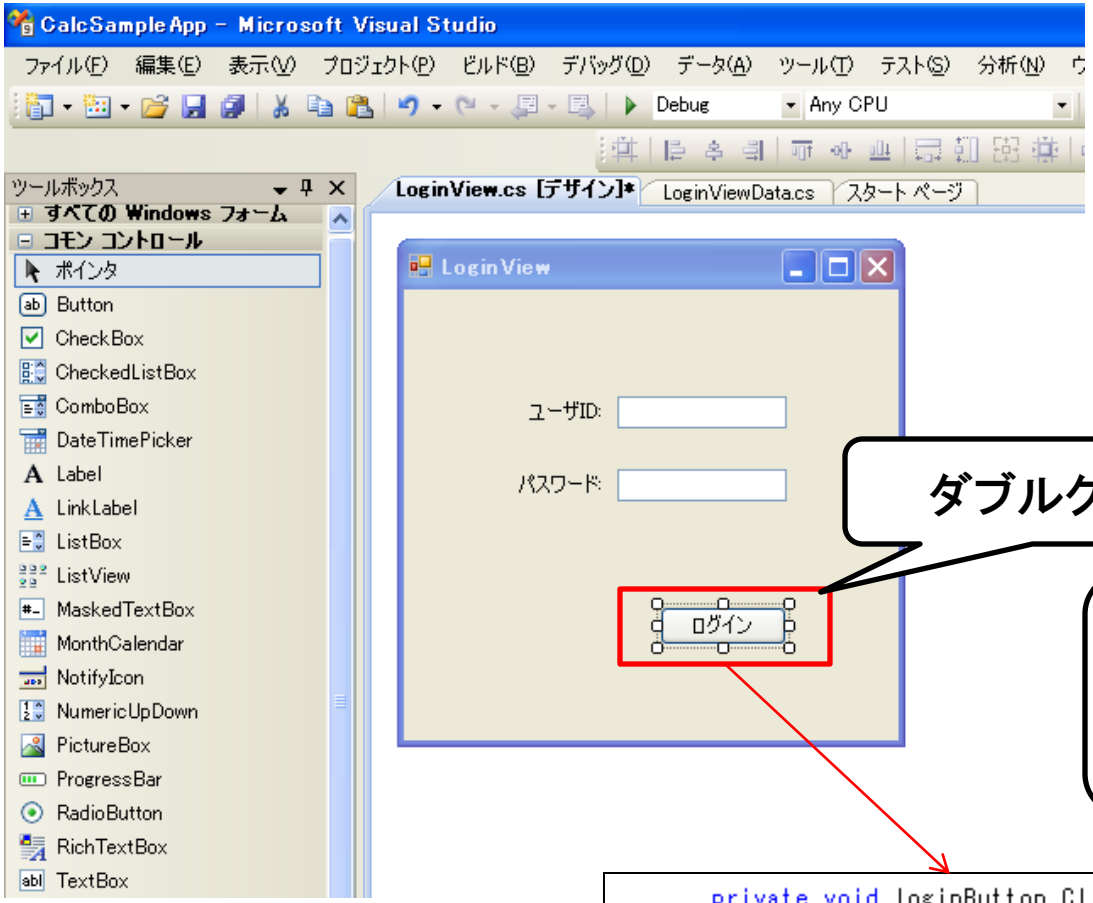
- EventProcessWorker(clientBizEventProcessWorker)のプロパティを設定します



“FormLock”は、イベント処理実行時に画面全体をEnable = falseにすることでロックします

イベント処理実行時にルールセット名=“RS01”で画面データの入力値検証をします

イベントハンドラの実装



ダブルクリック

ソースコードが表示され
クリックイベントの
イベントハンドラメソッドが生
成される

```
private void loginButton_Click(object sender, EventArgs e)
{

}
```

イベントハンドラの実装

- ボタンクリックイベントのメソッドを以下のように実装します
  - ボタンクリック時にイベント処理が同期実行されます

```
[ScreenId("LoginView")]
public partial class LoginView : Form
{
    ....
    private void loginButton_Click(object sender, EventArgs e)
    {
        clientBizEventProcessWorker.RunWorker();
    }
}
```

ログイン画面の作成

- 画面クラスのひな形コードのコメントを解除します

```
// ViewDataクラスを作成後、正しい名前空間を指定しコメントを解除してください
using CalcBusinessApplication.ViewData;

namespace CalcBusinessApplication.View
{
   // ScreenIdが決定後、指定して下さい。
    [ScreenId("LoginView")]
    public partial class LoginView : Form
    {
        // ViewDataクラスを作成後、コメントを解除してください
        public LoginViewData ViewData { get; set; }

        public LoginView()
        {
            InitializeComponent();
            // ViewDataクラスを作成後、コメントを解除して下さい。
            ViewData = ValidatableViewDataManager.CreateViewData<LoginViewData>();
        }
        private void LoginView_Load(object sender, EventArgs e)
        {
            // バインディングソース名が確定したら、修正後コメントを解除して下さい。
            loginViewDataBindingSource.DataSource = ViewData;
        }
}
```

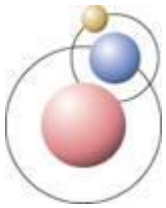
ひな型は大文字の「L」になっているので小文字（[l]）にします

TERASOLUNA

ログイン画面の実装

- AP動作確認時の画面入力操作を省略するため、Loadイベントのメソッドに画面初期表示処理を追加します
  - Loadイベント等で、画面データ(ViewData)のプロパティに値を設定することで、画面起動時にその値で初期表示されます。

```
private void LoginView_Load(object sender, EventArgs e)
{
    // バインディングソース名が確定したら、修正後コメントアウトして下さい。
    loginViewDataBindingSource.DataSource = ViewData;
    // 初期表示
    ViewData.UserId = "terasoluna";
    ViewData.Password = "password";
}
```



初期画面クラスの設定

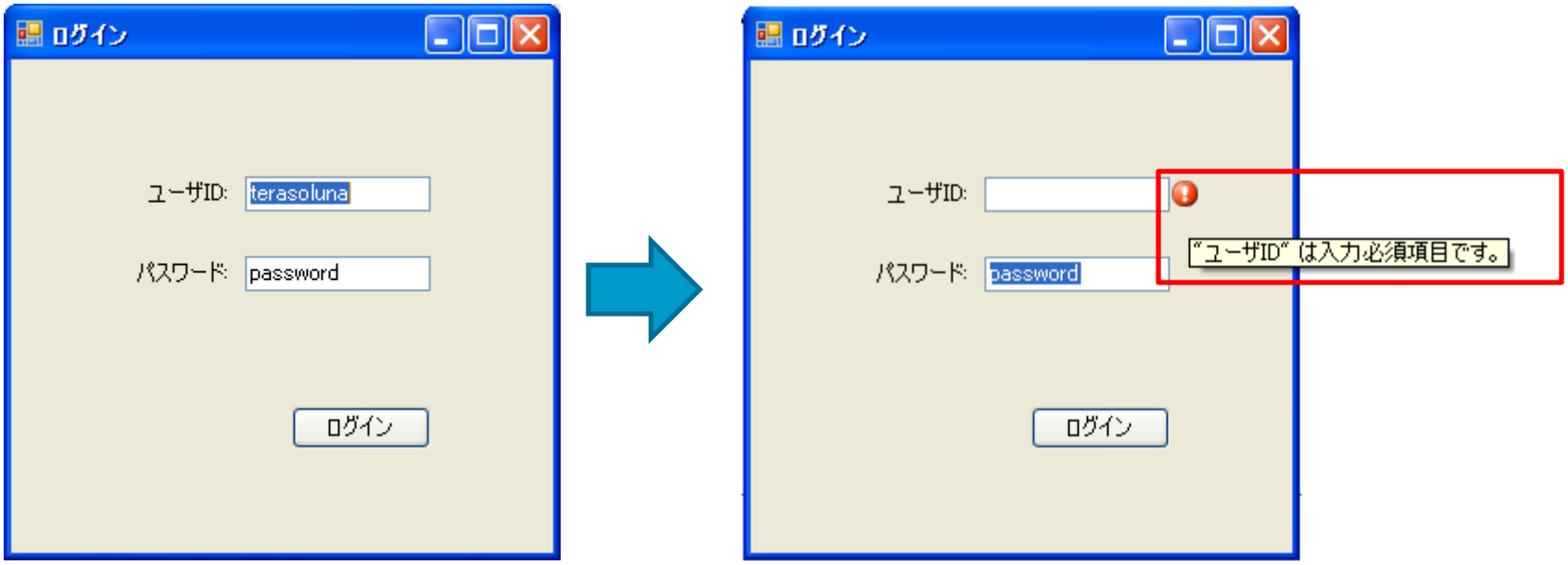
- 起動アプリケーションプロジェクト(CalcWinFormsApp)にあるTerasolunaBootstrap.configを修正します
  - /configuration/unity/containers/container/instances/add要素で初期画面の設定を行います
  - valueの値にログイン画面(LoginView)の完全修飾名を記述します

```
<!-- 起動画面の設定 -->
    <instances>
      <add name="StartFormType" type="Type"
          value="CalcBusinessApplication.View.LoginView, CalcBusinessApplication"
          typeConverter="TypeNameConverter"/>
    </instances>
```


TERASOLUNA

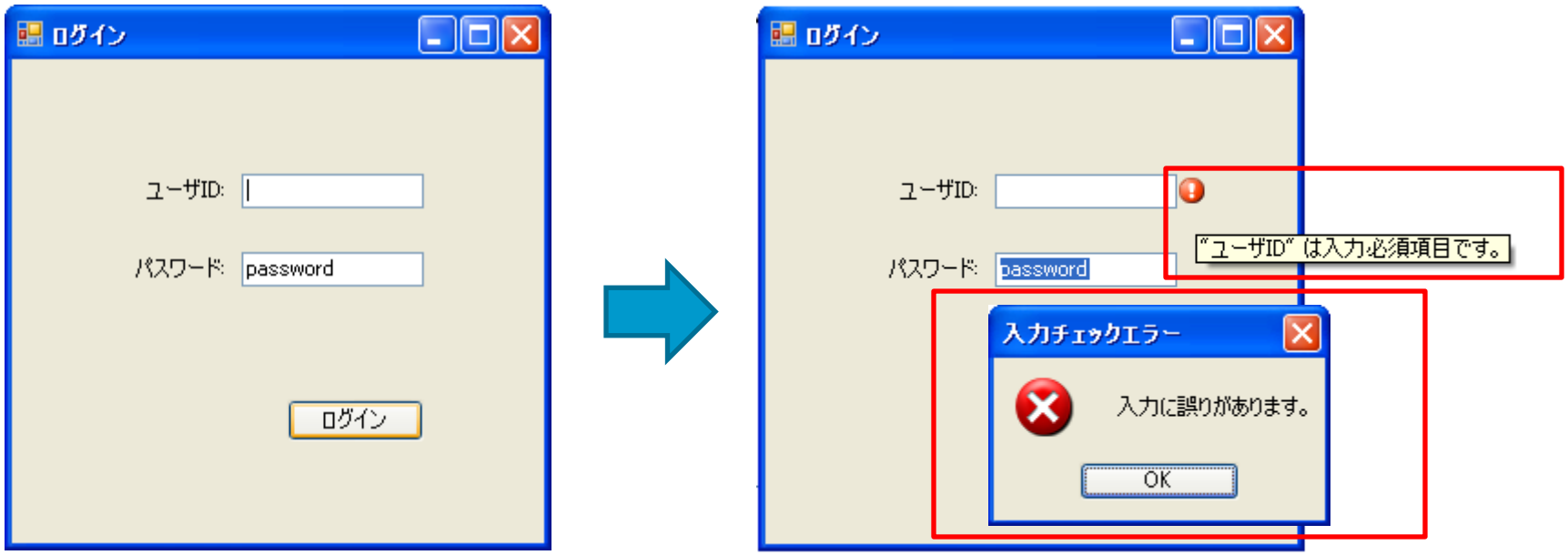
即値チェックの動作確認

- Visual Studioでデバッグ起動します
  - ログイン画面が起動します
- ユーザIDを消してロストフォーカスします
  - 直ちに入力値検証されエラー時には、対象のコントロールにErrorProviderが表示されます



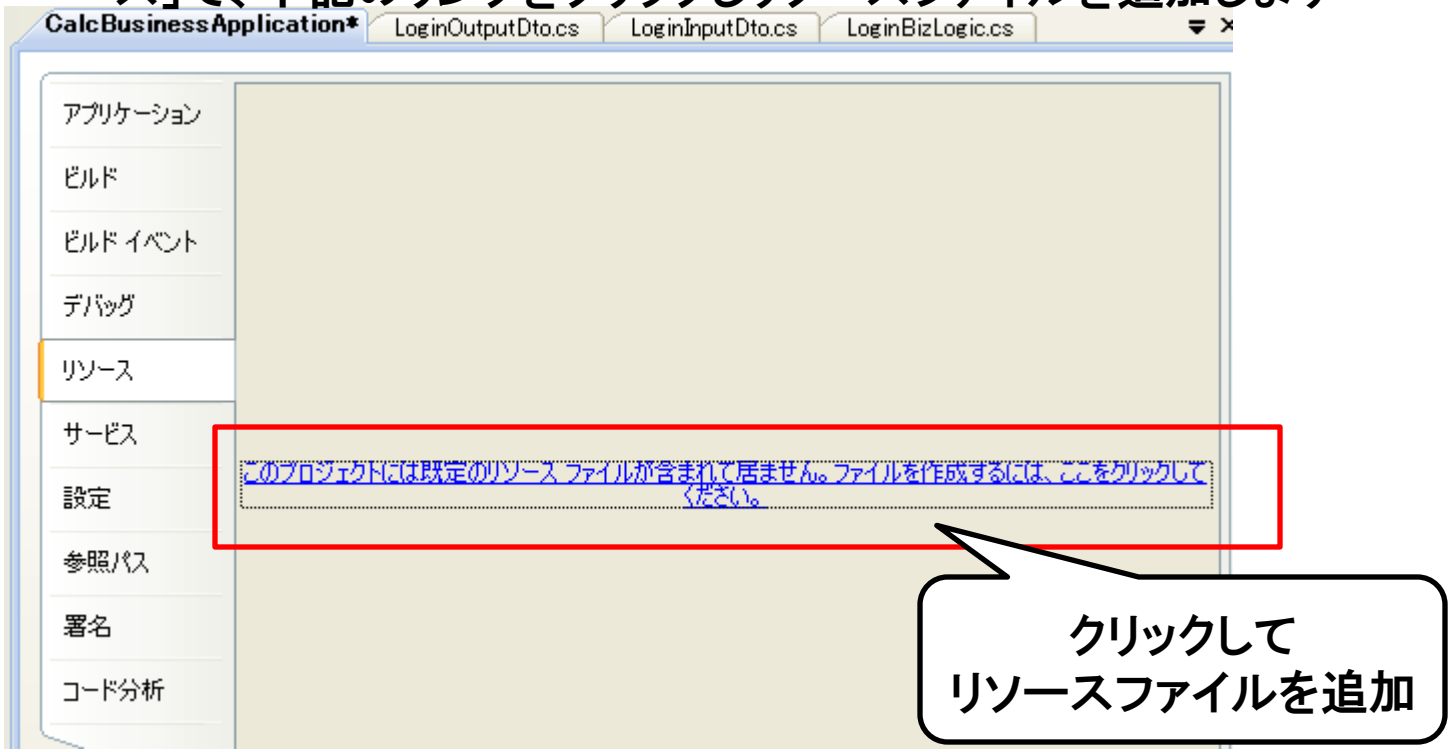
イベント処理時の入力値検証処理の動作確認

- ユーザIDを消したままログインボタンを押下します
  - エラーダイアログが表示されます
  - エラー対象のコントロールにErrorProviderが表示されます
    - イベント処理(EventProcessWoker)実行時に入力値検証が実施されたことが確認できます



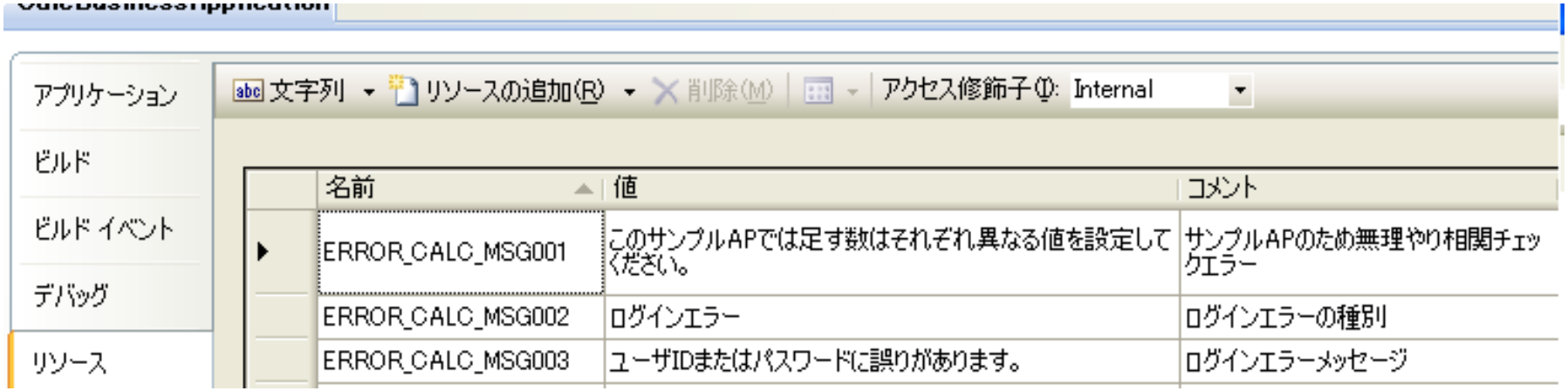
メッセージリソースの作成

- ユーザに表示するメッセージは、.NETが提供するリソースファイル(.resx)で作成します
  - 業務個別プロジェクト(CalcBusinessApplication)の「プロパティ」-「リソース」で、下記のリンクをクリックしリソースファイルを追加します



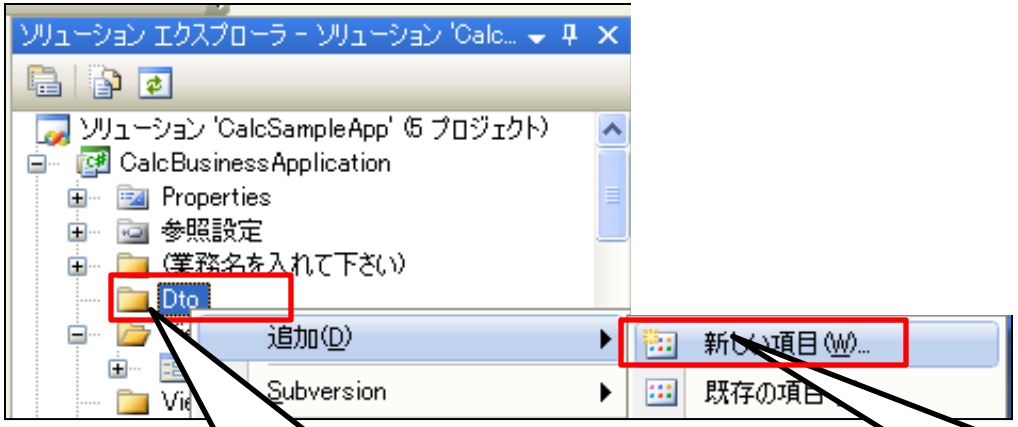
メッセージリソースの作成

- 以下のように、キーにメッセージを識別するID、値にメッセージを追加します



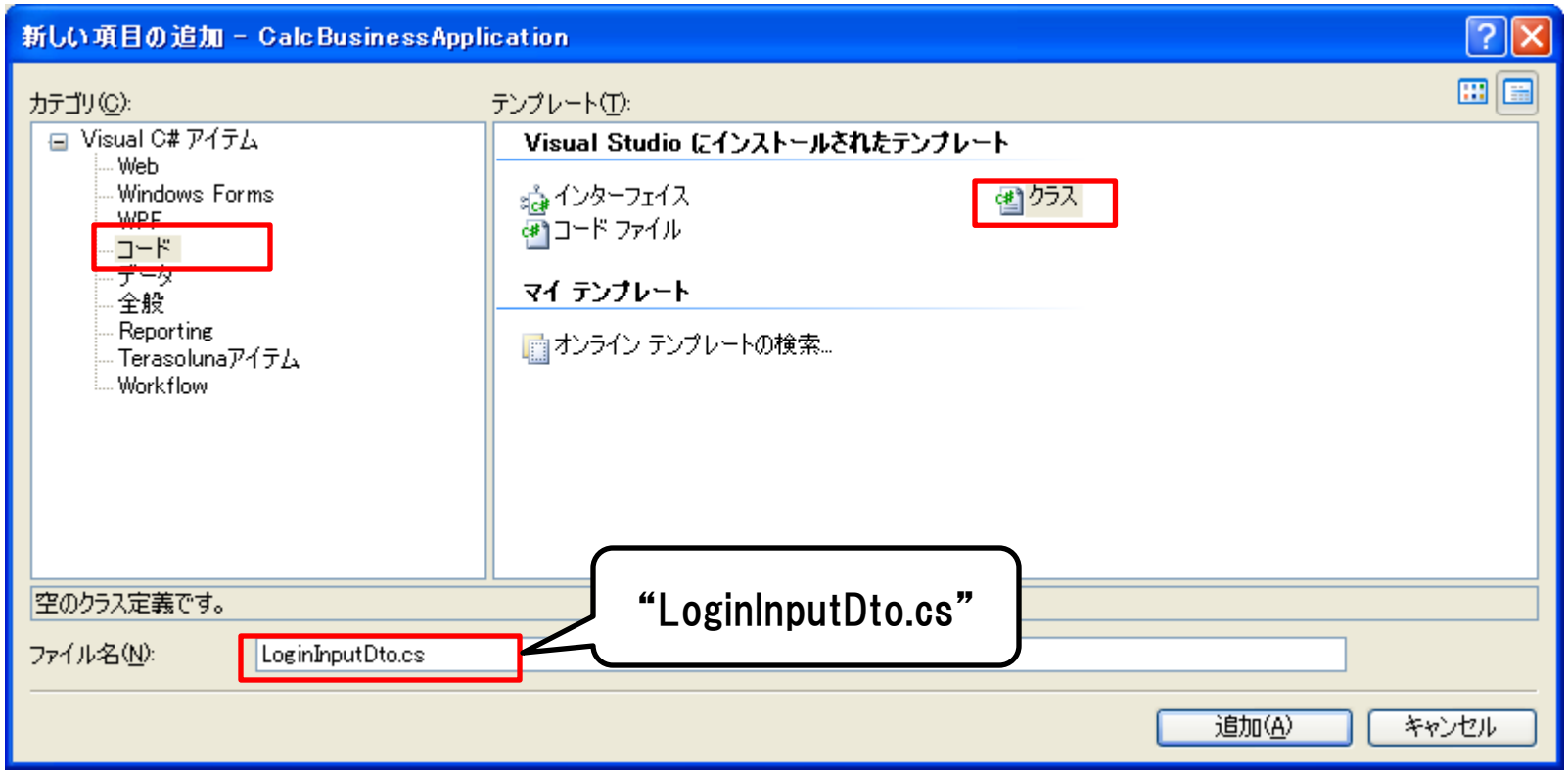
クライアントDTOの作成

- クライアントビジネスロジッククラスの入力DTOを作成します
  - Dtoフォルダを作成し、クラスを作成します



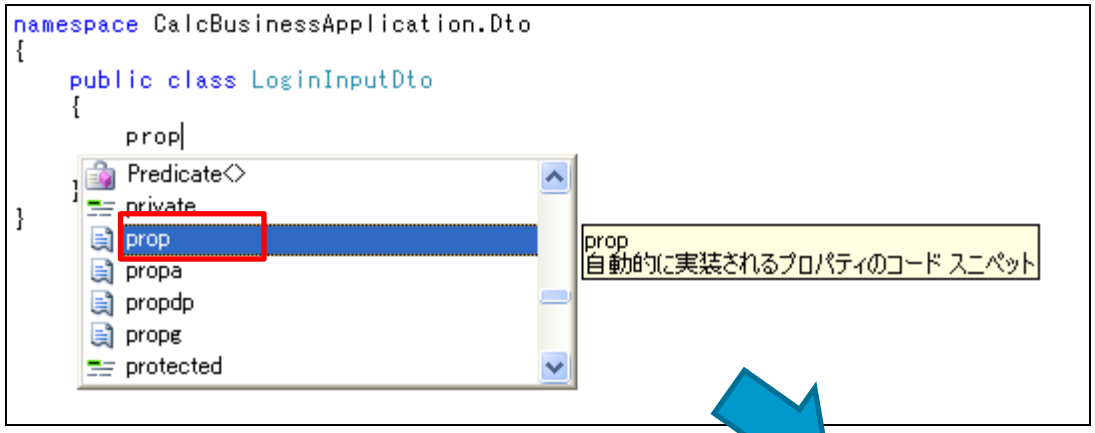
クライアントDTOの作成

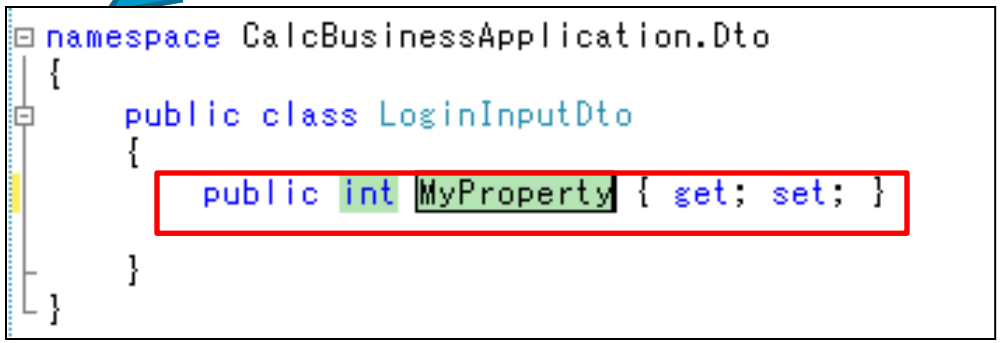
- 入力DTOクラスを作成します
  - Visual Studioの「クラス」テンプレートで作成します



クライアントDTOの作成

- DTOのプロパティを作成します
  - .NETのスニペット「prop」を使用しひな型を作成します





クライアントDTOの作成

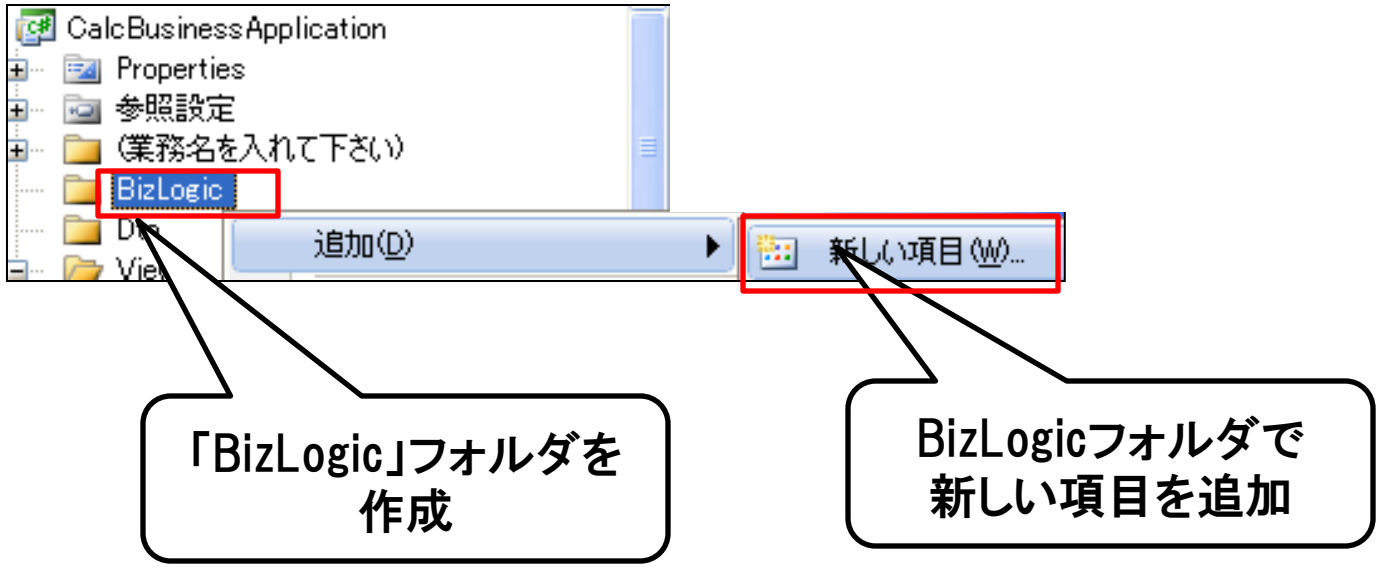
- DTOに以下のようにプロパティを実装します
  - 画面データのプロパティ名と一致させるようにします
    - TERASOLUNAフレームワークは、CoCにより設定の記述量を削減し生産性を向上する工夫をしており、クラスの階層構造とプロパティ名が一致すると画面データに格納された値を入力DTOに自動的にコピーするようにするためです(「データコピー機能」)

```
namespace CalcBusinessApplication.Dto
{
        public class LoginInputDto
        {
            public string UserId { get; set; }

            public string Password { get; set; }
        }
}
```

画面データクラス(LoginViewData)のプロパティ名を一致させる

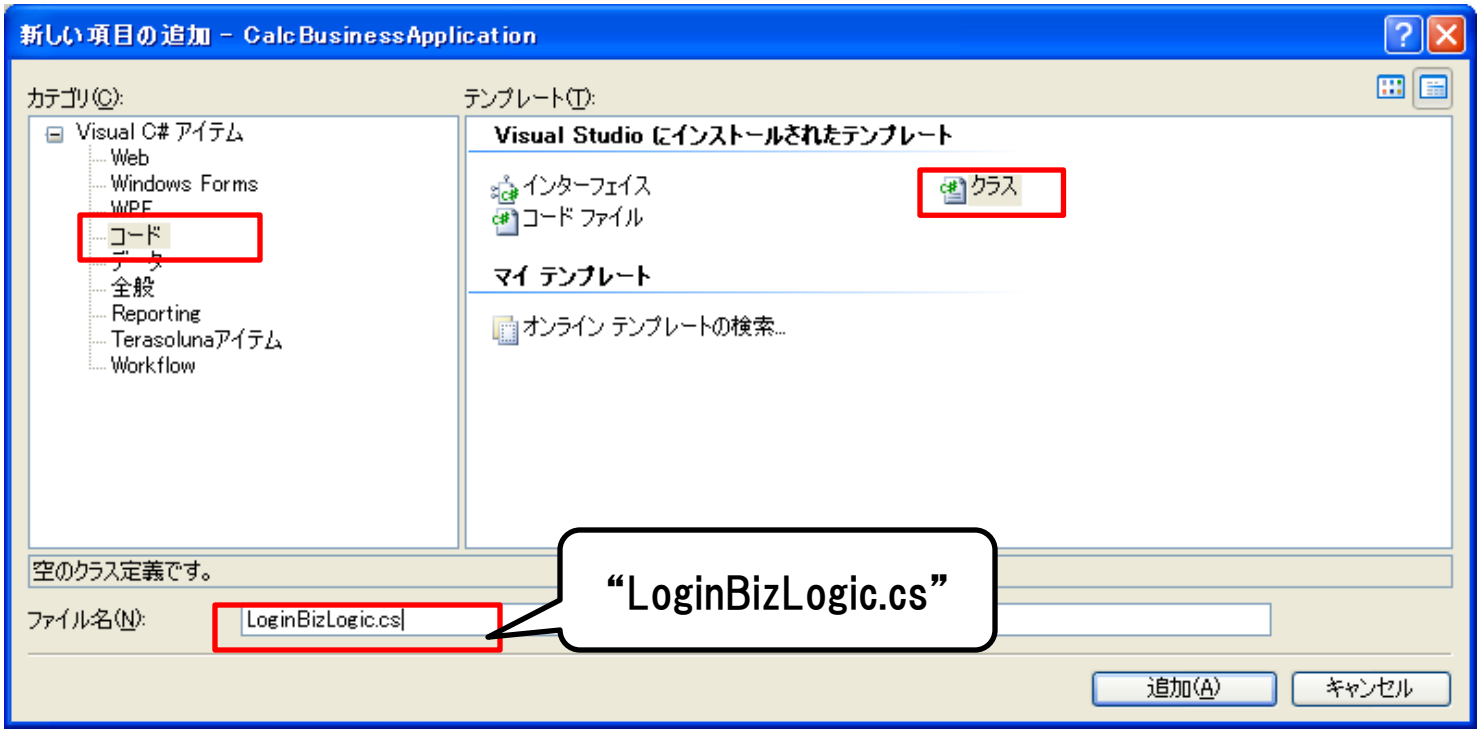
クライアントビジネスロジックの作成

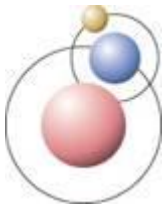
- 簡単なユーザ認証を実施するビジネスロジックを作成します
  - BizLogicフォルダを作成し、クラスを作成します



クライアントビジネスロジックの作成

- ビジネスロジッククラスを作成します
  - Visual Studioの「クラス」テンプレートで作成します





クライアントビジネスロジックの作成

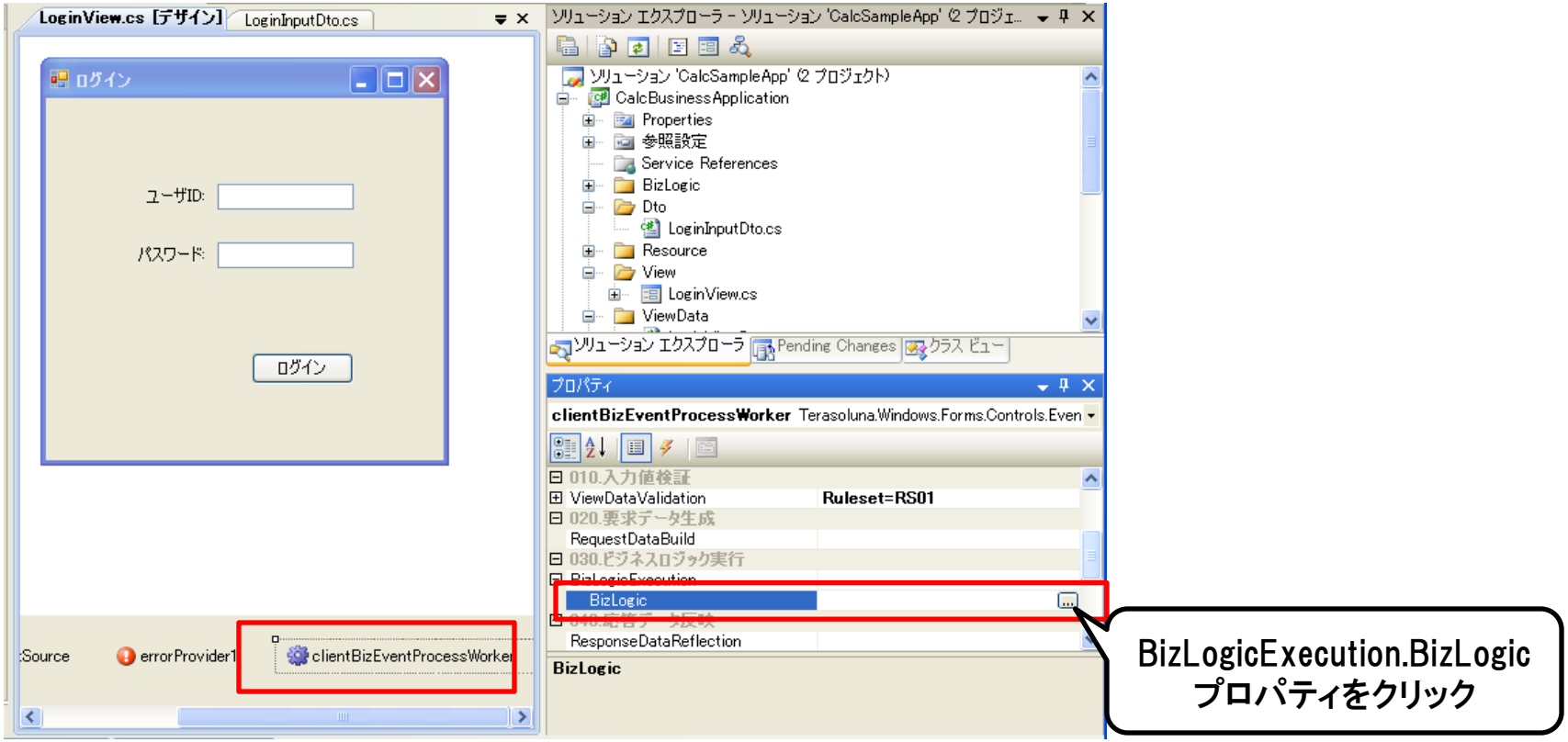
- LoginBizLogicクラスにLoginメソッドを以下のように実装します
  - ユーザIDまたはパスワードが一致しない場合は、業務エラーとしてBizLogicExceptionをスローするようにします

```
namespace CalcBusinessApplication.BizLogic
{
    public class LoginBizLogic
    {
        public void Login(LoginInputDto input)
        {
            ///簡単のため、ユーザID=“terasoluna”、パスワード=“passsword”であることをチェックする
            if (!"terasoluna".Equals(input.UserId, StringComparison.Ordinal)
                || !"password".Equals(input.Password, StringComparison.Ordinal))
            {
                ///業務エラーは、errorTypeをBizLogicExceptionErrorType.BizLogicFailureにして
                ///BizLogicExceptionをスローする
                throw new BizLogicException(
                    BizLogicExceptionErrorType.BizLogicFailure,
                    Resources.ERROR_CALC_MSG002,
                    new List<ErrorInfo>()
                    {
                        new ErrorInfo("ERROR_CALC_MSG003", null, Resources.ERROR_CALC_MSG003, null)
                    });
            }
        }
    }
}
```


TERASOLUNA

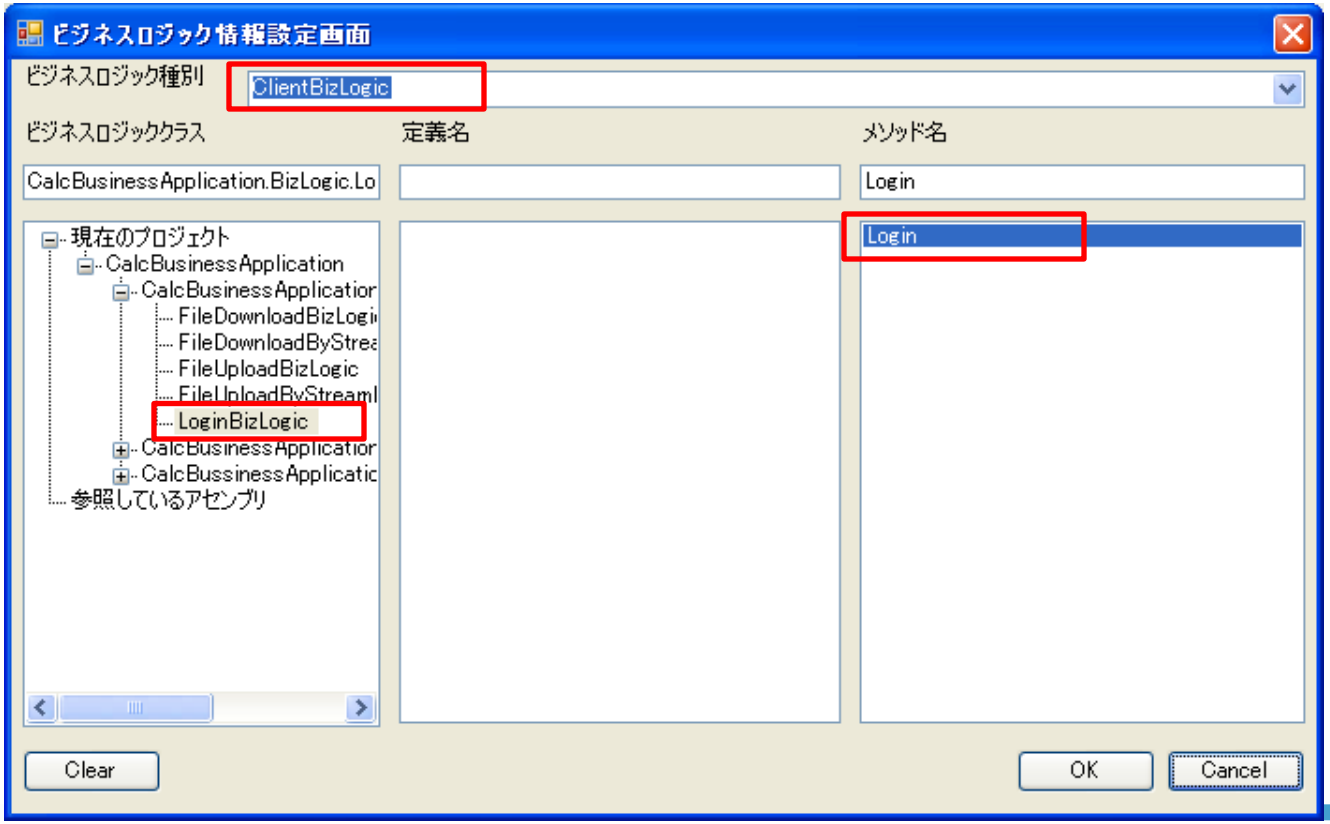
EventProcessWorkerの設定

- ビルドを実施し成功したら、EventProcessWorkerでビジネスロジック実行の設定をします
  - EventProcessWorker.BizLogicExecutionプロパティを設定します



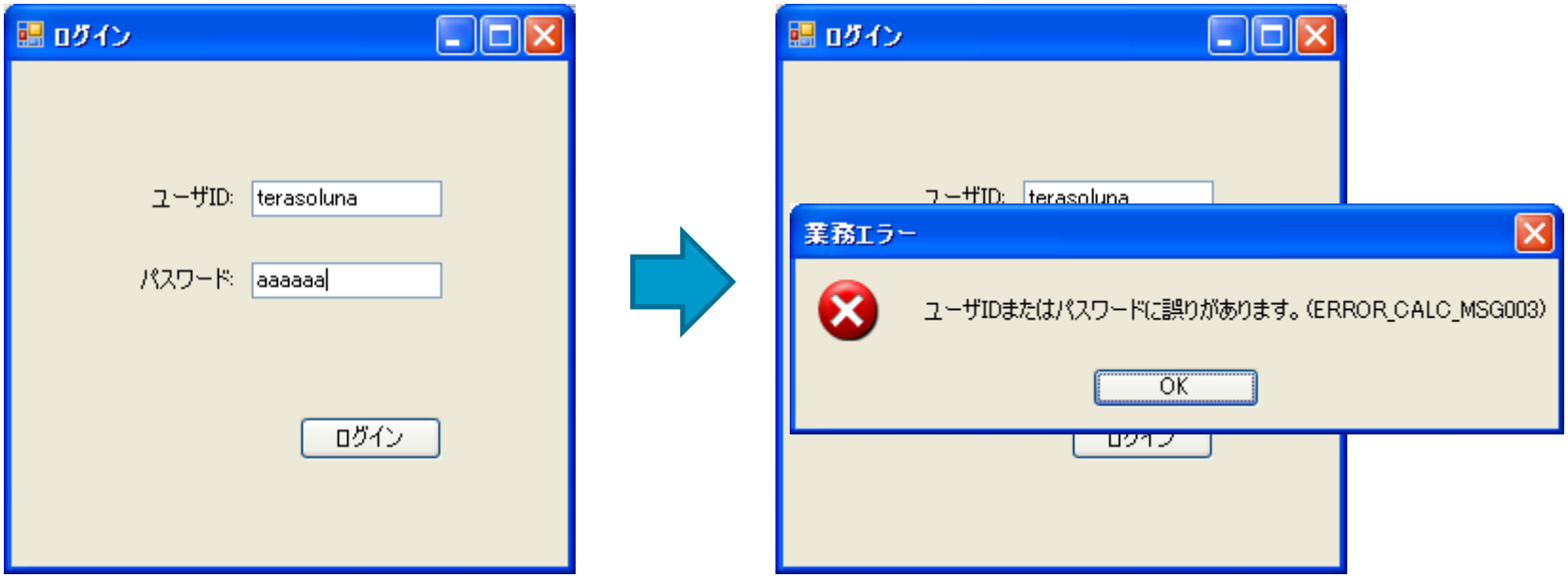
EventProcessWorkerの設定

- ビジネスロジック情報設定画面が表示されます
  - イベント処理時に実行するビジネスロジックを設定します
- ビジネスロジック種別を「ClientBizLogic」にして「LogicBizLogic」の「Login」メソッドを設定します
  - 「 ClientBizLogic 」はユーザ定義クラス、「WcfProxy」はWCFクライアントを実行する場合に使用します



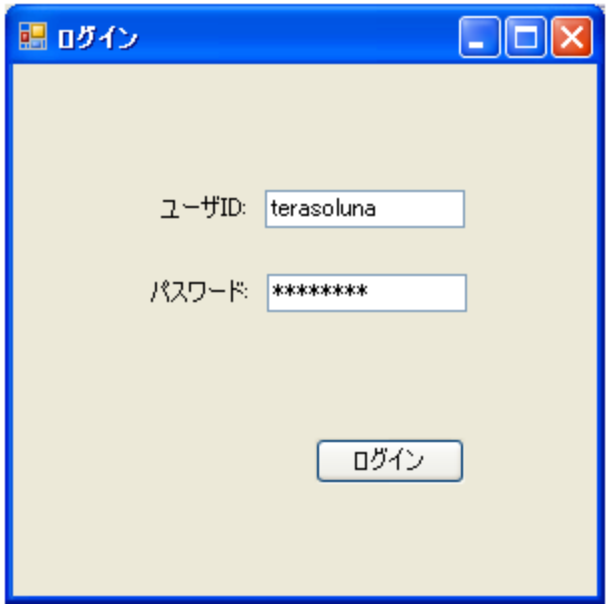
ビジネスロジック処理の動作確認

- Visual Studioでデバッグ起動します
  - ログイン画面が起動します
- パスワードを”aaaaa”など違う値に変更し、ログインボタンをクリックします
  - ビジネスロジックを実行し、パスワードが一致しなかったため、業務エラーがダイアログ表示されたことが確認できます



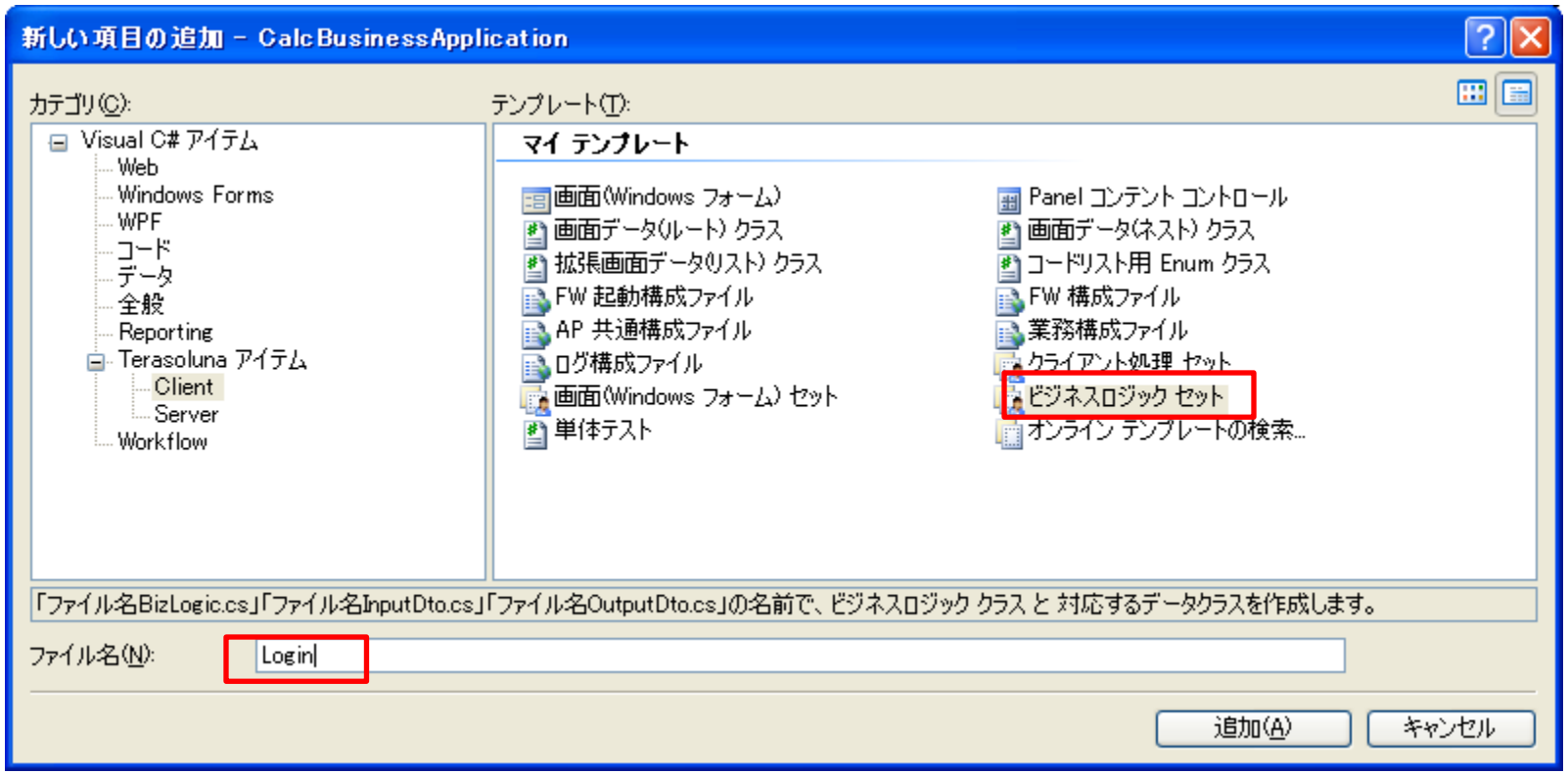
パスワードテキストボックスの設定

- ひと通り動作確認できたのでパスワードのテキストボックスのPasswordCharに「*」を設定します
  - パスワードが「*」で表示されます



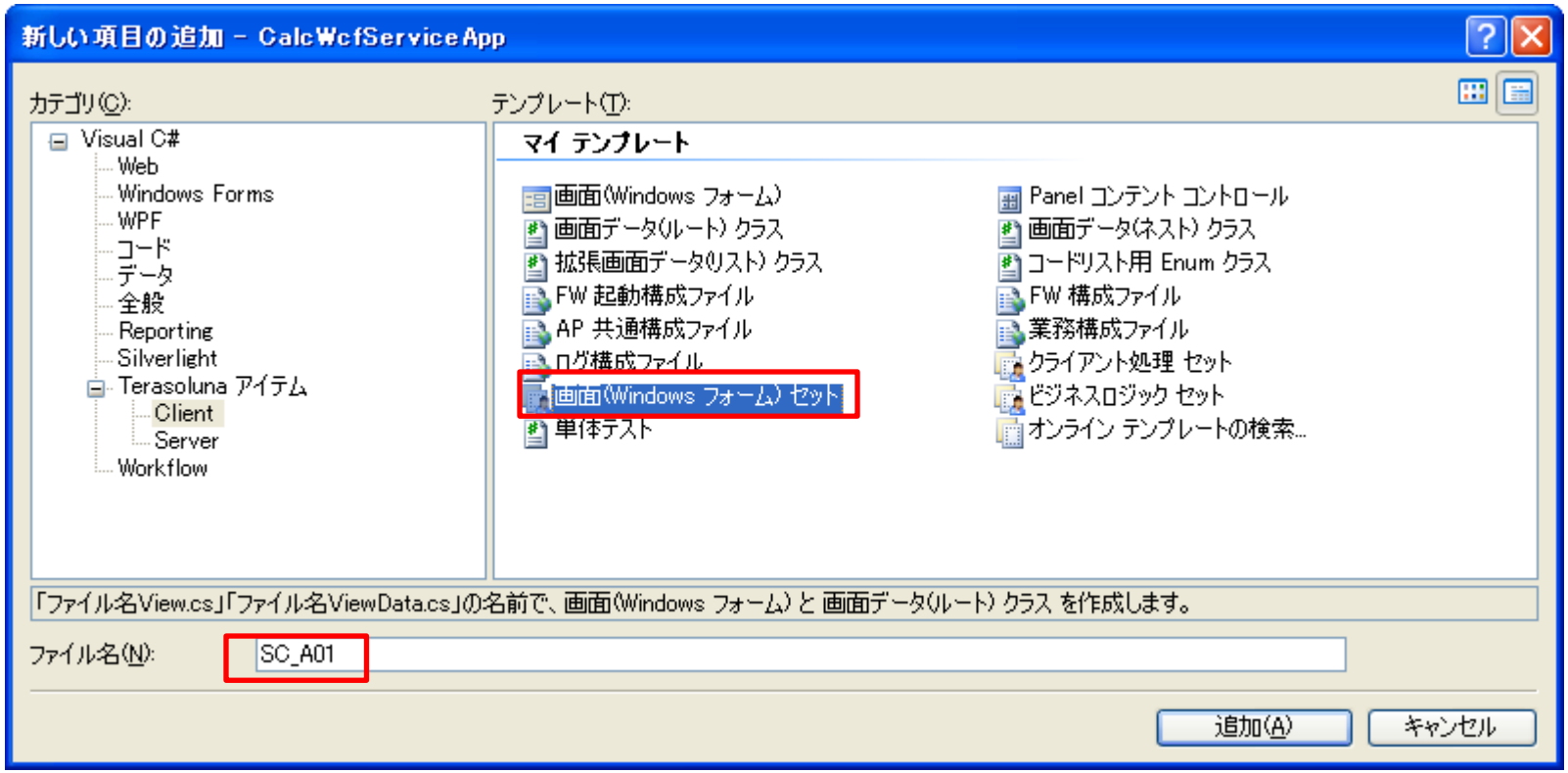
（参考）ビジネスロジックセット

- 一定の命名規約をもとにビジネスロジック、入出力DTOのセットを一度に生成できるカスタムテンプレート。



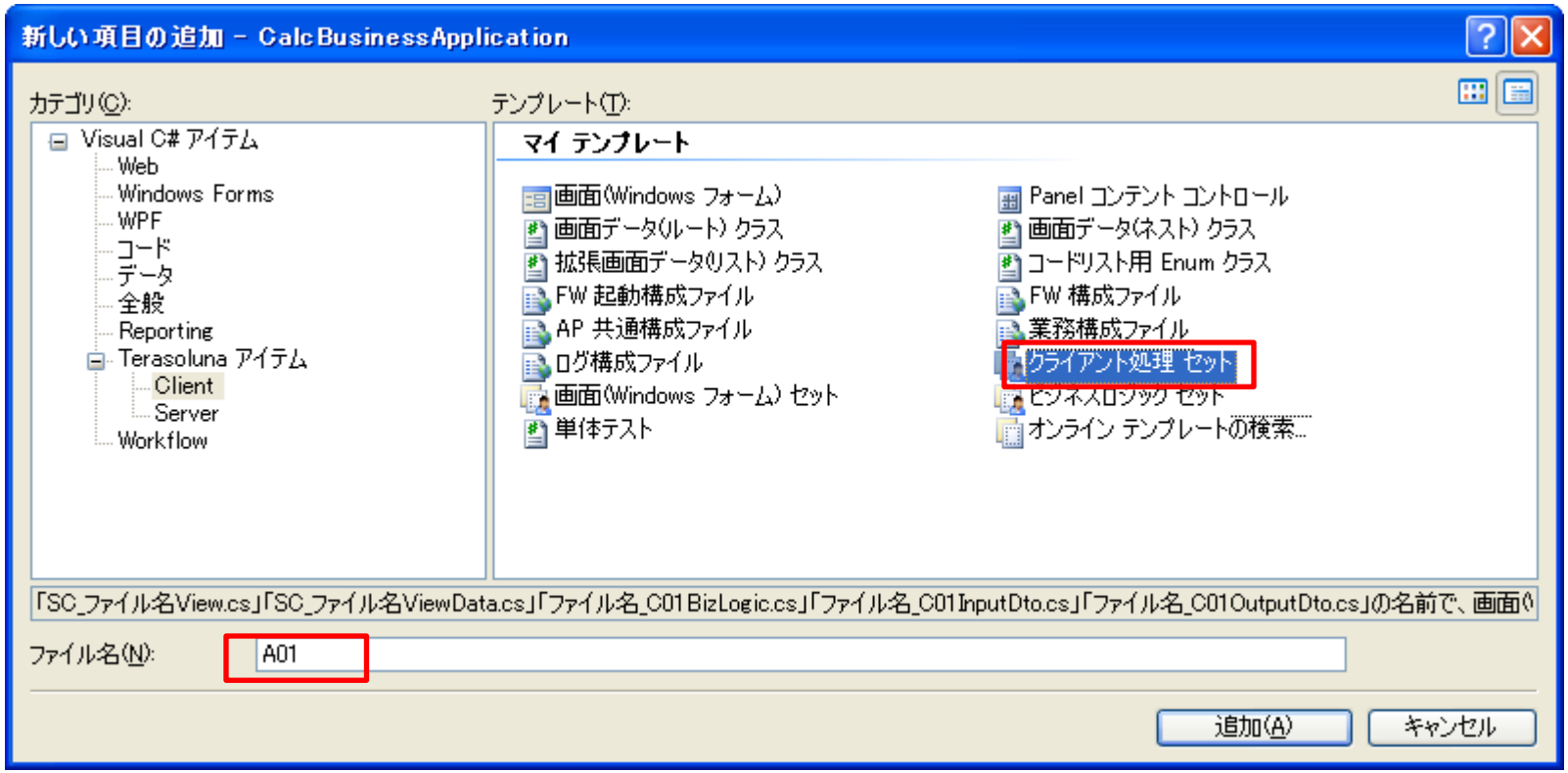
（参考）画面（Windowsフォーム）セット

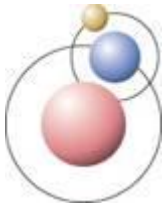
- 一定の命名規約をもとに画面、画面データのセットを一度に生成するカスタムテンプレート。通常の業務開発では、次頁の「クライアント処理セット」とこちらの併用を推奨します。



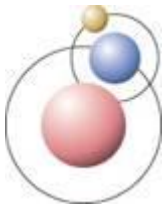
（参考）クライアント処理セット

- 一定の命名規約をもとに画面、画面データ、ビジネスロジック、入出力DTOのセットを一度に生成できるカスタムテンプレート。通常の業務開発ではこちらの利用を推奨します。



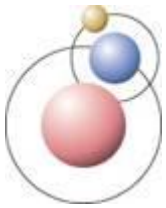


メニュー業務の作成
（.NETクライアントに閉じた処理）



メニュー画面

- ログイン画面と同様の手順でメニュー画面を作成します
  - 画面データクラス(MenuViewData.cs)の作成
    - TERASOLUNAのアイテムテンプレートで画面データ(ルート)を作成
      - バインドするプロパティを定義
  - 画面(MenuView.cs)の作成
    - TERASOLUNAのアイテムテンプレートで画面を作成
      - データソースウィンドウで、バインドする画面項目を画面にドラッグアンドドロップ
      - ボタンの作成
      - 画面データのコードを修正



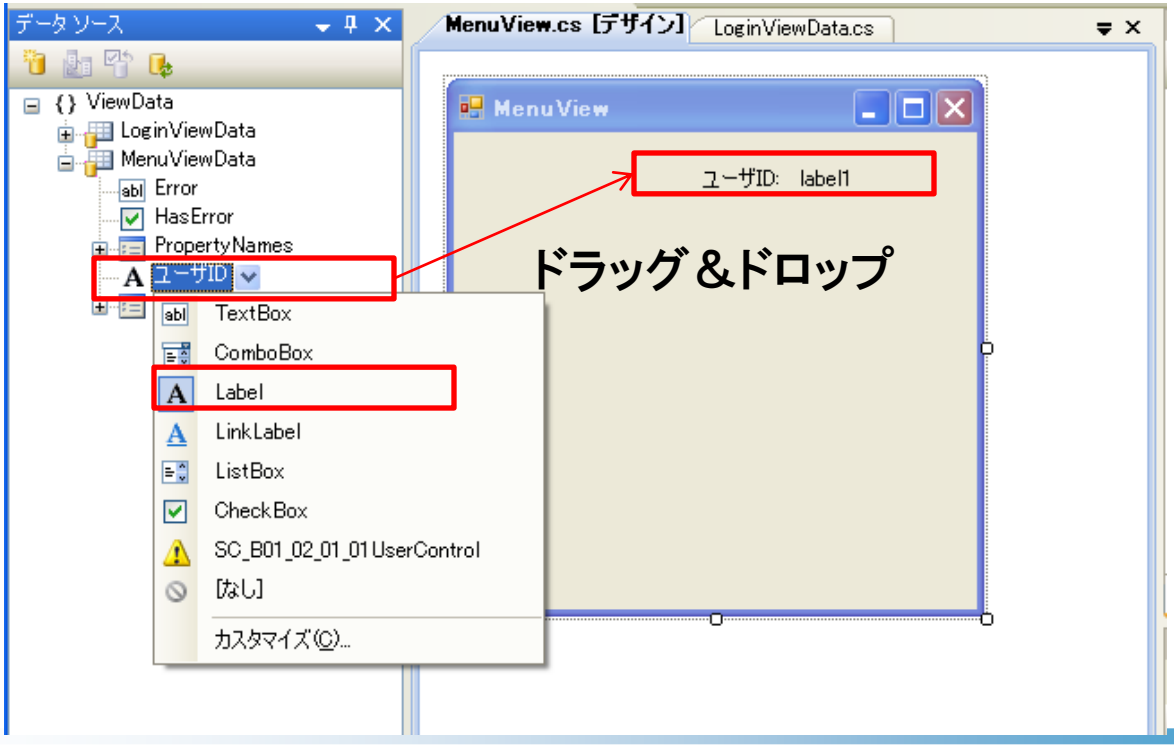
メニュー画面の画面データの作成

- UserIdプロパティを作成します
  - メニュー画面では入力チェック処理はないため、バリデータの属性の付与はありません
  - DefaultRulesetも不要なのでコメントアウトします

```
namespace CalcBusinessApplication.ViewData
{
    //[DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class MenuViewData : ValidatableRootViewData
    {
        [DisplayName("ユーザID")]
        public virtual string UserId { get; set; }
    }
}
```

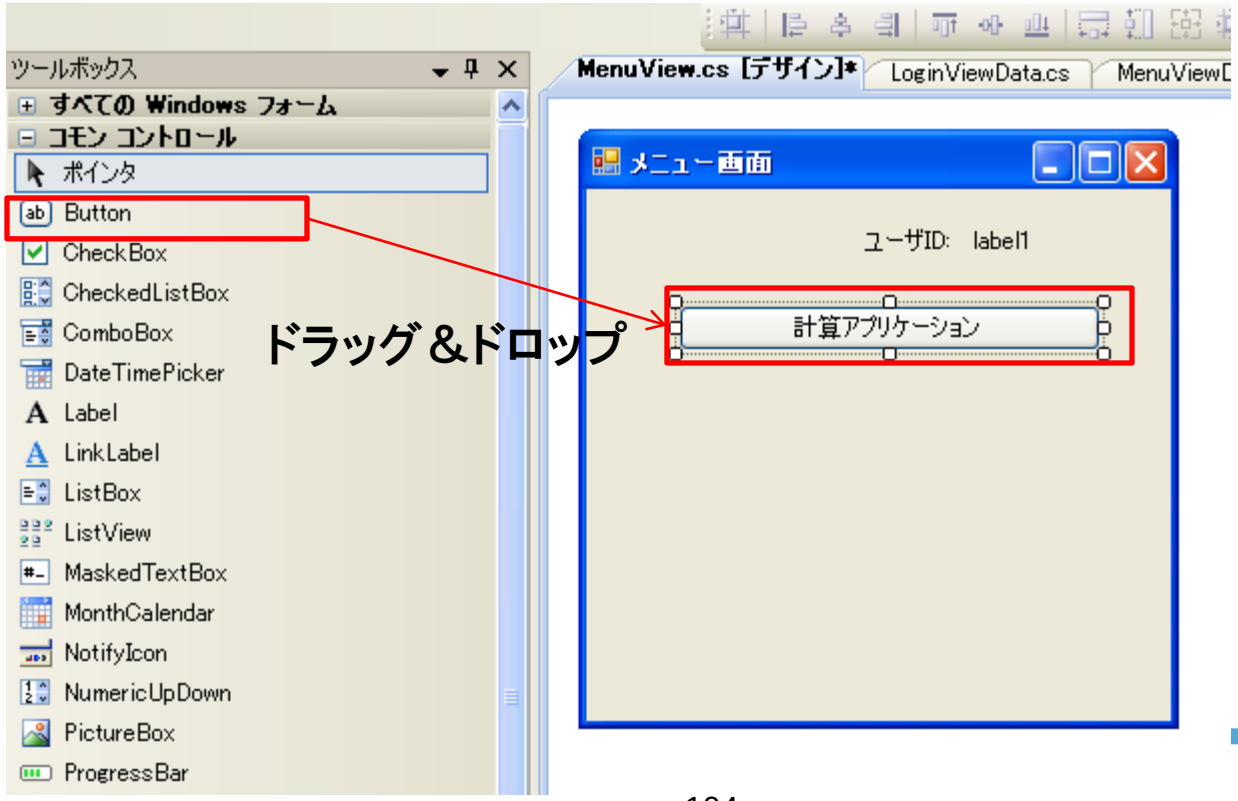
メニュー画面の作成

- MenuViewDataをデータソース構成ウィザードで追加して、データソース画面から、ユーザIDを画面へドラッグ＆ドロップします
  - 部品を「Label」に変更してから貼り付けます
  - BindingNavigaterは、不要なので削除します



メニュー画面の作成

- 画面のStartPositionプロパティを「CenterScreen」にします
- 計算画面へ遷移するためのボタンを作成します
  - nameを「forwardButton」、Textプロパティを「計算アプリケーション」とします



メニュー画面の作成

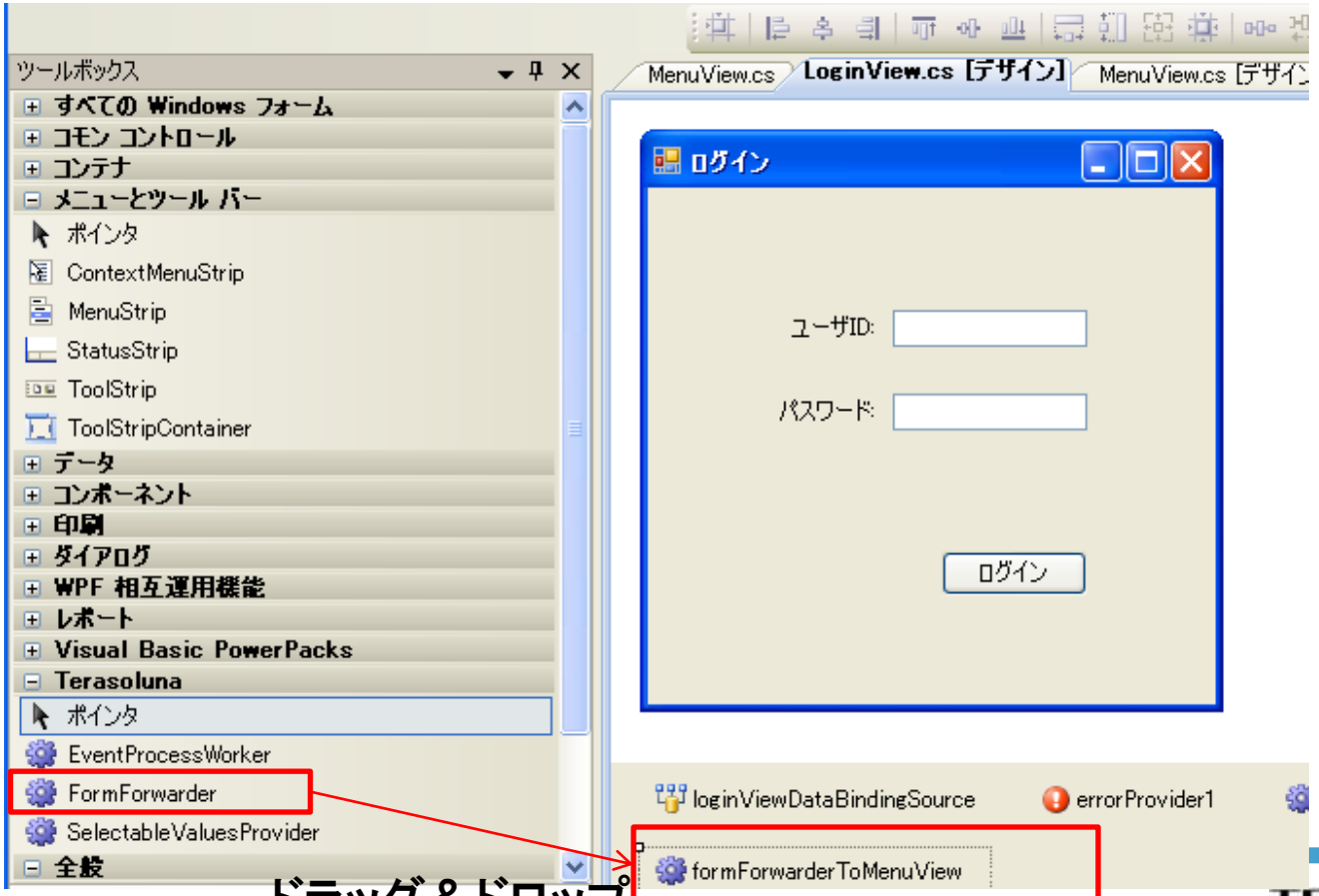
- テンプレートが生成したコードのコメントを外します

```
using CalcBusinessApplication.ViewData;

namespace CalcBusinessApplication.View
{
    // ScreenIdが決定後、指定して下さい。
    [ScreenId("MenuView")]
    public partial class MenuView : Form
    {
        // ViewDataクラスを作成後、コメント化を解除して下さい。
        public MenuViewData ViewData { get; set; }

        public MenuView()
        {
            InitializeComponent();
            // ViewDataクラスを作成後、コメント化を解除して下さい。
            ViewData = ValidatableViewDataManager.CreateViewData<MenuViewData>();
        }

        private void MenuView_Load(object sender, EventArgs e)
        {
            // バインディングソース名が確定したら、修正後コメント化を解除して下さい。
            menuViewDataBindingSource.DataSource = ViewData;
        }
    }
}
```

ひな型は大文字の「M」になっているので小文字（[m」）にします

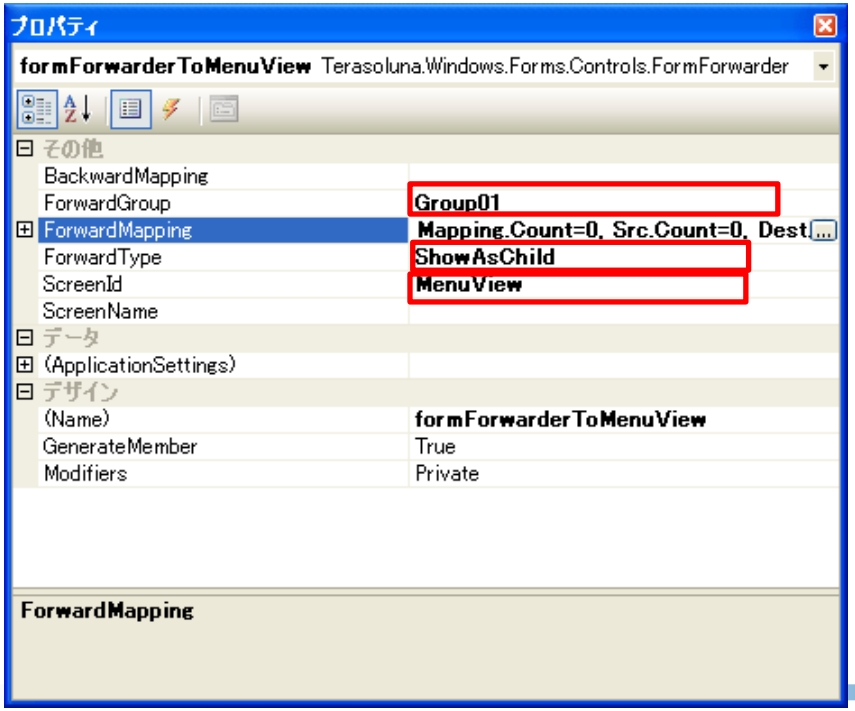
ログイン画面⇒メニュー画面への画面遷移

- ログイン画面へFormForwarderを貼り付けます
  - nameは「formFowarderToMenuView」にします



ログイン画面⇒メニュー画面への画面遷移

- FormForwarder(formFowarderToMenuView)のプロパティを設定します
  - FowardGroupプロパティに「Group01」を設定し、「スコープ」を作成します
    - 「ShowAndHide」による画面遷移には、スコープの作成が必須です
    - 同一のスコープ内では、全てのFormForwarderに同一の値を設定します
  - ForwardTypeプロパティは「ShowAsChild」にします
    - 「スコープ」開始する時は、ShowModelessまたはShowAsChildを指定します
  - ScreenIdプロパティは、「MenuView」を指定します。
    - 遷移先の画面ID(画面を一意に識別する文字列)を表します

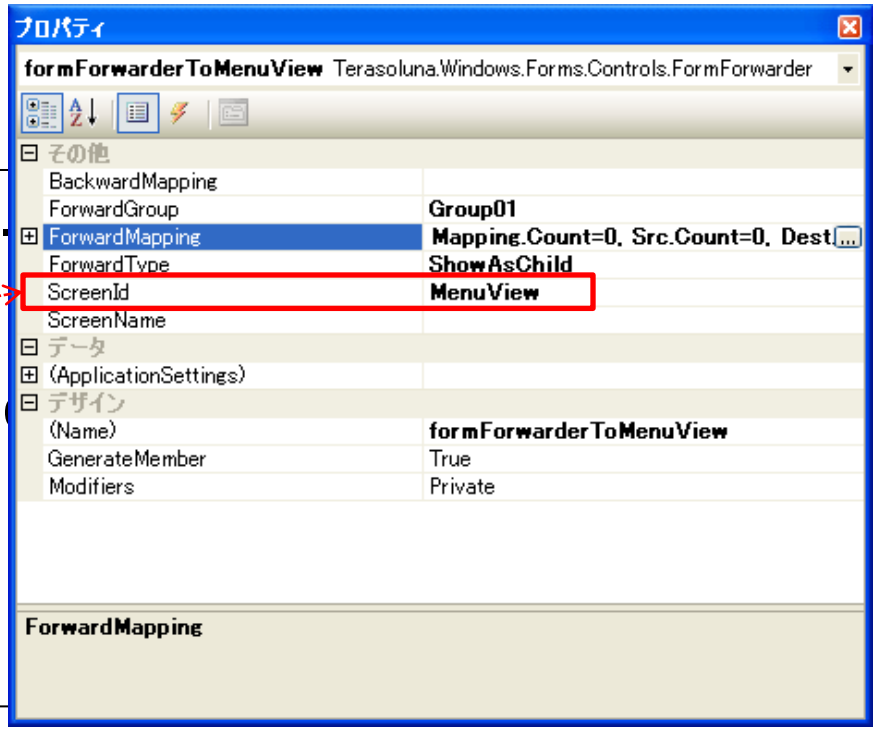


ログイン画面⇒メニュー画面への画面遷移

- 遷移先を特定するための「画面ID」は、各画面クラスのコード上にScreenId属性として定義されています
  - 画面クラスのテンプレートにより自動生成されるので、今回は特に編集の必要はありません

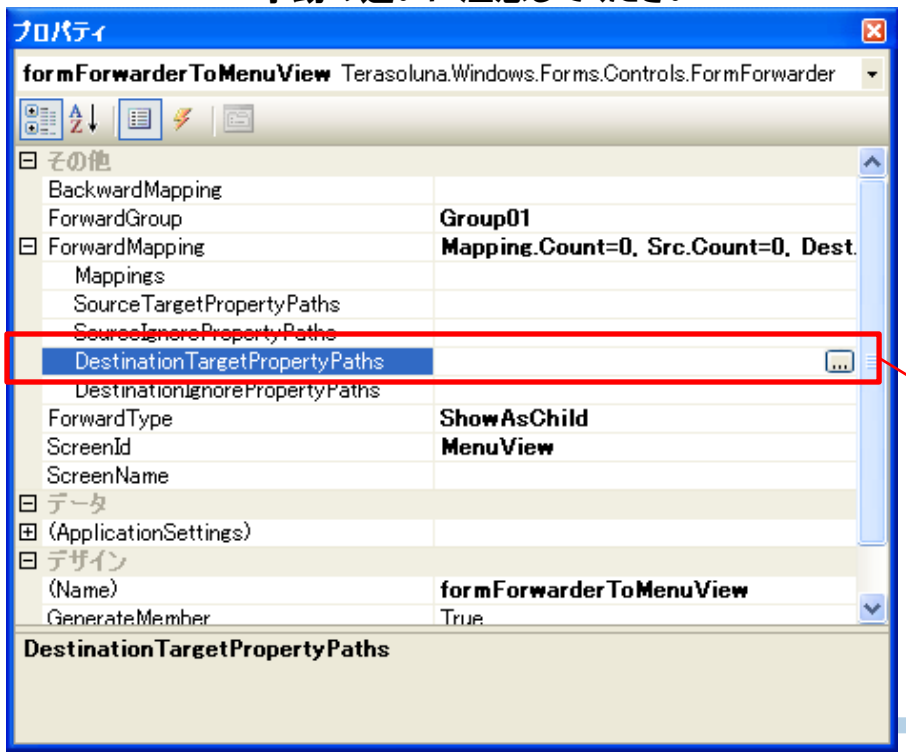
メニュー画面(MenuView.cs)のコード

```
namespace CalcBusinessApplication.
{
    [ScreenId("MenuView")]
    public partial class MenuView : F
    {
....
    }
}
```

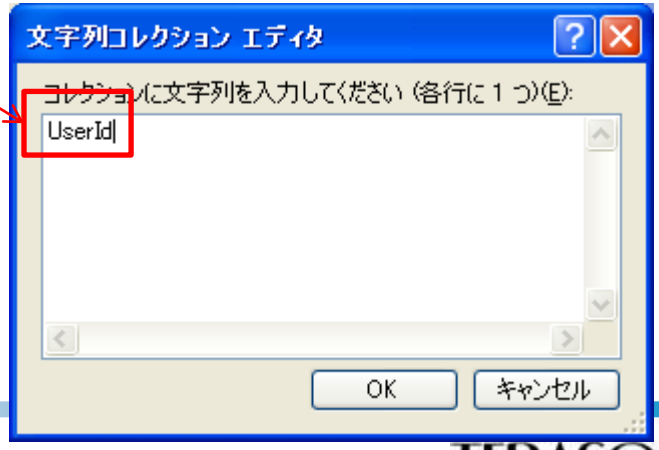


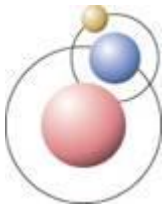
ログイン画面⇒メニュー画面への画面遷移

- ForwardMappingプロパティを設定します
  - 画面間での画面データコピー処理のマッピング設定が可能です
  - 「画面遷移機能」では、「DestinationTargetPropertyPaths」プロパティにコピー対象を明示した項目についてのみ、画面データ間のクラス構造とプロパティ名が一致すると自動的にコピーします。
    - 異なる画面では同じプロパティ名でも違う意味の入出力項目である可能性が高いため、コピー対象を明示するようにしています
    - イベント処理(EventProcessWorker)での画面データ⇔DTO間のコピーでは明示する必要がないため、挙動の違いに注意してください



このサンプルではUserIdプロパティをコピー対象とします





ログイン画面⇒メニュー画面への画面遷移

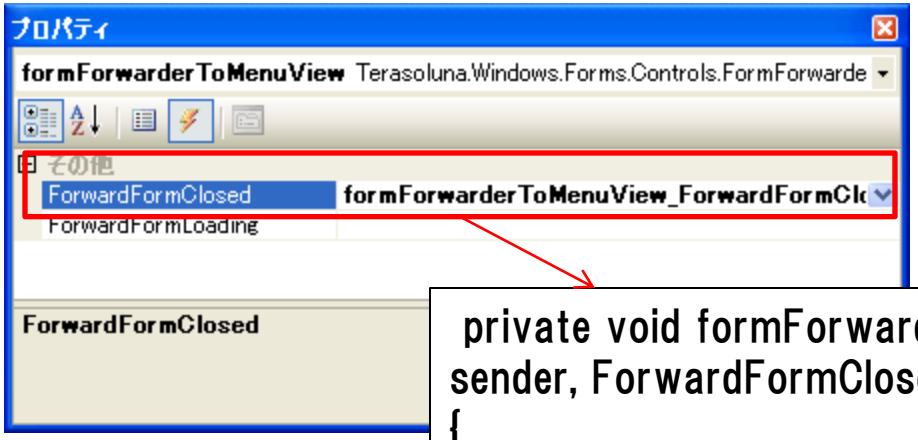
- ログインボタンのクリックイベントのハンドラメソッドを修正します。
  - FormForwarderのForwardメソッドを呼び出すことで画面遷移を実行します
  - EventProcessWorkerの処理結果をRunWorkerメソッドの戻り値として取得し、結果が成功したとき(IsSuccess プロパティ= true)だけ、画面遷移するようにします
  - ログイン画面を非表示するため、Hideします

```
private void loginButton_Click(object sender, EventArgs e)
{
    EventProcessResult result = clientBizEventProcessWorker.RunWorker();
    if (result.IsSuccess)
    {
        Hide();
        formForwarderToMenuView.Forward();
    }
}
```


TERASOLUNA

ログイン画面⇒メニュー画面への画面遷移

- メニュ画面が閉じられたときに再度ログイン画面を表示するようにします
- FormForwarderのFormwardFormClosedイベントを実装します
  - ForwardFormClosedEventArgsのCloseStateプロパティがFormState.Closed（クローズされた）ときにShow()します

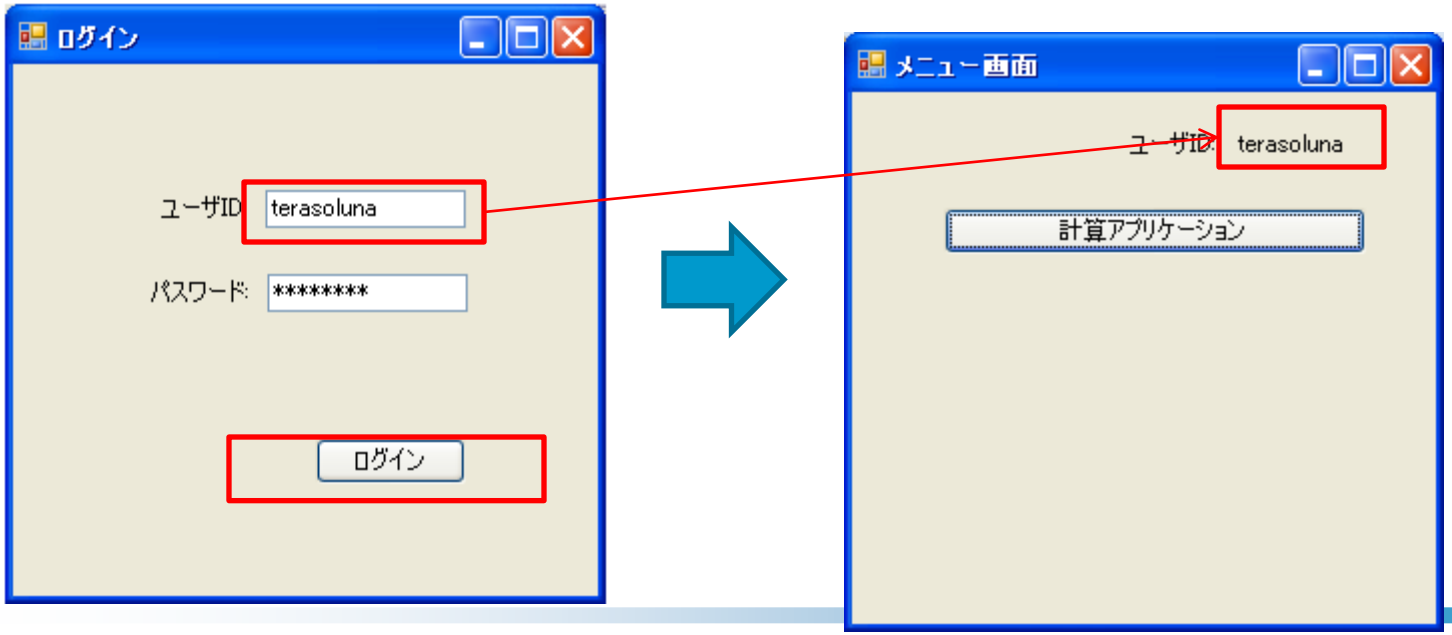


ダブルクリック

```
 private void formForwarderToMenuView_ForwardFormClosed(object
sender, ForwardFormClosedEventArgs e)
{
        if (e.CloseState == FormState.Closed)
        {
            Show();
        }
}
```

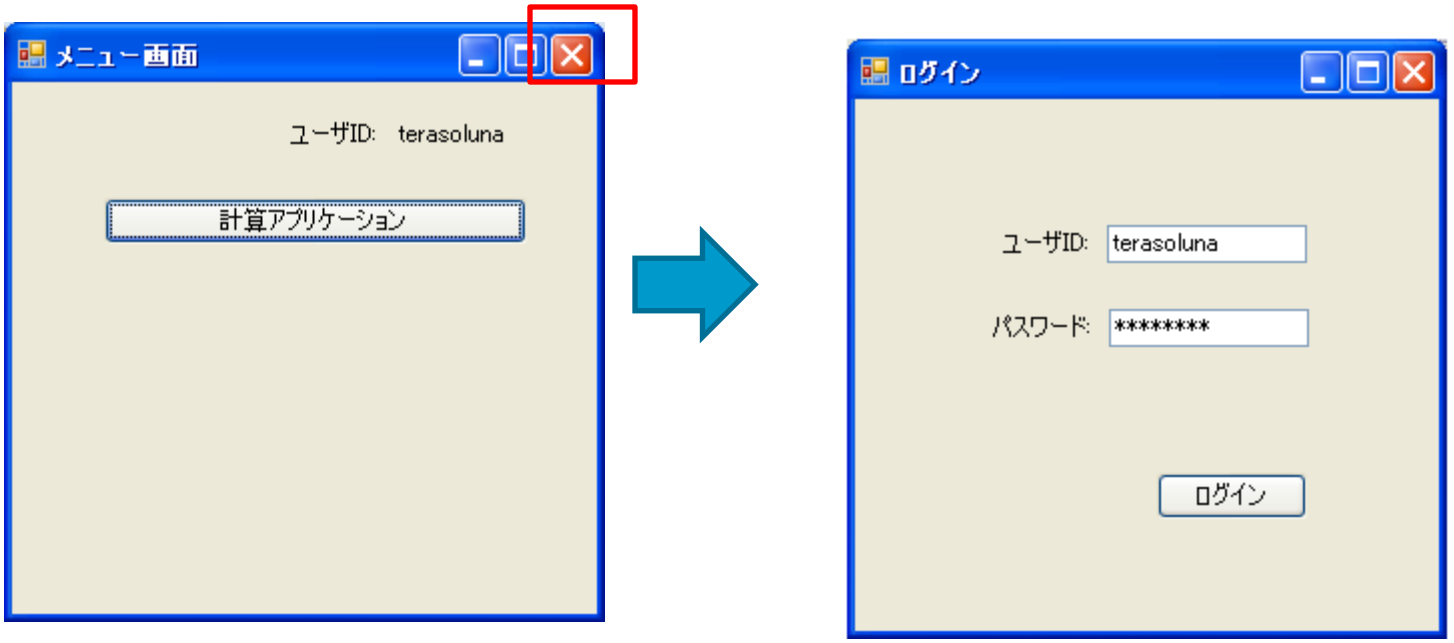
画面遷移処理の動作確認

- Visual Studioでデバッグ起動します
  - ログイン画面が起動します
- ログインボタンを押下します
  - ログイン画面が消えてメニュー画面が表示されます
  - 画面間でユーザIDの値が引き継がれているのが確認できます



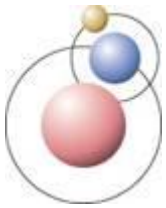
画面遷移処理の動作確認

- メニュー画面を「×」ボタンで閉じます
  - メニュー画面がクローズし、ログイン画面が再度表示されるのが確認できます



計算画面の画面データ・画面作成

- 前述の手順と同様に作成します
  - 画面データ(CalcViewData.cs)の作成
    - TERASOLUNAのアイテムテンプレートで画面データ(ルート)を作成
      - プロパティはのちほど定義するのでそのままでよいです
  - 画面(CalcView.cs)の作成
    - TERASOLUNAのアイテムテンプレートで画面を作成
      - 画面レイアウトはのちほど定義するのでそのままでよいです
    - 画面のStartPositionプロパティを「CenterScreen」にします



計算画面の作成

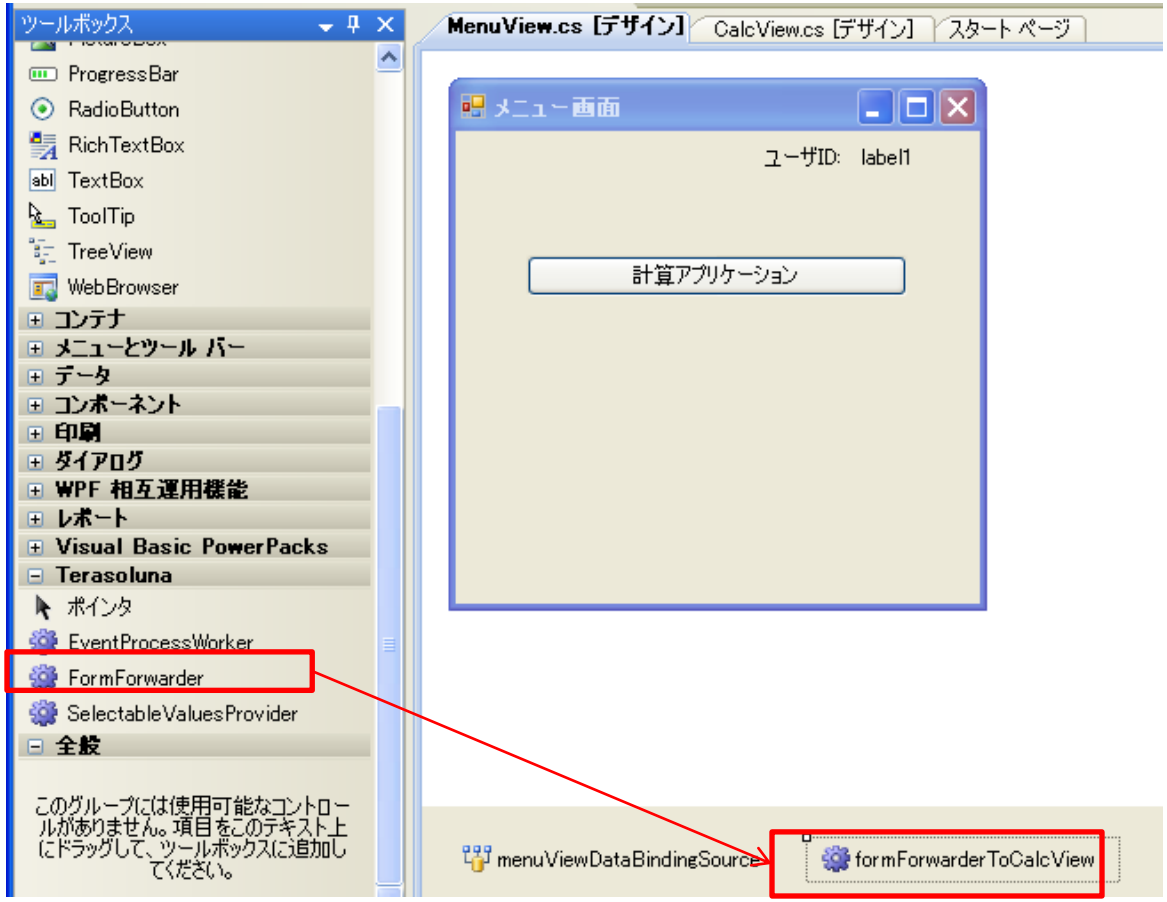
- 画面のコードのコメントを解除します

```
using CalcBusinessApplication.ViewData;

namespace CalcBusinessApplication.View
{
    // ScreenIdが決定後、指定して下さい。
    [ScreenId("CalcView")]
    public partial class CalcView : Form
    {
        // ViewDataクラスを作成後、コメント化を解除して下さい。
        public CalcViewData ViewData { get; set; }

        public CalcView()
        {
            InitializeComponent();
            // ViewDataクラスを作成後、コメント化を解除して下さい。
            ViewData = ValidatableViewDataManager.CreateViewData<CalcViewData>();
        }

        private void CalcView_Load(object sender, EventArgs e)
        {
            // バインディングソース名が確定したら、修正後コメント化を解除して下さい。
            calcViewDataBindingSource.DataSource = ViewData;
        }
    }
}
```

ひな型は大文字の「C」になっているので小文字（[c」）にします

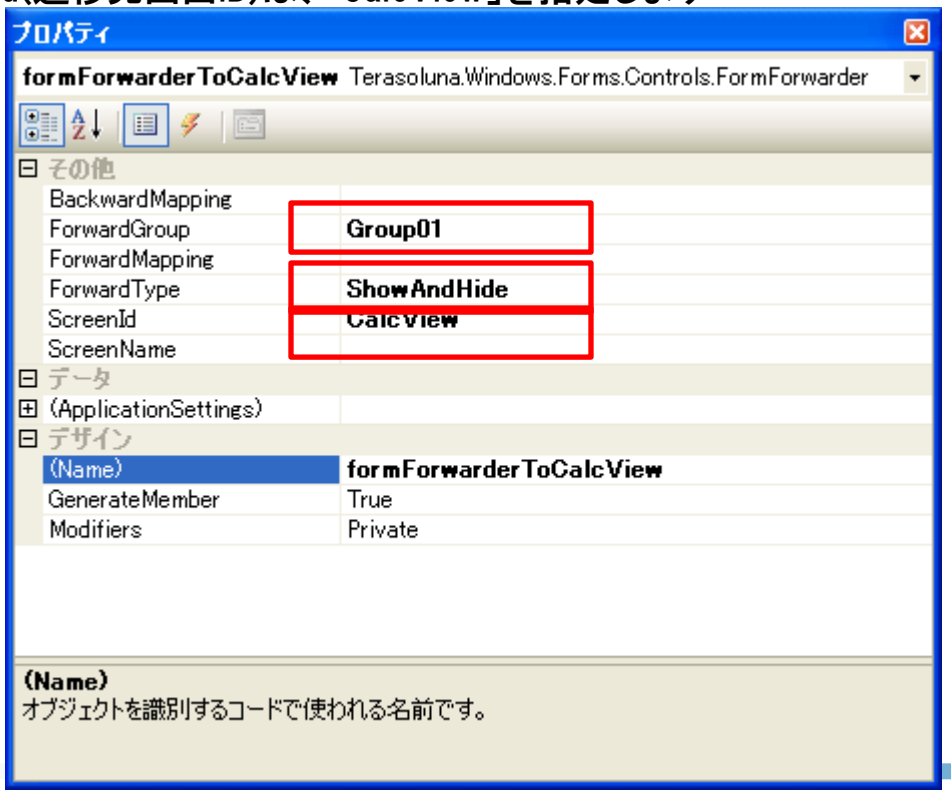
メニュー画面⇒計算画面への画面遷移

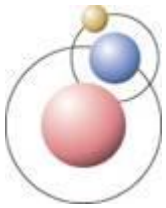
- メニュー画面にFormForwarder(forwarderToCalcView)を張り付けます



メニュー画面⇒計算画面への画面遷移

- 「ShowAndHide」を使った画面遷移を実施します
  - 遷移元画面を非表示し遷移先画面を表示する、遷移した画面を非表示する(または閉じる)と遷移元の画面を再表示するといった動作を自動的に実施します
  - ForwardTypeを「ShowAndHide」にします
  - 同一スコープを継続するため、FowardGroupに「Group01」と記述します
  - ScreenId(遷移先画面ID)は、「CalcView」を指定します





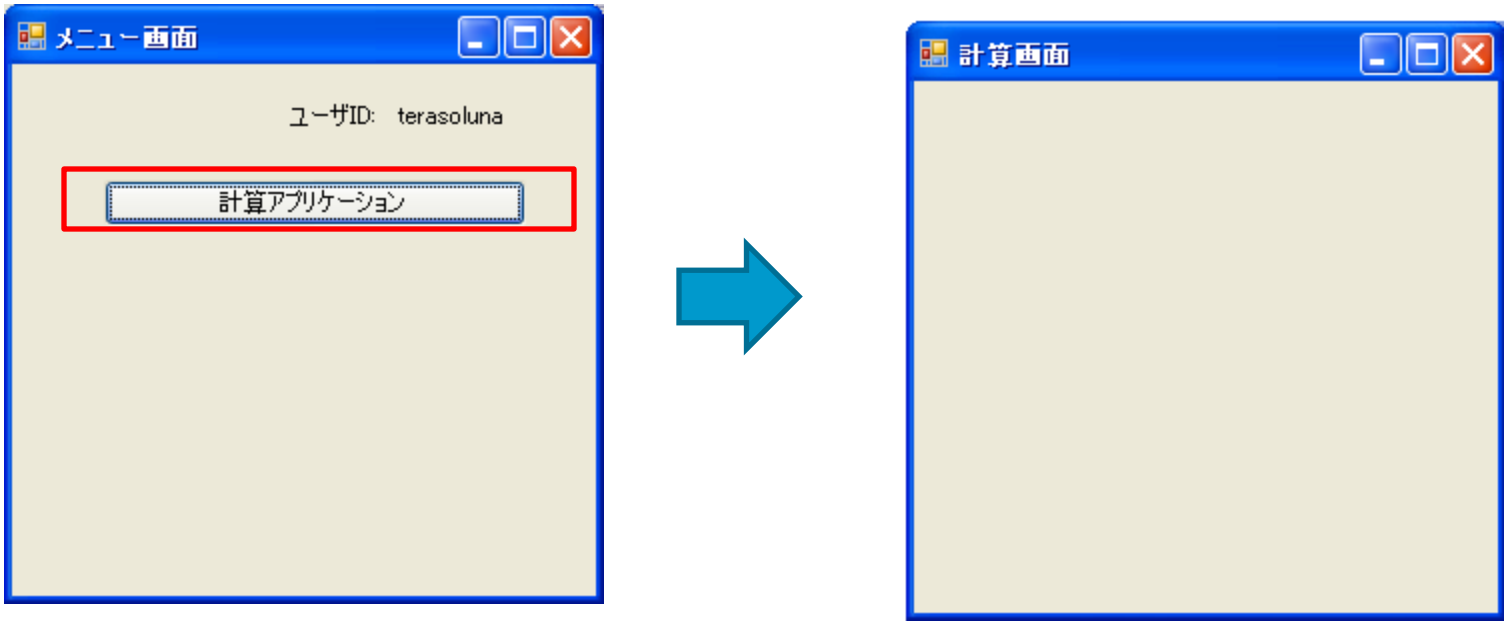
メニュー画面⇒計算画面への画面遷移

- メニュー画面のボタンのクリックイベントのハンドラメソッドにFormForwarderのForwardメソッドを記述します

```
 private void forwardButton_Click(object sender, EventArgs e)
{
    formForwarderToCalcView.Forward();
}
```

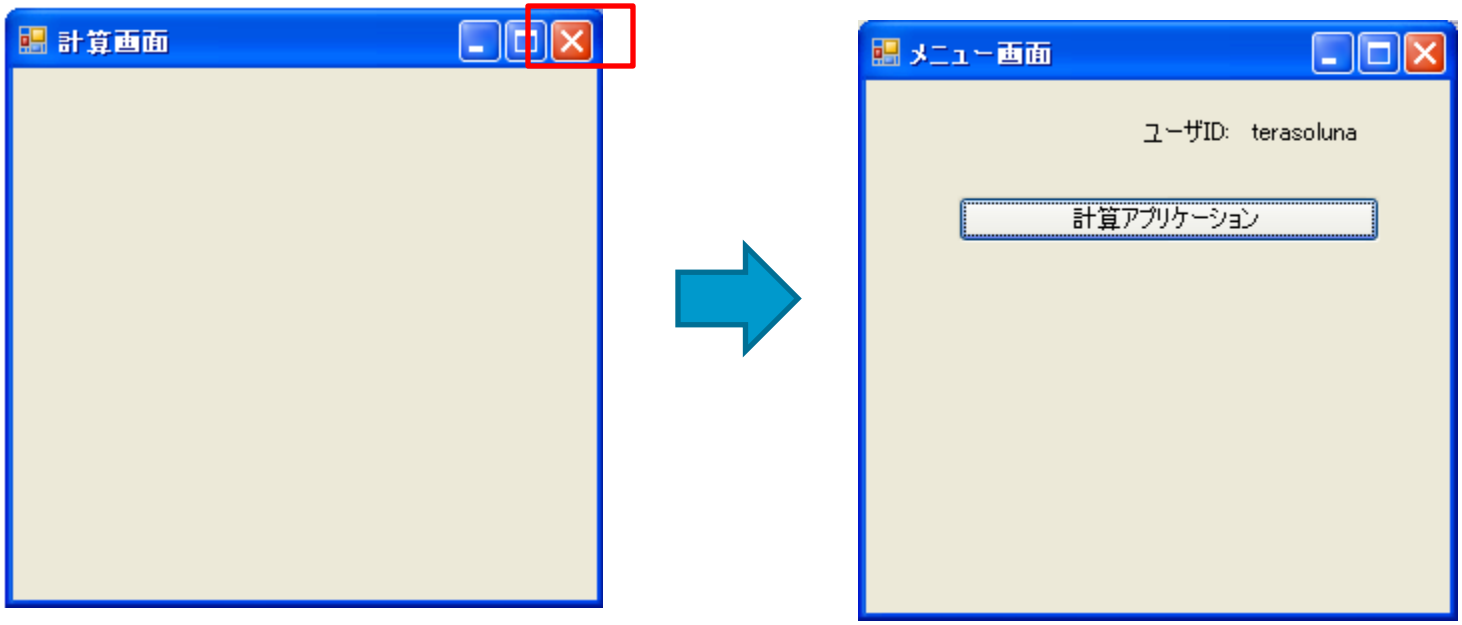
画面遷移処理の動作確認

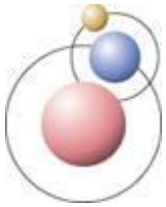
- Visual Studioでデバッグ起動し、メニュー画面へ進みます
- メニュー画面のボタンを押下します
  - メニュー画面が消えて計算画面が表示されます



画面遷移処理の動作確認

- 計算画面の「×」ボタンを押下します
  - 計算画面が消えてメニュー画面が再表示されます





計算処理の作成①
（.NETクライアント－.NETサーバの接続）

TERASOLUNA

計算画面の作成

- 前述の手順と同様に画面データを作成後、画面データを使って画面レイアウトを作成していきます

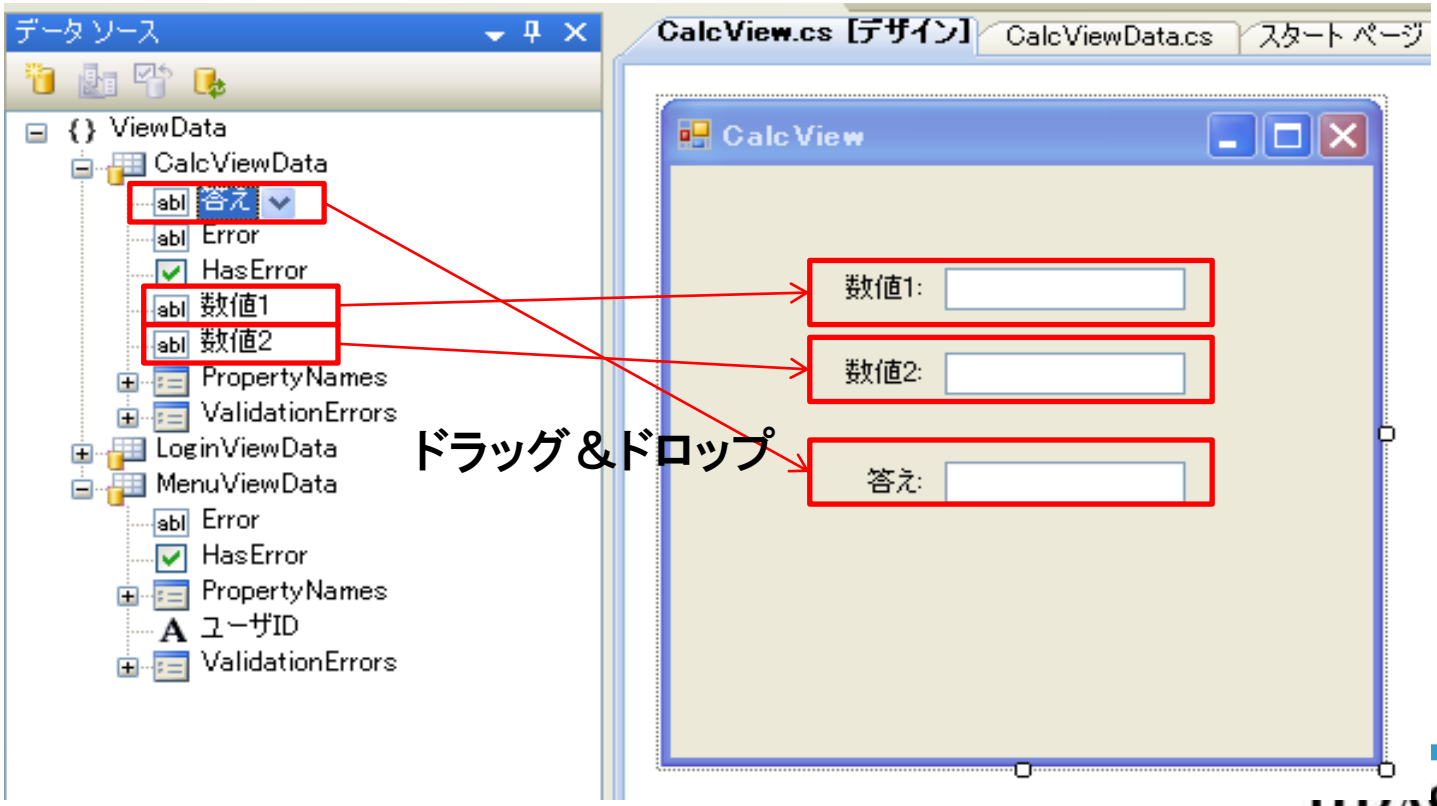
計算画面の画面データ作成

- 画面データ(CalcViewData)にプロパティを追加します
  - 「tvprop」スニペットで追加します

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class CalcViewData : ValidatableRootViewData
    {
        [DisplayName("数値1")]
        public virtual string Num1 { get; set; }

        [DisplayName("数値2")]
        public virtual string Num2 { get; set; }

        [DisplayName("答え")]
        public virtual string Answer { get; set; }
    }
}
```

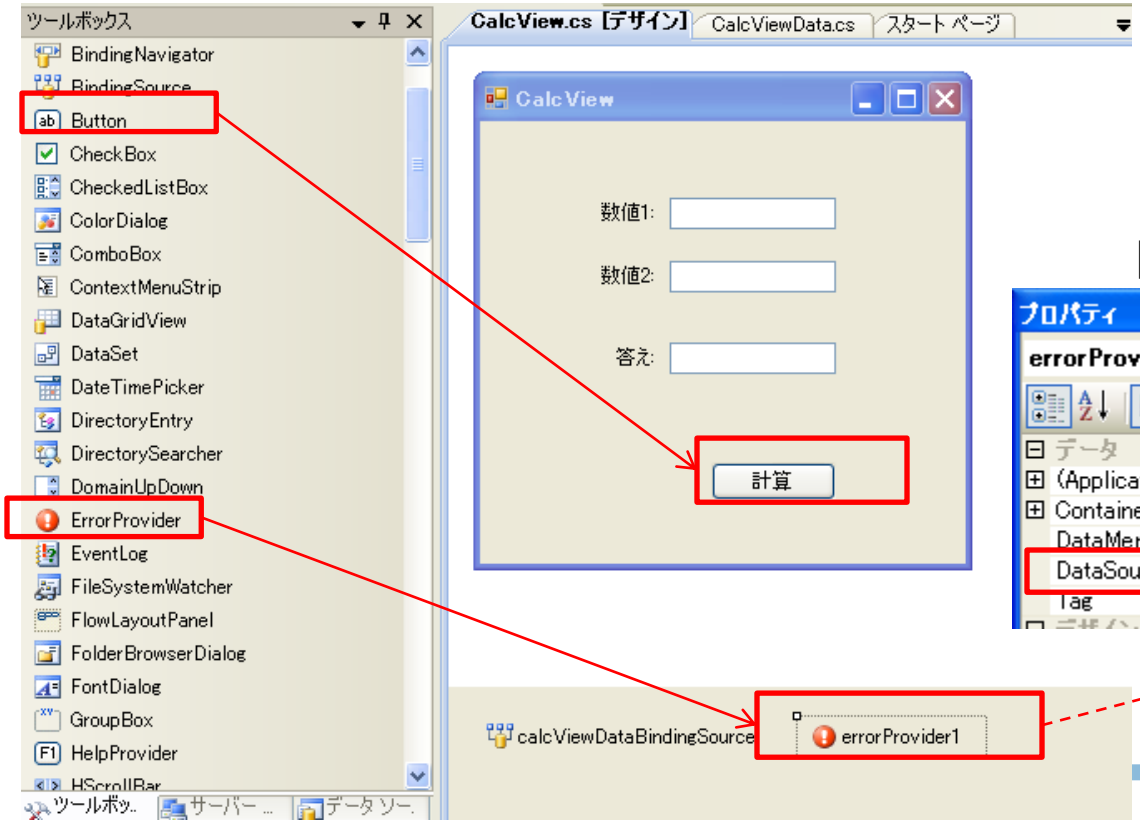
計算画面の作成

- 画面データ(CalcViewData)をデータソース構成ウィザードで追加し、コントロールを貼り付けます

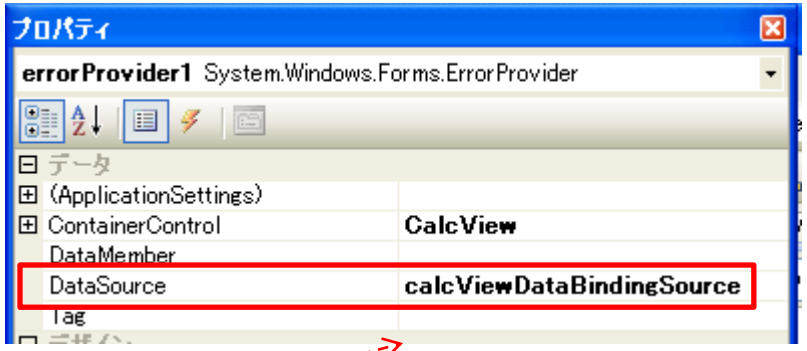


計算画面の作成

- ボタンを貼り付けます
  - nameは「calcButton」、Textプロパティを「計算」にします
- ErrorProviderを貼り付けます
  - DataSourceプロパティに、生成された BindingSource(calcViewDataBindingSource)を指定します

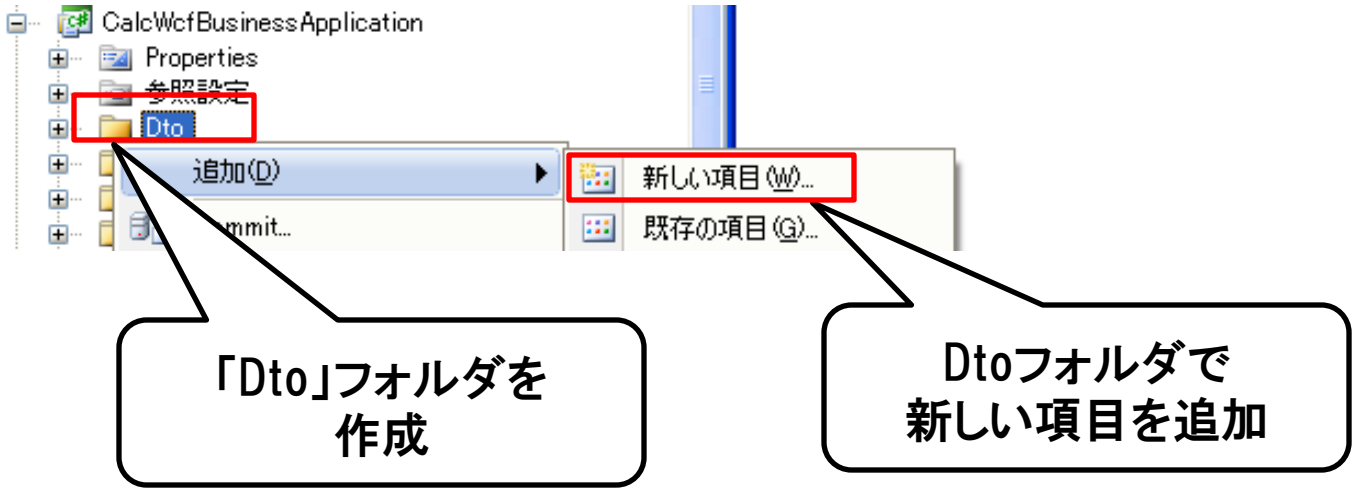


ErrorProviderのプロパティ



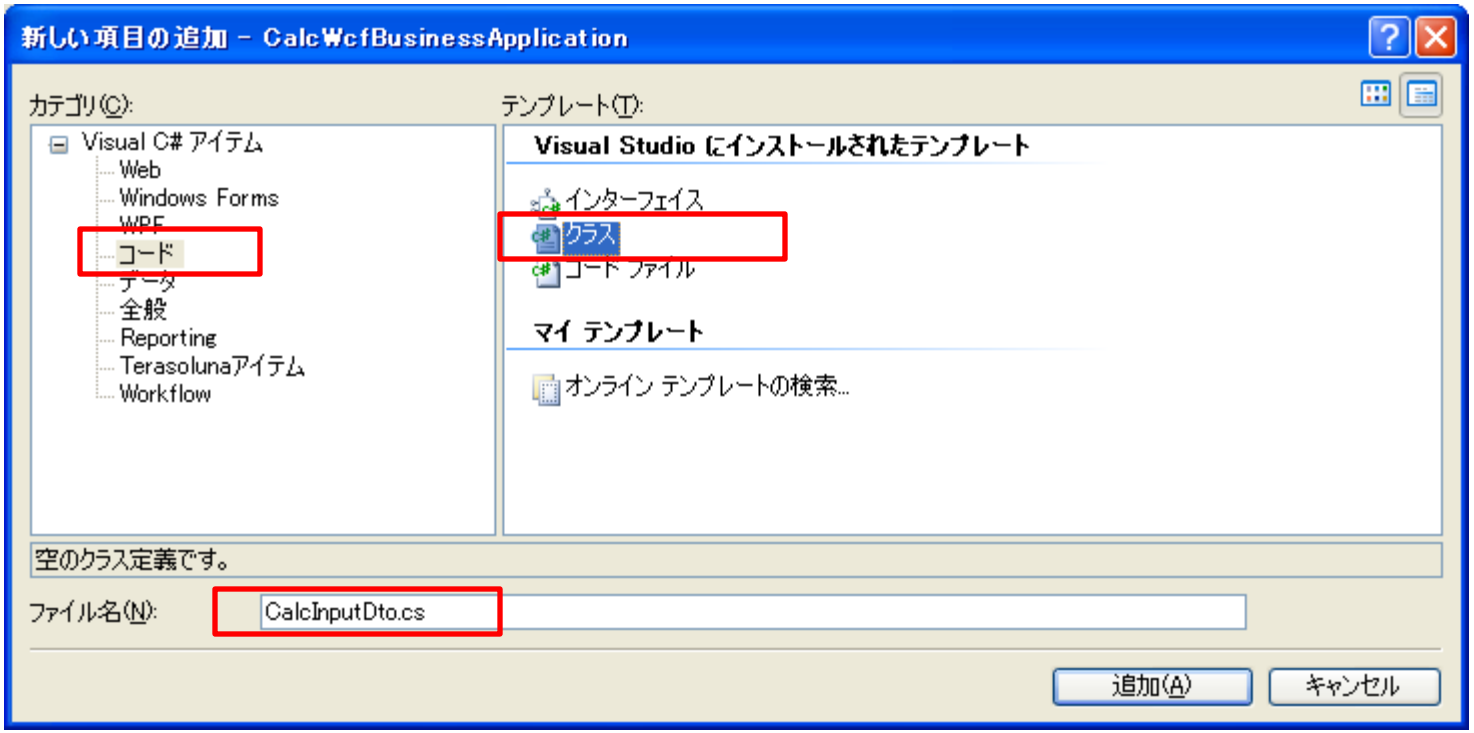
サーバ入出力DTOの作成

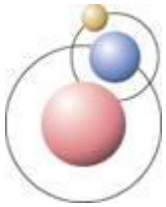
- WCFサービスの入出力DTOを作成します
  - 業務個別プロジェクト(CalcWcfBusinessApplication)に作成します



サーバ入出力DTOの作成

- .NET標準の「クラス」アイテムテンプレートを選択します

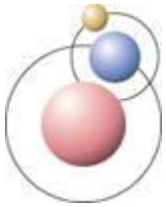




サーバ入力DTOの作成

- WCFサービスの入出力DTOを作成します
  - WCFデータコントラクトとして実装します
    - クラスに、DataContract属性を付与します
    - プロパティには、DataMember属性を付与します
    - 「tvdatamenber」スニペットを利用します(DataMember属性を自動付与)
  - 画面データのプロパティ名と一致させるようにします

```
namespace CalcWcfBusinessApplication.Dto
{
    [DataContract]
    public class CalcInputDto
    {
        [DataMember]
        public int Num1 { get; set; }
        [DataMember]
        public int Num2 { get; set; }
}
```

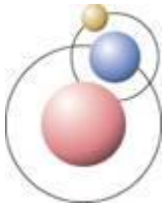


サーバ入力DTOの作成

- サーバ入力チェックを実装します
  - クライアントの画面データクラスと同様に、カスタム属性を付与します
  - TERASOLUNAのカスタムスニペットを使用して実装します
    - この例では、「tvint」を使ってIntRangeValidatorのひな形を生成します

```
namespace CalcWcfBusinessApplication.Dto
{
    [DataContract]
    public class CalcInputDto
    {
        [DataMember]
        [IntRangeValidator(LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive,
                           UpperBound = 100, UpperBoundType = RangeBoundaryType.Inclusive,
                           Tag = "数値1", Ruleset = "RS01")]
        public int Num1 { get; set; }
        [DataMember]
        [IntRangeValidator(LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive,
                           UpperBound = 100, UpperBoundType = RangeBoundaryType.Inclusive,
                           Tag = "数値2", Ruleset = "RS01")]
        public int Num2 { get; set; }
    }
}
```


TERASOLUNA



サーバ入力DTOの作成

- 相関項目チェックを追加します
  - 標準のValidatorのカスタム属性で実装できないカスタム入力チェックは、SelfValidationカスタム属性でメソッドを実装します
  - SelfValidationを使用する場合には、クラスにHasSelfValidationカスタム属性を付与します

```
namespace CalcWcfBusinessApplication.Dto
{
    [DataContract]
    [HasSelfValidation]
    public class CalcInputDto
    {
      ...
    }
}
```

サーバ入力DTOの作成

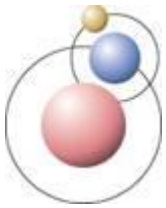
- SelfValidationカスタム属性でメソッドを実装します
  - SelfValidationメソッドの実装は、TERASOLUNAのスニペット「tvcustom」を使用します

```
namespace CalcWcfBusinessApplication.Dto
{
    [DataContract]
    [HasSelfValidation]
    public class CalcInputDto
    {
        [DataMember]
        public int Num1 { get; set; }
        [DataMember]
        public int Num2 { get; set; }

        t
```

tvbyte
tvcomparison
tvcustom
tvdate
tvdaterange
tvdecimal
tvdouble

tvcustom
カスタム入力チェックに対するコードスニペット

```
[SelfValidation(Ruleset = "RS01")]
public void CustomValidate01(ValidationResults results)
{
}
```

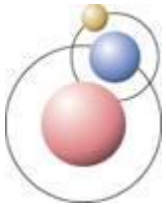


サーバ入力DTOの作成

- スニペットにより生成されたSelfValidationメソッドにカスタム入力チェック処理を実装します

```
[DataContract]
[HasSelfValidation]
public class CalcInputDto
{
    [SelfValidation(Ruleset ="RS01")]
    private void CustomValidate01(ValidationResults results)
    {
        ///数値1と数値2が同じ数だとエラーとする
        if (Num1 == Num2)
        {
            ValidationResult result =
                new ValidationResult(Resources.ERROR_CALC_MSG001,
                                                    this, string.Empty, string.Empty, null);
            results.AddResult(resultForNum1);
        }
    }
```


TERASOLUNA



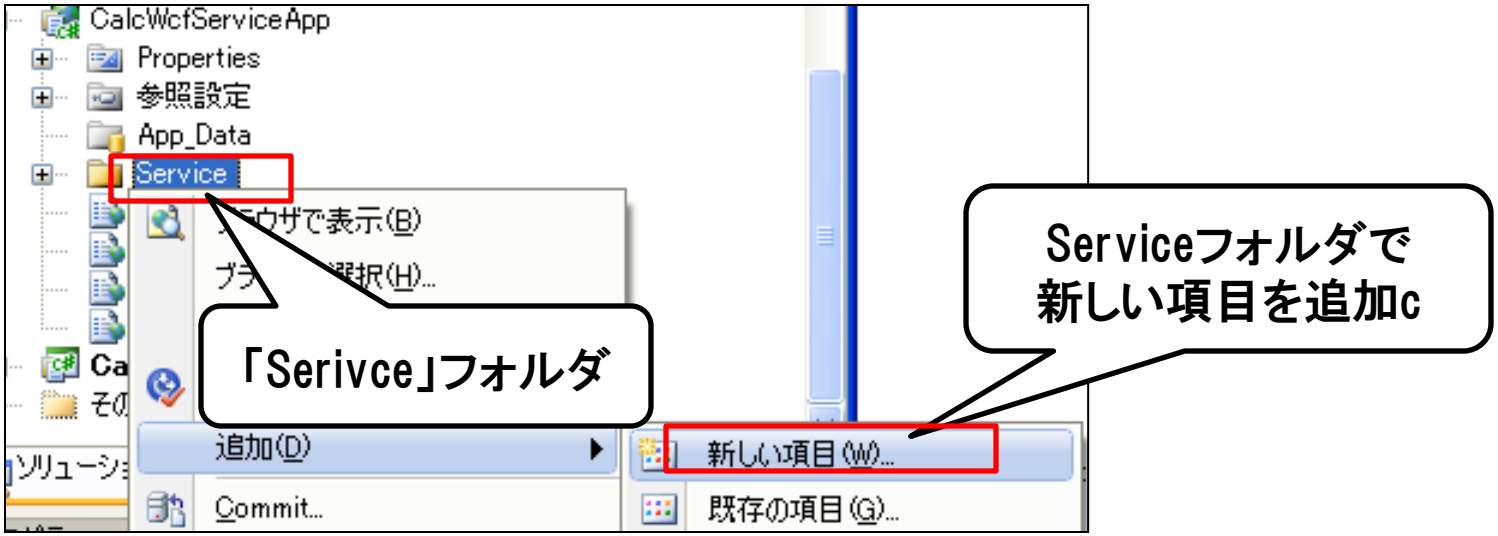
サーバ出力DTOの作成

- 入力DTOと同様に出力DTOも以下のように作成します

```
namespace CalcWcfBusinessApplication.Dto
{
    [DataContract]
    public class CalcOutputDto
    {
        [DataMember]
        public int Answer { get; set; }
    }
}
```

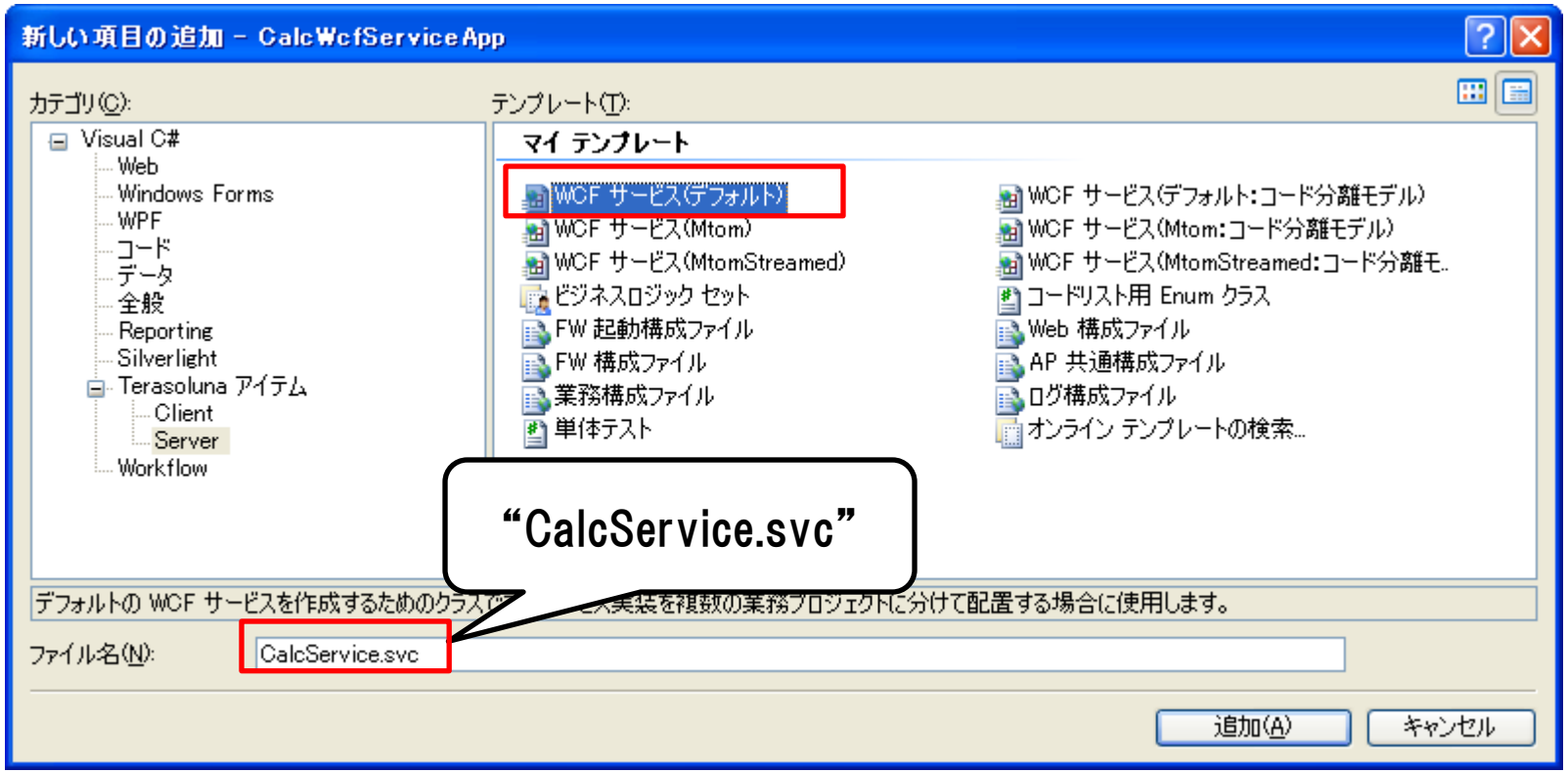
WCFサービスファイル(.svcファイル)の作成

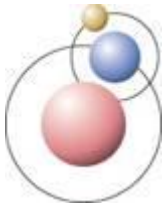
- WCFサービスを配備するためにWCFサービスファイル(.svcファイル)を作成します
  - WCFサービスアプリケーションプロジェクト(CalcWcfServiceApp)に、ファイルを追加します



WCFサービスファイル(.svcファイル)の作成

- 「WCFサービス(デフォルト)ファイル」
  - TERASOLUNAのテンプレートを使用して作成します





WCFサービスファイル(.svcファイル)の作成

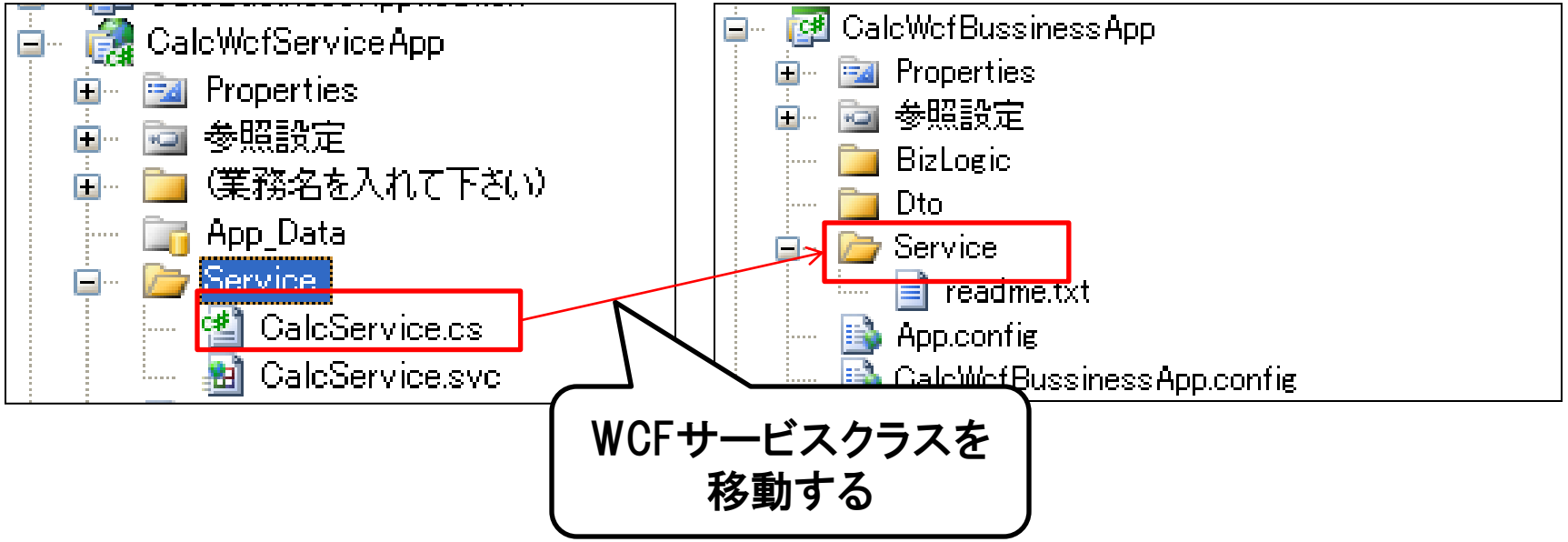
- テンプレートにより、以下の内容の.svcファイルが作成されます
  - ファイルを編集する必要はありません

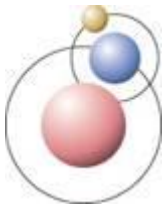
```
<%@ ServiceHost Language="C#" Debug="true"
      Factory="Terasoluna.Server.ServiceModel.DefaultServiceHostFactory" %>
```


TERASOLUNA

WCFサービスクラスの作成

- WCFサービスファイルのテンプレートにより、WCFサービスクラスも生成されます
- 業務個別プロジェクト(CalcWcfBusinessApplication)のServiceフォルダに、ファイルを移動します





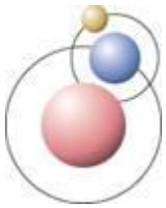
WCFサービスクラスの作成

- 生成されたWCFサービスコントラクト(CalcService.cs)のひな形を以下のように修正します
  - Addメソッドを追加します

WCFサービスクラス(CalcService.cs)

```
namespace CalcWcfBusinessApplication.Service
{
    [ServiceContract]
    public class CalcService
    {
        [OperationContract]
        CalcOutputDto Add(CalcInputDto inputDto)
        {
            int result = inputDto.Num1 + inputDto.Num2;
            return new CalcOutputDto() { Answer = result };
        }
    }
}
```


TERASOLUNA



WCFサービスクラスの作成

- サーバ入力チェックを処理を有効にするため ValidationBehavior属性を付与します
  - ルールセット名は、”RS01”にしておきます

WCFサービスクラス(CalcService.cs)

```
namespace CalcWcfBusinessApplication.Service
{
    [ServiceContract]
    public class CalcService
    {
        [OperationContract]
        [ValidationBehavior(RuleSet = "RS01")]
        CalcOutputDto Add(CalcInputDto inputDto)
        {
            ...
        }
    }
}
```


TERASOLUNA

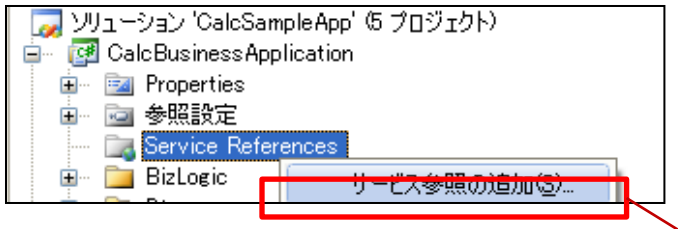
TerasolunaFramework.configの設定

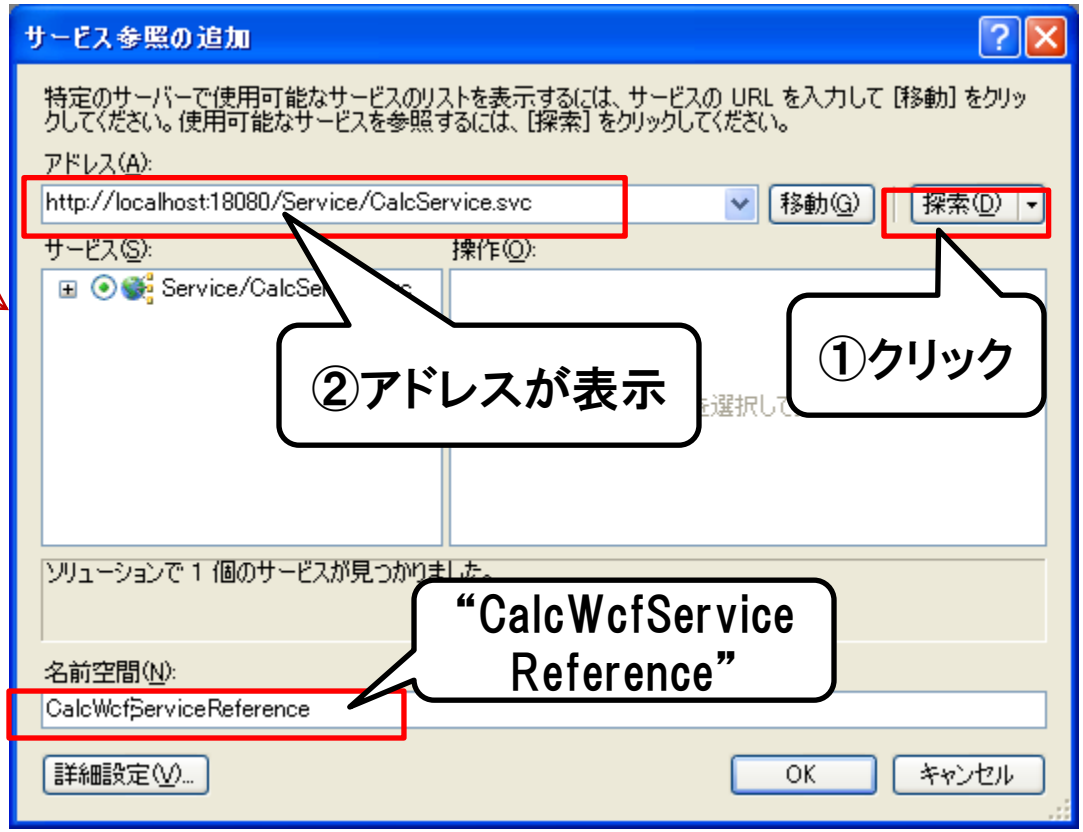
- WCFサービスアプリケーションプロジェクト(CalcWcfService)のTerasolunaFramework.configで、WCFサービスファイルのURIのパスからWCFサービスクラス名を決定する際に使用する「基底名前空間」(BaseNamespace)を設定します
  - 個別業務プロジェクトの基底名前空間である「CalcWcfBussinessApplication」に設定します

```
<containers>
  <container>
    <types>
    </types>
    <instances>
      <!-- svcファイルからWCFサービスを取得するためのベース名i
      <add name="BaseNamespace" type="string"
           value="CalcWcfBusinessApplication" />
    </instances>
    <extensions>
```

WCFクライアントの生成

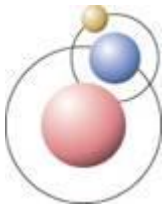
- クライアントの業務個別プロジェクト(CalcBusinessApplication)で、「サービス参照の追加」を実行し、WSDLよりプロキシコード(WCFクライアント)を生成します





WCFクライアントの生成

- 「サービス参照」が追加されます



App.configのコピー

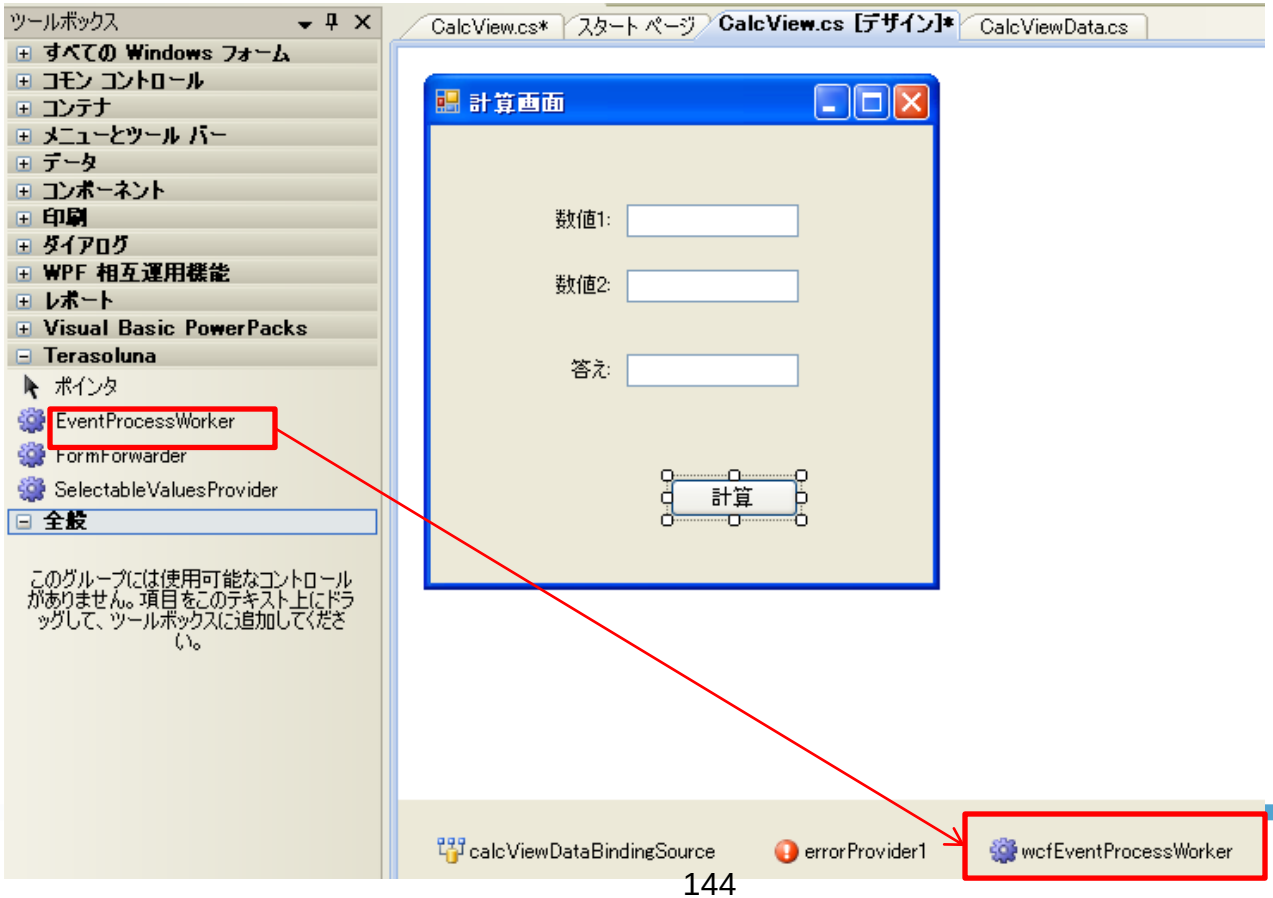
- 「サービス参照」の追加により、クライアントの業務個別プロジェクト(CalcBusinessApplication)のApp.configにWCFの設定が生成されます
- 実行時には、起動アプリケーションプロジェクト(CalcWinFormsApp)のApp.configに同様の設定をしておく必要があります
- <system.serviceModel>要素配下をCalcWinFormsAppのApp.configにコピーします

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <wsHttpBinding>
            …
      </wsHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:18080/Service/CalcService.svc"
          binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_CalcService"
          contract="CalcWcfServiceReference.CalcService" name="WSHttpBinding_CalcService">
        <identity>
          <dns value="localhost" />
        </identity>
      </endpoint>
    </client>
  </system.serviceModel>
…
```


TERASOLUNA

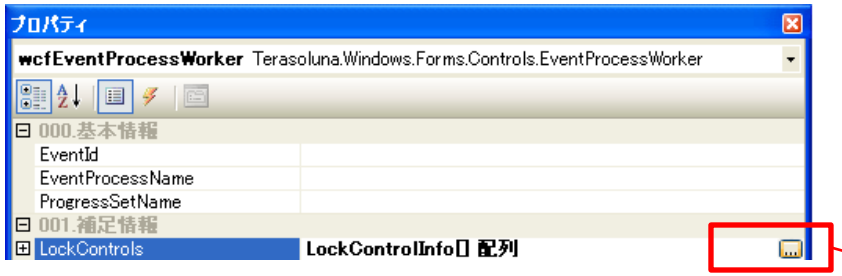
EventProcessWorkerの設定

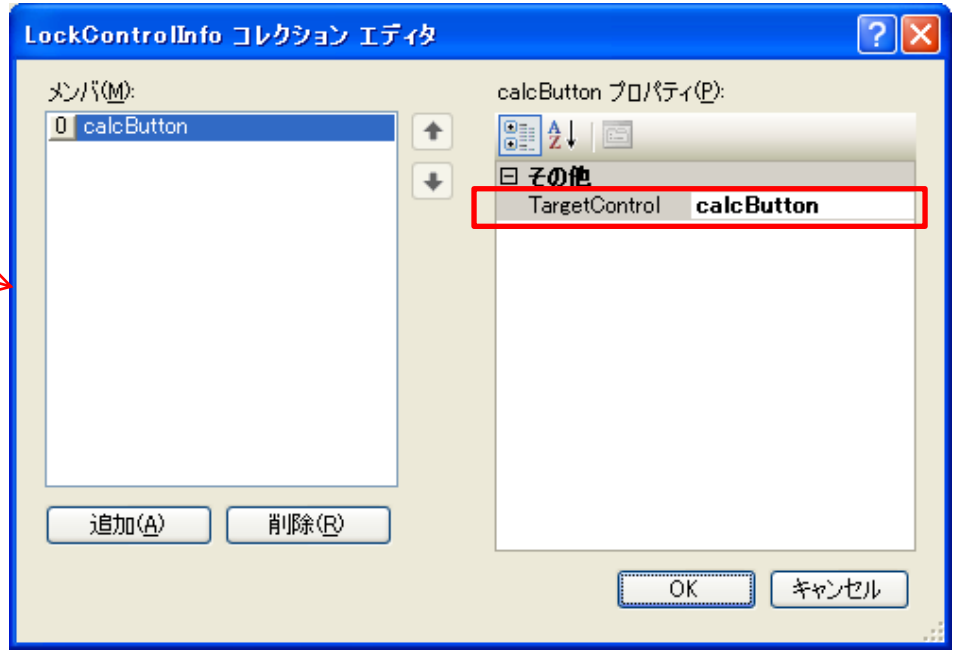
- 計算画面にEventProcessWorkerを追加します
  - nameをwcfEventProcessWorkerとします



EventProcessWorkerの設定

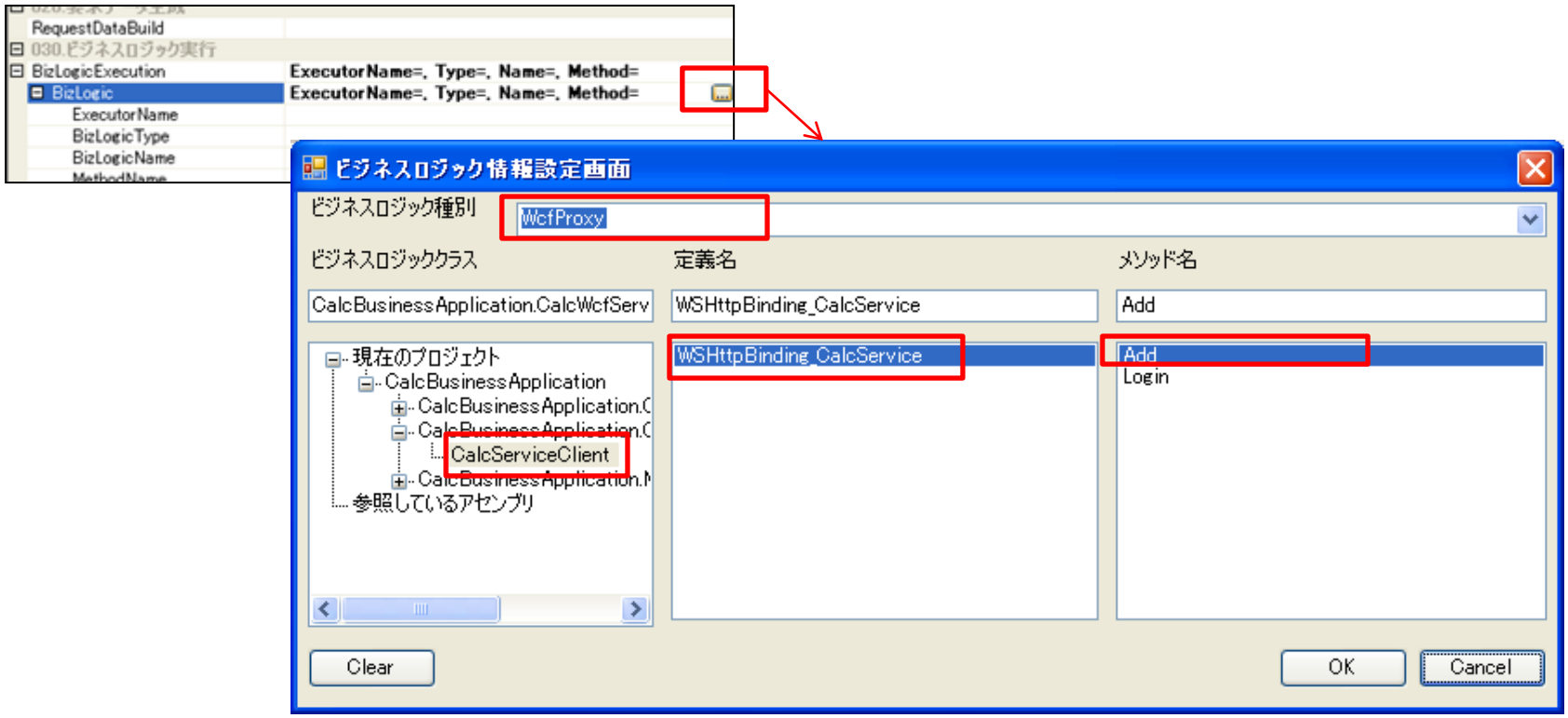
- 計算画面にEventProcessWorkerを追加します
- LockControlsプロパティに「calcButton」(計算ボタン)を追加します
  - EventProcesssNameプロパティを指定しない場合、「ControlsLock」(指定した画面上のコントロールを無効化しイベントの多重実行防止する)になります
  - 「ControlsLock」の場合、イベント処理中に、LockControlsプロパティに指定したUIコントロールのみ無効化します





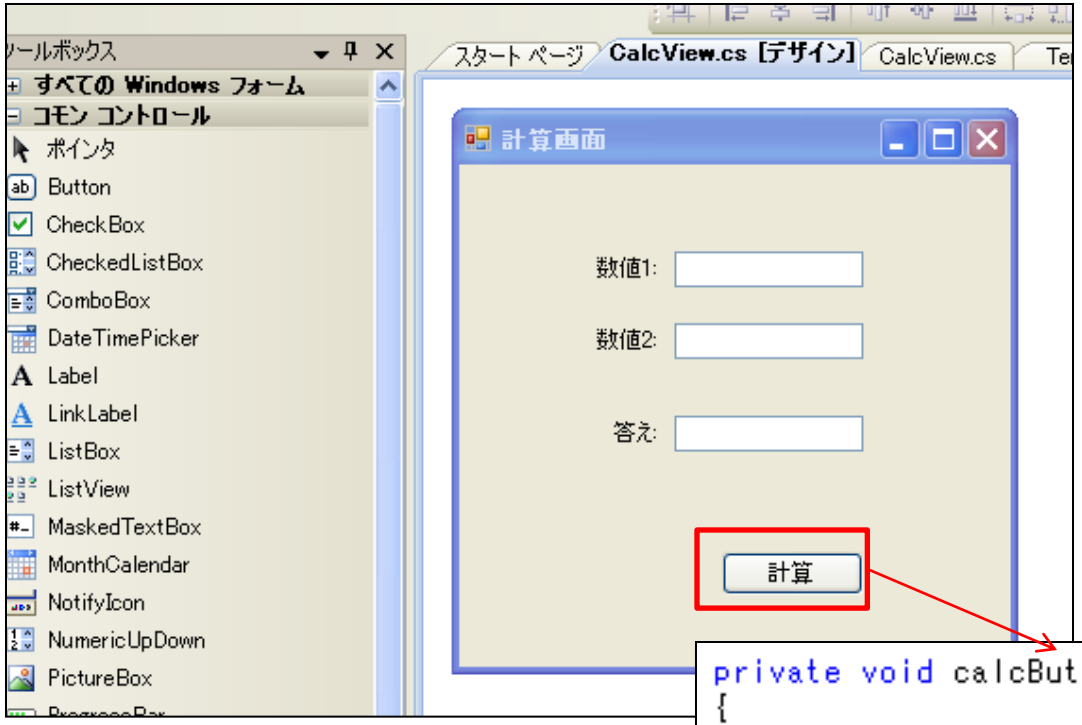
EventProcessWorkerの設定

- 計算画面にEventProcessWorkerを追加します
  - BizLogicExecutionプロパティに、実行するWCFクライアントを設定します

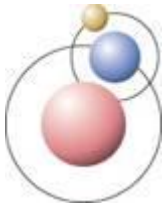


EventProcessWorkerの実行

- 計算ボタンをダブルクリックします
  - ボタンクリックイベントのハンドラメソッドが生成されます




```
private void calcButton_Click(object sender, EventArgs e)
{
    |
}
```



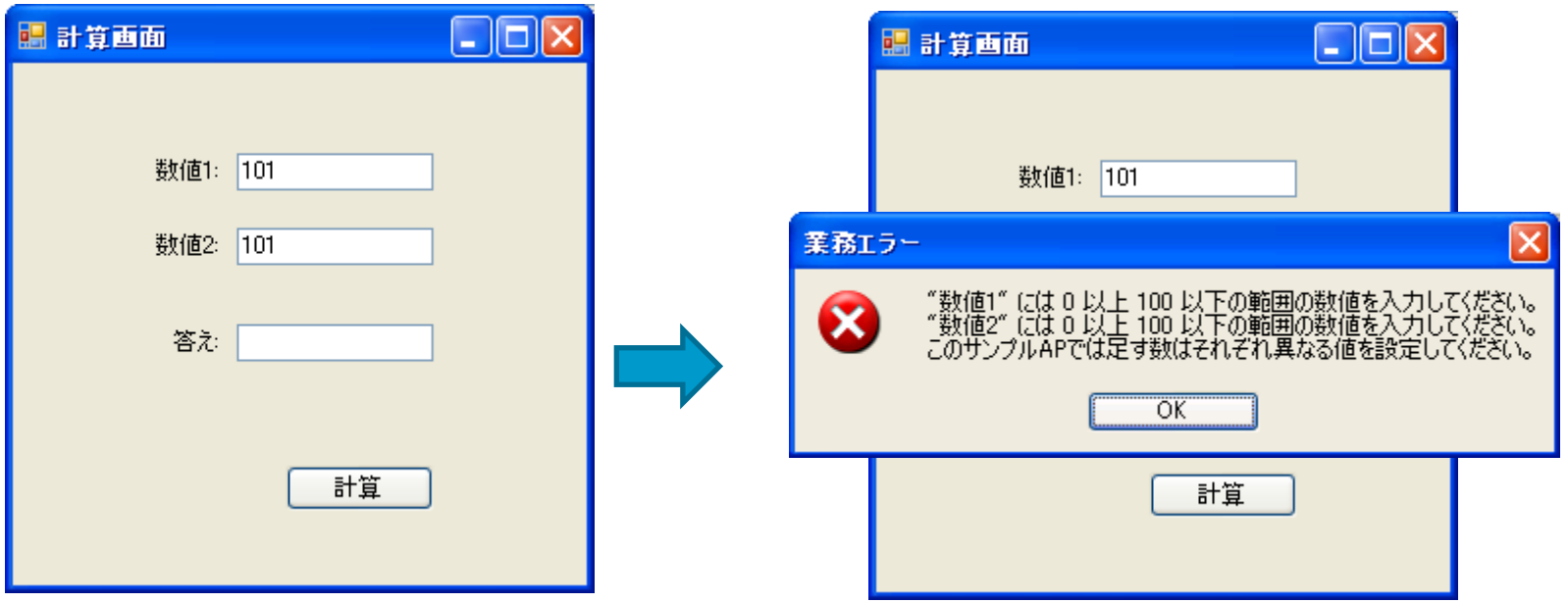
EventProcessWorkerの実行

- ボタンクリックイベントのハンドラメソッドでイベント処理の非同期実行を実施します
  - EventProcessWorkerのRunWorkerAsyncメソッドを使用します

```
private void calcButton_Click(object sender, EventArgs e)
{
    wcfEventProcessWorker.RunWorkerAsync();
}
```

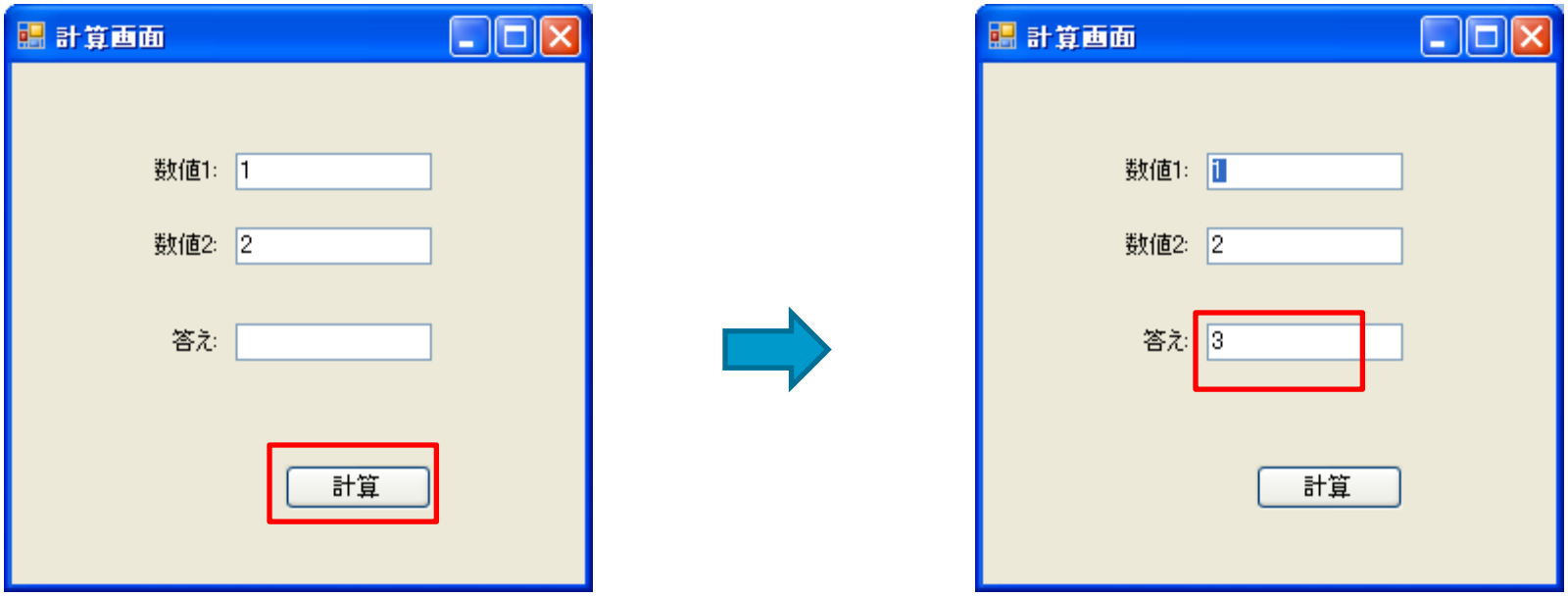
サーバ入力チェックの動作確認

- Visual Studioでデバッグ実行します
  - 計算画面まで進みます
  - 数値1、数値2に「101」を入力します
  - 単項目チェックとカスタム入力チェックエラー（相関項目チェック）が出力され、サーバ側で入力チェック処理が実行されたことが確認できます



サーバ計算処理の確認

- 数値1に「1」、数値2に「2」を入力します
  - 答えに「3」が表示され、サーバ側で計算処理が正しく実行されたことが確認できます。



イベントの非同期処理の動作確認

- CalcServiceのAddメソッド内に「Thread.Sleep(5000); 」を追記し再度実行します
  - イベント処理中でも画面の移動が可能であることが確認できます
    - イベントを非同期実行(EventProcessWorker.RunWorkerAsyncメソッド)したためです
  - イベント処理中は、「計算ボタン」が無効化されることが確認できます
    - LockControlsプロパティを指定したためです

計算画面の画面データ作成

- 画面データ(CalcViewData)を以下のように作成します
  - IntRangeValidatorのスニペットは「tvint」を使います
    - 0以上100以下の数値かを検証するようにします

```
namespace CalcBusinessApplication.ViewData
{
    [DefaultRuleset("RS01")]
    // [RulesetMapping("RS01", "", "")]
    public class CalcViewData : ValidatableRootViewData
    {
        [DisplayName("数値1")]
        [RequiredValidator(Tag="数値1", Ruleset="RS01")]
        [IntRangeValidator(LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive,
                UpperBound = 100, UpperBoundType = RangeBoundaryType.Inclusive, Tag="数値1", Ruleset="RS01")]
        public virtual string Num1 { get; set; }

        [DisplayName("数値2")]
        [RequiredValidator(Tag = "数値2", Ruleset = "RS01")]
        [IntRangeValidator(LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive,
                 UpperBound = 100, UpperBoundType = RangeBoundaryType.Inclusive, Tag = "数値2", Ruleset = "RS01")]
        public virtual string Num2 { get; set; }

        [DisplayName("答え")]
        public virtual string Answer { get; set; }
    }
}
```

計算画面の画面データ作成

- 相関項目チェックを追加します
  - 標準のValidatorのカスタム属性で実装できないカスタム入力チェックは、SelfValidationカスタム属性でメソッドを実装します
  - SelfValidationを使用する場合には、クラスにHasSelfValidationカスタム属性を付与します

```
namespace CalcBusinessApplication.ViewData
{
    [HasSelfValidation]
    [DefaultRuleset("RS01")]
    public class CalcViewData : ValidatableRootViewData
    {
      ...
    }
}
```

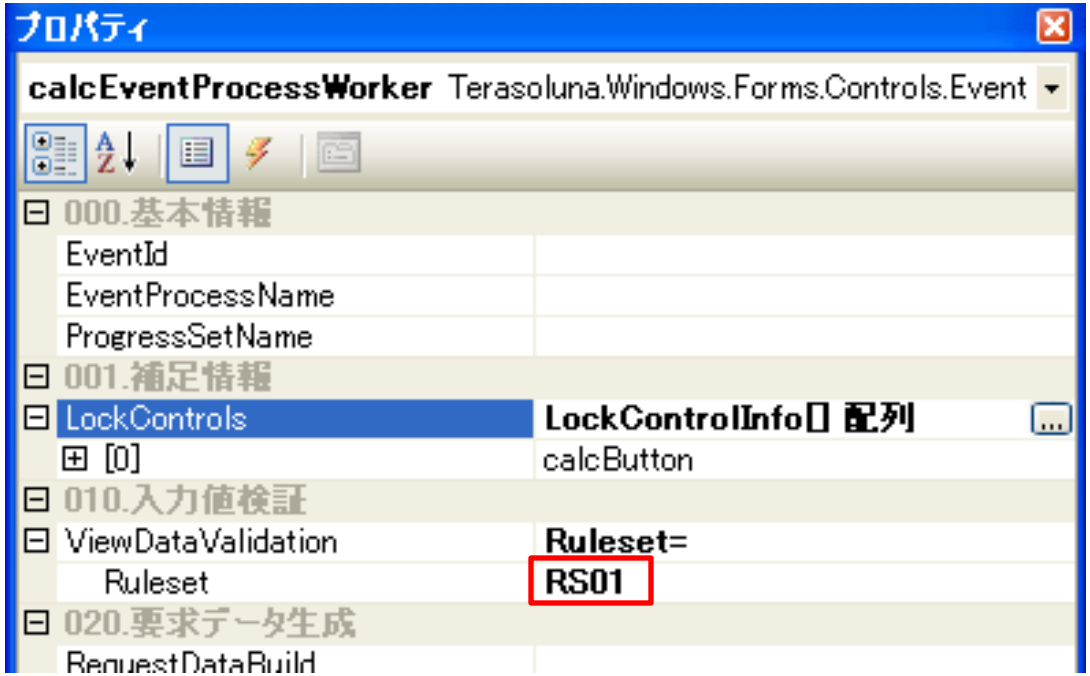
計算画面の画面データ作成

- SelfValidationカスタム属性でメソッドを実装します
  - SelfValidationメソッドの実装は、TERASOLUNAのスニペット「tvcustom」を使用します

```
[HasSelfValidation]
[DefaultRuleset("RS01")]
public class CalcViewData : ValidatableRootViewData
{
    …
    [SelfValidation(Ruleset ="RS01")]
    private void CustomValidate01(ValidationResults results)
    {
        ///数値1と数値2が同じ数だとエラーになる
        if (Num1.Equals(Num2, StringComparison.Ordinal)) {
            ///相関チェックでもエラーを各コントロールに表示するなら
            ///keyにプロパティ名を指定してValidationResultを作成
            ValidationResult resultForNum1 =
                new ValidationResult(Resources.ERROR_CALC_MSG001, this,
                    "Num1", null, EventSpecificValidator.DefaultInstance);
            ValidationResult resultForNum2 =
                new ValidationResult(Resources.ERROR_CALC_MSG001, this,
            "Num2", null, EventSpecificValidator.DefaultInstance);
            results.AddResult(resultForNum1);
            results.AddResult(resultForNum2);
    }
}
```

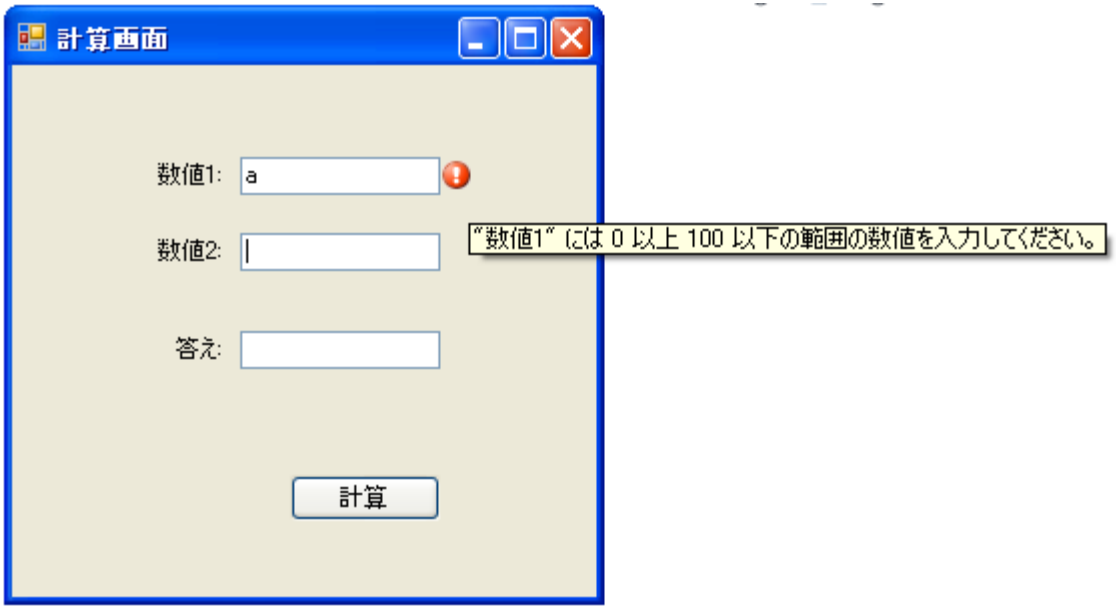
EventProcessWorkerの設定

- クライアントの入力チェック処理の設定をします
  - VaiewDataValidationプロパティで、実行するルールセット名を指定します
  - ここでは、「RS01」を設定します



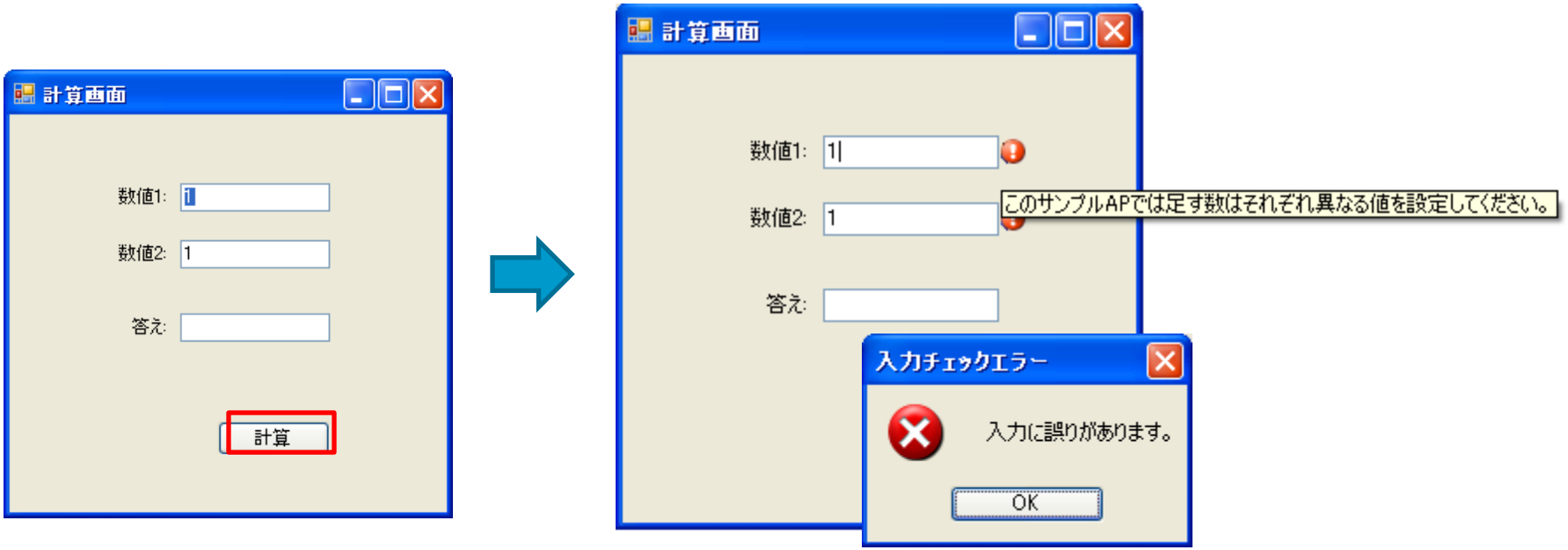
即値チェック処理の動作確認

- 計算画面で数値に「a」など数値以外の文字を入力しロストフォーカスします
  - 入力チェック処理が実行され、直ちにErrorProviderが表示されます



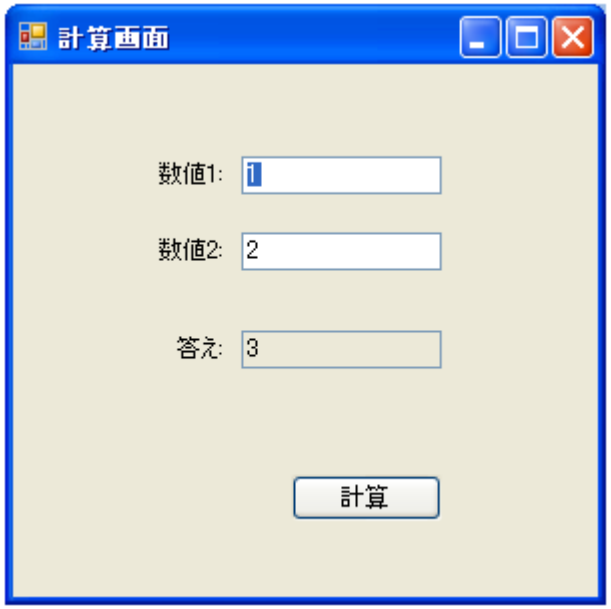
イベント処理実行時の入力値検証処理の動作確認

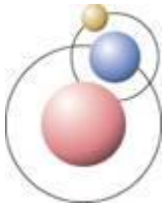
- 計算画面で数値1に「1」、数値2に「1」を入力し計算ボタンを押下します
  - イベント処理実行時には、SelfValidationメソッドによるカスタム入力チェック(相関項目チェック)も含めて入力チェック処理が実施されることが確認できます



「答え」のTextBoxのReadOnly設定

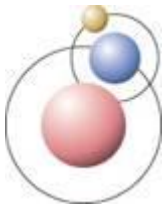
- ひとおり動作確認できたので、「答え」のTextBoxのReadOnlyプロパティをtrueにします





計算処理の作成②
（.NETクライアント-Javaサーバの接続）

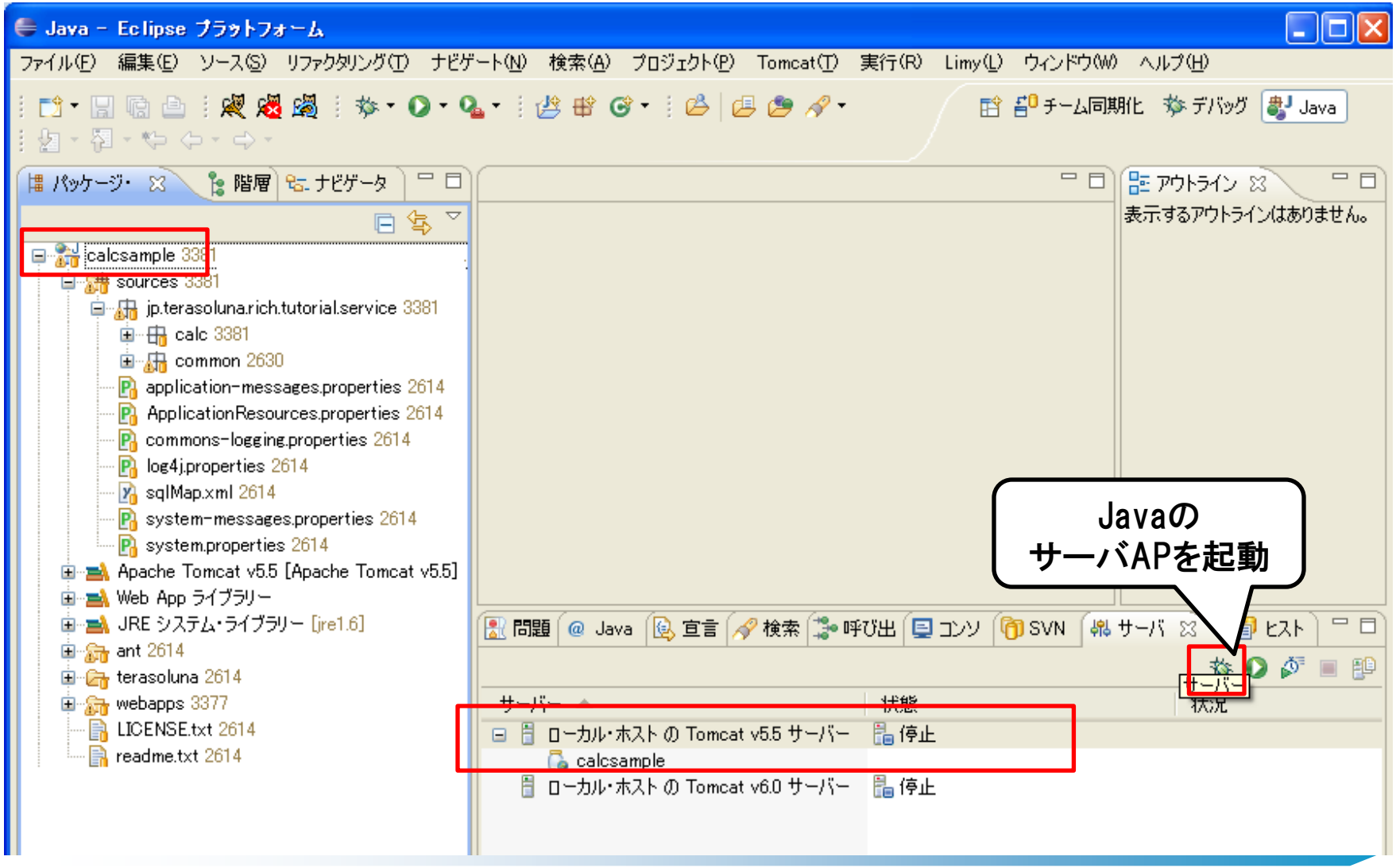
TERASOLUNA



JavaサーバAPに接続する

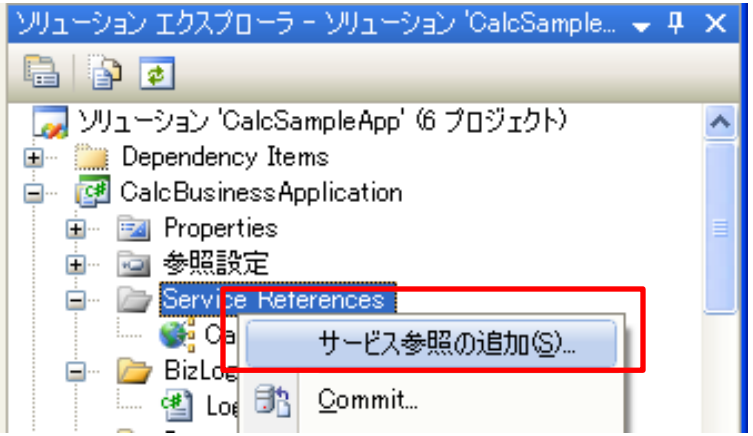
- 計算処理を機能拡張し、JavaのサーバAP(JAX-WS Webサービス)に接続します
  - 前述のJavaの環境構築が必要です
- JavaサーバAP(calcsample)を起動します
  - EclipseのWTPで、Tomcatにcalcsampleを配備し、起動します
  - または、Antを使ってTomcatに配備し、起動します
    - ant¥build.xmlのdeployタスクを実行します
    - Tomcatを起動します。

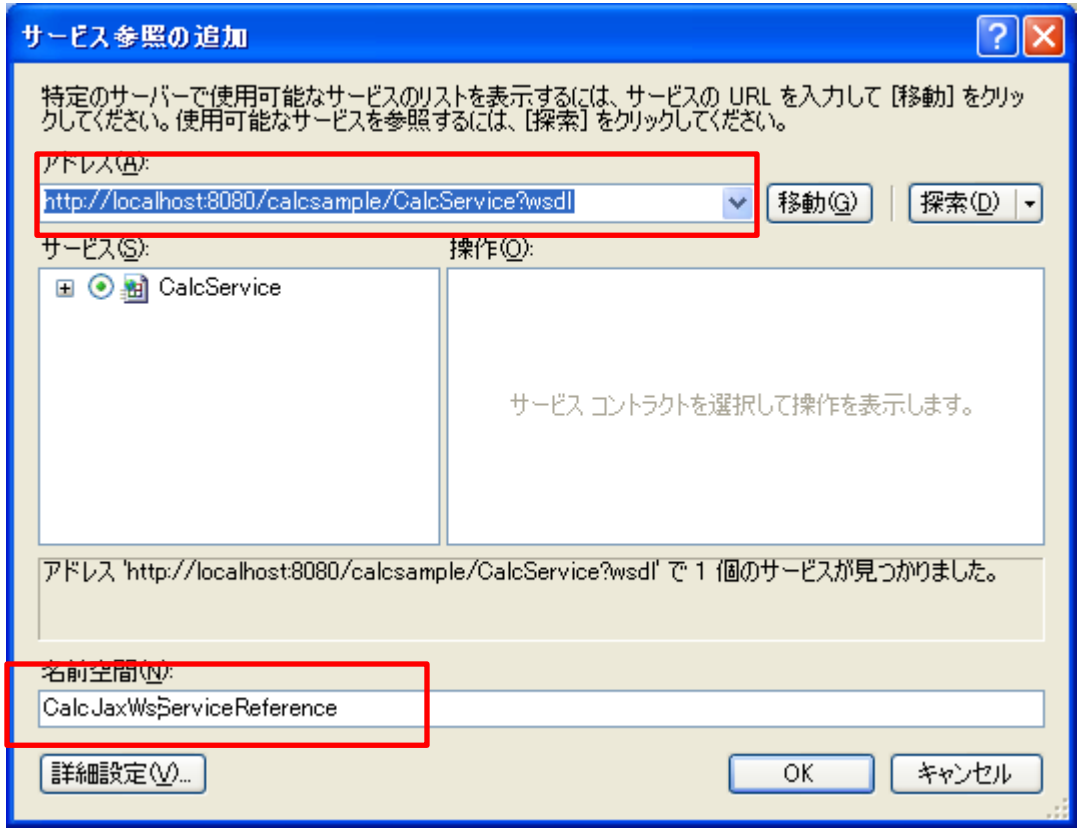
JavaサーバAPに接続する



サービス参照の追加

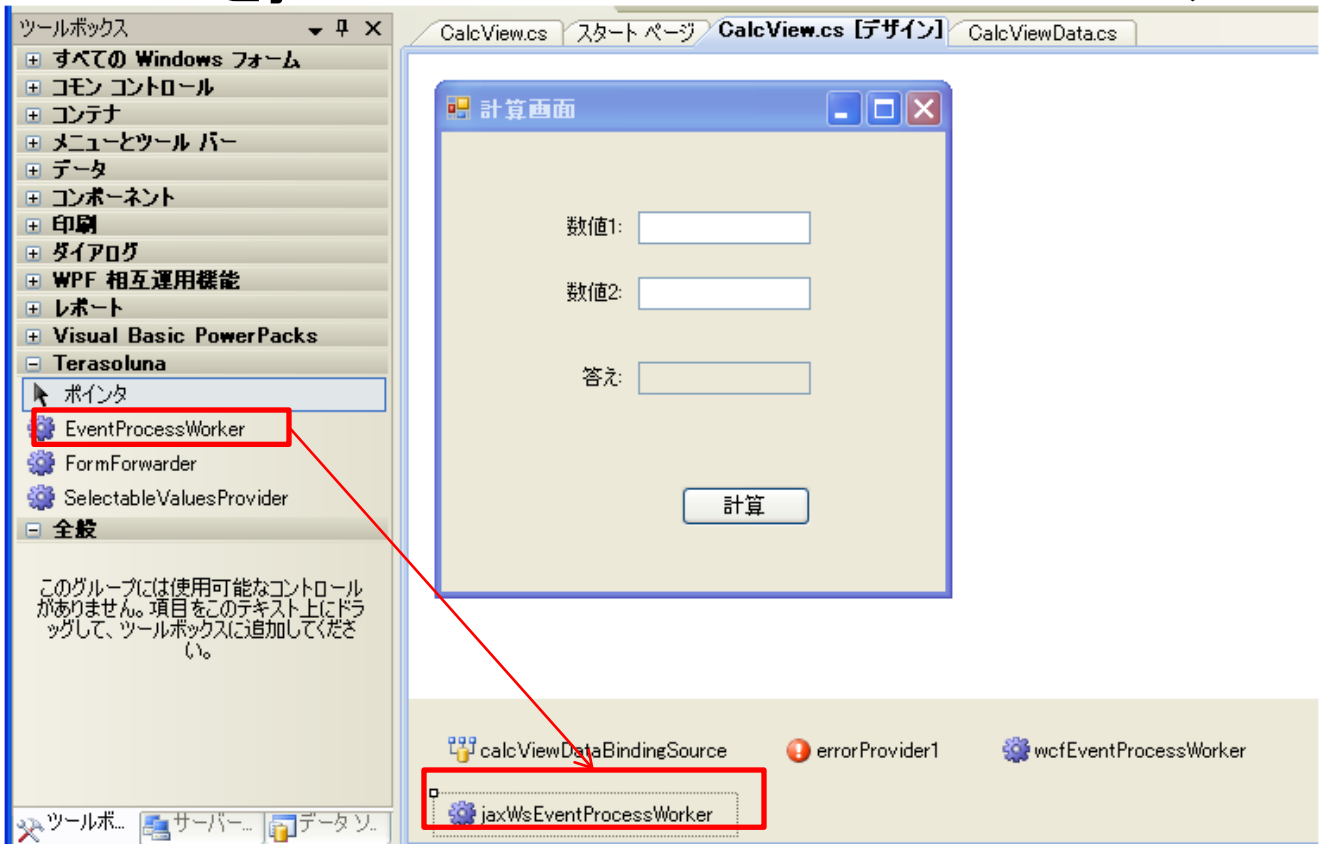
- 「サービス参照の追加」でWCFのプロキシコードを生成します
  - アドレスに「http://localhost:8080/calcsample/CalcService?wsdl」を入力します





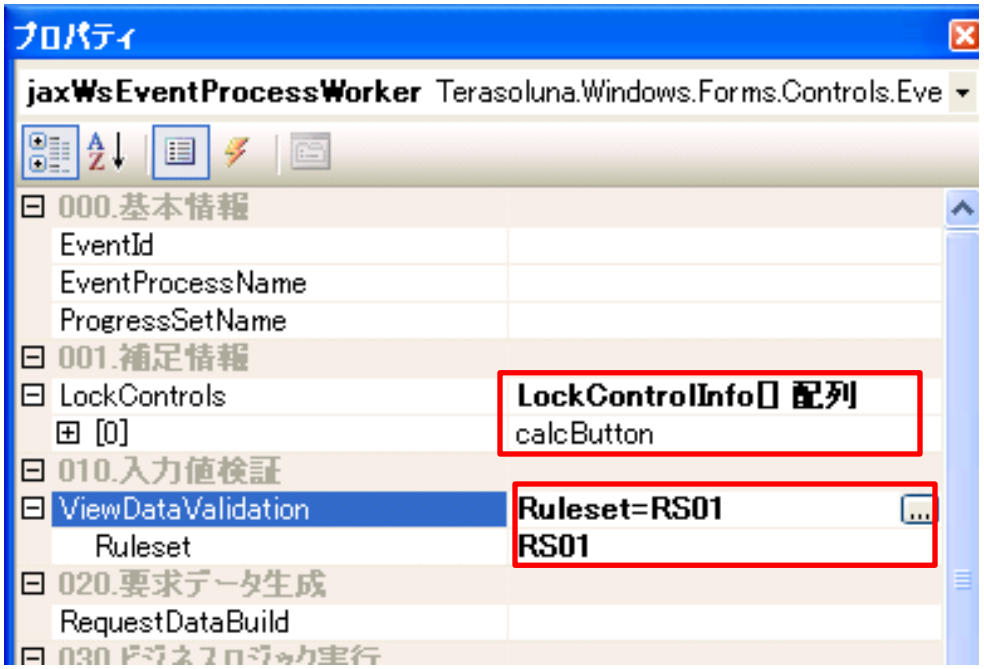
EventProcessWorkerの作成

- 計算画面にEventProccessWorkerを作成します
  - nameをjaxWsEventProcessWorkerとします



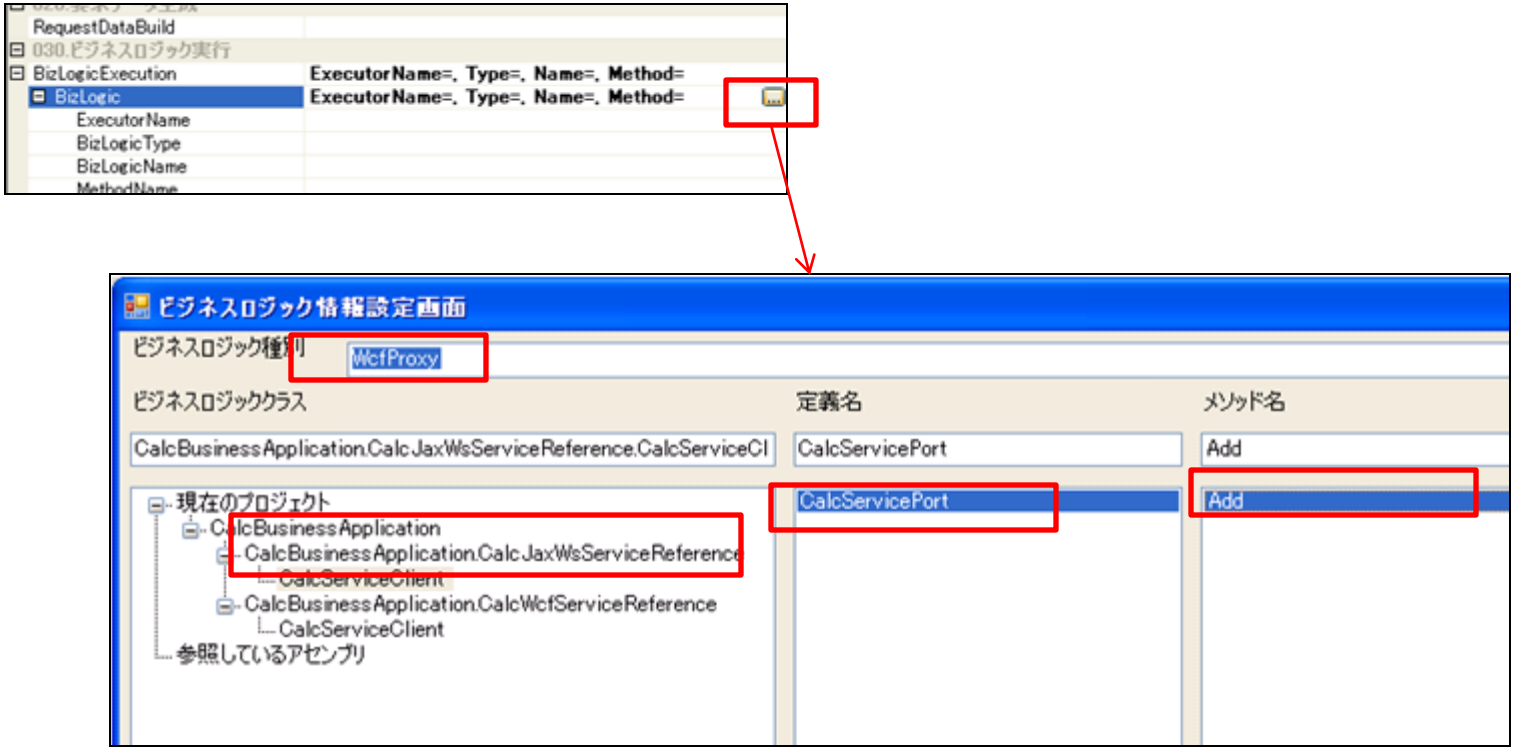
EventProcessWorkerの設定

- LockControlsとViewDataValidationプロパティを設定します
  - wcfEventProcessWorkerと同じ値を設定します



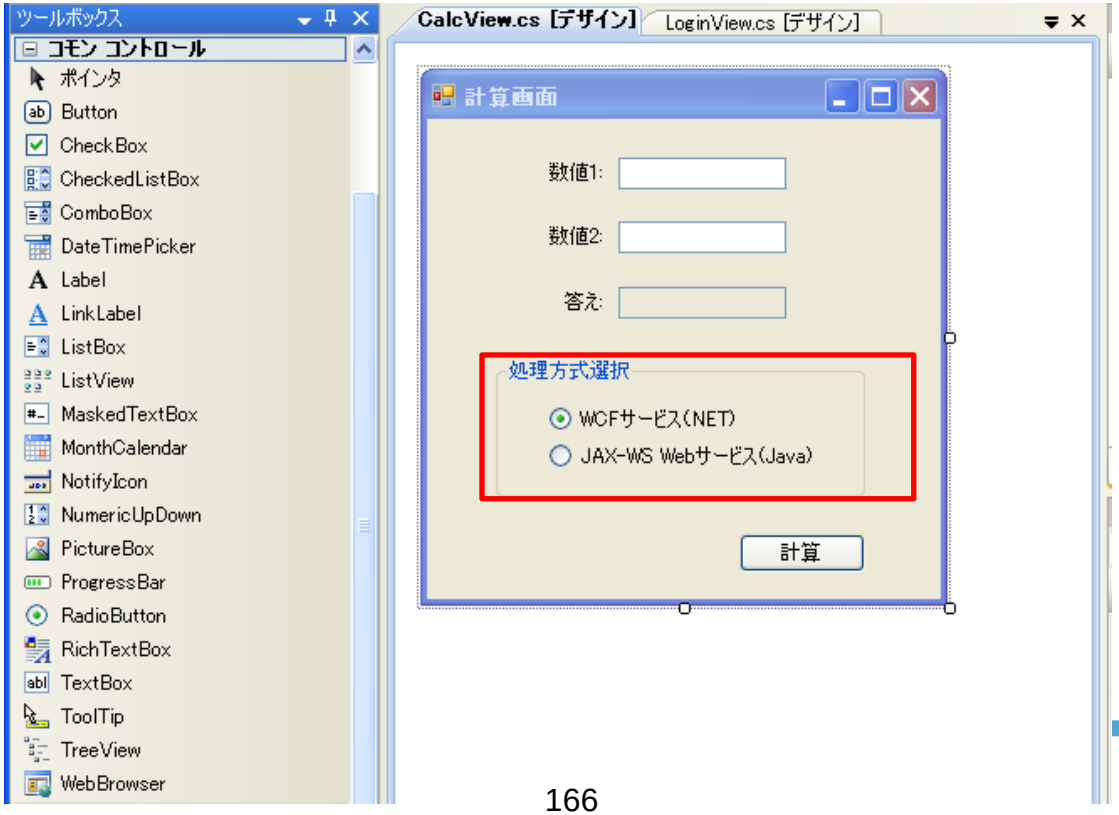
EventProcessWorkerの設定

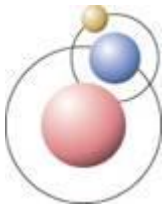
- BizLogicExecutionプロパティにJAX-WS Webサービスから生成したWCFクライアントを設定します



RadioButtonの貼り付け

- GroupBoxおよびRadioButtonを2つ貼り付け処理方式を選択できるようにします
  - WCFサービス用のラジオボタンのnameをwcfRadioButton
  - JAX-WS用のラジオボタンのnameをjaxWsRadioButton



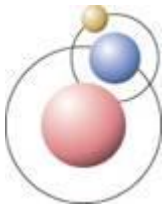


EventProcessWorkerの実行

- 計算ボタンのクリックイベントのハンドラメソッドを修正し、選択したRadioButtonによって処理を振り分けます

```
private void calcButton_Click(object sender, EventArgs e)
        {
            if (wcfRadioButton.Checked)
            {
                wcfEventProcessWorker.RunWorkerAsync();
            }
            else if (jaxWsRadioButton.Checked)
            {
                jaxWsEventProcessWorker.RunWorkerAsync();
            }
        }
```


TERASOLUNA



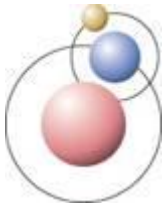
App.configのコピー

- 「サービス参照」の追加により、クライアントの業務個別プロジェクト(CalcBussinessApplication)のApp.configにWCFの設定が生成されるので、再度、起動アプリケーションプロジェクトのApp.configをコピーします

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
          …
    </bindings>
    <client>
      <endpoint address="http://localhost:18080/Service/CalcService.svc"
          binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_CalcService"
          contract="CalcWcfServiceReference.CalcService“  name="WSHttpBinding_CalcService">
        <identity>
          <dns value="localhost" />
        </identity>
      </endpoint>
      <endpoint address="http://localhost:8080/calcsample/CalcService"
          binding="basicHttpBinding" bindingConfiguration="CalcServicePortBinding"
          contract="CalcJaxWsServiceReference.CalcService" name="CalcServicePort" />
    </client>
  </system.serviceModel>
</configuration>
```

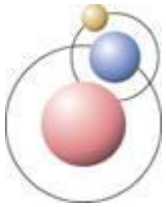
JavaサーバAP接続の動作確認

- Tomcatを起動します
- Visual Studioをデバッグ起動します
- 数値を入力し、計算画面で「JAX-WS Webサービス」のラジオボタンを選択後、計算ボタンを押下します
  - 答えにサーバでの処理結果が表示されて、JavaサーバAPと接続できたことが確認できます



ログイン処理の作成②
（.NETクライアント－.NETサーバの接続
サーバ上でのDBアクセス処理）

TERASOLUNA



DBにアクセスする

- ログイン処理を機能拡張し、WCFサービスでDBにアクセスするビジネスロジックによりユーザ認証できるようにします
- DBアクセスには、ADO.NET(TableAdapter&型付きデータセット)を利用してSQLServer Expressにアクセスします

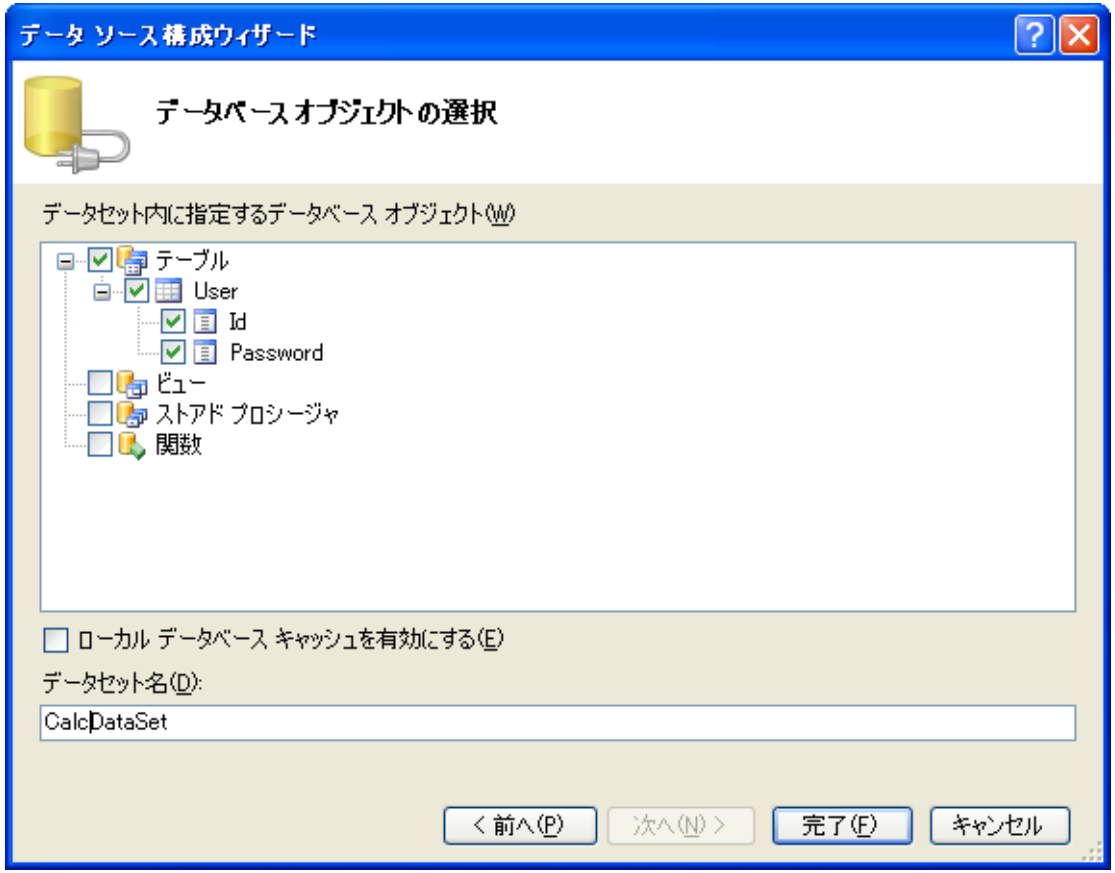
DBの作成

- 完成版チュートリアルAPのCalcDBAccessプロジェクトにあるmdfファイル、ldfファイルを作業中の同フォルダにコピーします
  - SQLServer2008 Expressの場合
    - CalcDBForSql2008.mdf
    - CalcDBForSql2008_log.ldf
  - SQLServer2005 Expressの場合
    - CalcDBForSql2005.mdf
    - CalcDBForSql2005_log.ldf

※チュートリアルの手順および完成版のチュートリアルAPはSQLServer 2008 Expressのmdf/ldfを使った例ですので、SQLServer2005 Expressの場合は、接続文字列のファイル名等は適宜読み替えてください

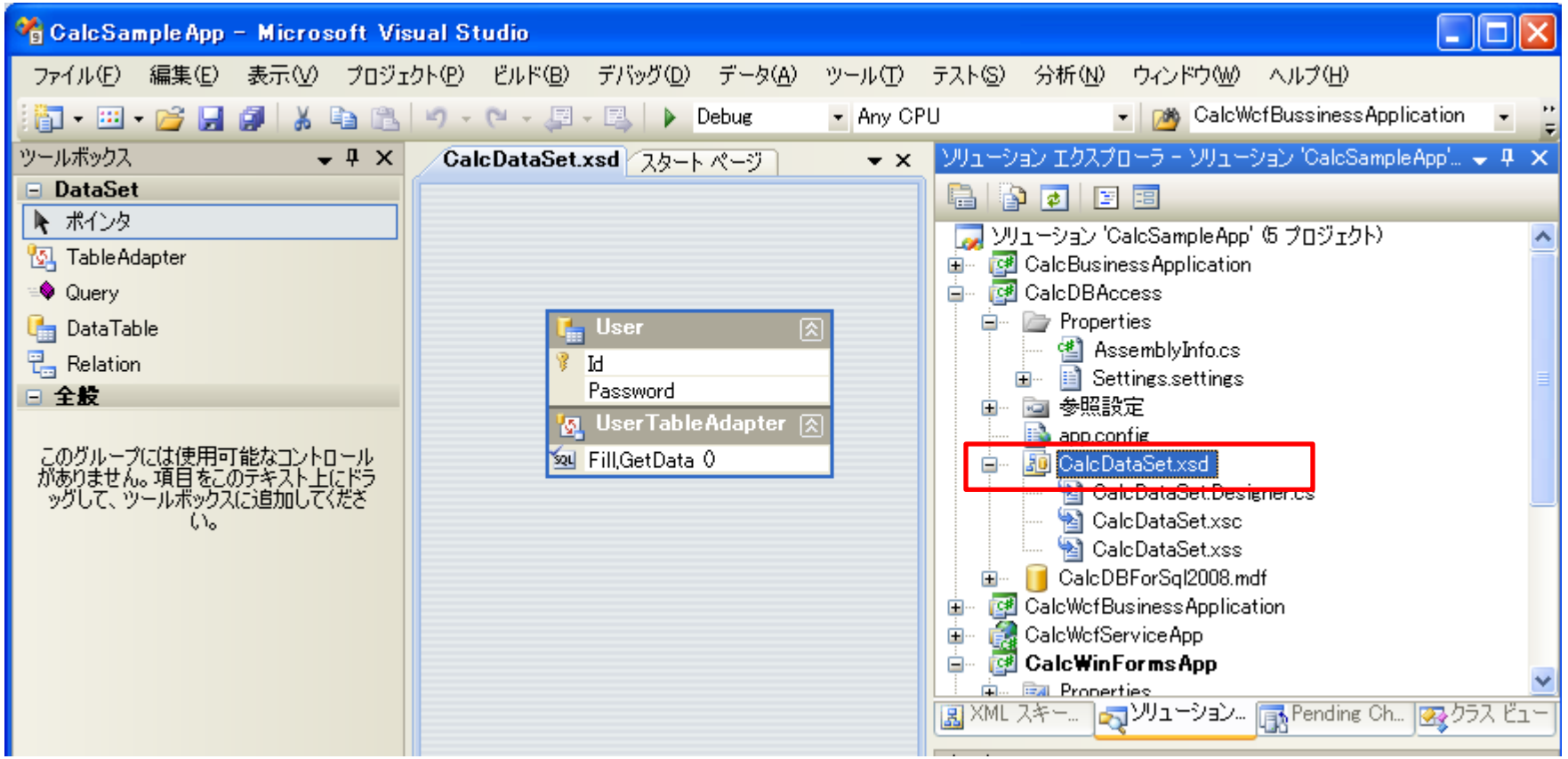
TableAdapter&型付きDataSetの作成

- データソース構成ウィザードが起動するのでUserテーブルを追加します



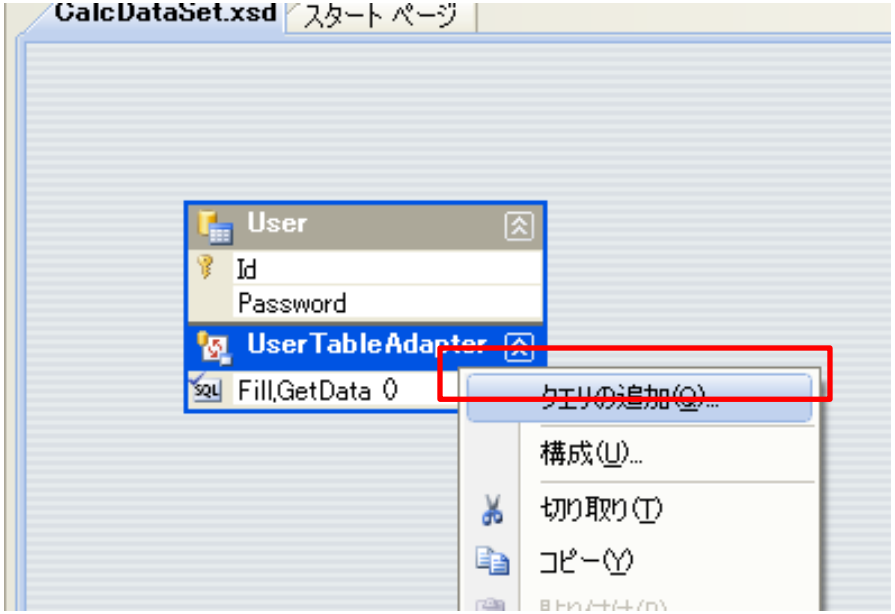
TableAdapter&型付きDataSetの作成

- CalcDataSet.xsdが生成されます
  - UserDataTableとUserTableAdapterが作成されます



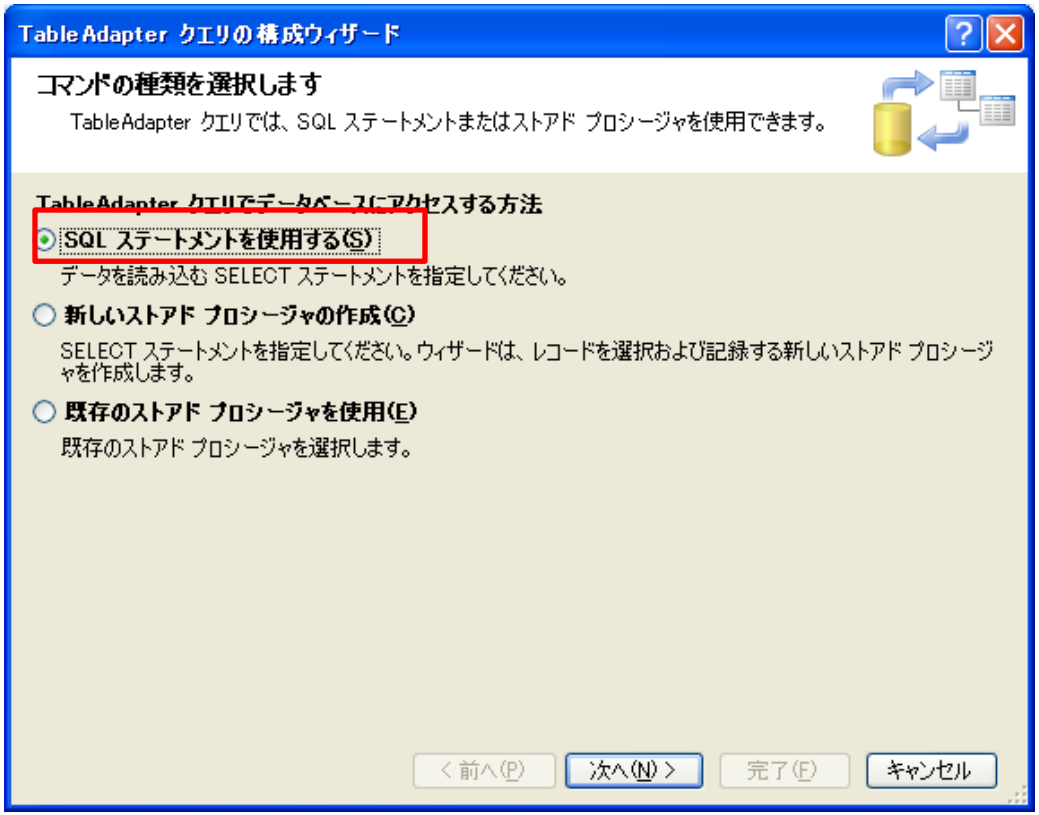
クエリの追加

- ユーザIDとパスワードが一致するユーザを検索するSQLを作成します
  - UserTableAdapterで右クリックし「クエリの追加」を選択



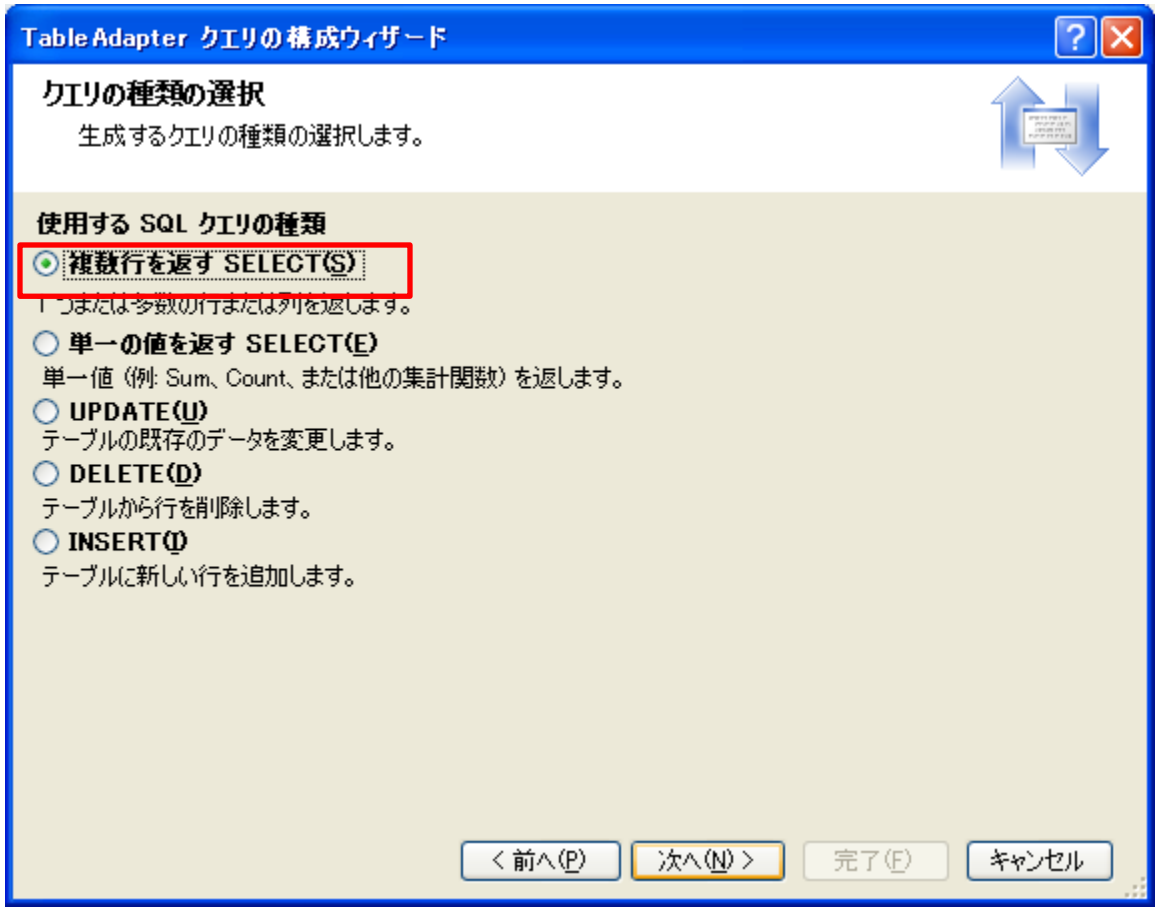
クエリの追加

- 「SQLステートメントを使用する」を選択します



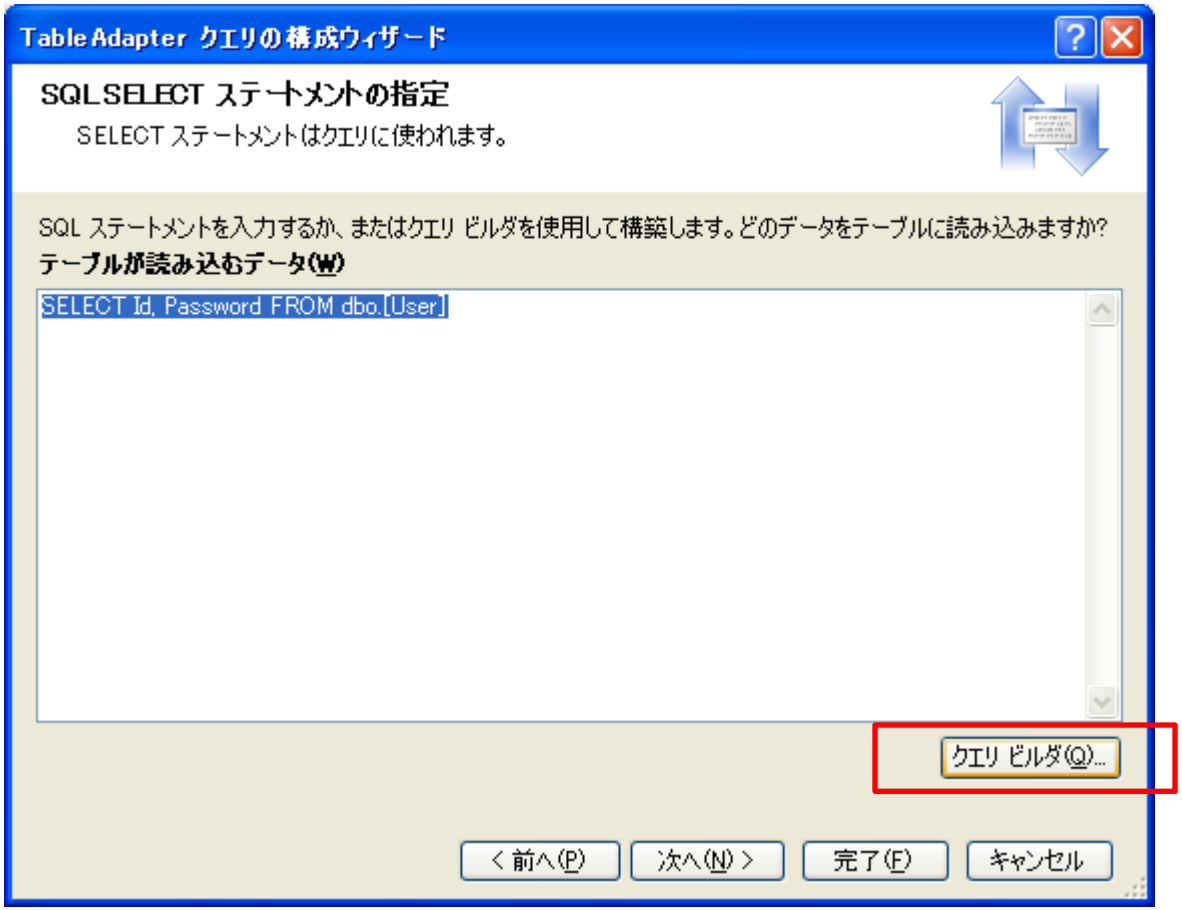
クエリの追加

- 「複数行返すSELECT」を選択します



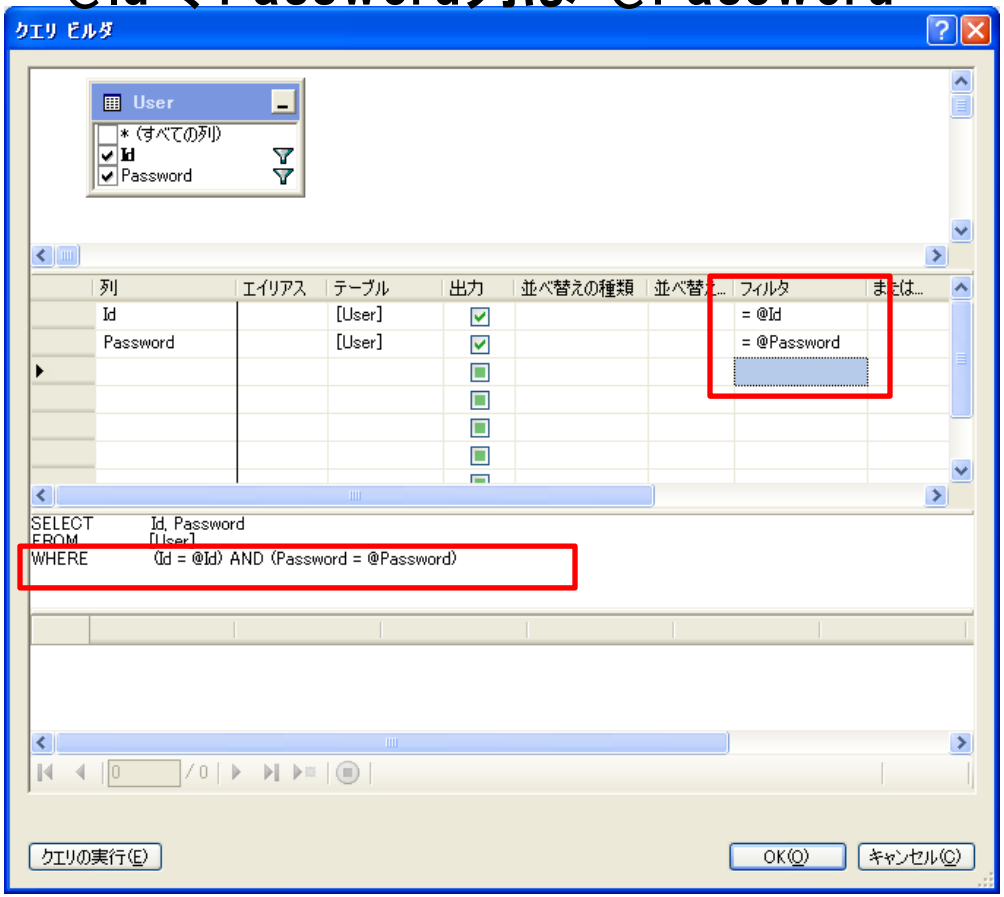
クエリの追加

- 「クエリ ビルダ」ボタンをクリックします



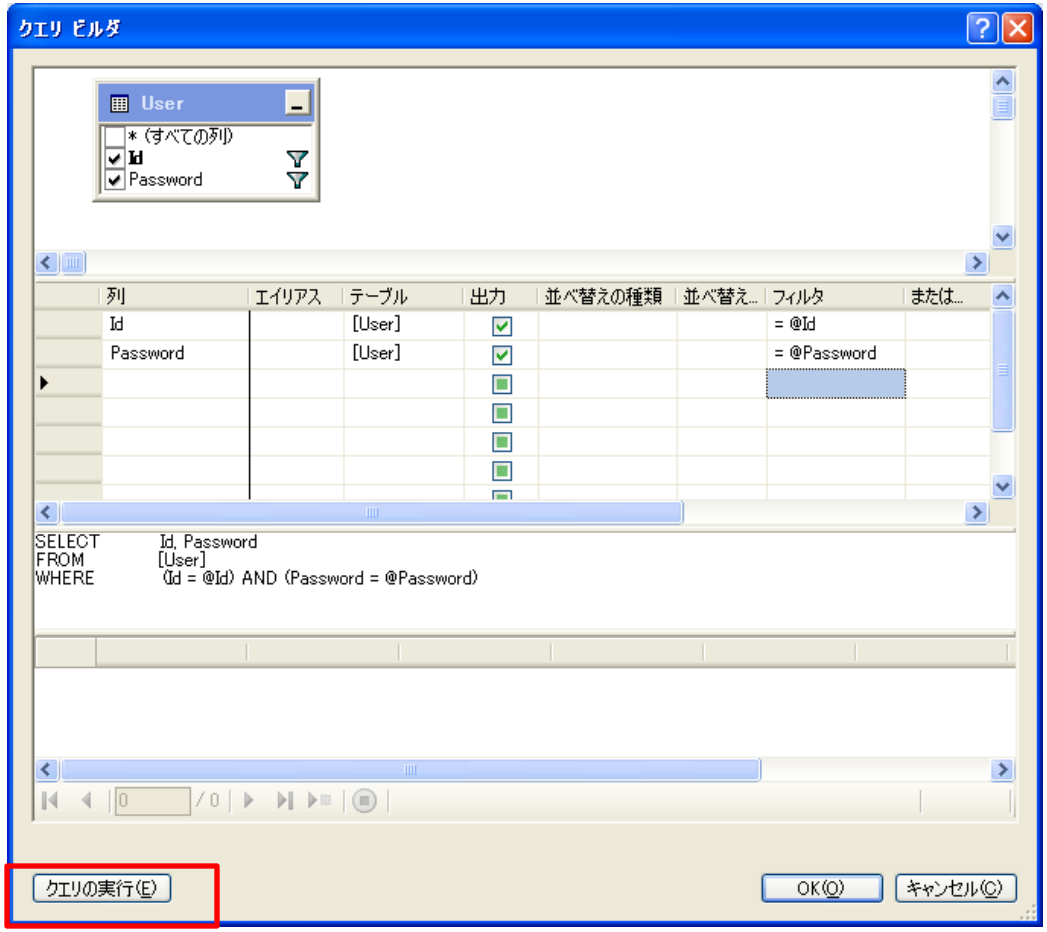
クエリの追加

- フィルタを記述し、クエリにWHERE句を追加します
  - Id列 は“= @Id”、Password列は”@Password”



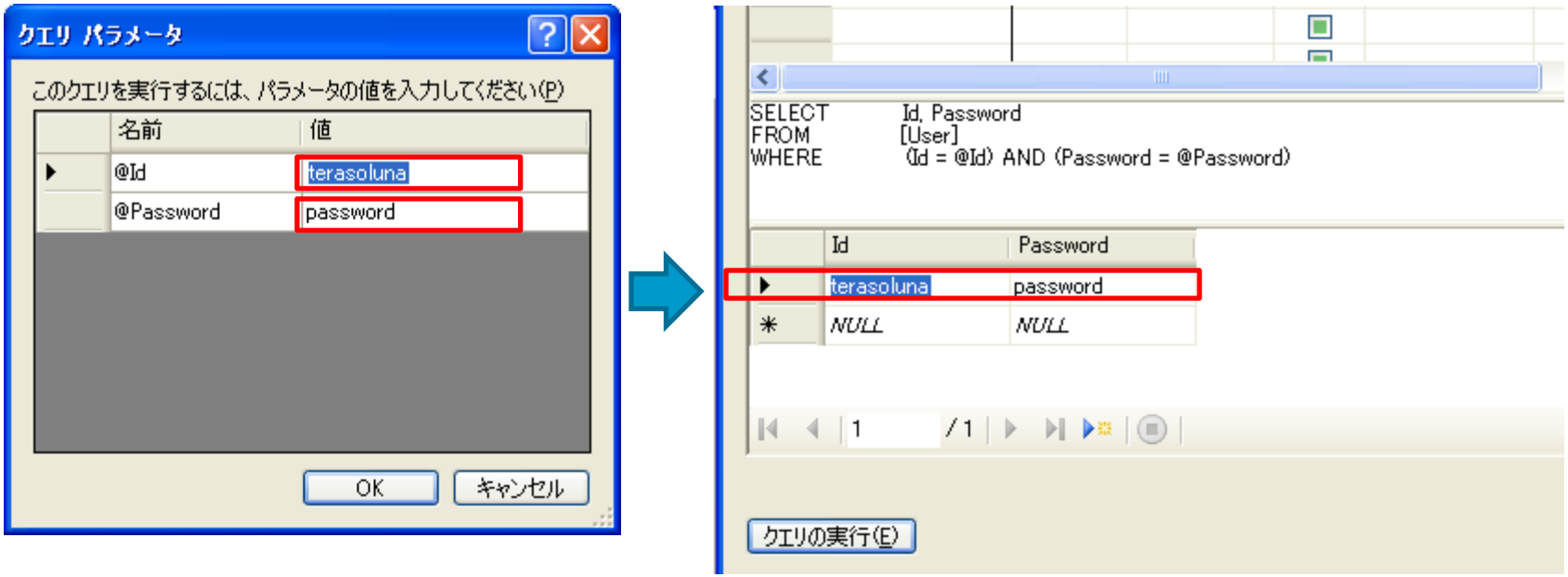
クエリの確認

- 「クエリの実行」ボタンをクリックします



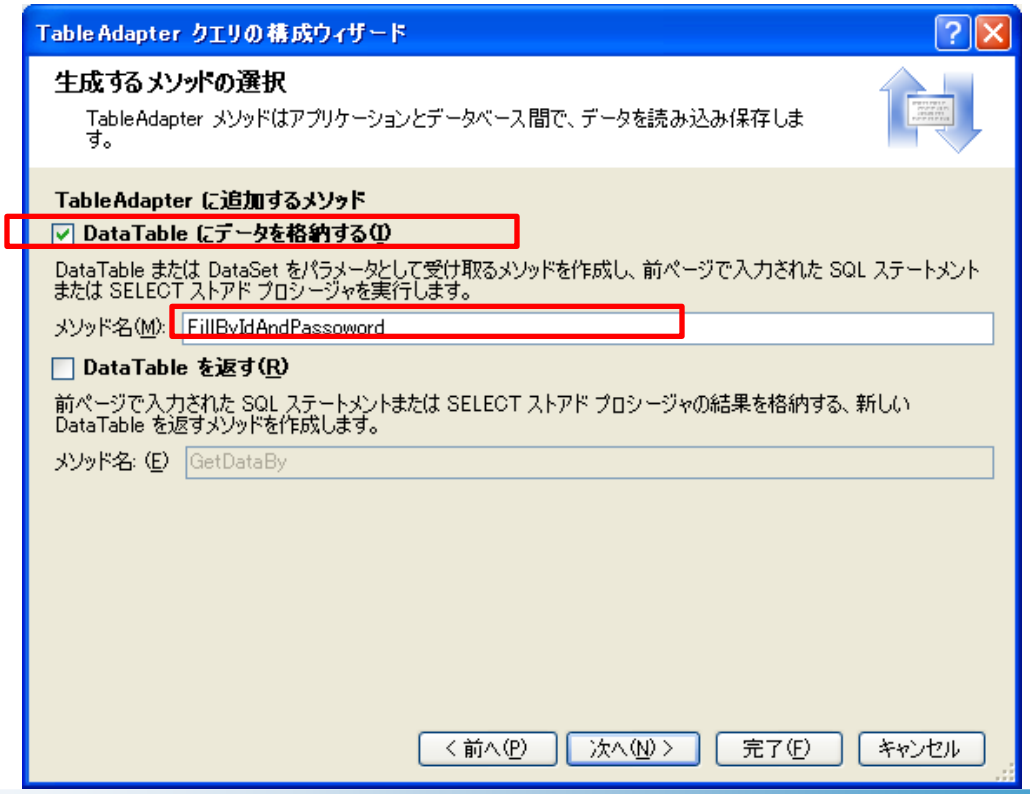
クエリの確認

- @Id =“terasoluna”、@Password =“password”と入力し、クエリを実行します
  - レコードが1件返り、SQLが正しいことを確認します



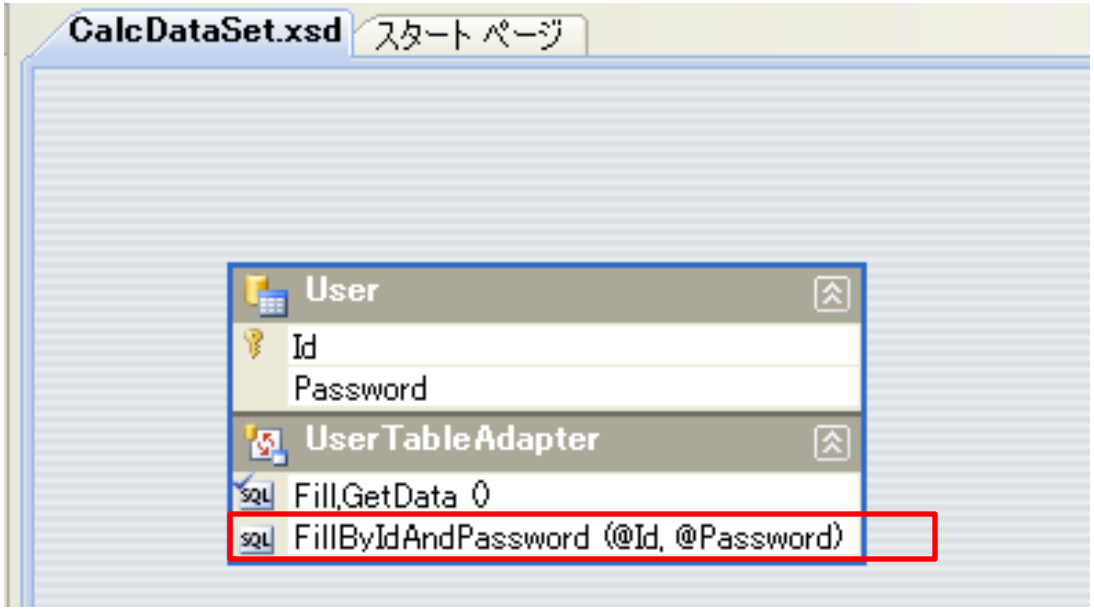
Fillメソッドの作成

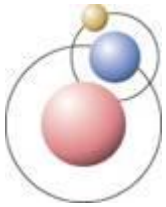
- ウィザードを進んでいき、下記画面で、「DataTableにデータ格納する」のみにチェックし、メソッド名に「FillByIdAndPassword」を入力します



Fillメソッドの作成

- デザイナ上でFillByIdAndPasswordメソッドが追加されたことが確認できます





Web.configへのコピー

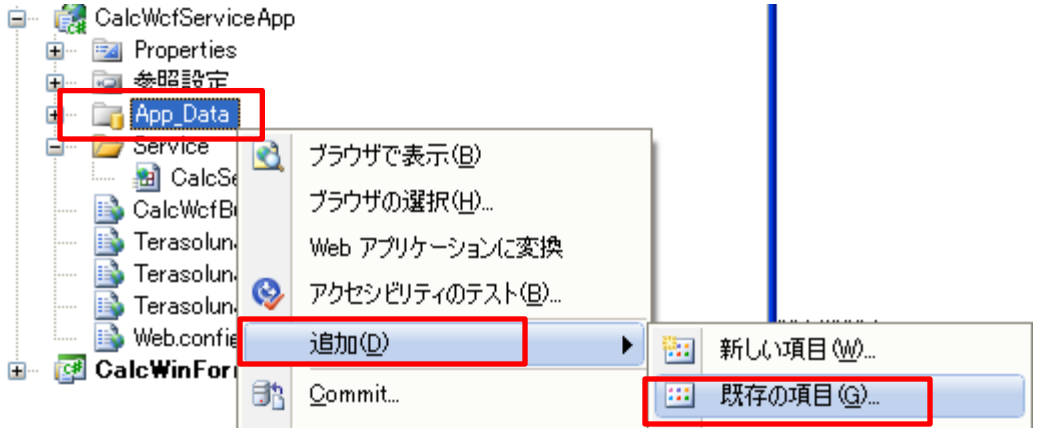
- データアクセスプロジェクト(CalcDBAccess)のApp.configに生成された接続文字列の設定(<connectionStrings>要素>)をWCFサービスアプリケーションのWeb.configにコピーします

```
<?xml version="1.0"?>
<configuration>
  <!-- DBの接続文字列 -->
  <connectionStrings>
    <add
name="CaclDBAccessClassLibrary.Properties.Settings.CalcDBForSql2008ConnectionString"
        connectionString="Data
Source=.¥SQLEXPRESS;AttachDbFilename=|DataDirectory|¥CalcDBForSql2008.mdf;Integrated
Security=True;User Instance=True"
        providerName="System.Data.SqlClient" />
  </connectionStrings>
...
```


TERASOLUNA

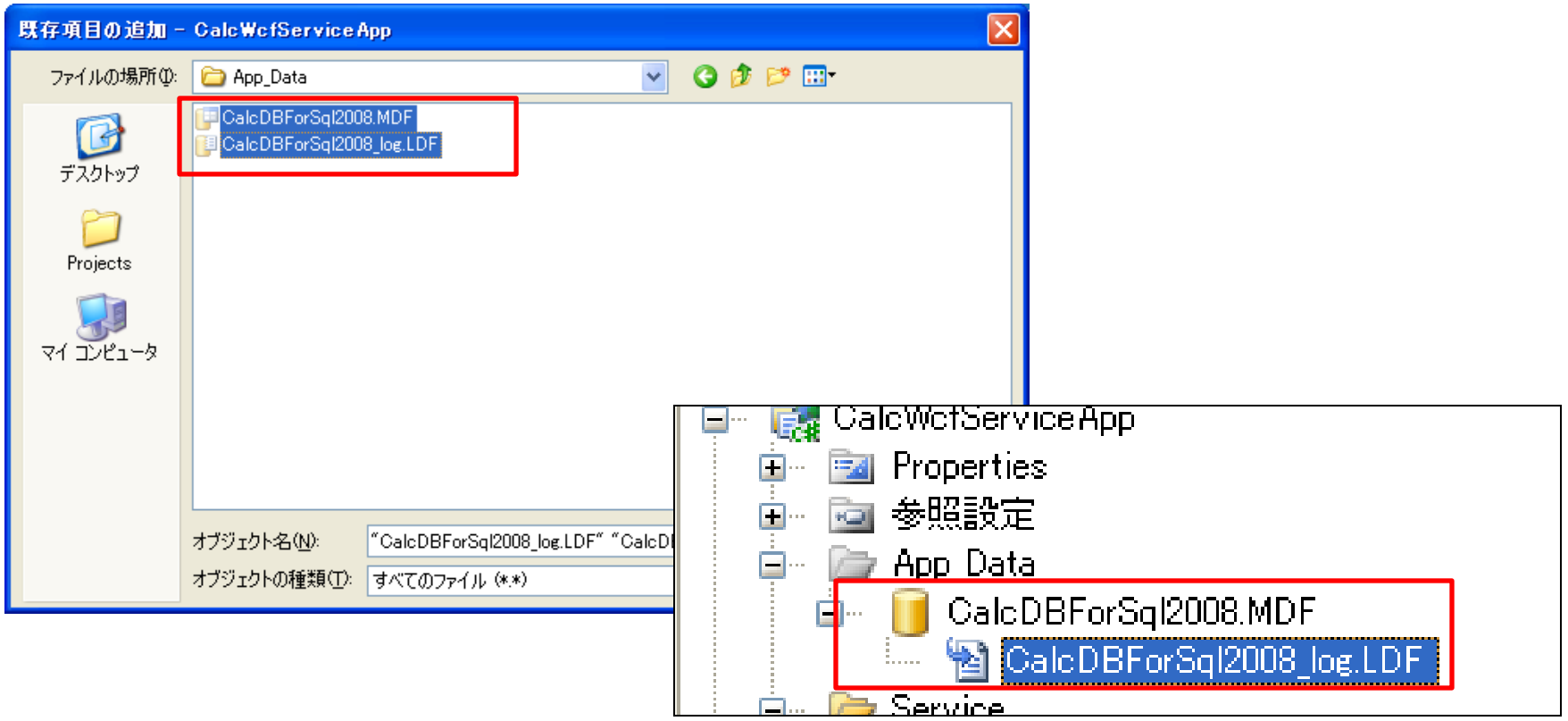
mdf/ldfファイルの追加

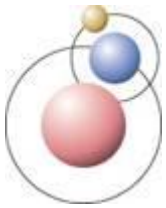
- ソリューションをビルドします
  - データアクセスプロジェクト(DBAccess)のビルドイベントにより mdf/ldfファイルがWCFサービスアプリケーションプロジェクト (CalcWcfServiceApp)のApp_Dataフォルダにコピーされます
- App_Dataフォルダで、「追加」-「既存の項目」でmdf/ldfファイルをプロジェクトに追加します



mdf/ldfファイルの追加

- App_Dataフォルダに生成されたmdf/ldfファイルを選択します

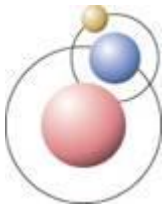




サーバ入力DTOの作成

- 業務個別プロジェクト(WcfBussinessApplication)に、サーバ入力DTO(LoginInputDto)を作成します

```
namespace CalcWcfBusinessApplication.Dto
{
    [DataContract]
    public class LoginInputDto
    {
        [DataMember]
        [RequiredValidator(Tag = "ユーザID", Ruleset = "RS01")]
        public string UserId { get; set; }

        [DataMember]
        [RequiredValidator(Tag = "パスワード", Ruleset = "RS01")]
        public string Password { get; set; }
    }
}
```



WCFサービスクラスの作成

- WCFサービスコントラクト(CalcService)に、以下のようにLoginメソッドを追加で実装します

```
namespace CalcWcfBusinessApplication.Service
{
    [ServiceContract]
    public class CalcService
    {
        ...
        [OperationContract]
        [ValidationBehavior(RuleSet = "RS01")]
        void Login(LoginInputDto inputDto)
        {
        }
    }
}
```


TERASOLUNA

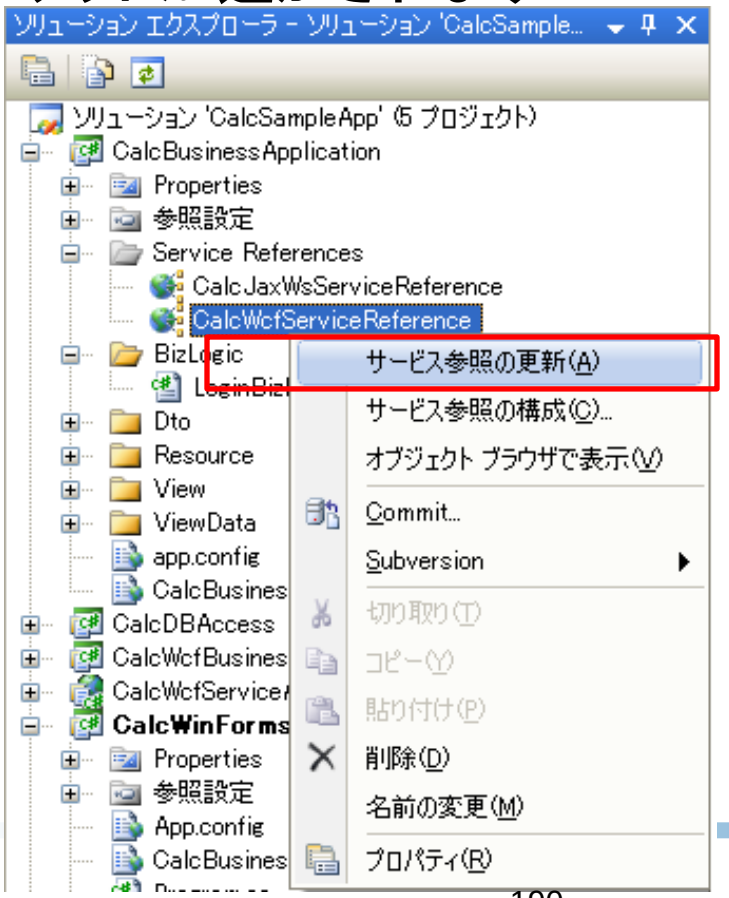
WCFサービスクラスの作成

- WCFサービスコントラクト(CalcService)に、以下のようにLoginメソッドを追加で実装します

```
[OperationContract]
[ValidationBehavior(RuleSet = "RS01")]
public void Login(LoginInputDto inputDto)
{
    ///自動トランザクションの設定
    using (TransactionScope scope = new TransactionScope())
    {
        UserTableAdapter ta = new UserTableAdapter();
        CalcDataSet.UserDataTable userTable = new CalcDataSet.UserDataTable();
        ta.FillByIdAndPassword(userTable, inputDto.UserId, inputDto.Password);
        if (userTable.Count == 0)
        {
            ///IDとパスワードが合致しなかった場合、業務エラーとしてBizLogicExceptionをスロー
            throw new BizLogicException(
                BizLogicExceptionErrorType.BizLogicFailure,
                Resources.ERROR_CALC_MSG002,
                new List<ErrorInfo>()
                {
                   new ErrorInfo("ERROR_CALC_MSG003", null,  ResourcesERROR_CALC_MSG003, null)
                });
        }
    }
}
```

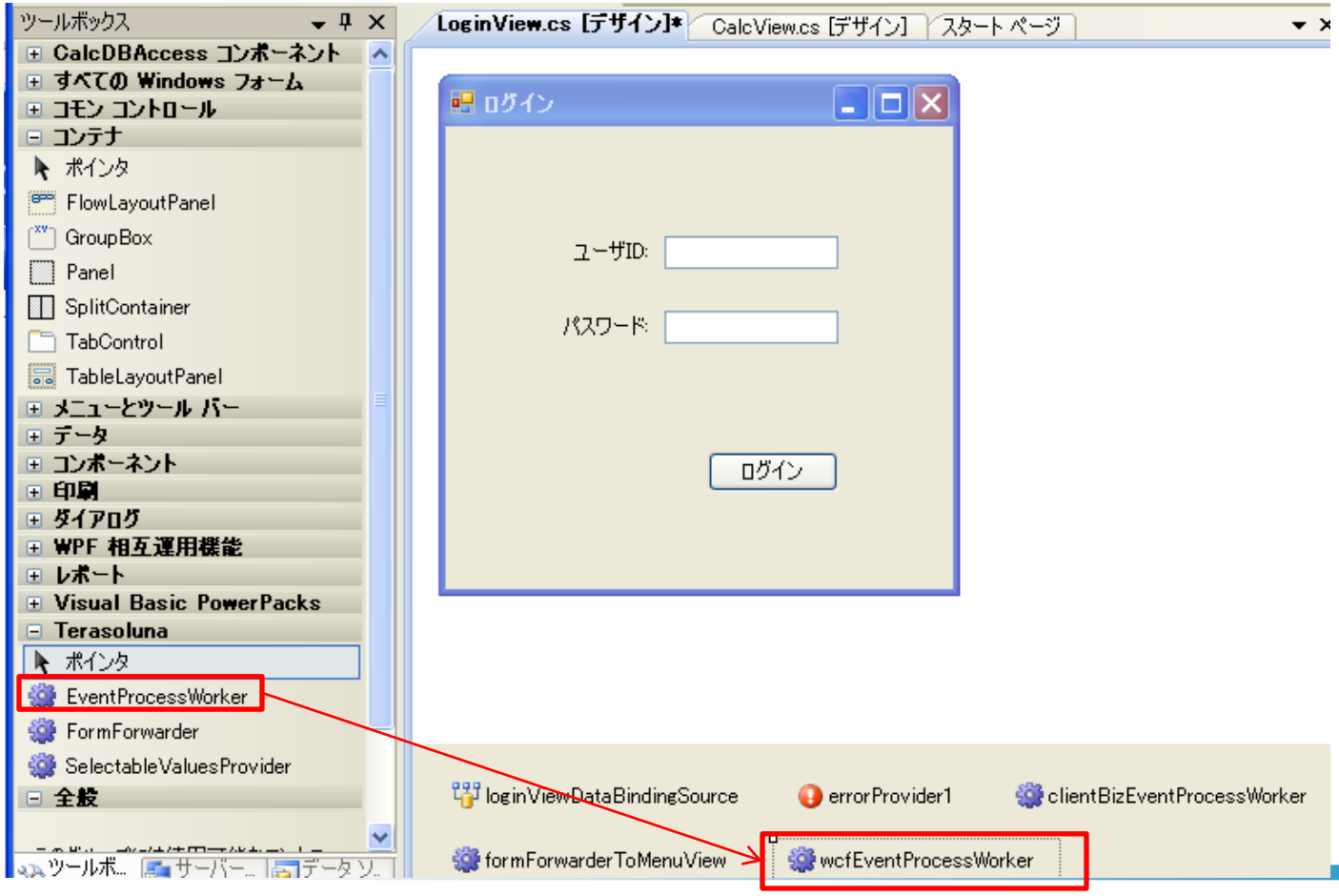
サービス参照の更新

- クライアントの個別業務AP(CalcBusinessApplication)で作成したサービス参照(CalcWcfServiceReference)を更新します
  - Loginメソッドが追加されます



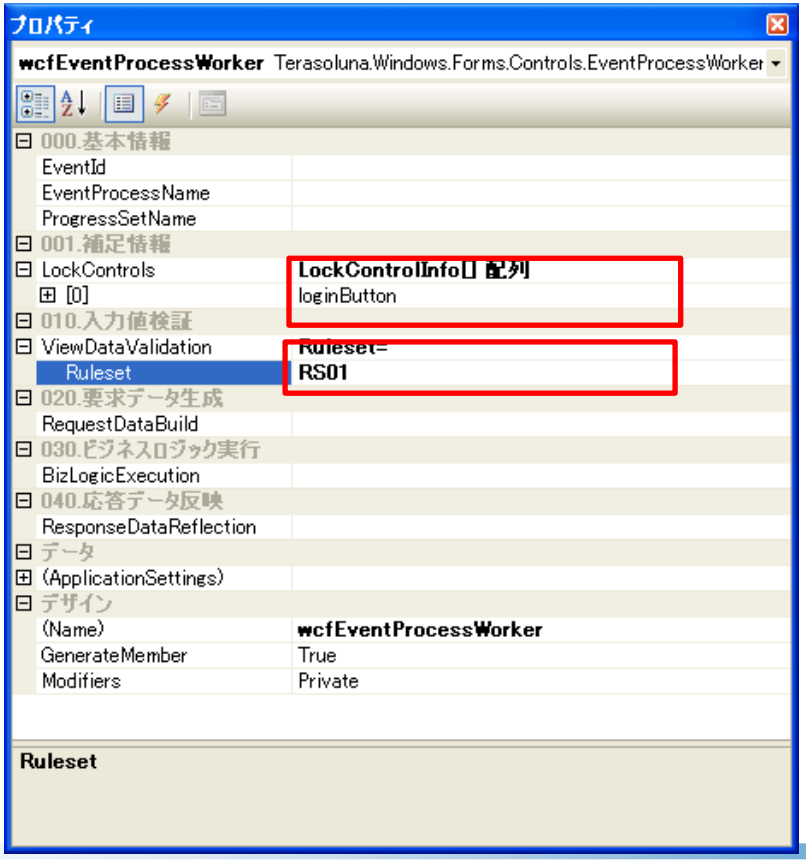
EventProcessWorkerの作成

- ログイン画面にEventProcessWorkerを作成します
  - nameを「wcfEventProcessWorker」にします



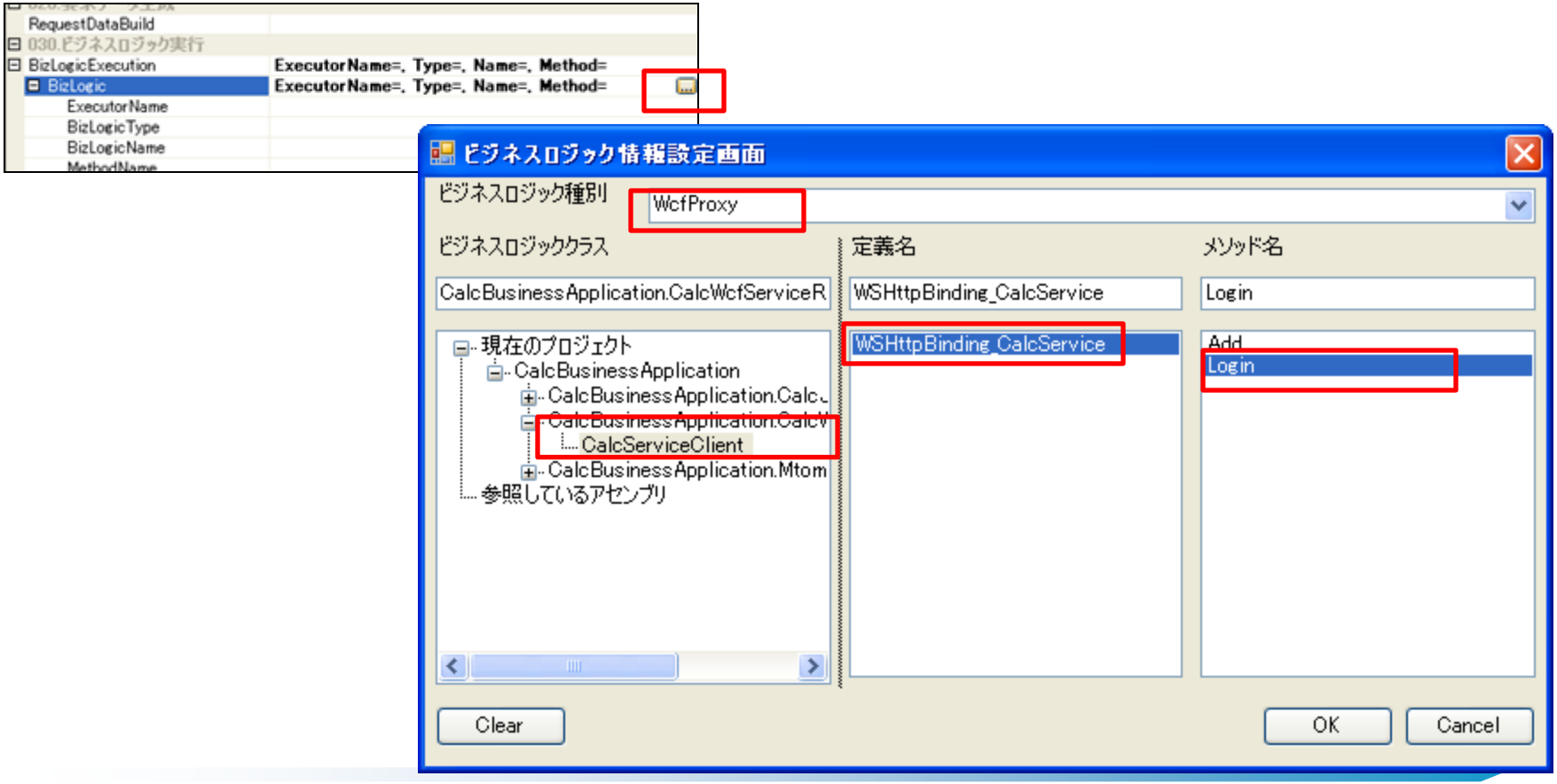
EventProcessWorkerの設定

- LockControlsとViewDataValidationプロパティを設定します
  - LockControlsプロパティ
  - ViewDataValidationプロパティ



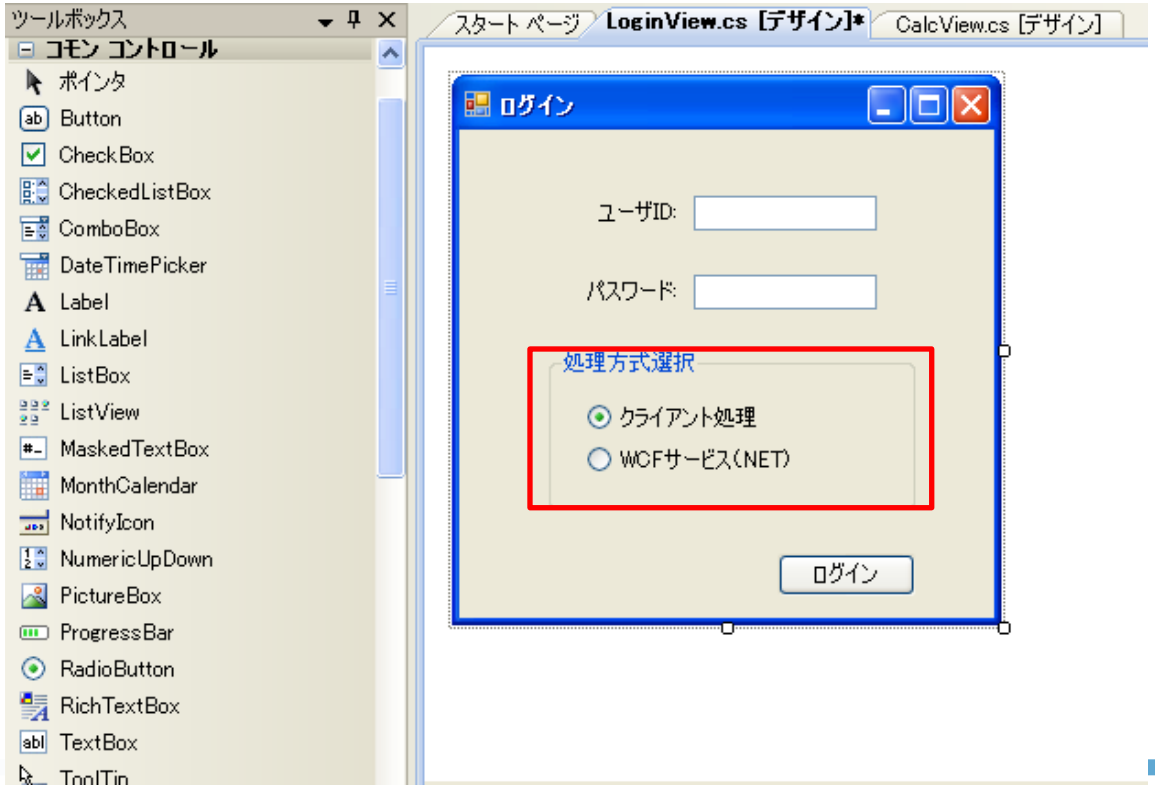
EventProcessWorkerの設定

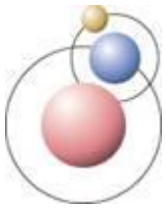
- BizLogicExecutionプロパティに、実行するWCFプロキシを設定します
  - 追加されたLoginメソッドを選択します



RadioButtonの貼り付け

- GroupBoxとRadioButtonを2つ貼り付け処理方式を選択できるようにします
  - クライアント処理用のラジオボタンのnameをclientBizRadioButton
  - WCFサービス用のラジオボタンのnameをwcfRadioButton





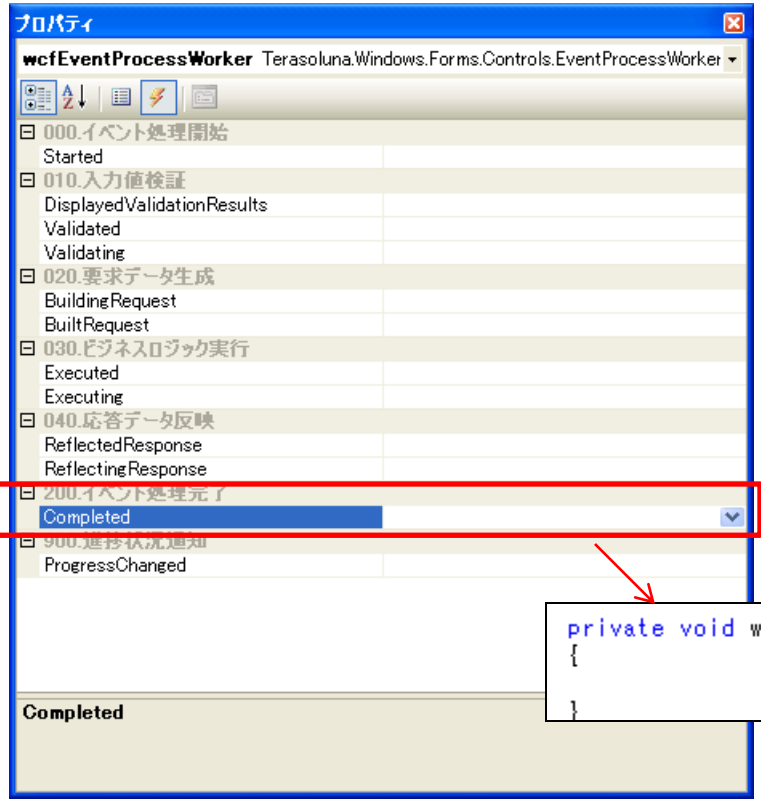
EventProcessWorkerの実行

- ログインボタンのクリックイベントのハンドラメソッドを修正し、選択したRadioButtonによって処理を振り分けます

```
private void loginButton_Click(object sender, EventArgs e)
{
    if (clientBizRadioButton.Checked)
    {
        EventProcessResult result = clientBizEventProcessWorker.RunWorker();
        if (result.IsSuccess)
        {
            Hide();
            formForwarderToMenuView.Forward();
        }
    }
    else if (wcfRadioButton.Checked)
    {
        wcfEventProcessWorker.RunWorkerAsync();
    }
}
```

EventProcessWorkerのイベントハンドラメソッド実装

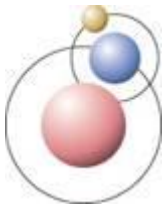
- wcfEventProcessWorkerは、非同期実行(RunWorkerAsyncメソッド)で実行するため、イベントの処理結果を確認するには、EventProcessWorker.Completedイベントを実装して確認します



ダブルクリック

イベントハンドラを生成

```
private void wcfEventProcessWorker_Completed(object sender, EventProcCompletedEv
{

}
```



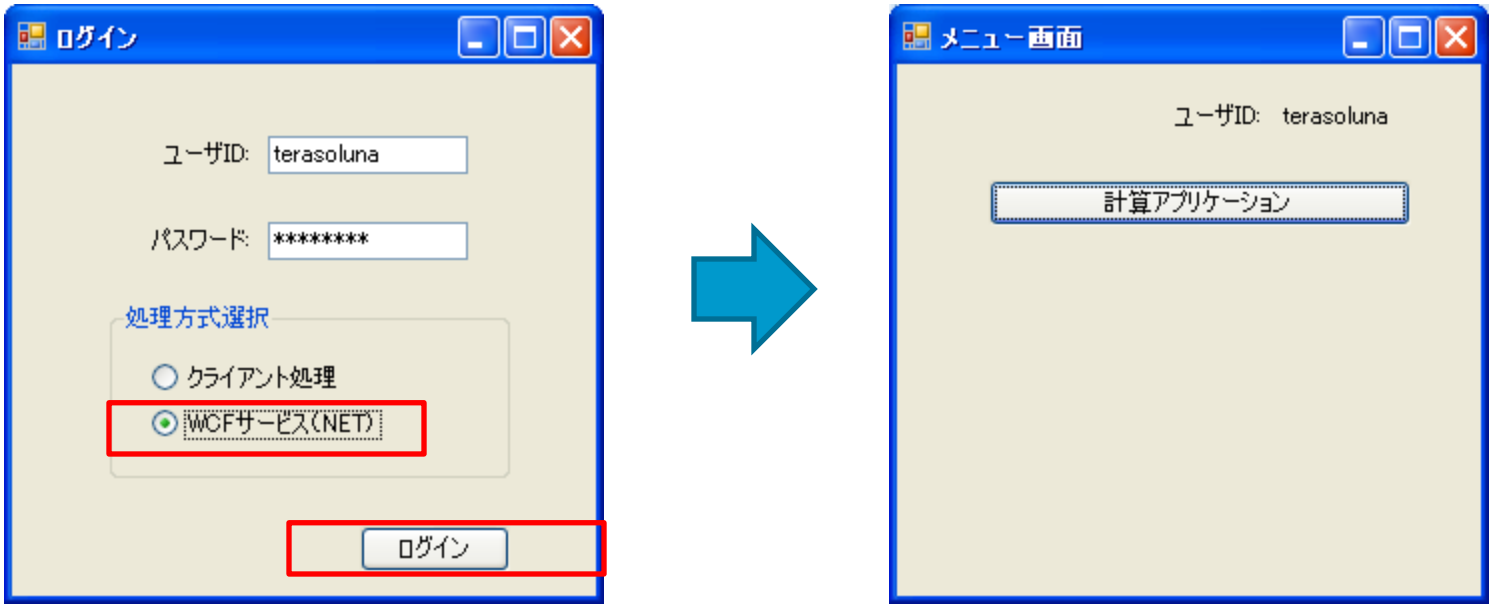
EventProcessWorkerのイベントハンドラメソッド実装

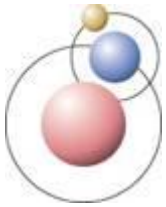
- Completedイベントを以下のように実装します
  - clientBizEventProcessWorkerの同期処理実行と同様に、処理結果成功時に画面遷移処理を実装します
  - 処理結果は、EventArgsのResultプロパティを使用して確認できます

```
private void wcfEventProcessWorker_Completed(object sender,
                                             EventProcCompletedEventArgs e)
{
    if (e.Result.IsSuccess)
     {
            Hide();
            formForwarderToMenuView.Forward();
      }
}
```

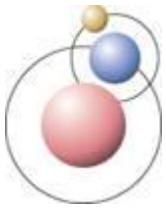
DBアクセス処理の動作確認

- Visual Studioをデバッグ起動します
  - WCFサービス用のラジオボタンを選択しログインボタンを押下します
    - ログインに成功しメニュー画面へ遷移します
  - WCFサービスアプリケーションからDBアクセスを実施し、ログインに成功することが分かります





Click Onceの実行



Click Onceの実行

- Click Onceにより簡単にクライアントAPを配信することができます
  - TERASOLUNAで作成したクライアントAPもClick Onceを利用可能です
- チュートリアルではVisual Studioの機能を使った簡単なClick Onceの手順を示します
  - .NETクライアントAPのみ実装した場合は、.NETクライアントをIIS上に発行し、Click Onceでインストールし、動作確認します
  - .NETサーバAPも実装した場合はIIS上に発行し、クライアントと接続する手順を実施します
  - クライアントAPに.NET-JAVA接続を実装した場合には、Tomcat上のJAVAサーバAPに接続する手順を実施します

Click Onceの実行

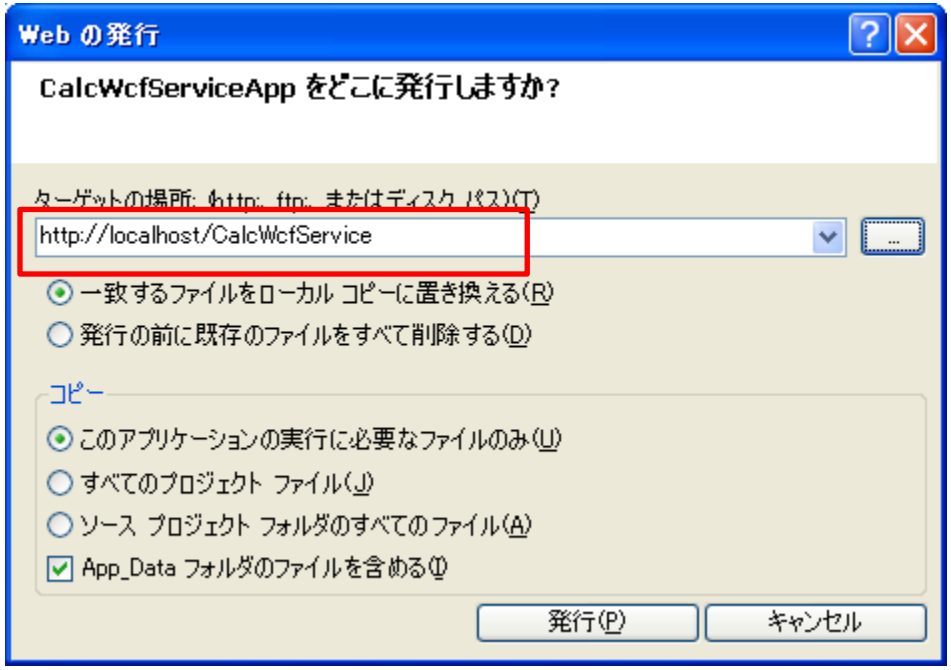
- IISの環境構築
  - 本手順を実行するためには、IISをインストールします
  - .NETサーバAPが、DB(SQLServer Express)へ統合Windows認証でアクセスするため、IISのプロセスワーカの実行ユーザをSQLServerへアクセス可能なユーザに設定する必要があります
    - .NETサーバAPをISSに発行する場合に必要な手順です
    - IIS5の場合
      - 「C:¥WINDOWS¥Microsoft.NET¥Framework¥v2.0.50727¥CONFIG¥machine.config」のprocessModel要素にuserName、password要素を追記します
        - <processModel autoConfig="true" userName="ユーザ名" password="パスワード"/>
      - 修正後、「コントロールパネル」-「管理ツール」-「サービス」より、「IIS Admin」を再起動します
    - IIS6、IIS7の場合の手順
      - 「インターネットインフォメーションサービス(IIS)マネージャ」から、「アプリケーションプール」のプロパティの「識別」タブで「セキュリティアカウント」を変更します

.NETサーバAPの発行

- .NETサーバAPの場合は、Visual Studioより発行し、IISに配備します

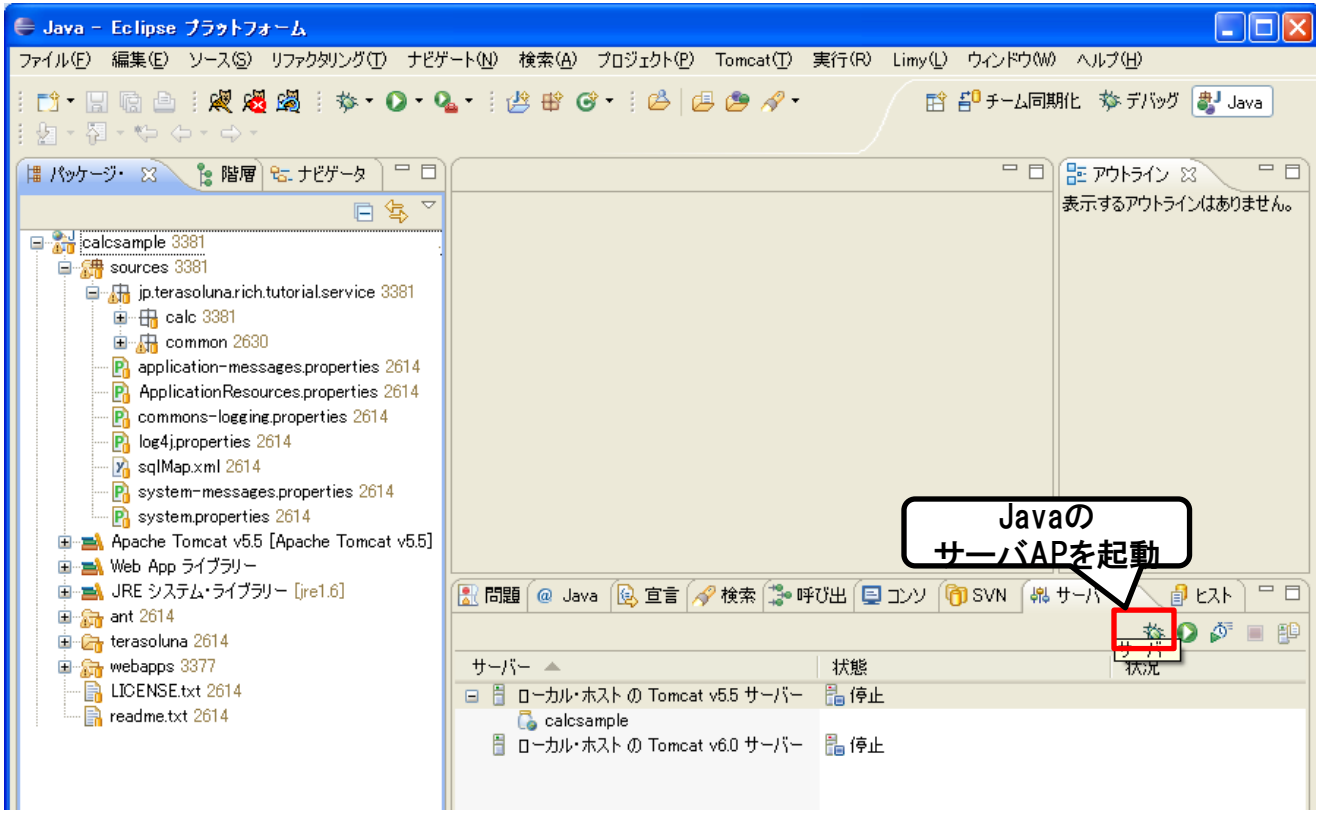
.NETサーバAPの発行

- 発行先URLを指定し、発行ボタンをクリックします



JavaサーバAPの起動

- JavaサーバAPは、いままでの手順と同様EclipseのWTPまたはAntではAPを配備しTomcatを起動します



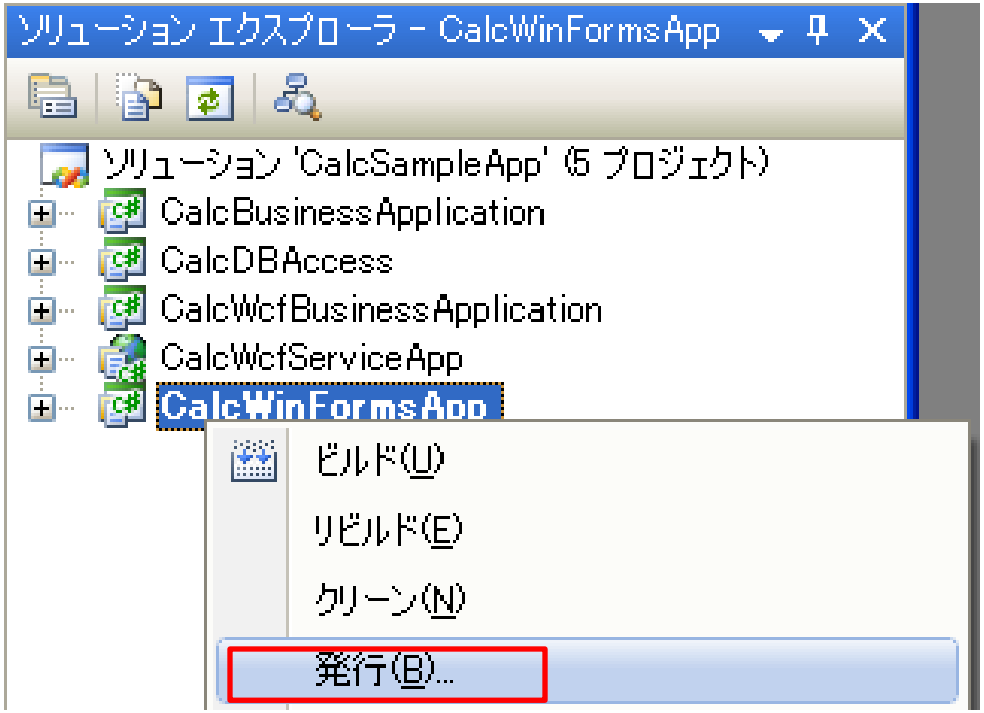
.クライアントAPの設定変更（.NETサーバAPの接続先URL置換の設定）

- 起動アプリケーションプロジェクト(CalcWinFormsApp)のTerasolunaApplication.configに以下を追記し、.NETサーバ接続先URLをIIS上のサーバAPのURLに一括置換する設定をします(“http://localhost:18080/”→“http://localhost/CalcWcfServce/”)
  - 「サーバ通信機能」が提供する機能です

```
...
 <containers>
    <container>
     <types>
      <!-- サーバ通信用のアドレス置換設定 -->
      <type name="localhostToDnsName"
           type="IAddressReplacer" mapTo="MappingAddressReplacer">
       <lifetime type="singleton" />
       <typeConfig>
        <property name="Order" propertyType ="int">
         <value value="0" type="int"/>
        </property>
        <property name="FromUri" propertyType ="Uri">
         <value value="http://localhost:18080/" type="Uri"/>
        </property>
        <property name="ToUri" propertyType ="Uri">
         <value value="http://localhost/CalcWcfService/" type="Uri"/>
        </property>
       </typeConfig>
      </type>
     </types>
```

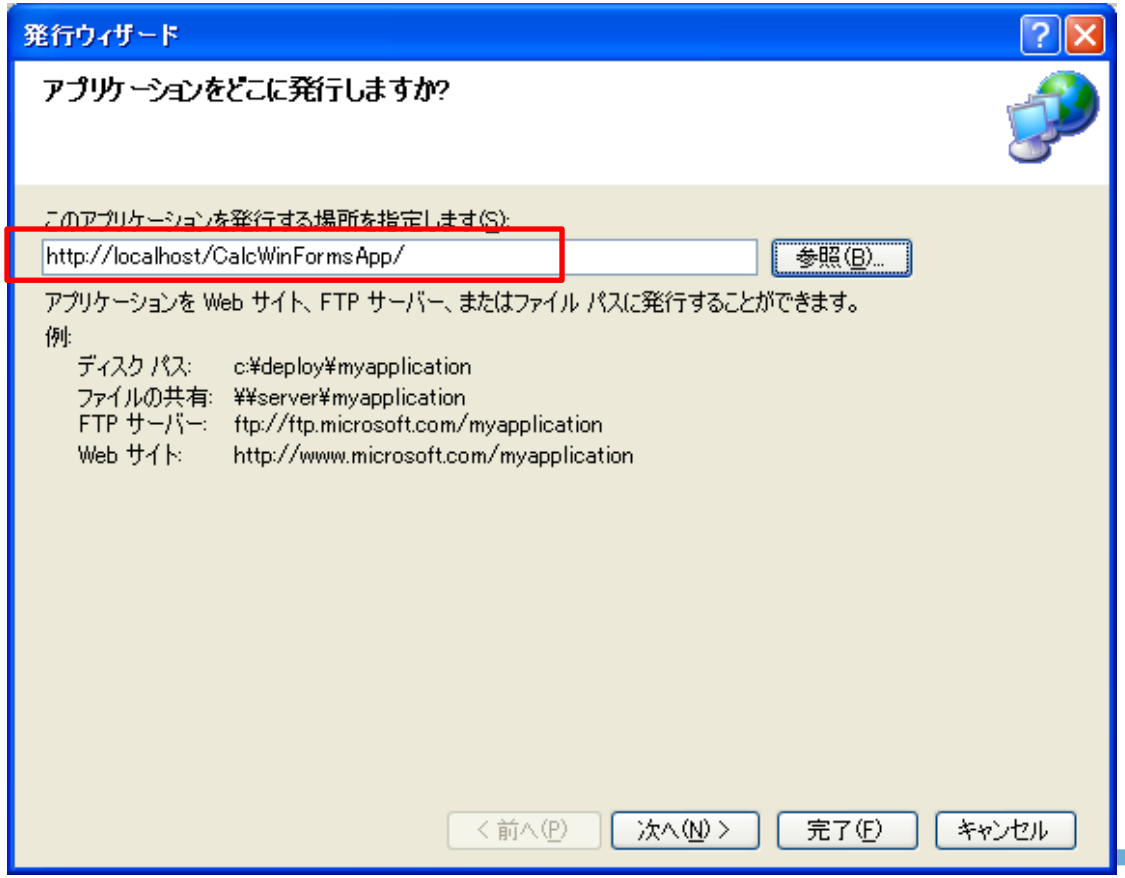
クライアントAPの発行

- 起動アプリケーションプロジェクトで右クリックし、「発行」を選択します



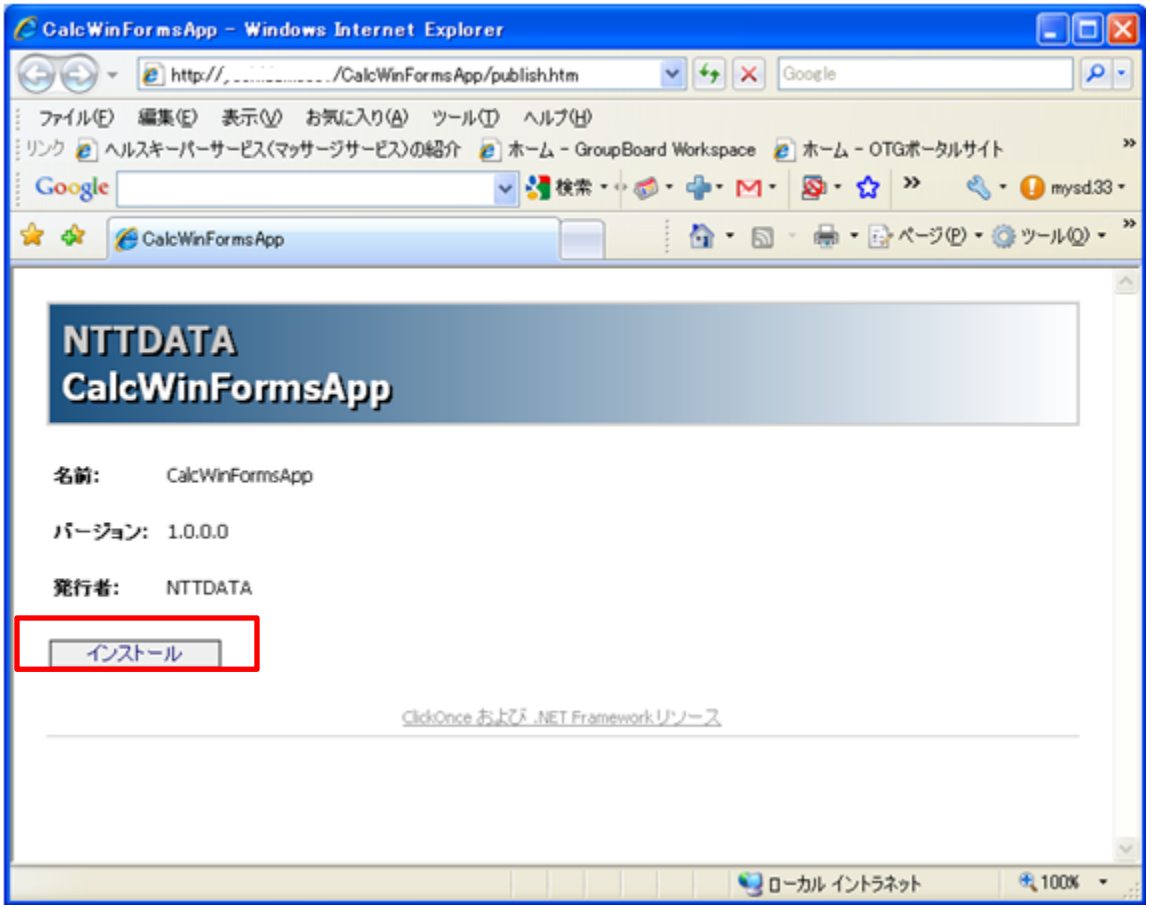
クライアントAPの発行

- アプリケーションを発行する場所を指定し、完了ボタンをクリックします



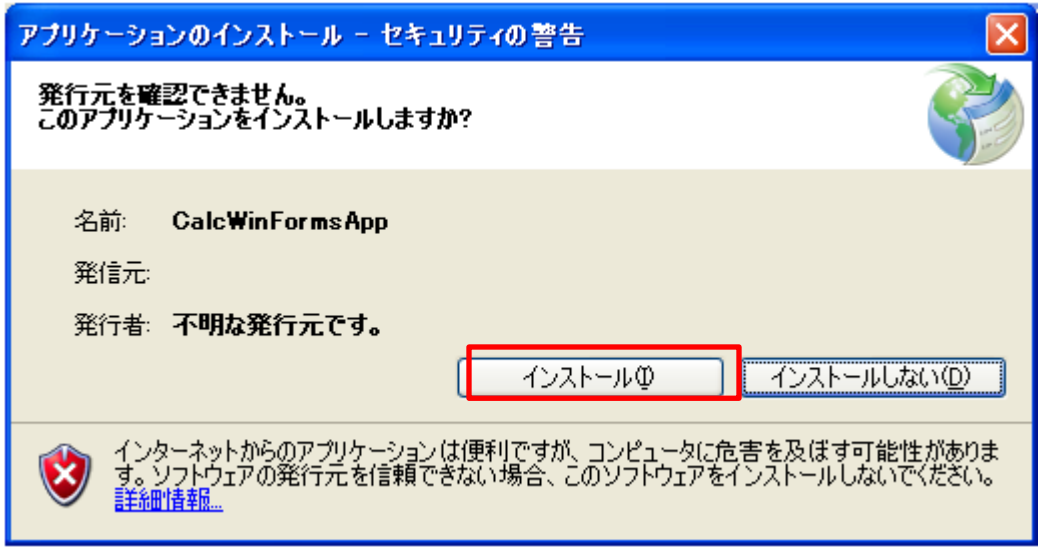
ClickOnceの動作確認

- ブラウザが表示されるので「インストール」をクリックします



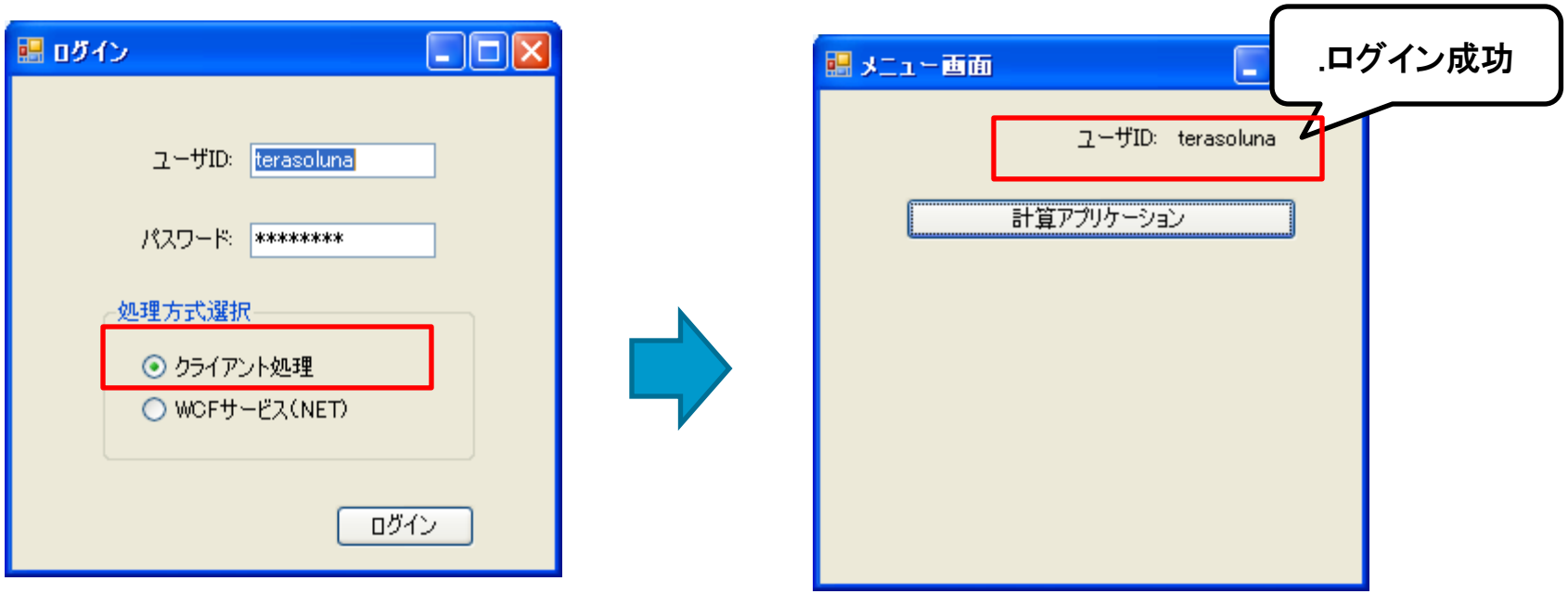
ClickOnceの動作確認

- アプリケーションのインストール画面で「インストール」を選択します



ClickOnceの動作確認

- Click Onceでインストールしたクライアントapの確認
  - インストール後、アプリケーション起動します
  - ログイン処理(クライアント処理)を実施して、クライアントAPに閉じた処理が正しく動作することを確認できます



ClickOnceの動作確認

- .NETサーバ/Javaサーバ接続の確認
  - 計算画面で足し算を実施して、IIS上またはTomcat上のサーバAPに接続し、ビジネスロジックを実行できたことが確認できます