CL-01 画面データ機能

■ 概要

TERASOLUNA フレームワークでは.NET の双方向データバインドの機能を利用し、画面の入出力データを「画面データ」クラスとして管理する。「画面データ」クラスは、画面の入出力表示とリアルタイムで同期をとるとともに、ユーザが画面より入力した値を検証する。

- 双方向データバインドの実装支援
  - 双方向データバインドは、UI コントロールのプロパティとデータソース(「画面データ」クラス)のプロパティを紐づけてプロパティの変更をリアルタイムで同期させる。双方向データバインドは、Visual Studio の GUI エディタにより設定可能であり、特別な実装を必要としないため開発生産性の向上につながる。
  - ビジネスロジックは、「画面データ」クラスを介して UI コントロールにアクセスするため、「画面データ」クラスにより画面とビジネスロジックの依存関係を疎結合にすることができる。
  - 「画面データ」クラスは、POCO(Plain Old CLR Object)ライク[1]なクラスとする。テーブル構造をもつ型付き DataSet をデータソースとして利用する場合と異なり、クラスの構造化や共通化が容易である。また、Silverlight などのプレゼンテーションテクノロジーや、標準通信基盤である WCF(Windows Communication Foundation)、LINQ、EntityFramework といった.NET Framework 3.0 以降のデータアクセステクノロジーなどとも親和性が高い。

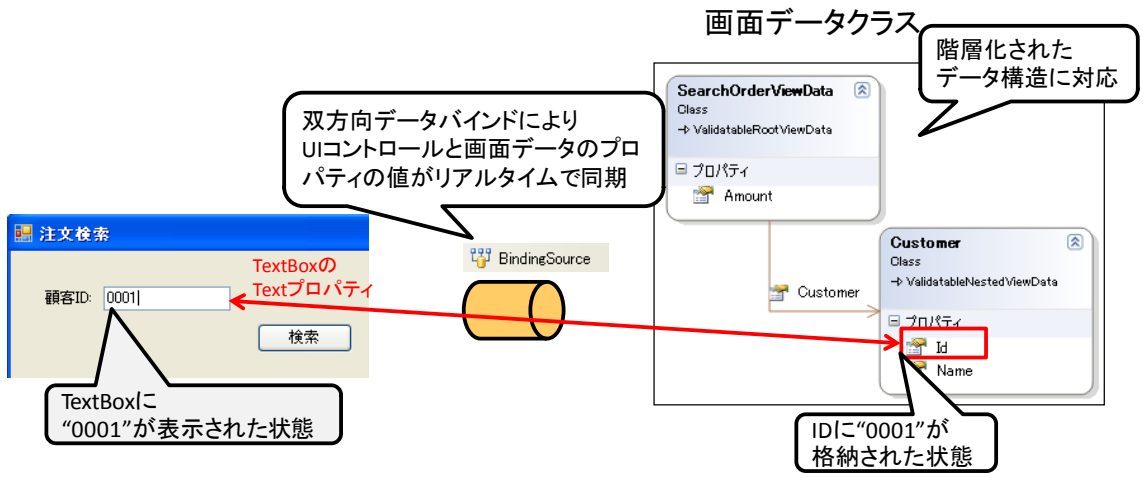

図 1 双方向データバインドの動作概念図

[1] 「画面データ」クラスは、TERASOLUNA フレームワークが提供するクラスを継承して作成するため厳密には POCO ではないが、型付き DataSet のようなテーブル構造をもった.NET 特有のクラスと比較して、POCO「ライク」という言い方をしている。

- 双方向データバインドを実現するためには、以下の実装が必要である。
  - UI コントロール側
    - 「"バインド対象プロパティ名"+Changed」イベントを実装し、データソース側へプロパティ値の変更を通知する必要がある（*PropertyName*Changed パターン[2]と呼ばれる）
    - 通常、標準提供の UI コントロールについては実装済みであるので 、バインド可能なプロパティを追加したカスタムコントロールを作成する際に必要となる。
  - 「画面データ」クラス
    - System.ComponentModel.INofityPropertyChanged インタフェースを実装し、UI コントロール側へプロパティ値の変更を通知する必要がある[3]。
    - 各プロパティの set アクセサで、PropertyChanged イベントを発生させる処理の記述が必要である。

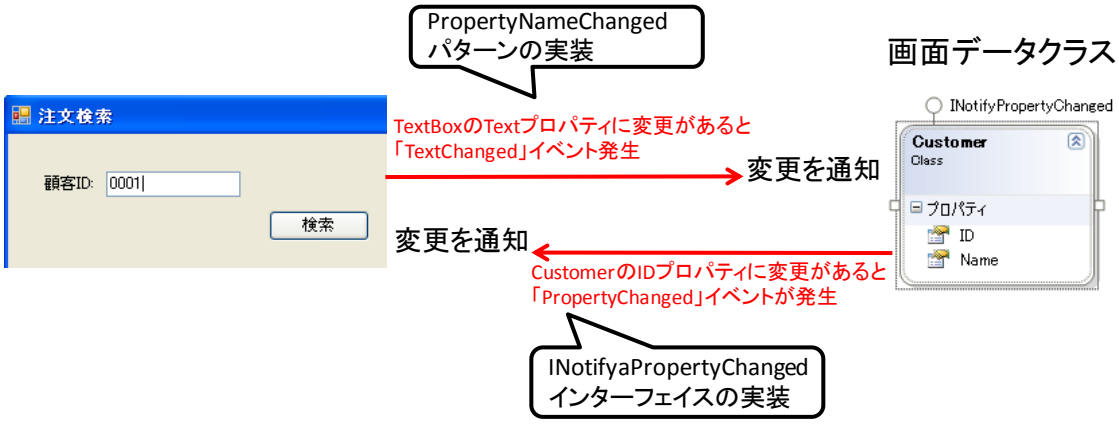

図 2 双方向データバインドにおけるプロパティの変更通知

- *PropertyName*Changed パターンの実装は、アーキテクトや業務共通開発者がシステムで共有するカスタムコントロールを作成する際にのみ必要であるため、それを利用する業務個別開発者の開発には影響しない。アーキテクトや業務共通開発者は、通常通り *PropertyName*Changed パターンによりカスタムコントロールを実装する。
- 一方、INotfiyPropertyChanged インタフェースの実装は、業務開発者が「画面データ」クラスを開発する際に必ず実装しなければならないため、双方向データバインドを実現するのための負荷は非常に高い。、
  そこで、本機能では、「CM-02 インスタンス管理機能」の AOP 機能を利用し、set アクセサ実行後に PropertyChanged イベントを自動的に発生させること、開発者による INotifyPropertyChanged インタフェースに関わる実装作業を不要としている。これにより、業務開発者による実装作業を大幅に軽減できるとともに PropertyChanged イベントを発生させるコードの書き忘れによる不具合発生を事前に防止することができる。

---

[2]方法 ：PropertyNameChanged パターンを適用する(MSDN)
http://msdn.microsoft.com/ja-jp/library/ms229615.aspx
[3]方法 ：INotifyPropertyChanged インターフェイスを実装する(MSDN)
http://msdn.microsoft.com/ja-jp/library/ms229614.aspx

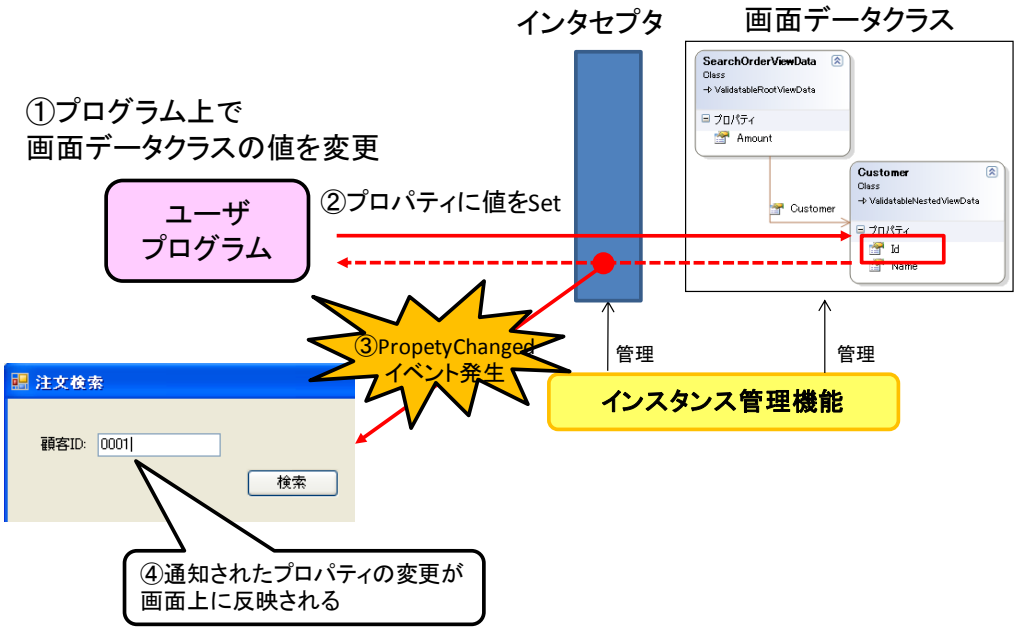

図 3 PropertyChanged イベント自動発生の動作概念図

- 画面の入力値検証
  - Windows Formsの入力値検証処理は、通常であれば Control.Validating イベント等で実装する。このため、画面クラス(Form)に入力値検証処理のためのイベントハンドラメソッドが大量に記述され、ソースコードの見通しが悪く、メンテナンス性も悪い。TERASOLUNA フレームワークでは、入力値検証のロジックを「画面データ」クラスに集約することで、画面側のコードがすっきりし、メンテナンス性の高いコードにすることができる。
  - 入力値検証処理は、「CM-05 入力値検証機能」を利用し、「画面データ」クラスのプロパティを検証する。検証は、以下のタイミングで実施する。
    - UI コントロールのロストフォーカス(当該コントロールから、フォーカスが外れること)時(以下、「即値チェック」と呼ぶ)
    - ボタン押下などによる業務処理実行時
  - 「即値チェック」では、ユーザによる値の入力後、ロストフォーカスのタイミングで単項目チェックを実施し、即座にエラー表示する。
    双方向データバインドにより、ロストフォーカス時に入力データが即座に「画面データ」クラスのプロパティに反映され、同時にフレームワークにより変更があったプロパティに対して自動的に入力値検証処理が実施される。「画面データ」クラスは、System.ComponentModel.IDataErrorInfo インタフェースを実装しているため、入力検証エラーのメッセージを ErrorProvider に表示することができる。

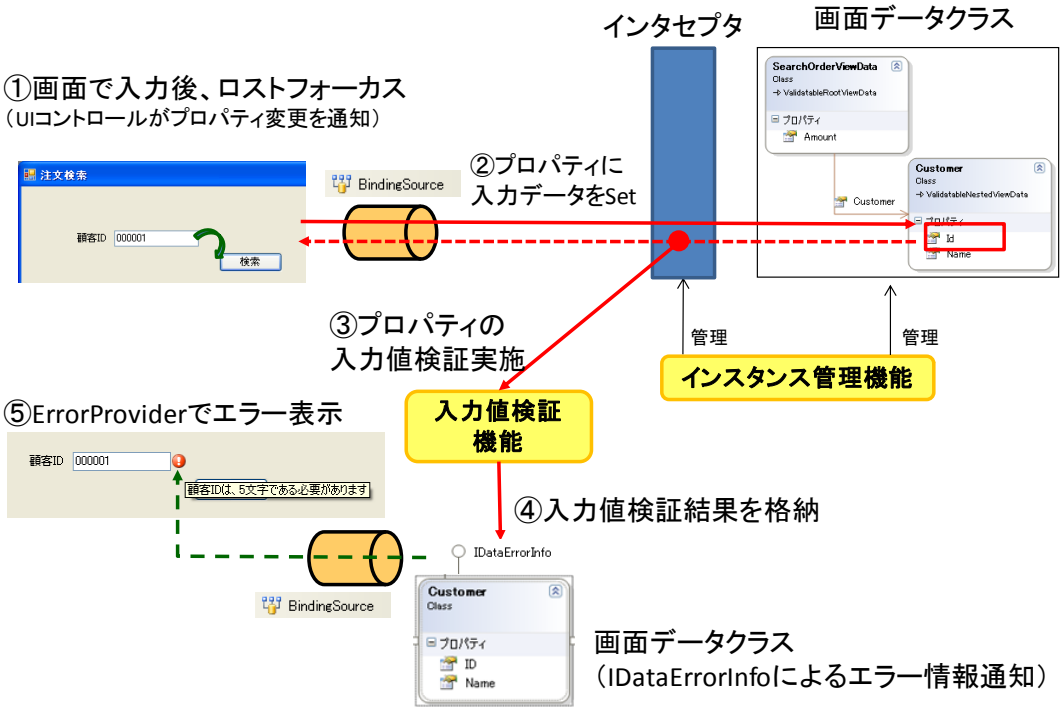

図 4 即値チェックの動作概念図

- ボタン押下などによる業務処理実行時は、「CL-03 イベント処理実行機能」を呼び出し[4]、入力値検証処理を実施する。業務処理実行時は、即値チェックと異なり画面全体の入力値検証を実施する。また、単項目チェックだけなく、相関項目チェック等のカスタム入力チェックも実行する。入力値検証エラー時には、即値チェックと同様の仕組みにより、エラーメッセージを ErrorProvider に表示することができる。

[4] TERASOLUNA フレームワークでは、通常、「XX-99 イベント処理実行機能」を使ってビジネスロジックを実行する。このとき、ビジネスロジック実行前に入力値検証を実施する仕組みになっている。

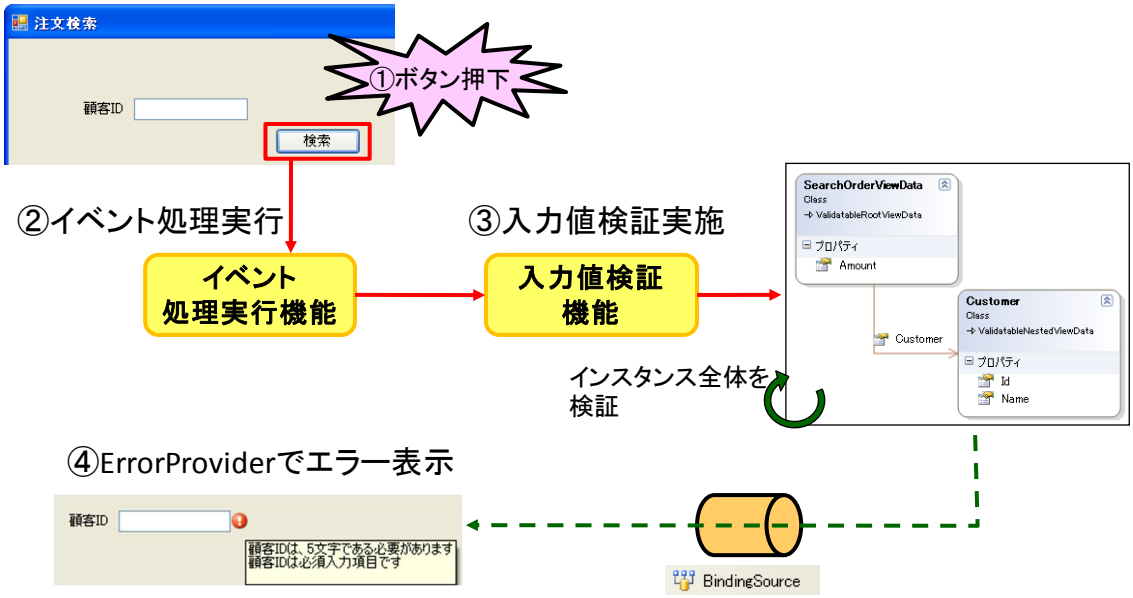

図 5 イベント処理実行時の入力値検証の動作概念図

- 画面データの階層化への対応
  - 「画面データ」クラスは意味のある単位でクラスを分割し構造化したり、複数の画面から共通的に利用したりするケースが出てくる。また、DataGridView などの UI コントロールと複合データバインドする場合、「画面データ」クラスのプロパティとして、DataGridView に表示するコレクションデータを保持することになる。このため、「画面データ」クラスのあるプロパティが別の「画面データ」クラス（または、そのリスト）であるといった階層化されたクラス構造にも対応する必要がある。
  - 「画面データ」クラスの入力値検証処理は、Validation Application Block(以下、Validation AB)をベースに実現している。Validataion AB は、各プロパティに対する入力チェックルールの組み合わせ集合を「ルールセット」として識別することができる。これにより、異なるルールセットを指定して入力値検証処理を実施することで、同じ「画面データ」クラスに対して複数の検証処理を切り替えて実行することができる。
  - Validation AB の入力チェックはクラス単位であり、クラスごとに複数のルールセットを定義する。この際、階層化された異なる「画面データ」クラス間（親子）で、ルールセットを識別する名前（ルールセット名）には何の関連もない。
    このため、「画面データ」クラスが構造化されると、「画面データ」クラス間で、ルールセットの対応関係を定義する必要がある。
    通常、1画面に対して1つの「画面データ」クラスが紐づく。この「画面データ」クラスは、クラス構造の最上位になる。これを「「画面データ（ルート）」と呼ぶ。一方、ネストした「画面データ」クラスを「「画面データ（ネスト）」と呼ぶことにする。クラスの共通化や階層化が進むと、1つの「画面データ（ルート）」は、複数の「画面データ（ネスト）」を保持したり、「画面データ（ネスト）」からさらにネストした「画面データ（ネスト）」を保持することがある。

TERASOLUNA フレームワーク では、クラス階層が深くなっても各画面に対する入力検証処理として何が定義されているか見通しがよくなるように、画面に 1 対1で紐づく「「画面データ(ルート)」に対して RulesetMapping 属性[5]を付与し、「「画面データ(ネスト)」とのルールセットの対応関係を定義する。
また、異なる画面から利用される共通の「画面データ(ネスト)」の場合、ルールセット(検証パターン)を共有できるので、検証パターンや検証処理の共通化を図ることも可能となる。

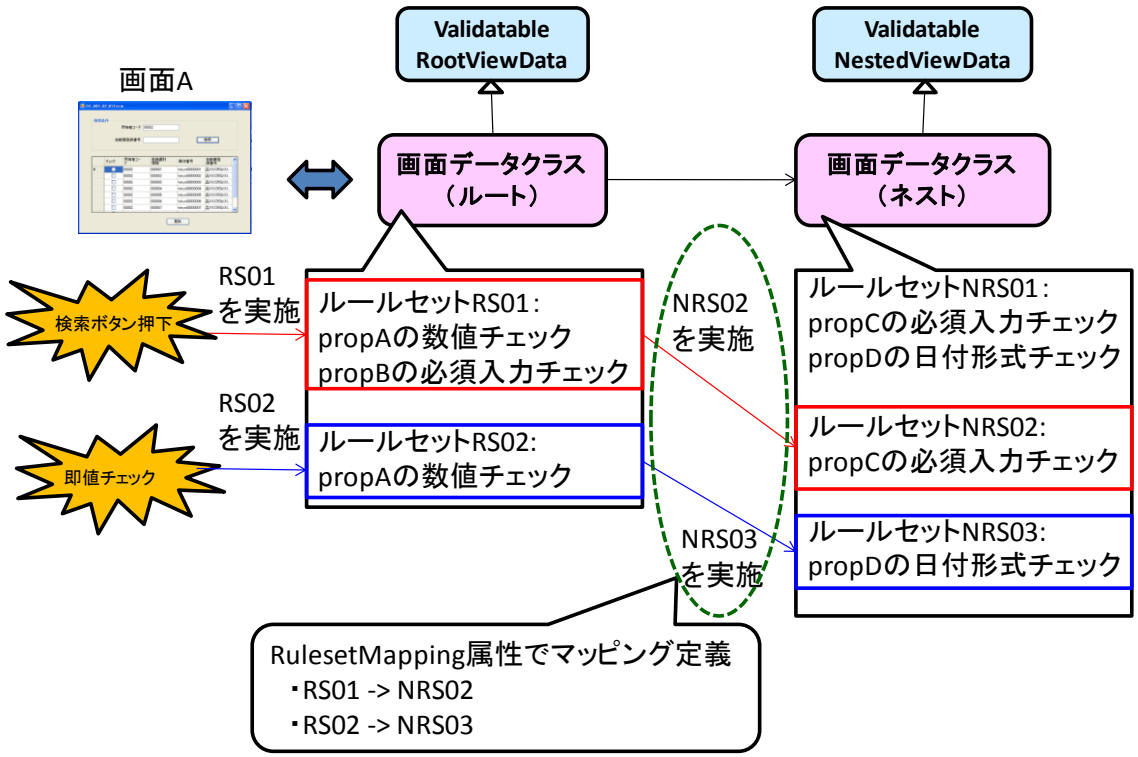

図 6 RulesetMapping 属性によるルールセットの対応関係の定義

[5] Validation AB の標準でも、ObjectValidator、ObjectCollectionValidator といったクラス間のルールセットマッピングが可能な機能が存在するが、ネストした画面データプロパティに対して付与するため、画面全体に対してどのような入力値検証処理がされているのか見通しが悪くなってしまう。このため、TERASOLUNA フレームワークは、Validation AB を拡張し、RulesetMapping によるルールセットのマッピング定義機能を追加した。

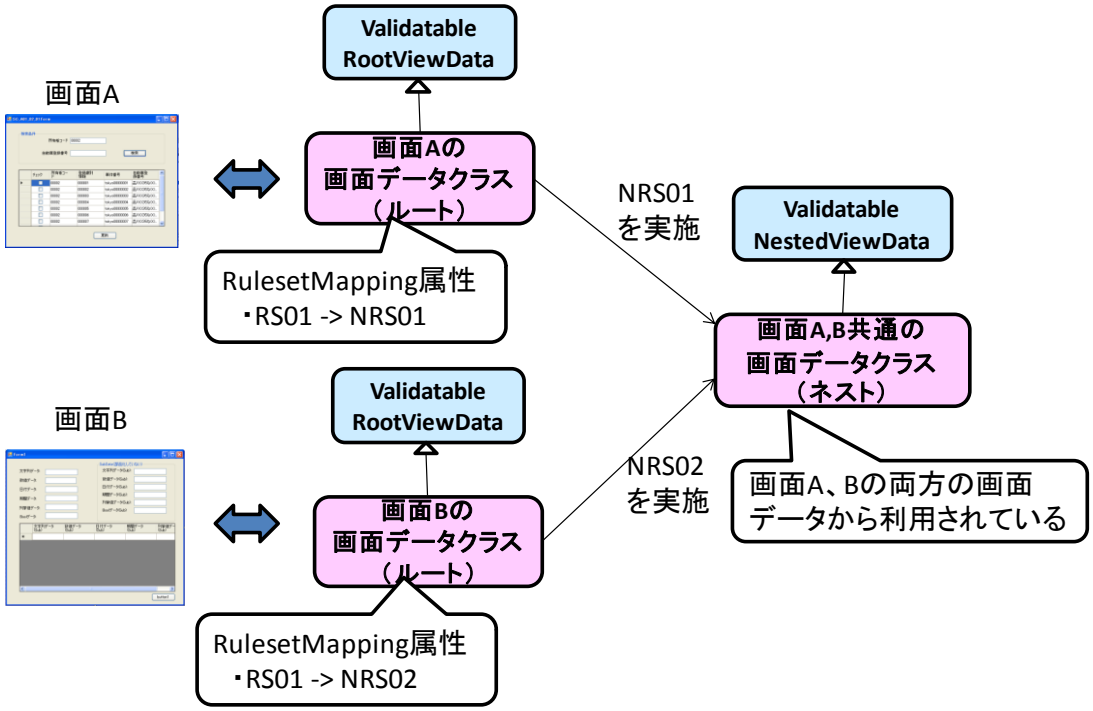

図 7 画面データの共通化と RulesetMapping の定義イメージ

■ 使用方法

◆「画面データ」クラスの作成

「画面データ」クラスは3種類に分類され、それぞれ以下のクラスを利用（継承）して作成する。

表 1 本機能が提供する「画面データ」クラス

項番	画面データの種類	利用クラス	説明
1	画面データ（ルート）	ValidatableRootViewData	「画面データ」の階層構造の最上位となるクラス。画面クラスと 1 対 1 に対応する。
2	画面データ（ネスト）	ValidatableNestedViewData	「画面データ」クラスを構造化する場合に、「画面データ（ルート）」の配下に配置される「画面データ」クラス。
3	画面データ（ネスト）のリストデータ	ValidatableNestedViewDataList<T>	DataGridView 等のコレクション系 UI コントロールと複合データバインドするリストデータを定義する場合に利用するクラス。型パラメータ T は、Terasoluna.ViewModel.Validation.ValidatableNestedViewData の継承クラス（「画面データ（ネスト）」）でなければならない。

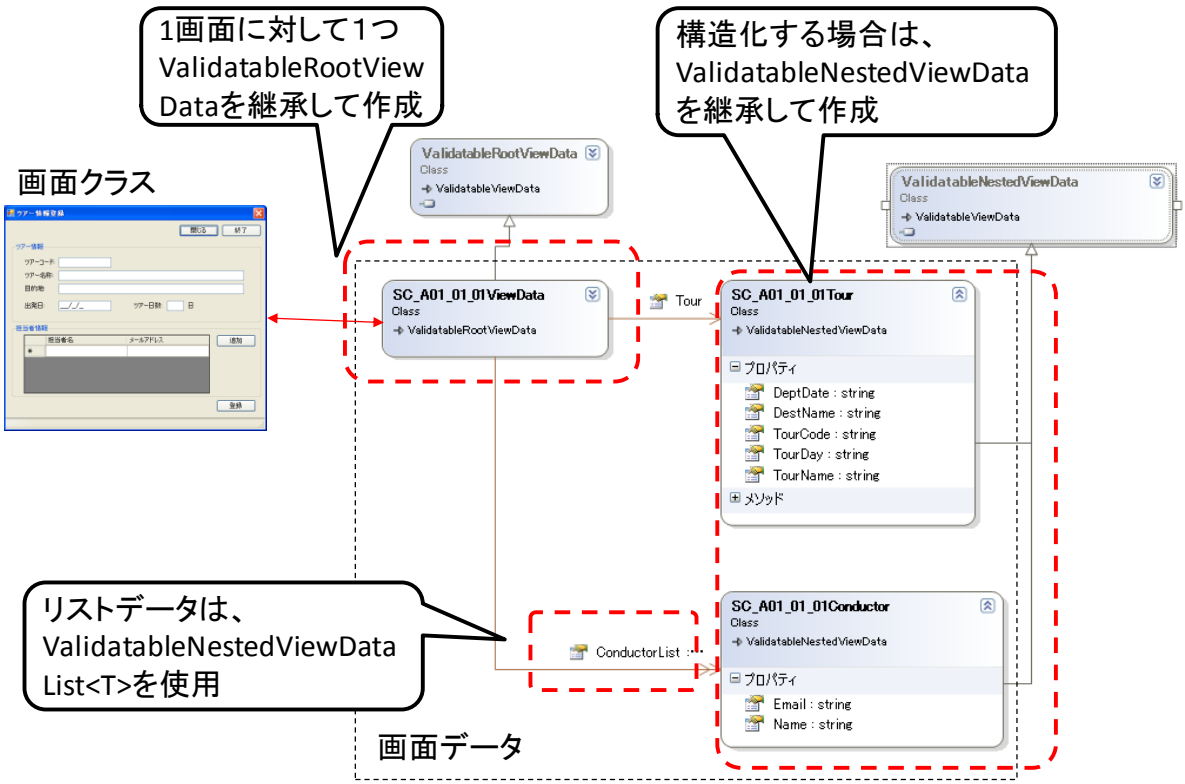

図 8 画面と「画面データ」クラスの対応関係

「画面データ」クラスを作成するには、TERASOLUNA フレームワークが提供するカスタムアイテムテンプレートを使用する。Visual Studio のプロジェクト上で、クラスを作成するフォルダを選択し右クリックメニューで「追加」-「新しい項目」を選択する。

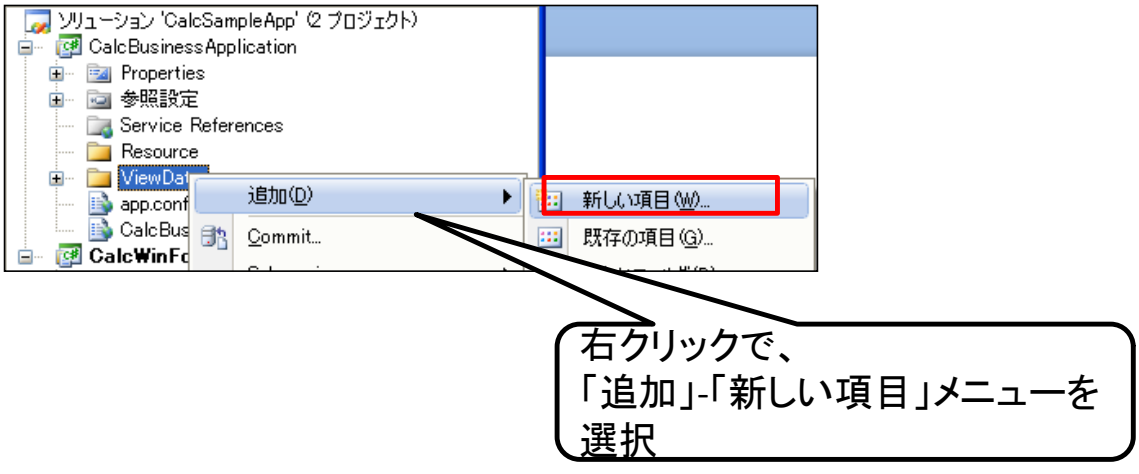

図 9 「画面データ」クラスの作成イメージ1

「新しい項目の追加」ウィンドウで、「TERASOLUNA アイテム」-「Client」-「「画面データ(ルート)クラス」」(または「画面データ(ネスト) クラス」)を選択するとひな型が作成される[6]。

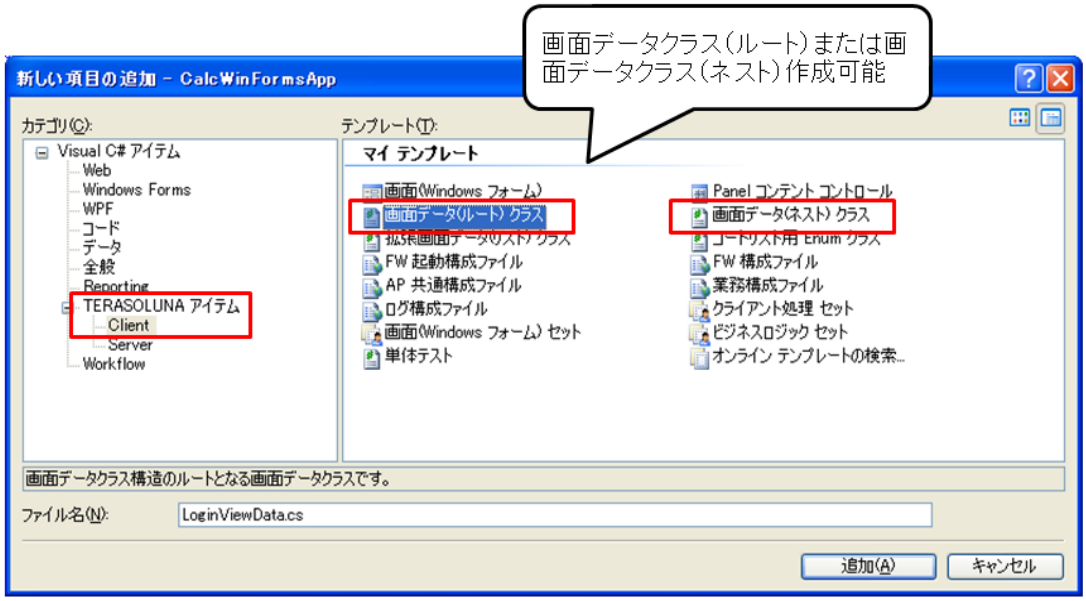

図 10 「画面データ」クラスの作成イメージ2

```
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; }
    }
}
```

図 11 テンプレートより生成された「画面データ(ルート)」

[6] 他にも「クライアント処理 セット」、「画面(Windows フォーム) セット」テンプレートからも「画面データ」クラスを作成することができる。これらの使い方は後述する。

次に、TERASOLUNA フレームワークが提供するカスタムスニペット「tvprop」を使用して、各プロパティを定義する。スニペットは、プロパティが守らなければならないルールに従ってひな形を作成する。開発者は、生成されたひな形を修正するs。

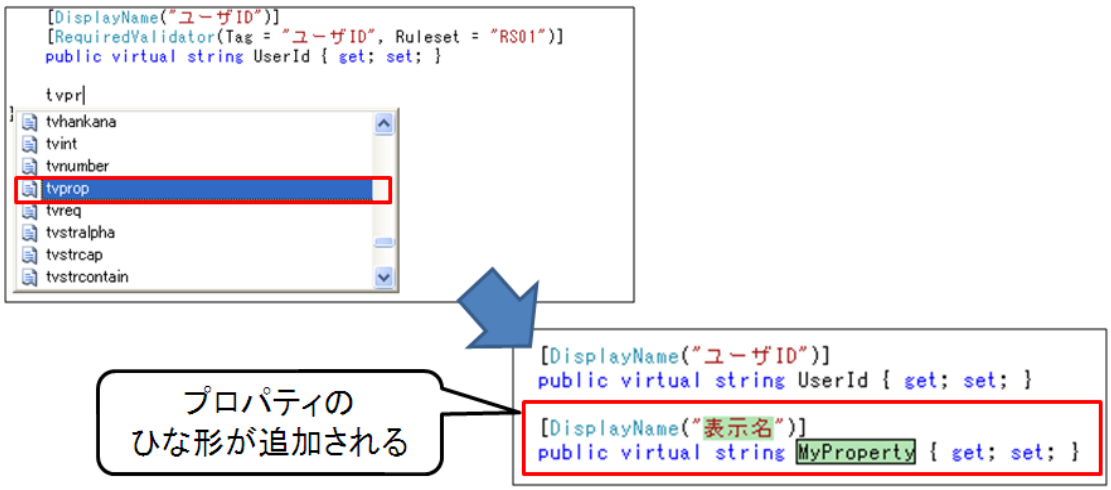

図 12 「tvprop」カスタムスニペットの利用イメージ

なお、プロパティの作成時に守らなければならないルールは以下の通りである。ただし、カスタムスニペットにより生成される部分であるため、開発者が意識することは通常ない。

- 各プロパティは、string 型で定義すること。

例えば、「画面データ」クラスを int 型や DateTime 型といったプロパティで定義し、双方向データバインドした場合、UI コントロールに int 型や DateTime 型で型変換できないデータが入力されると「画面データ」クラスのプロパティと同期がとれないためデータバインドされない。また、デフォルトでは、ロストフォーカスや「×」ボタンの押下もできない状態に陥ってしまい、ユーザビリティも良くない。そこで、「画面データ」クラスのプロパティは全て string 型とし、誤ったデータが入力されてもいったんデータバインドを許し、入力仕掛かり状態として操作を継続させるようにする。
このため、TERASOLUNA フレームワークでは、(ネストした画面データ以外の)プロパティは、全て string 型で定義することとし、誤ったデータが入力された場合には、即値チェックまたはイベント処理時チェックにより、入力値検証エラーを表示することで対応する。

- 各プロパティには、virtual 修飾子を付与すること。

本機能では Unity Application Block の Virtual Method Interceptor により「画面データ」クラスの継承クラスを動的に生成し、AOP による処理(PropertyChanged イベントの発生や入力値検証処理(即値)など)を挟みこんでいる。そのため、virtual 修飾子を付与しないと、双方向データバインドや即値チェックが動作しないので注意すること。

以下に、「画面データ」クラスの作成例を示す。「画面データ」クラスに属性が付与されているが、各属性の記述方法については、後述する。

```
/// <summary>
///  「画面データ（ルート）」の例
/// </summary>
[HasSelfValidation]
[DefaultRuleset("RS01")]
[RulesetMapping("RS01", "NRS01", "Person")]
[RulesetMapping("RS01", "NRS01", "PersonList[]")]
public class SampleViewData : ValidatableRootViewData
{
    /// <summary>
    ///  単純データバインドするデータの例
    /// string型で定義
    /// </summary>
    [DisplayName("ID")]
    [AlphaNumericStringValidator(Ruleset = "RS01")]
    public virtual string Id { get; set; }

    ///<summary>
    ///  単純データバインドするデータの例
    ///  数値データでもstring型で定義
    /// </summary>
    [DisplayName("番号")]
    [IntRangeValidator(LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive,
                       UpperBound = 100, UpperBoundType = RangeBoundaryType.Inclusive,
                       Tag = "番号", Ruleset = "RS01")]
    public virtual string Num { get; set; }

    /// <summary>
    ///  画面データ（ネスト）の例
    /// ValidatableNestedViewData継承クラスのプロパティにする
    /// </summary>
    public virtual Person Person { get; set; }

    /// <summary>
    ///  画面データ（ネスト）のリストデータの例
    /// ValidatableNestedViewDataList<T>クラスのプロパティにする
    /// </summary>
    public virtual ValidatableNestedViewDataList<Person> PersonList { get; set; }

    /// <summary>
    ///  相関項目チェック等のカスタム入力値検証
    /// </summary>
    [SelfValidation(Ruleset = "RS01")]
    public void CustomValidator01(ValidationResults results)
    {
        //TODO:  カスタム入力値検証処理
    }
}
```

画面データ(ルート)
ValidatableRootViewData
を継承
1画面に対して1クラス作成

virtual 修飾子
を付与

リスト 1 「画面データ(ルート)」の作成例

```
/// <summary>
///  「画面データ（ネスト）」の例
/// </summary>
[HasSelfValidation]
public class Person : ValidatableNestedViewData
{
    [DisplayName("名前")]
    [ZenkakuStringValidator(Tag = "名前", Ruleset = "NRS01")]
    public virtual string Name { get; set; }

    [DisplayName("住所")]
    [ZenkakuStringValidator(Tag = "住所", Ruleset = "NRS01")]
    public virtual string Address { get; set; }

    /// <summary>
    ///  相関項目チェック等のカスタム入力値検証
    /// </summary>
    [SelfValidation(Ruleset = "NRS01")]
    public void CustomValidator01(ValidationResults results)
    {
        //TODO: カスタム入力値検証処理
    }
}
```

「画面データ(ネスト)」は ValidatableNestedViewData を継承

リスト 2 「画面データネスト)」の作成例

◆ 入力値検証処理の実装

画面データのプロパティの値を検証することで、画面の入力値を検証する。検証処理は「CM-05 入力値検証機能」を利用して実装する。

(1) 単項目チェック

単項目チェックは、検証対象の画面データのプロパティに対して、「CM-05 入力値検証機能」が提供する Validator の属性を付与することで実装する。
カスタム属性の種類や各属性の詳細な使用方法については、「CM-05 入力値検証機能」の機能説明書を参照すること。
以下に、「画面データ」クラスの単項目チェックの記述例を示す。

```
[RequiredValidator(Tag="担当者名", Ruleset="RS01")]
[ZenkakuStringValidator(Tag = "担当者名", Ruleset = "RS01")]
public virtual string ConductorName { get; set; }
```

リスト 3 「画面データ」クラスに記述する単項目チェックの記述例

単項目チェックエラーが発生すると、「画面データ」クラスの内部では、IDataErrorInfo インタフェースの Item プロパティ(インデクサ)に対象プロパティのエラーメッセージを格納する。これにより、ErrorProvider が各コントロールにエラーメッセージを表示する仕組みになっている。

(2) 相関項目チェック等のカスタム入力チェック

相関項目チェックや複雑な単項目チェックを実施しなければならない場合は、カスタム入力チェック処理を実装する。カスタム入力チェック処理は、イベント処理実行時のみ実施される(ロストフォーカス時には実施されない)。カスタム入力チェックは以下のように実装する。使用方法の詳細は、「CM-05 入力値検証機能」を参照のこと。

- 対象「画面データ」クラスに HasSelfValidation 属性を付与する。
- カスタム入力チェック処理は、対象「画面データ」クラスのメソッドとして実装し、メソッドに SelfValidation 属性を付与する。
- カスタム入力チェックを実装するメソッドのシグニチャは、引数を Microsoft.Practices.EnterpriseLibrary.Validation.ValidationResultsクラス、戻り値をvoidとしなければならない。
- 入力値検証エラーの場合、Microsoft.Practices.EnterpriseLibrary.Validation. ValidationResultクラスのインスタンスを生成しエラーメッセージを格納後、メソッドの引数であるValidationResultsオブジェクトにAddメソッドで追加する。

この時、ValidationResultクラスのコンストラクタの第3引数には、検証対象のキー値を設定する。キー値の設定方法によって、画面へのエラー表示処理の動作が異なるので注意すること。

- 画面項目に対するエラーメッセージとしたい場合
  - 「画面データ」の対象プロパティ名の文字列を指定する。
  - 前述の単項目チェックと同様に、ErrorProviderにより各コントロールにエラーメッセージを表示することができる。
    - 「画面データ」クラス内部では、IDataErrorInfoインタフェースのItemプロパティに対象プロパティ名をキーとしてエラーメッセージが格納されるため、ErrorProviderで表示可能である。

- 画面全体に対するエラーメッセージとしたい場合
  - 「画面データ」の対象プロパティが存在しないため、空の文字列(string.Empty)を指定する。
  - ErrorProviderによるエラーメッセージは表示されない。
    - 「画面データ」クラス内部では、IDataErrorInfoインタフェースの(Itemプロパティではなく)Errorプロパティにインスタンスに対するエラーメッセージが格納されるため、ErrorProviderでは表示されない。
  - エラー発生時にエラーメッセージを取得し画面表示するように別途実装する必要がある。
    - ただし、DataGridViewの場合には、行単位のエラーとして、エラーアイコンおよびエラーメッセージが自動的に表示される(図 13)。

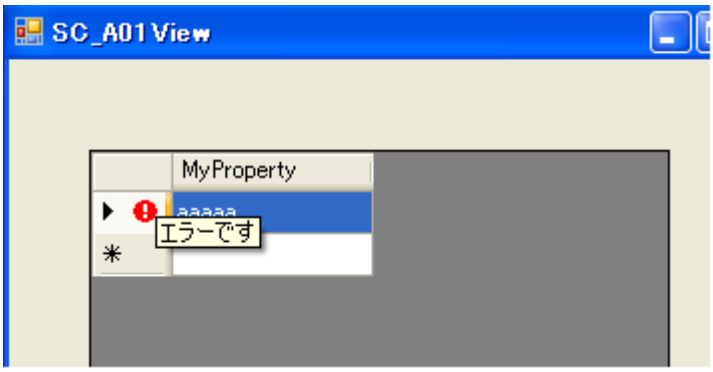

図 13 カスタムチェック時に対象プロパティを指定しない場合の DataGridView のエラー表示例

また、即値チェックを利用する場合、ValidationResult クラスのコンストラクタの第5引数(validator)に渡す値に注意が必要である。
即値チェック時に、ErrorProvider 等で表示したカスタムチェックエラーのメッセージだけクリアされないようにしたい場合には Validation 型の値である、
「EventSpecificValidator.DefaultInstance」プロパティを渡す必要がある。
「画面データ」クラスは、即値チェック時に、検証対象の「画面データ」のプロパティに発生した入力値検証エラーを一旦全てクリアしてから再検証する。一方、カスタム入力チェックはイベント処理実行時のみに実行される入力値検証処理で、即値チェック時には実行されない。このため、再度即値チェックが行われることでカスタム入力チェック結果のエラーメッセージが消えてしまうとエラーメッセージの整合性がとれなくなる。
この対処策として上記の設定をすることで、次回イベント処理実行時までカスタム入力チェックのエラーメッセージを消さないようにすることができる。
即値チェックを実施しない場合や、即値チェック時にカスタム入力チェックエラーのメッセージが消えても問題ない場合には、第5引数にnullを渡してもよいが、今後の機能拡張時などの保守性を考え、上記設定をしておくことを推奨する。

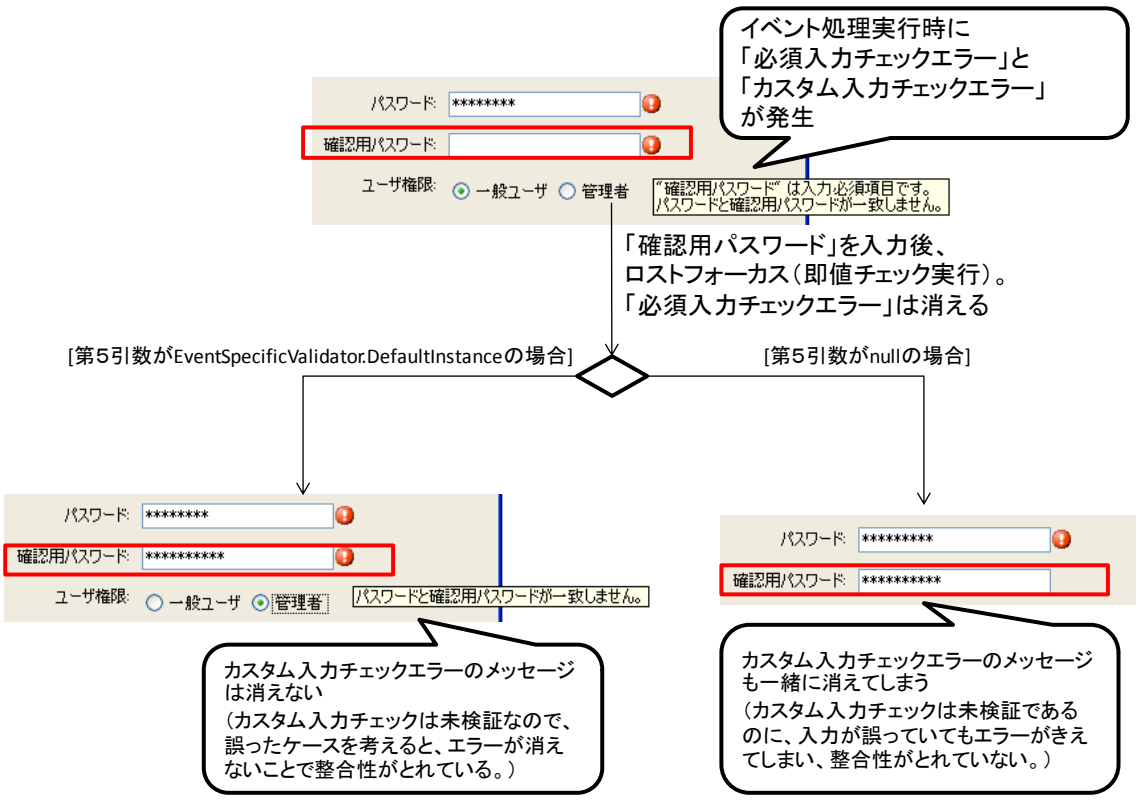

図 14 ValidationResult のコンストラクタの第 5 引数による挙動の違い

以下に、カスタム入力チェックの実装例を示す。

```
    /// カスタム入力値チェックが存在するクラスにHasSelfValidtionを付与
    [HasSelfValidation]
・・・
    public class SampleViewData : ValidatableRootViewData
    {

        /// <summary>
        /// カスタム入力チェック。SelfValidaion属性を付与
        /// ErrorProviderに表示する例
        /// </summary>
        [SelfValidation(Ruleset = "RS01")]
        public void CustomValidator01(ValidationResults results)
        {
            ///TextFieldとNumberFieldのどちらか入力されていることをチェック
            if (!string.IsNullOrEmpty(Id) || !string.IsNullOrEmpty(Num))
            {
                ValidationResult result =
                    new ValidationResult(Resource1.MSG0002, this, "TextField", null,
                            EventSpecificValidator.DefaultInstance);
                ValidationResult result2 =
                     new ValidationResult(Resource1.MSG0002, this, "NumberField", null,
                             EventSpecificValidator.DefaultInstance);
                results.AddResult(result);
                results.AddResult(result2);
            }
        }
        /// <summary>
        /// カスタム入力チェック。SelfValidaion属性を付与
        /// ErrorProviderに表示しない例
        /// </summary>
        [SelfValidation(Ruleset = "RS02")]
        public void CustomValidator02(ValidationResults results)
        {
            ///IdとNumのどちらか入力されていることをチェック
            if (!string.IsNullOrEmpty(Id) || !string.IsNullOrEmpty(Num))
            {
                ValidationResult result =
                    new ValidationResult(Resource1.MSG0001, this, string.Empty, null,
                            EventSpecificValidator.DefaultInstance);
                results.AddResult(result);
            }
        }

    }
```

ErrorProvider でエラー表示する場合は、ValidationResult 作成時、該当の各プロパティ名を設定

ErrorProvider でエラー表示しない場合、「string.Empty」を設定

リスト 4 カスタム入力値検証処理の実装例

(3) 即値チェックのルールセット定義

即値チェックで使用するルールセット名は、DefaultRuleset 属性で設定する。DefaultRuleset 属性は、「「画面データ(ルート)」」(ValidatableRootViewData 継承クラス)に1つだけ付与することができる[7]。
DefaultRuleset 属性の引数には、即値チェックとして実施するルールセット名を記述する。

以下に、DefaultRuleset 属性の記述例を示す。

```
/// 「画面データ （ルート）」に対してDefaultRuleset属性を付与
/// この例では、即値チェックはルールセットRS01で定義したバリデータが実行される
 [DefaultRuleset("RS01")]
 public class SC_A01_01_02ViewData : ValidatableRootViewData
 {
      [DisplayName("担当者名")]
      [RequiredValidator(Tag="担当者名", Ruleset="RS01")]
      [ZenkakuStringValidator(Tag = "担当者名", Ruleset = "RS01")]
      public virtual string ConductorName { get; set; }
       ・・・
 }
```

リスト 5 DefaultRuleset 属性の記述例

(4)「画面データ」クラス間のルールセットマッピング定義

「画面データ(ルート)」で定義されるルールセットに対して、「画面データ(ネスト)」では、どのルールセットで入力値検証を実施するかといった、ルールセットの対応関係を RulesetMapping 属性で設定する。RulesetMapping 属性は、「画面データ(ルート)」のみに付与することができる。

表 2 RulesetMapping 属性

項番	属性名	説明
1	RulesetMappingAttribute	「画面データ(ルート)」のルールセットに対して「画面データ(ネスト)」およびそのリストデータについて、実施すべきルールセットの対応関係を定義する。 【第1引数】: 「画面データ(ルート)」が実施するルールセット名 【第2引数】: 「画面データ(ネスト)」が実施するルールセット名 【第3引数】: 「画面データ(ネスト)」のプロパティ名 2階層以上ネストしている場合は、「.」(ピリオド)で連結して表現する。また、リスト型の画面データであれば、「[]」(角かっこ)で指定する。

以下に、RulesetMapping 属性の設定例を示す。

---

[7]即値チェックのルールセットは、1画面に対して1つしか存在しないためである。

```
[DefaultRuleset("RS01")]
[RulesetMapping("RS01", "NRS01", "SubData")]
[RulesetMapping("RS01", "NRS01", "SubData.SubSubData")]
[RulesetMapping("RS01", "NRS02", "SubList[]")]
[RulesetMapping("RS01", "NRS01", "SubList[].SubSubData")]
[RulesetMapping("RS02", "NRS02", "SubData")]
[RulesetMapping("RS02", "NRS02", "SubData.SubSubData")]
[RulesetMapping("RS02", "NRS01", "SubList[]")]
[RulesetMapping("RS02", "NRS02", "SubList[].SubSubData")]
 public class SampleViewData : ValidatableRootViewData
{
    [AlphaNumericStringValidator(Ruleset = "RS01")]
    [AlphaNumericStringValidator(Ruleset = "RS02")]
    [RequiredValidator(Ruleset = "RS02")]
    public virtual string Id { get; set; }

    public virtual SampleSubViewData SubData { get; set; }

    public virtual ValidatableNestedViewDataList<SampleSubViewData2>   SubList { get; set; }
}

public class SampleSubViewData : ValidatableNestedViewData
{
    [AlphaNumericStringValidator(Ruleset = "NRS01")]
    [AlphaNumericStringValidator(Ruleset = "NRS02")]
    [RequiredValidator(Ruleset = "NRS02")]
    public virtual string Id { get; set; }

    public virtual SampleSubSubViewData SubSubData { get; set; }
}

public class SampleSubViewData2 : ValidatableNestedViewData
{
    [AlphaNumericStringValidator(Ruleset = "NRS01")]
    [AlphaNumericStringValidator(Ruleset = "NRS02")]
    [RequiredValidator(Ruleset = "NRS02")]
    public virtual string Id { get; set; }

    public virtual SampleSubSubViewData2 SubSubData { get; set; }
}

public class SampleSubSubViewData : ValidatableNestedViewData
{
    [AlphaNumericStringValidator(Ruleset = "NRS01")]
    [AlphaNumericStringValidator(Ruleset = "NRS02")]
    [RequiredValidator(Ruleset = "NRS02")]
    public virtual string Id { get; set; }
}

public class SampleSubSubViewData2 : ValidatableNestedViewData
{
    [AlphaNumericStringValidator(Ruleset = "NRS01")]
    [AlphaNumericStringValidator(Ruleset = "NRS02")]
    [RequiredValidator(Ruleset = "NRS02")]
    public virtual string Id { get; set; }
}
```

リスト 6 DefaultRuleset 属性と RulesetMapping 属性の記述例

上記のサンプルコードの場合、「画面データ(ルート)」で実施するルールセット名を「RS01」から「RS02」で切り替えることで、「画面データ(ネスト)」で実施するルールセットも図 15 のように切り替わる。これにより、画面データが構造化され場合にも、複数の入力値検証ルールを適用することができる。

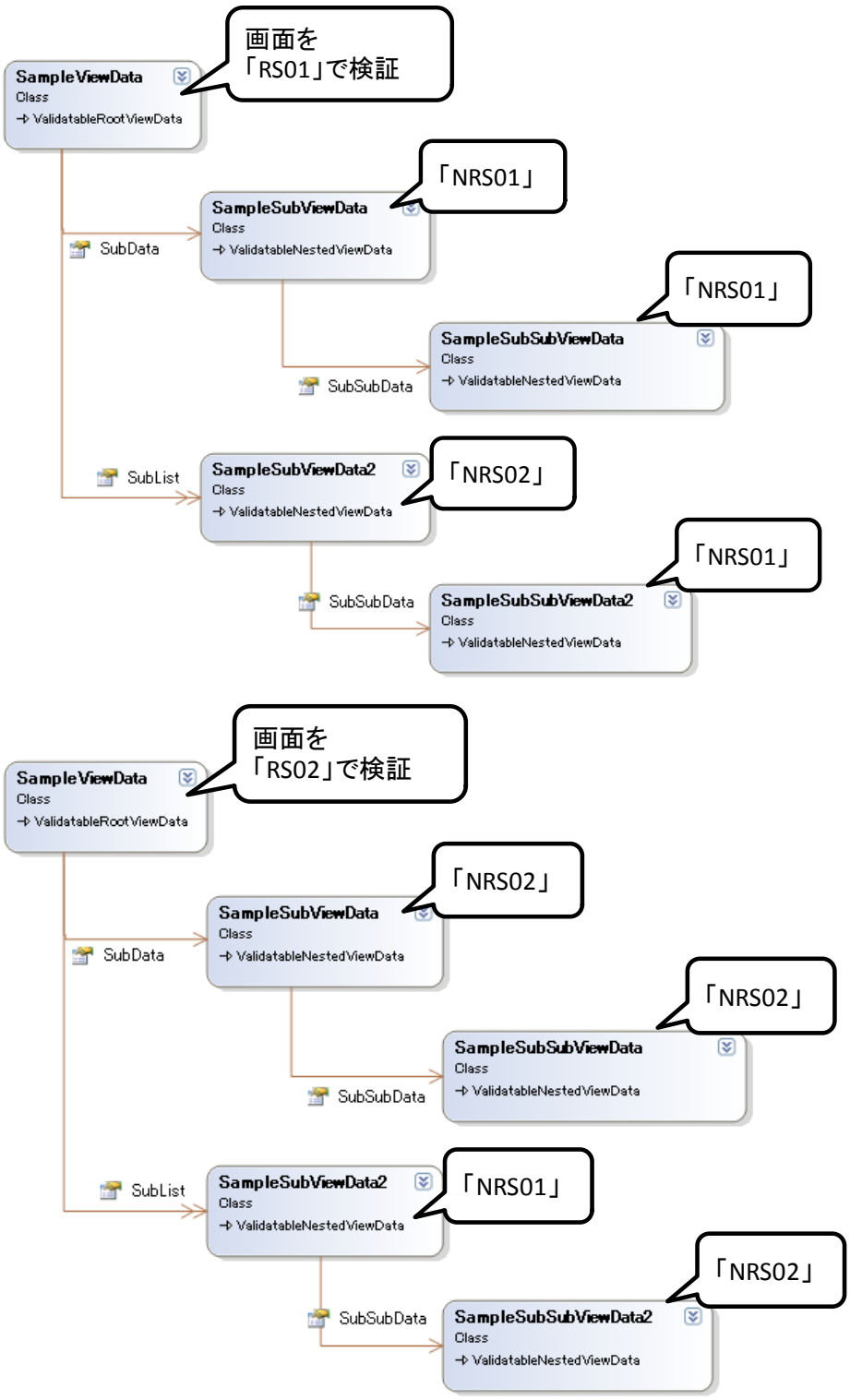

図 15 RulesetMapping 属性によるルールセットの切り替えイメージ

◆ 画面の作成

「画面データ」クラスをバインドする画面を作成する。
画面クラスを作成するには、TERASOLUNA フレームワークが提供するカスタムテンプレート「画面(Windows フォーム)」を使用する。
テンプレートを使用して作成することで、画面クラスの初期化コードのひな形が追加された Form クラスが生成される。

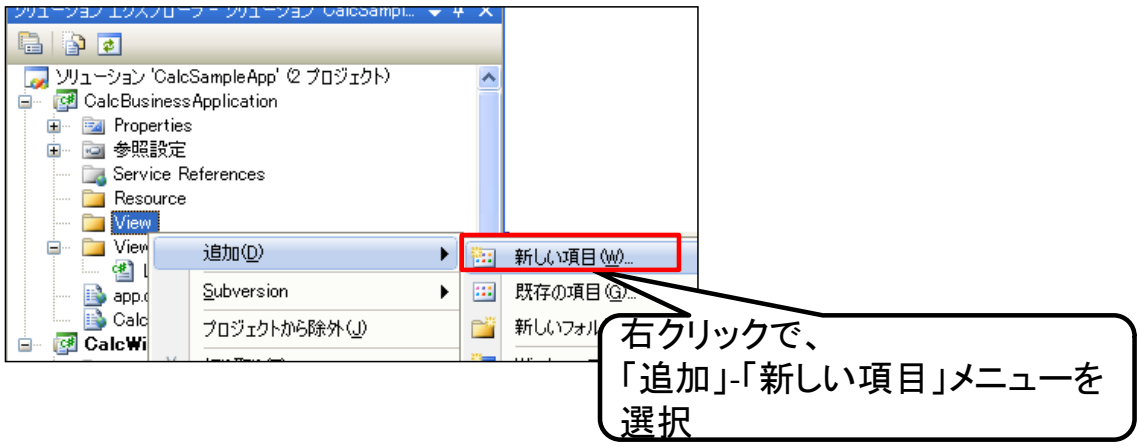

図 16 画面クラスの作成イメージ1

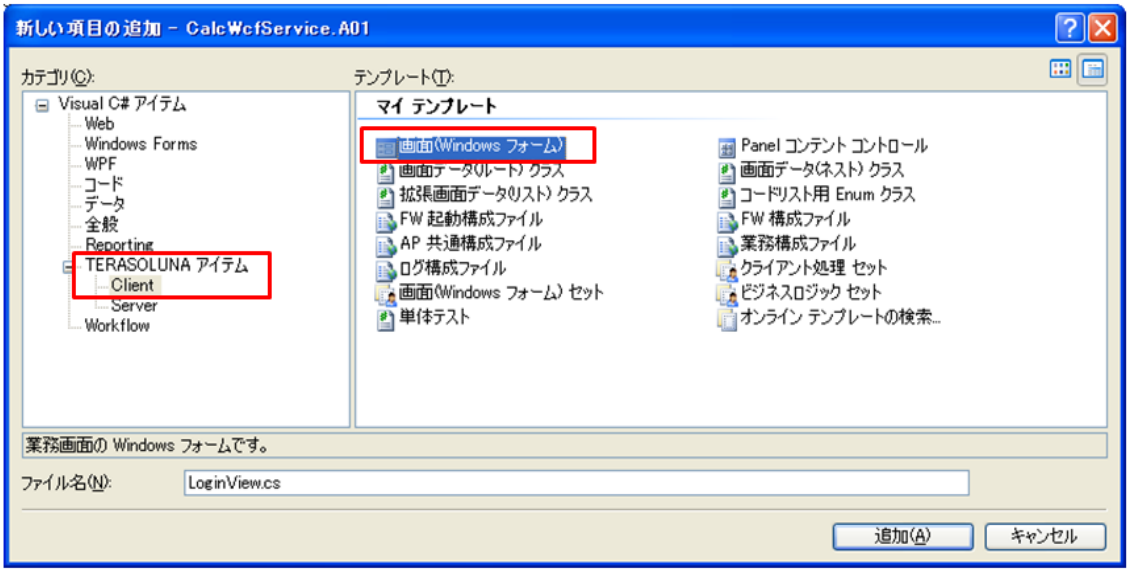

図 17 画面クラスの作成イメージ 2

テンプレートにより、以下のような画面クラスに紐づく「画面データ」クラスの初期化ロジックが生成され、業務開発者の実装作業を支援する。必要に応じてコメントを解除の上、適切な変数名、

クラス名に修正する。

```
// ViewDataクラスを作成後、正しい名前空間を指定しコメント化を解除して下さい。
// 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;
        }
    }
}
```

リスト 7 テンプレートにより生成されたひな型

画面クラスの初期化処理の実装には、以下のルールを守らなければならない。ただし、カスタムテンプレートがルールに従ってひな形を生成するので、開発者による実装作業はかなり少なくなっている。

- 「画面データ(ルート)」型の「ViewData」プロパティを定義する。
  - 「CL-03 イベント処理実行機能」や「CL-02 画面遷移機能」(FormFowarder)が、リフレクションを使って、画面クラスにある「ViewData」という名前のプロパティを「画面データ」クラスとして自動的に取得するためである。
- 「画面データ」クラスのインスタンス生成する方法として、new 演算子は使用せず、ファクトリメソッドである、ViewDataManager.CreateViewData メソッドを使用する。
  - PropertyChanged イベントの自動発生や即値チェックには、「CM-02 インスタンス管理機能」の AOP／インスタンス生成機能を利用している。
  - 上記メソッドにより、DI/AOP コンテナで管理されたインスタンスを生成する必要があるためである。

表 3 ViewDataManager クラスの CreateFormData メソッド

項番	メソッド	型パラメータ	戻り値の型
1	public static T CreateViewData<T>()	ValidatableViewData の継承クラス	型パラメータで指定されたクラスのインスタンス

- ViewDataManager.CreateViewData メソッドの型パラメータには、「「画面データ(ルート)」を設定」しメソッドを実行する。
  - 上記メソッドは、生成する「画面データ(ルート)」に「画面データ(ネスト)」が存在した場合に、ネストクラスのインスタンスも再帰的に生成しプロパティにセットした状態でインスタンスを返却する。
- 画面クラスのコンストラクタで「画面データ」クラスはインスタンスを生成し、ViewData プロパティにセットする。
  - 「CL-02 画面遷移機能」において、遷移元画面から遷移先画面へ「画面データ」クラスをコピーすることができるが、コピーするためには、遷移先画面のインスタンス生成時(コンストラクタ実行時)に、「画面データ」クラスのインスタンスも生成しておく必要があるためである。
- 画面クラスの Load イベントで、BindingSource の DataSource プロパティに「画面データ」クラスをセットし、バインド設定する。この BindingSource は、後述する双方向バインドの設定により生成された画面データ(ルート)に対応する BindingSource である。

- 上記は、「CL-02 画面遷移機能」の FormFowarder を使用した画面(Form)の切り替えによる画面遷移を想定した「画面データ」の設計を前提にしたルールである。「CL-02 画面遷移機能」の ContentPlaceHolder を使用したパネル切り替えによる画面遷移を実装する場合には、上記のルールをベースにパネル切り替え固有のルールに従う必要がある。詳細については、「CL-02 画面遷移機能」の機能説明書を参照のこと。

なお、その他、「CL-02 画面遷移機能」に関連したルールとして「ScreenId 属性の付与」がある。ScreenId 属性については、「CL-02 画面遷移機能」の機能説明書も合わせて参照すること。
以下に、画面クラスの初期化処理の実装例を示す。

```
[ScreenId("CalcView")]
public partial class CalcView : Form
{
    // 「画面データ（ルート）」型のViewDataプロパティ
    public CalcViewData ViewData { get; set; }

    public CalcView()
    {
        InitializeComponent();
        // ViewDataのインスタンス生成
        ViewData = ValidatableViewDataManager.CreateViewData<CalcViewData>();
    }

    private void CalcView_Load(object sender, EventArgs e)
    {
        // BindingSource.DataSourceプロパティの設定
        calcViewDataBindingSource.DataSource = ViewData;
    }
・・・
}
```

リスト 8 画面クラスの初期化処理の記述例

通常、「画面」と「画面データ(ルート)」は対で利用する。カスタムテンプレート「クライアント処理 セット」または「画面(Windows フォーム) セット」を使用すると、対応する「画面」と「画面データ(ルート)」を同時に生成することができる。

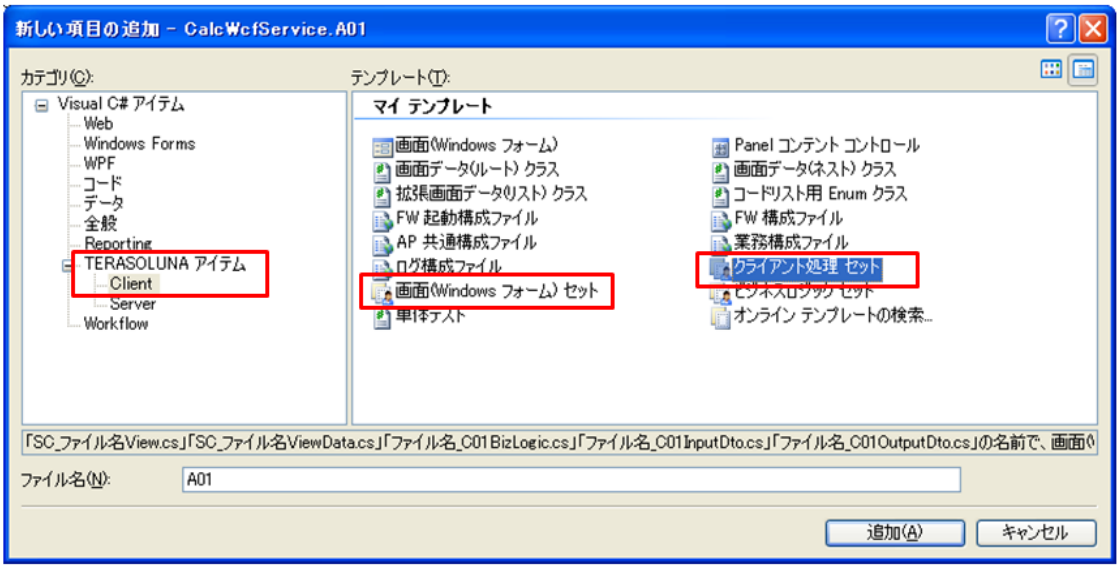

図 18「クライアント処理 セット」と「画面(Windows フォーム)セット」アイテムテンプレート

これら 2 つのテンプレートは、入力されたファイル名に基づいてクラス名を決定し、「View」フォルダと「ViewData」フォルダの中に、「画面」と「画面データ(ルート)」を作成する。以下に、2 つのテンプレートの利用イメージを示す。

「クライアント処理 セット」は入力されたファイル名に従って、「画面」と「画面データ(ルート)」の他に、「ビジネスロジッククラス」と「DTO クラス」も含めた 4 種類のファイルを一度に作成する。

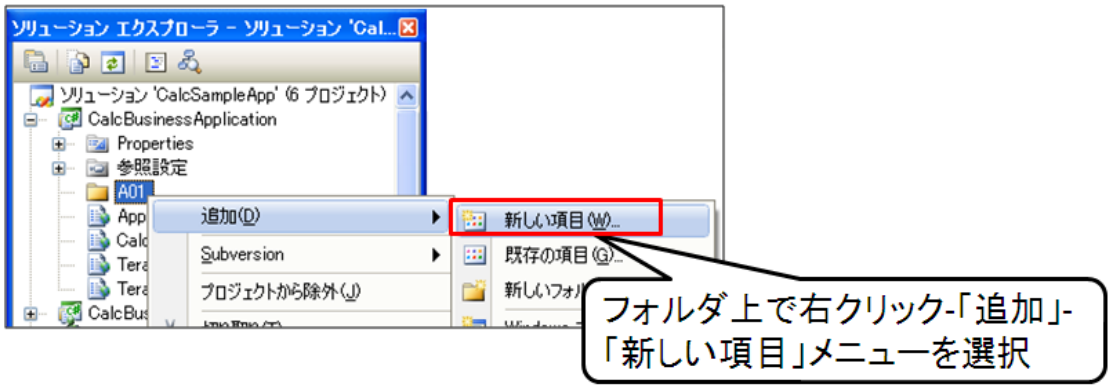

図 19「クライアント処理 セット」の作成イメージ 1

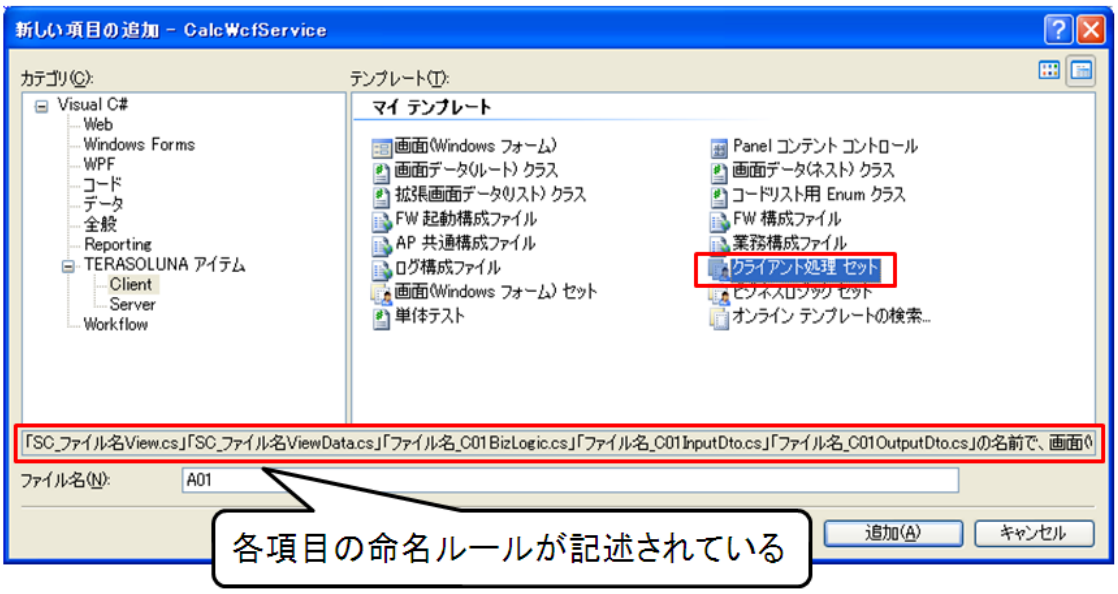

図 20「クライアント処理 セット」の作成イメージ 2

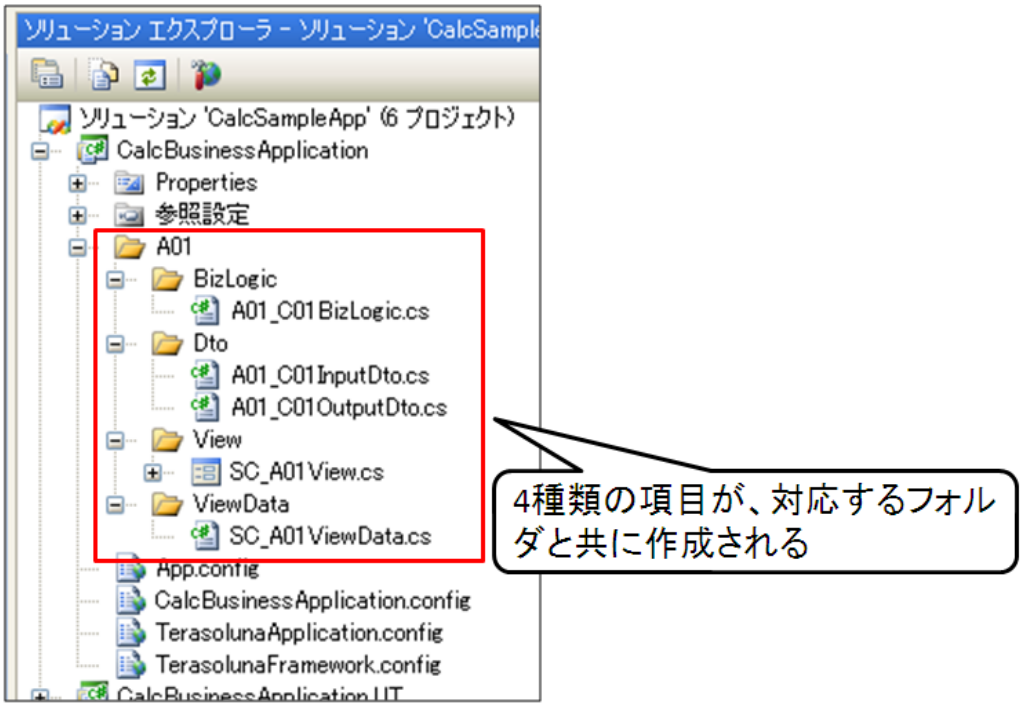

図 21「クライアント処理 セット」の作成イメージ 3

最初の業務作成時には、対応する 4 種類の「フォルダ」も一緒に作成されるが、以降の業務セット追加時には、既存のディレクトリ配下にファイルのみが追加される。
なお、「画面(Windows フォーム)」テンプレートは、上記のうち「画面」と「画面データ(ルート)」の 2 種類のみを作成する。利用方法は「クライアント処理 セット」と同様である。

◆ UI コントロールと画面データの双方向バインド設定

Visual Studio を使った双方向バインドの設定方法は、大きく分けて3種類ある。
詳細な設定手順については、MSDN のドキュメント[8]を参照のこと。

表 4 双方向バインドの設定方法

項番	設定方法	利用シーン
1	データソースウィンドウを使って、UI コントロールを貼り付ける。	「画面データ」の設計／製造を、画面クラスよりも先行できる場合。以下の 2 つのケースがある。 ・Visio 等のツールで作成した画面レイアウトで上流工程を行った場合など、製造工程まで Visual Studio による画面クラスの開発を遅延させられる場合 ・上流工程で、画面レイアウトの画面項目等をもとにして、「画面データ」クラスの設計／製造が可能な場合
2	データソースウィンドウを使って、既存の UI コントロールにバインド設定をする。	・製造工程以前の段階で、Visual Studio を使った画面クラスの製造が完了している場合など、画面クラスに対して後から「画面データ」の製造およびバインド設定を行う場合。
3	プロパティエディタを使って UI コントロールのバインド設定をする。	項番1、2の方法により自動生成されたバインド設定を個別に変更する等、よりきめ細かいバインド設定をする場合。

(1) データソースウィンドウを使った UI コントロールの貼り付け

「データソース」ウィンドウから、画面データのプロパティをドラッグ＆ドロップすることで、「画面データ」クラスのプロパティとバインド設定済みの状態で UI コントロールを配置することができる。
画面レイアウト作成とバインド設定が一度に自動生成されるため、最小の手順で画面作成が可能である。ただし、製造工程まで Visual Studio による画面クラスの開発を遅延させられる場合もしくは、上流工程で画面レイアウトの画面項目等をもとに「画面データ」クラスの設計／製造して、画面クラスを作成する場合に有効な方法である。

[8] データの表示の概要:
http://msdn.microsoft.com/ja-jp/library/2b4be09b.aspx

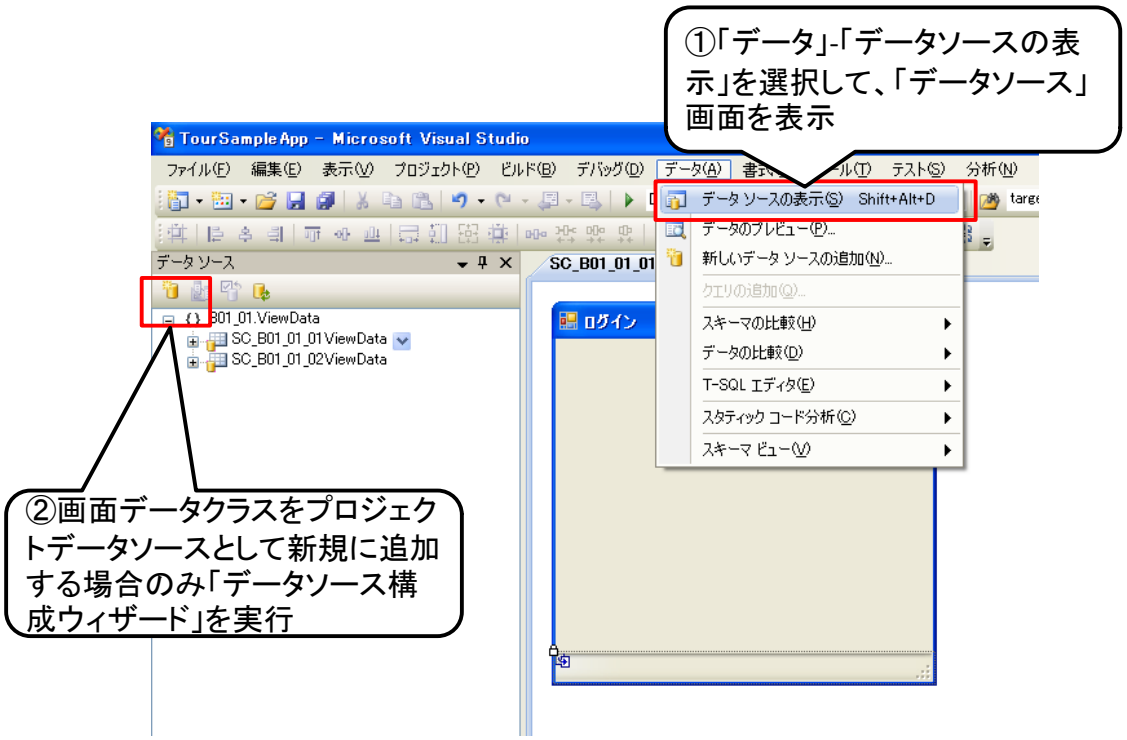

図 22 データソースへの画面データクラスの追加 1

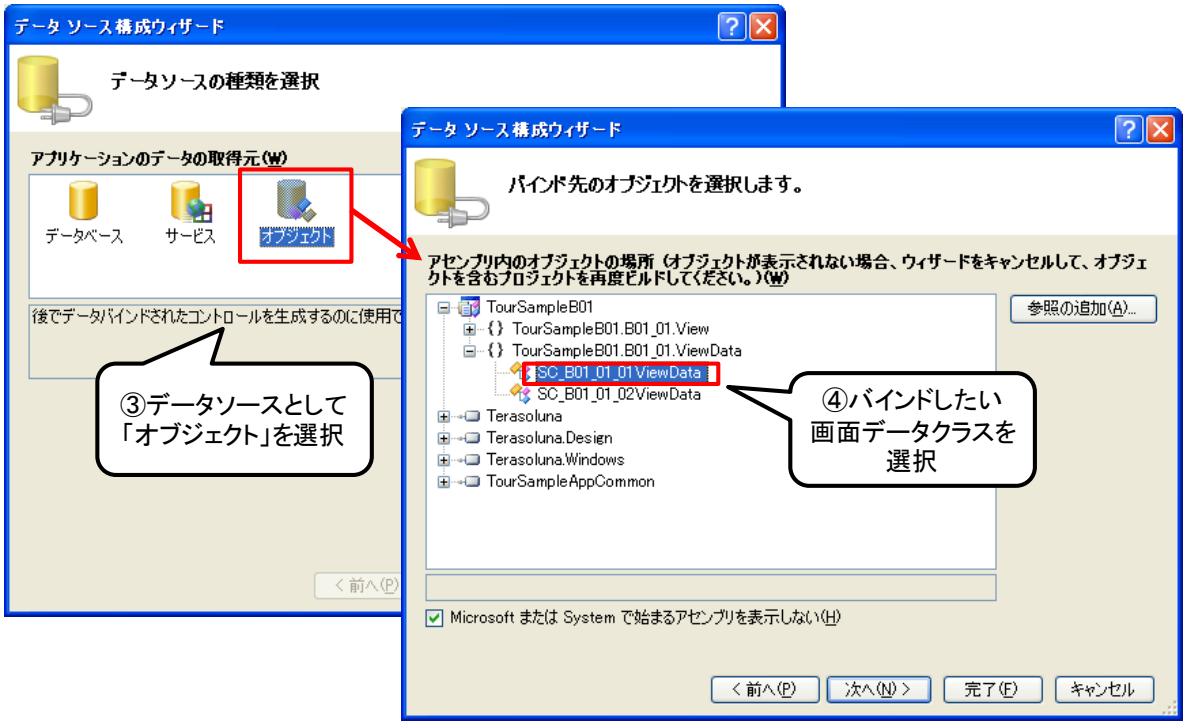

図 23 データソースへの画面データクラスの追加 2

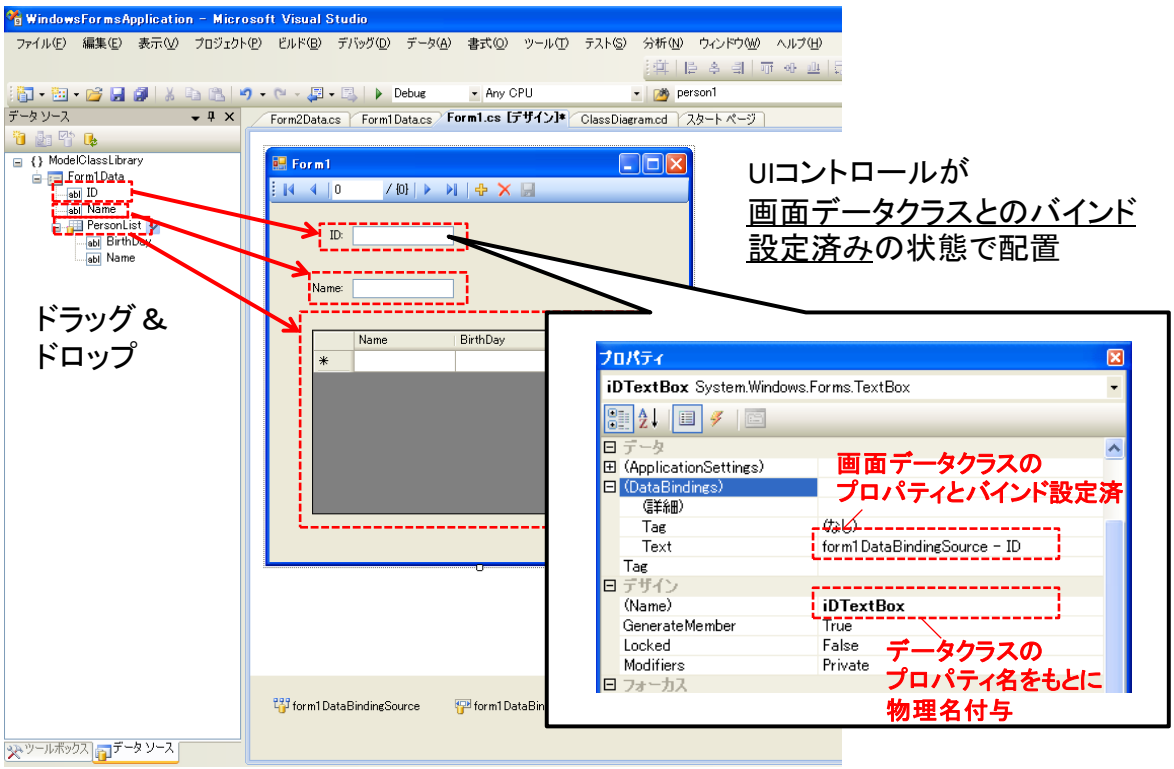

図 24 データソースウィンドウを使った UI コントロールの貼り付け

※DisplayName 属性について

データソースウィンドウから張り付けたときに、Label が一緒に生成されるが、あらかじめ「画面データ」のプロパティに DisplayName 属性を付与しておくと、DisplayName 属性で指定した日本語名で Label が生成されるため、画面レイアウトの開発作業を軽減することができる。
なお、前述のカスタムスニペット「tvprop」を使ってプロパティを生成すると、DisplayName カスタム属性のひな形が一緒に生成される。

画面データクラス

```
public class A010101ViewData : ValidatableRootViewData
{
    [DisplayName("申請番号")]
      public virtual string  ShiseiNo { get; set; }
    ・・・
}
```

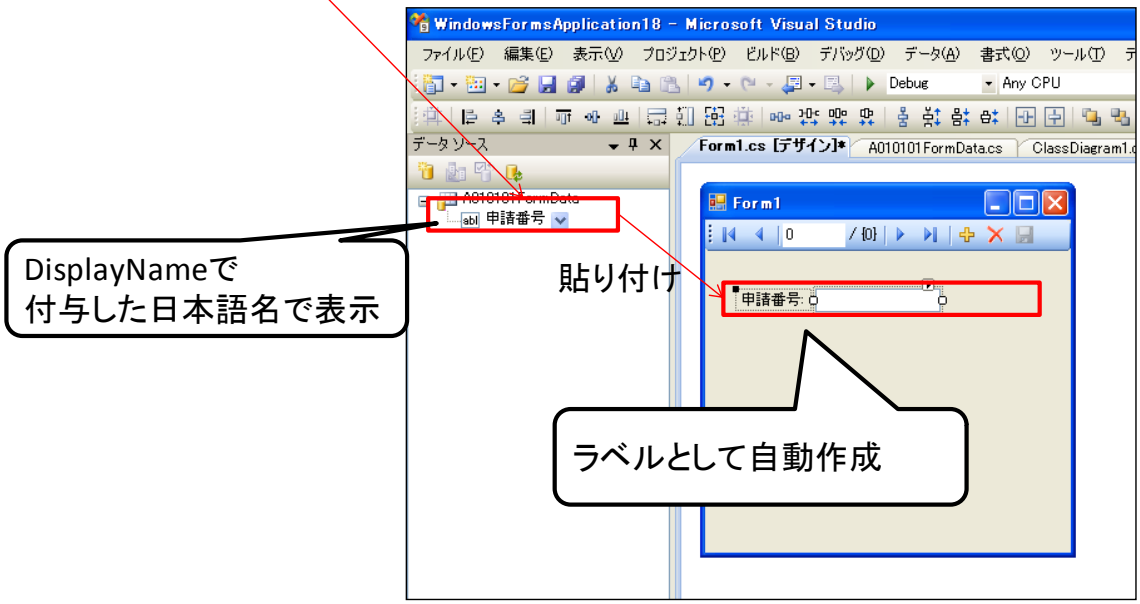

図 25 DisplayName 属性による日本語名でのラベル作成

(2) データソースウィンドウを使った既存の UI コントロールのバインド設定

「データソース」ウィンドウからデータソースに追加した画面データクラスのプロパティを、画面へ配置済みの UI コントロールにドラッグ＆ドロップすることでバインド設定される。
手順は項番 1 とほぼ同じであるが、画面へ配置済みの UI コントロールに対してバインド設定を自動生成することができるため、製造工程以前に作成済みの画面クラスを利用して、製造工程開始後に「画面データ」を作成しバインド設定のみ実施する場合に有効な設定方法である。

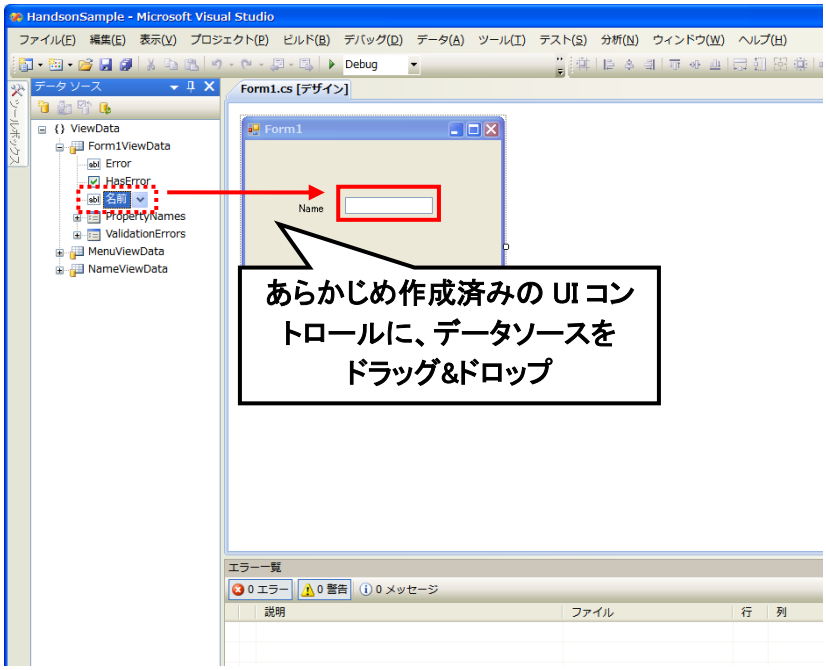

図 26 データソースウィンドウを使った既存の UI コントロールのバインド設定

(3) プロパティエディタを使った UI コントロールのバインド設定

項番 1,2 の方法により自動生成されたバインド設定を個別に変更する等、項番 1,2 の手順ではできない、よりきめ細かいバインド設定をする場合に実施する。
UI コントロールのプロパティエディタで、「(DataBindings)/バインド対象のプロパティ名」を選択し、バインディングソースまたは、プロジェクトデータソースで、バインドする「画面データ」クラスのプロパティ名を選択する。

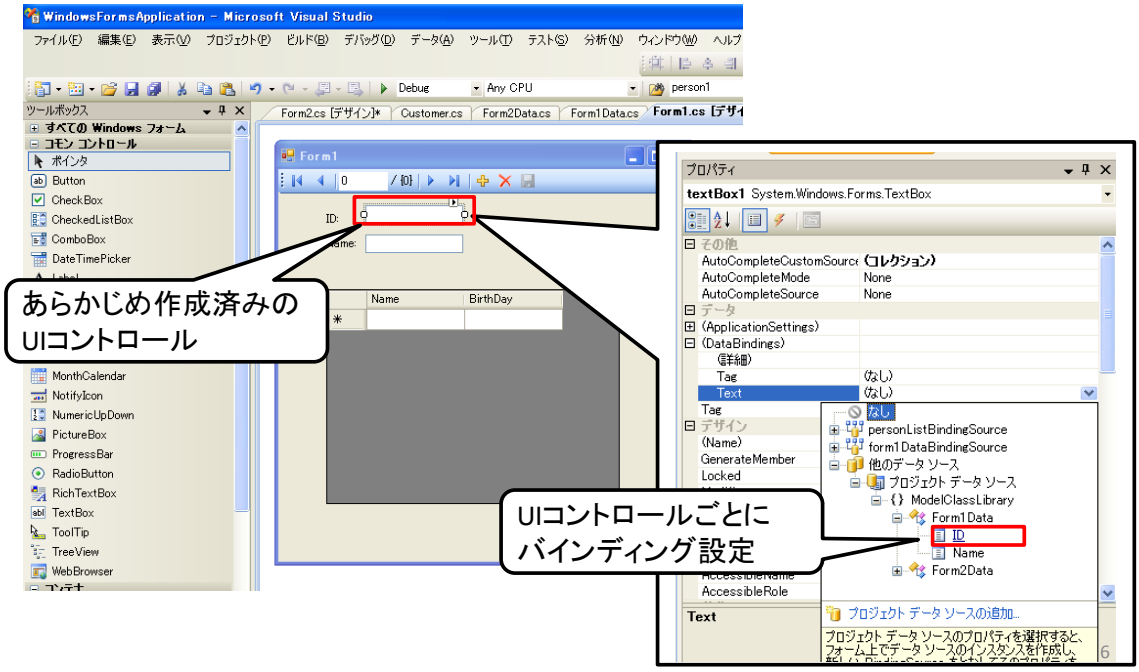

図 27 プロパティエディタを使った UI コントロールのバインド設定

◆ ErrorProvider の設定

ErrorProvider の DataSource プロパティに、バインド設定時に生成された(もしくは作成した)BindingSource クラスを設定する。
通常、入力値チェックが必要な「画面データ」クラスと対応する BindingSource の数だけ、ErrorProvider を貼り付け、DataSource プロパティにそれぞれ BindingSource を設定する。

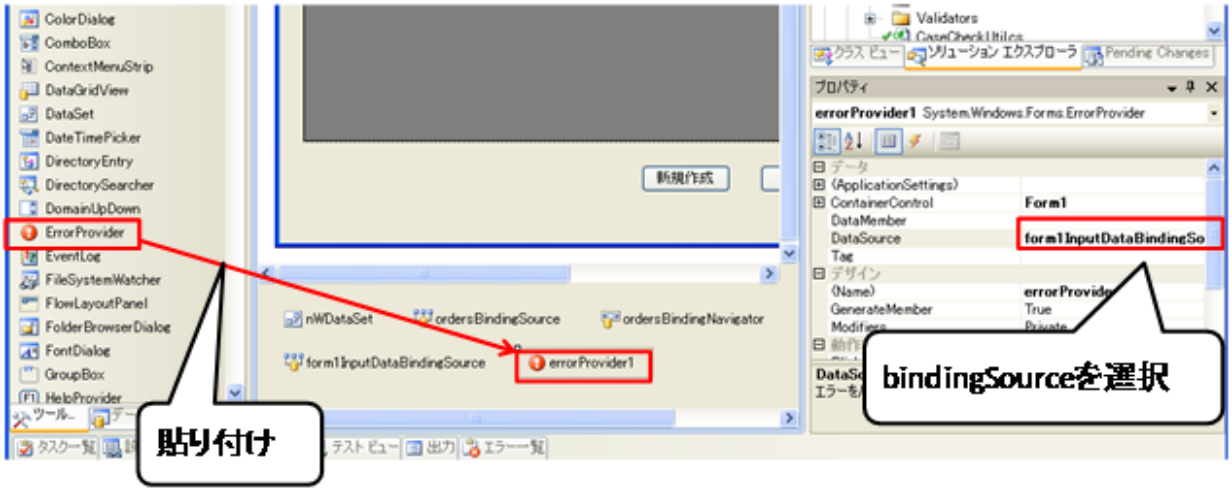

図 28 ErrorProvider の設定

なお、DataGridView にバインドしたリストデータに対応する BindingSource に対しては、DataGridView 自体がエラー表示機能持っているため、ErrorProvider の設定は不要である。

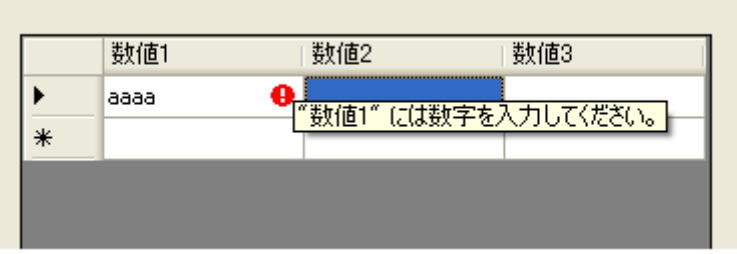

図 29 DataGridView のエラー表示

◆ FW 構成ファイル(TerasolunaFramework.config)の設定

本機能を利用するためには、FW 構成ファイル(TerasolunaFramework.config)の /configuration/unity/containers/container/extensions/add タグで、以下の UnityContainerExtension 継承クラスを記述する。

- Microsoft.Practices.Unity.InterceptionExtension.Interception クラス
  - 「CM-02 インスタンス管理機能」の AOP 機能を利用するため必要な Extension
- Terasoluna.Unity.SetInterceptorExtension クラス
  - 「CM-02 インスタンス管理機能」の AOP 機能を利用するため必要な Extension
- Terasoluna.ViewModel.Validation.ValidatableViewDataExtension クラス
  - 本機能を利用するため必要なデフォルトの Extension

以下に、TerasolunaFramework.config の記述例を示す。なお、TERASOLUNA フレームワークが提供するカスタムテンプレートを利用して当該ファイルを生成した場合、下記内容はデフォルト設定されている。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
       <extensions>
          <!-- AOPの設定 -->
          <add type="Microsoft.Practices.Unity.InterceptionExtension.Interception,
                    Microsoft.Practices.Unity.Interception,
                    Version=1.2.0.0,
                    Culture=neutral,
                    PublicKeyToken=31bf3856ad364e35" />
          <add type="Terasoluna.Unity.SetInterceptorExtension,
                    Terasoluna" />
          <!-- 画面データ機能の設定 -->
          <add type="Terasoluna.Windows.ViewModel.Validation.ValidatableViewDataExtension,
                    Terasoluna.Windows.ViewModel.Validation" />
       </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 9 TerasolunaFramework.config の記述例

◆ 入力値検証処理の実行

入力値検証処理は、通常、ロストフォーカス時、または「CL-03 イベント処理実行機能」による業務処理実行時に自動的に実施されるため、開発者は入力値検証処理を明示的に実行する必要はない。
もし、開発者が明示的に入力値検証処理のみを実施したい場合は、ValidatableViewDataManager クラスが提供する入力値検証メソッドを実行する。
以下に、ValidatableViewDataManager の主な入力値検証メソッドを示す。

表 5 ValidatableViewDataManager クラスの主な入力値検証メソッド

項番	メソッド	第 1 引数	第 2 引数	第 3 引数	説明
1	public static void ValidateAll(ValidatableRootViewData viewData, string rootRuleset)	画面データ（ルート）	ルールセット名	-	指定したルールセットで、ネストした画面データも含めて画面データ（ルート）全体を入力値検証する。
2	public static void ValidateAllByDefaultRuleset(ValidatableRootViewData viewData)	画面データ（ルート）	-	-	DefaultRuleset 属性で指定されたルールセットで、ネストした画面データも含めて画面データ（ルート）全体を入力値検証する。
3	public static void ValidateProperty(ValidatableViewData viewData, string propertyName, string ruleset)	画面データ	プロパティ名	ルールセット名	指定されたルールセットで、指定したプロパティのみを入力値検証する。
4	public static void ValidatePropertyByDefaultRuleset(ValidatableViewData viewData, string propertyName)	画面データ	プロパティ名	-	DefaultRuleset 属性で指定されたルールセットで、指定したプロパティのみを入力値検証する。

◆ 入力値検証エラーメッセージの取得

入力値検証エラーメッセージは、ロストフォーカス時、または「CL-03 イベント処理実行機能」による業務処理実行時に ErrorProvider に表示される。
ErrorProvider 以外の方法でエラーメッセージを表示したい場合は、入力値検証エラーメッセージを取得し画面表示するよう実装する必要がある。

- IDataErrorInfo インタフェース
  エラーメッセージは、「画面データ」クラスに対して、IDataErrorInfo インタフェースが定義するプロパティから取得可能である。
  IDataErrorInfo のプロパティは文字列型であるため、対象の画面データのプロパティについて複数のエラーメッセージがあった場合、デフォルトでは各メッセージを改行コード（System.Environment.NewLine）で連結したものを返却する。

表 6 IDataErrorInfo が定義するプロパティ

項番	プロパティ	説明
1	string this[string columnName] { get; }	Item プロパティ(インデクサ) 指定したプロパティ名(columnName)に対応するエラーを返却する。当該プロパティに対する単項目チェックエラーやカスタム入力チェック（ValidationResult に当該プロパティ名を指定した場合）で発生したエラーメッセージについて文字列で返却する。
2	string Error { get; }	インスタンス全体に対するエラー カスタム入力チェックで、ValidationResult に空の文字列(string.Empty)でプロパティ名を指定したエラーメッセージについて文字列で返却する。

- IViewData インタフェース
  プログラム上で、各エラーメッセージをオブジェクトとして扱いたい場合は、画面データのインタフェースである IViewData が定義するプロパティまたはメソッドを利用する。

表 7 IViewData が定義するプロパティおよびメソッド

項番	プロパティまたはメソッド	第 1 引数	説明
1	IEnumerable<ValidationErrorInfo> ValidationErrors { get; }	-	対象画面クラスの全てのエラー情報（各プロパティに対するエラーおよびインスタンスに対するエラー）のリストを返却する
2	ValidationErrorInfo GetValidationError(string propertyName)	プロパティ名	指定したプロパティに対するエラー情報を返却する。
3	ValidationErrorInfo GetValidationErrorForWholeViewData()	-	インスタンスに対するエラー情報を返却する。

ValidationErrorInfo は画面データの入力値エラー情報を1つ1つ保持するクラスで、以下のプロパティによりエラー情報を取得することができる。

表 8 ValidationErrorInfo のプロパティ

項番	プロパティ	説明
1	public string PropertyName { get; }	エラーとなったプロパティ名を取得する。 インスタンス全体に対するエラーは、空の文字列(string.Empty)を返却する。
2	public string Message { get; }	エラーメッセージを取得する。
3	public object RawData { get; }	ValidationErrorInfo に加工する前のエラー情報 (デフォルトでは、Validation AB の ValidationResults オブジェクト)

- ValidatableViewDataUtility クラス

IDataErrorInfo や IViewData のインタフェースが提供する API を使用する場合、対象の「画面データ」クラスのエラー情報しか取得できない。つまり、ネストした「画面データ」クラスについては、開発者がネストした「画面データ」クラスのプロパティをたどり、個別にエラー情報を取得する必要がある。
そこで、TERASOLUNA フレームワークでは ValidatableViewDataUtility クラスに、「画面データ」のクラス階層をたどりネストした「画面データ」の分を含む全てのエラーメッセージを取得するユーティリティメソッドを提供している。

表 9 ValidatableViewDataUtility クラスのメソッド

項番	メソッド	第 1 引数	説明
1	public static Dictionary<string,List<ValidationErrorInfo>> CollectValidationErrors(ValidatableRootViewData rootViewData)	画面データ (ルート)	ネストした「画面データ」クラスを含む全てのエラーメッセージを取得する。

戻り値は、Dictionary<string,List<ValidationErrorInfo>>型で返却されている。
Dictionary のキーは、「画面データ(ネスト)」のプロパティパスを表しており、クラス階層は半角ピリオド(”.”)、配列やコレクションの要素は角かっこ(”[]”)で表す。例えば、キーは以下のように表現される。

- キーが空の文字列(string.Empty)なら、画面データ(ルート)自身
- キーが「Person」なら、Person プロパティである画面データ(ネスト)
- キーが「PersonList[0]」なら、PersonList プロパティであるコレクションにおいて 0 番目の要素の画面データ(ネスト)
- キーが「Person.Country」なら、Person プロパティの画面データ(ネスト)にある Country プロパティである「画面データ(ネスト)」

一方、値は、キーに対応する「画面データ」で発生した入力値検証エラーのリストである。

以下に、ValidatableViewDataUtility.CollectValidationErrors メソッドを使ったエラー情報の取得例を示す。
CollectValidationErrors メソッドを使って取得したエラー情報をもとに、画面データのプロパティパスとエラーメッセージを単純にダイアログに表示している。

```
StringBuilder sb = new StringBuilder();
///ユーティリティクラスを使ってネストした画面データも含めた全てのエラーを集めて返却する
Dictionary<string, List<ValidationErrorInfo>> errorDict =
                        ValidatableViewDataUtility.CollectValidationErrors(rootViewData);
foreach (KeyValuePair<string, List<ValidationErrorInfo>> error in errorDict)
{
        string key = error.Key;
        foreach (ValidationErrorInfo errorInfo in error.Value)
        {
                sb.AppendFormat("{0}.{1}プロパティ:{2}",key, errorInfo.PropertyName,
errorInfo.Message).AppendLine();
        }
        MessageNotifier.ShowErrorMessage(context.TargetForm, sb.ToString(), "入力チェックエラー");
}
```

リスト 10　ValidatableViewDataUtility.CollectValidationErrors メソッドの使用例

出力結果は、以下の通り。

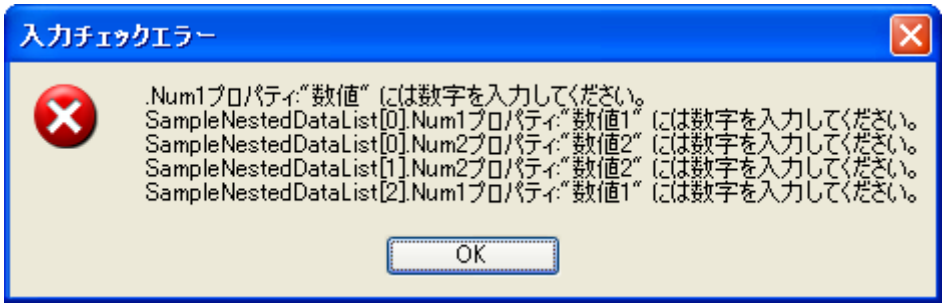

図 30 出力結果

なお、出力結果を見て分かるとおり、リストデータの入力チェックエラーに関しては、エラーメッセージだけを表示しても、ユーザが画面上の DataGridView の何列目かを特定できないなど、非常に分かりにくい。この場合、プロパティパスの文字列の要素番号を加工するなどして対処するか、このメソッドにより取得したメッセージは使用せずに、双方向バインドで DataGridView のエラー表示機能を使って自動的に対象カラムのそばにエラー表示することで対処する必要がある。

図 31 DataGridView のエラー表示

◆「画面データ」を介した DataGridView への行データの追加

プログラムから DataGridView の行(Row)データを追加したい場合は、「画面データ（ネスト）」のリストデータに新しい「画面データ（ネスト）」のオブジェクトを追加する。
追加には、ValidatableNestedViewDataList<T>.AddNew メソッド(IBindingList.AddNew メソッド)を使用する。
new 演算子を使って画面データ（ネスト）オブジェクトを生成して追加するとしても AOP が適用されず即値チェックや PropertyChanged イベントが自動発生しないので必ず上記メソッドを利用すること。

実装例を以下に示す。

```
SC_A01_01_01Conductor newConductor = ViewData.ConductorList.AddNew();
```

リスト 11　ValidatableNestedViewDataList<T>.AddNew メソッドの実行例

■ TIPS

以下では、本機能を利用する際、開発者がよく直面する問題について対処方法をまとめている。開発時に実装方法に困った場合に参考にするとよい。

◆ 画面間のデータ引き継ぎについて

TERASOLUNA フレームワークにおける画面間のデータ引き継ぎの考え方については、「CL-02 画面遷移機能」の機能説明書を参照のこと。

- AP 全体で共有するデータ
  - Singleton クラスなど、アプリケーション全体で1つしかなく、かつどこからでもアクセス可能な static な共通データクラスを実装する。
- ユースケースなど一連の業務処理を形成する画面遷移グループ内で共有するデータ
  - 「CL-02 画面遷移機能」の FormForwarder クラスを使用した画面(Form)の切り替えによる遷移の場合、「スコープ」を利用することで、スコープ内の画面間でのみアクセス可能な共通データ領域に保存することができる。
  - 「CL-02 画面遷移機能」の ContentPlaceHolder クラスを使用したパネル(Control)の切り替えによる遷移の場合、ContentPlaceHolder が各パネルのインスタンス管理をしているため、各パネルが保持する画面データを任意の場所で取得できる。
- 遷移元画面と遷移先画面間でのみ引き継ぎが必要なデータ
  - 「CL-02 画面遷移機能」では画面遷移時に、「CM-04 データコピー機能」を使用した、遷移元と遷移先の画面（または画面部品）間での「画面データ」のコピーが可能である。

◆ 階層構造をもつ画面データの双方向データバインド設定

図 32 のように、「画面データ」クラスが階層構造をもつ場合には双方向バインドの設定に注意が必要である。

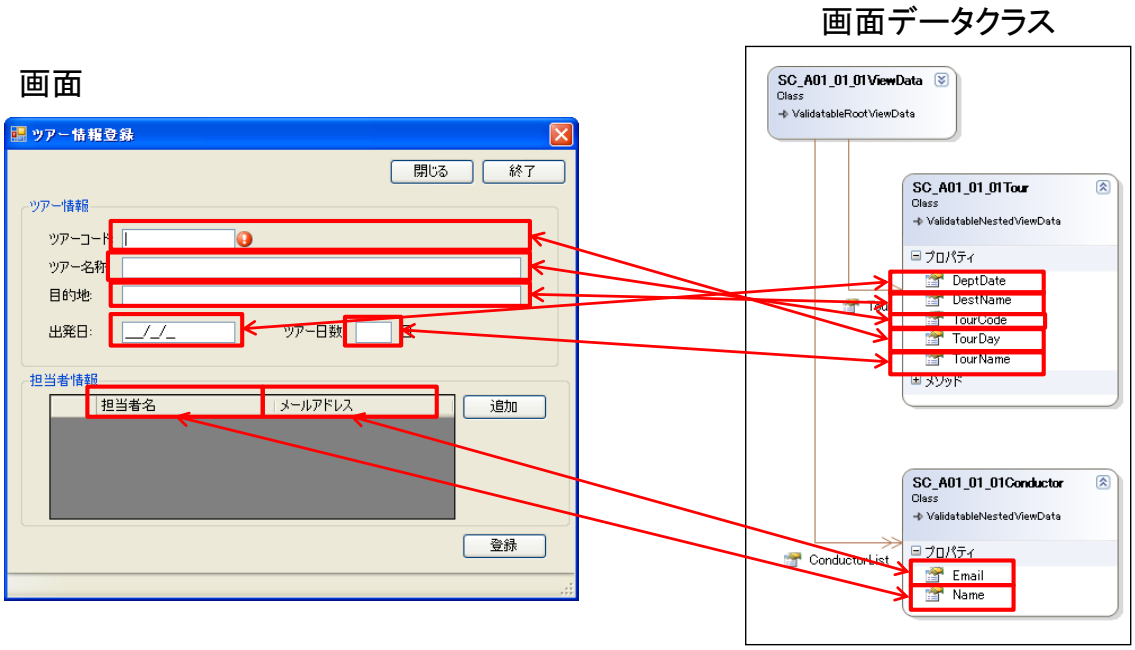

図 32 階層化された画面データのバインドの例

ネストした「画面データ」のプロパティと、画面上にある UI コントロールをデータバインドする場合、UI コントロールのバインドしたプロパティの値に、半角ピリオド(".")が含まれることがある。(手順1「データソースウィンドウによる UI コントロールの貼り付け」により、ネストした「画面データ」クラスを、通常の方法1の手順でバインド設定を実施していると発生することがある。)
このようなバインド設定になっている場合、入力チェックエラー時に ErrorProvider によるエラーが表示されない事象が発生する。

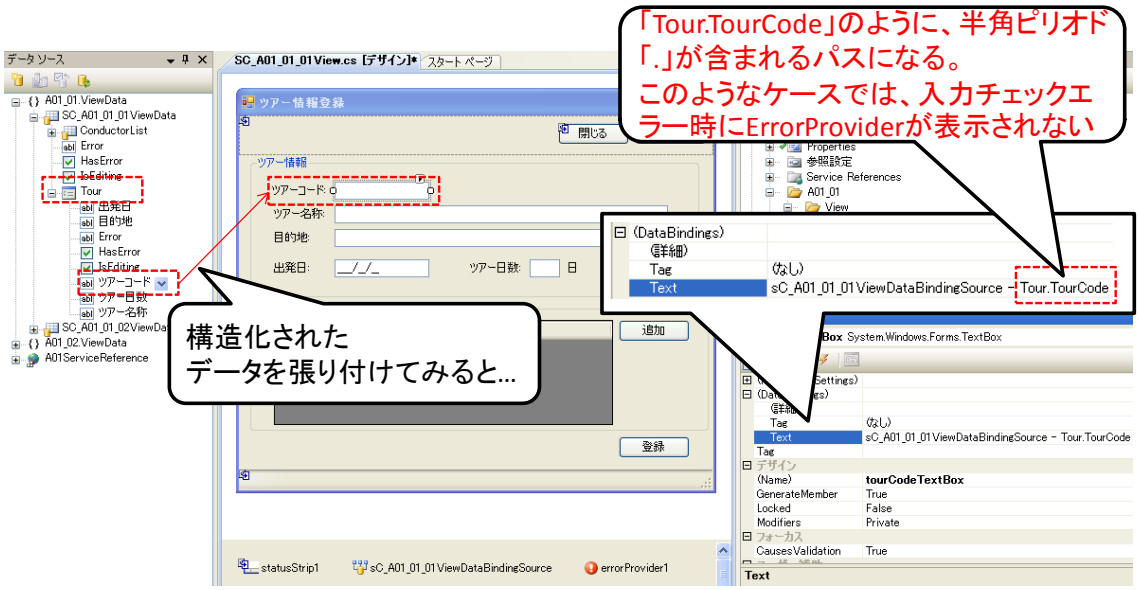

図 33 構造化された画面データのバインド設定の NG 例

このような場合は、「画面データ」クラスと同じ階層構造になるように BindingSource をチェーンさせて作成しバインドすることで、UI コントロールのプロパティのバインド設定が半角ピリオドを含むパス形式にならないようにする。

まず、「画面データ(ルート)」に対応する BindingSource を生成後、ツールボックスより BindingSource を張り付けて、ネストした画面データのプロパティに対応する BindingSource を作成する。そして、新たに作成した BindingSource の DataSource プロパティに、「画面データ(ルート)」に対応する BindingSource を設定する。また、DataMember プロパティには、ネスト「画面データ」クラスのプロパティを指定する。この時、DataMember プロパティには、リストデータ以外のプロパティが選択肢として出現しないので、プロパティ名を直接手入力する。

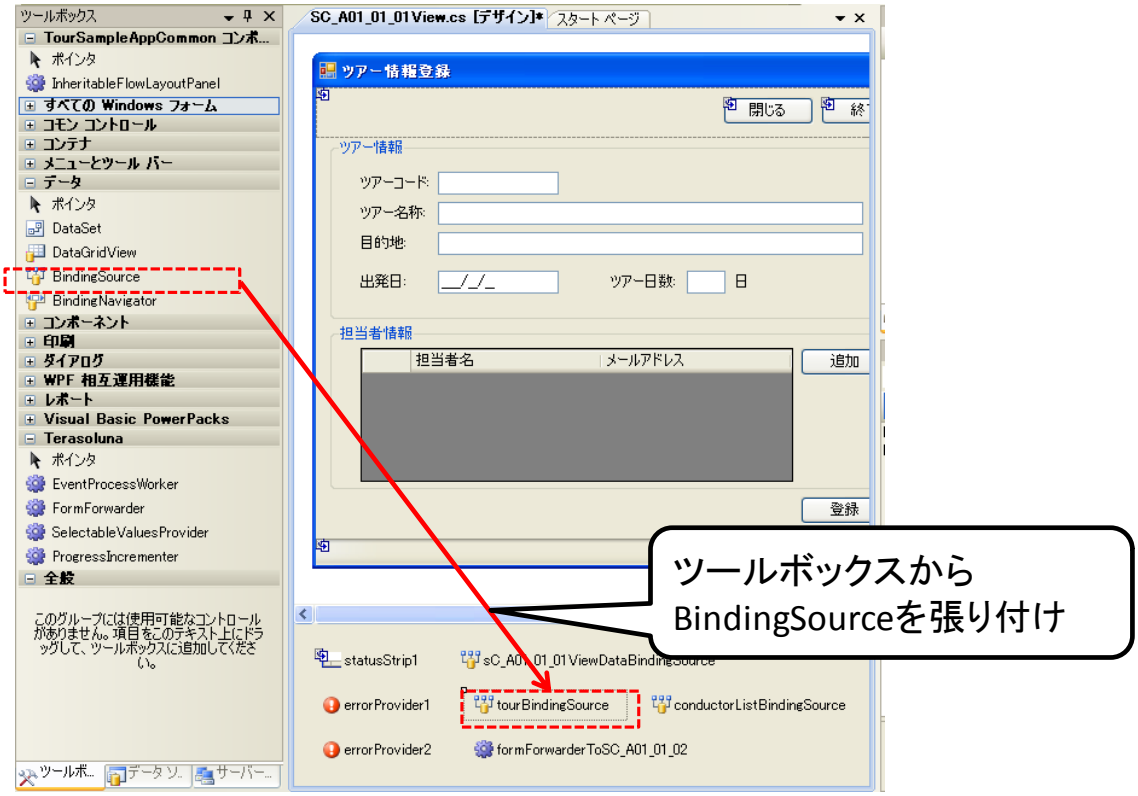

図 34 構造化された画面データのバインド設定 1

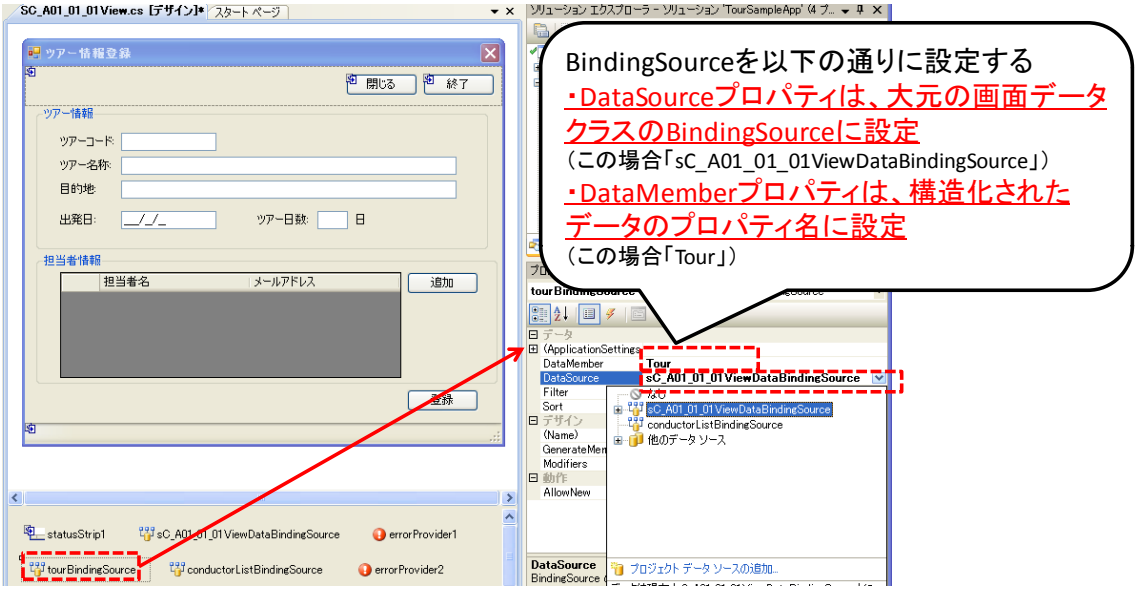

図 35 構造化された画面データのバインド設定 2

BindingSource を作成後、UI コントロールのバインド設定を、新たに作成した BindingSource を介して対象の「画面データ」クラスのプロパティ名を設定し直す。

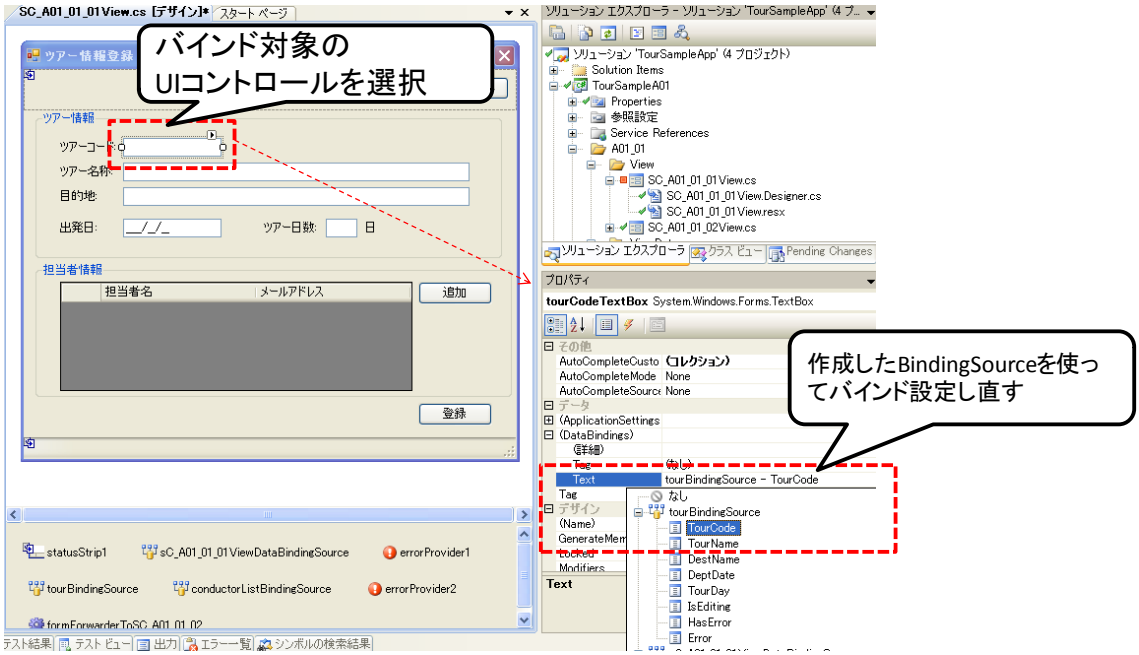

図 36　構造化された画面データのバインド設定 3

このように、UI コントロールのプロパティのバインド設定がパス形式になっている場合は、「画面データ」クラスの階層構造と同じ階層構造になるように BindingSource をチェーンさせて作成しバインドすることで、UI コントロールのプロパティのバインド設定が半角ピリオドを含むパス形式にならないようにする。
また、階層が増えた場合でも、前述の手順を繰り返すことで BindingSource をチェーンさせるよう複数設定する。

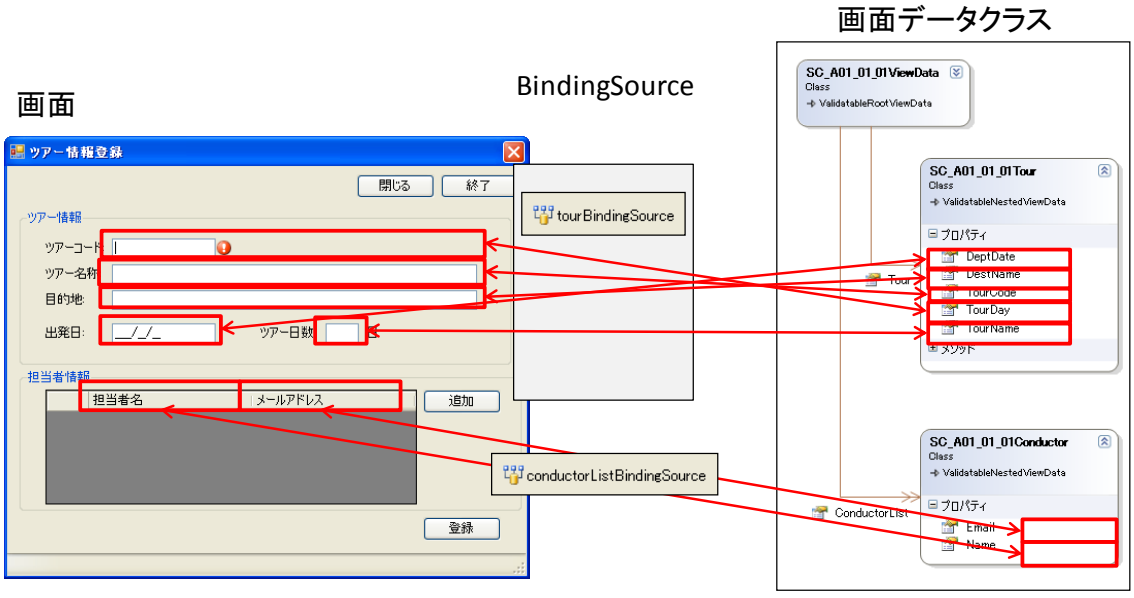

図 37　UI コントロール、画面データ、Bindingsource の対応関係

この際、各 BindingSource に対応するよう、ErrorProvier を複数作成し、DataSource プロパティに BindingSource を設定するが、DataGridView に対応する BindingSource には、ErrorProvider は不要である。

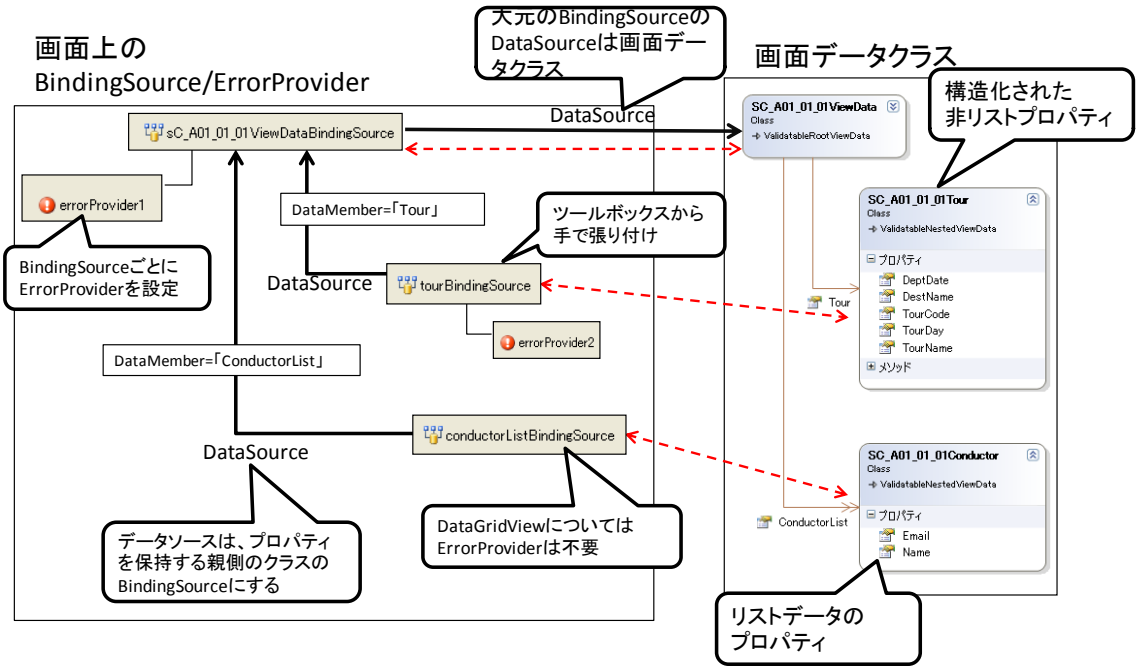

図 38 画面データ、BindingSource、ErrorProvider の対応関係

◆ ツールバー上のボタン(ToolStripButton)やメニューバー上のアイテム(ToolStripMenuItem)などを使った業務処理の注意事項

この注意事項を理解するためには、画面上の入力項目（UI コントロール）に入力した値が、なぜ即時に「画面データ」（データソース）に反映されるか、Windows Formsのデータバインドの仕組み[9]を詳しく説明する必要がある。
ユーザが画面から値を入力すると、UI コントロールのプロパティが変更される（例えば、TextBox に"aaa"と入力されると、Text プロパティが"aaa"に変わる）。プロパティに変更があると、*PropertyName*Changed パターンにより、「画面データ」クラスのバインドされたプロパティに変更が通知される（例えば、TextBox.Text なら TextChanged イベントが発生する）。
変更があったデータが「画面データ」に反映されるタイミングは、デフォルトでは、Form.Validating イベントの実行時である。このことは、Visual Studio の「フォーマットと詳細バインド」の画面で、「データソース更新モード」がデフォルトで「OnValidation」になっていることから分かる。

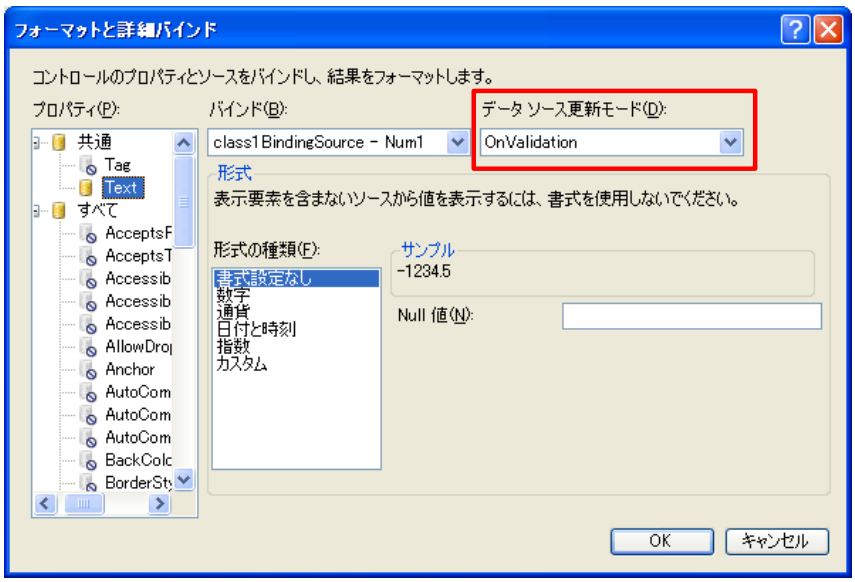

図 39 フォーマットと詳細バインドの画面

一方、Form クラスには AutoValidate プロパティが存在し、このプロパティの値に応じて、暗黙的に各コントロールの Control.Validating/Validated イベントが発生する。AutoValidate プロパティのデフォルト値は「EnablePreventFocusChange」であるが、この場合、ロストフォーカス時に暗黙的に Control.Validating/Validated イベントが発生する。
このため、ロストフォーカス時にユーザが入力した内容が「画面データ」に即時に反映される。

---

[9]・Windows フォームでのユーザー入力の検証(MSDN)
(http://msdn.microsoft.com/ja-jp/library/ms229603.aspx)
・Control.Validating イベント(MSDN)
(http://msdn.microsoft.com/ja-jp/library/system.windows.forms.control.validating.aspx)

ここで注意すべき点は、業務処理要求のためユーザがボタンクリックした時に、直前で入力した UI コントロールは、ロストフォーカスしないことがあるということである。
クリックしたボタンが System.Windows.Forms.Button クラスの場合は、クリック時にテキストボックスからボタンへフォーカスが移動するため、最後に入力した UI コントロールはロストフォーカスし画面データへの反映が正常に行われているので問題ない。

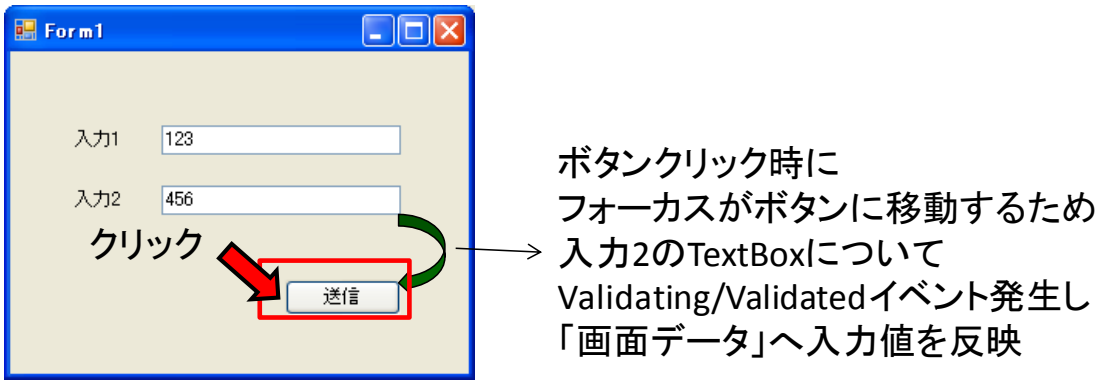

図 40 イベント処理実施のトリガが Button の場合

しかし、ツールバー（ToolStrip）上のボタン(ToolStripButton)やメニューバー（MenuStrip）上のアイテム(ToolStripMenuItem)などをクリックした場合には、ボタンをクリックしてもフォーカスは移動しないので、テキストボックスのロストフォーカスは発生しない。このため、この時点では入力したはずのデータが「画面データ」クラスに反映されていないということになる。

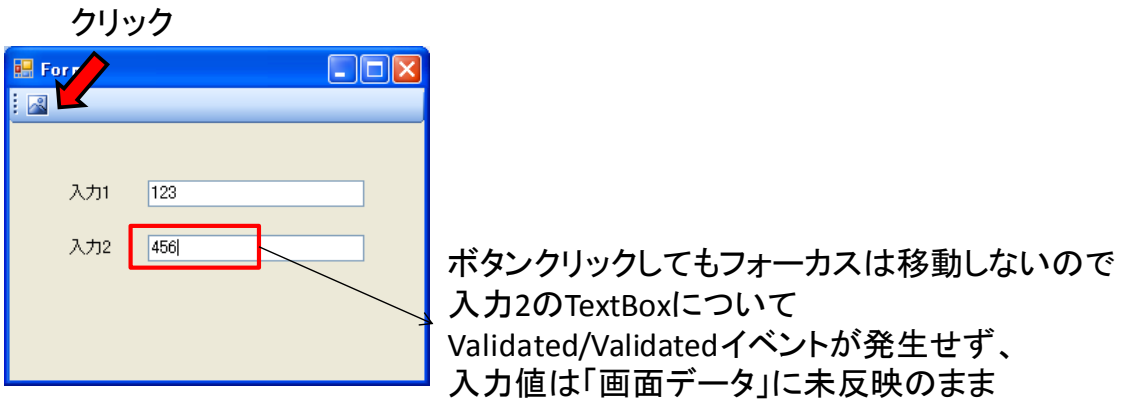

図 41 イベント処理実施のトリガが ToolStripButton の場合

このままでは、最後に入力した入力データが「画面データ」クラスに反映されない状態で、イベント処理が実行されてしまい、アプリケーションが誤動作する。
対処策として、ToolStripButton や ToolStripMenuItem の Click イベントのイベントハンドラ内で、イベント処理の実行前に Form.Validate メソッドを実行して、ロストフォーカスした最後のコントロールについて明示的に Validating/Validated イベントを発生させるようにする。
以下に、Validate メソッドの実行例を示す。

```
    public partial class Form1 : Form
    {
・・・
        private void toolStripButton1_Click(object sender, EventArgs e)
        {
            //toolStripButtonのClickイベントの最初でValidateメソッドを実行
            Validate();
            eventProcessWorker1.RunWorkerAsync();
        }
    }
```

リスト 12 Form.Validate メソッドの実行例

◆ 行の並び替えや検索可能な DataGridView の実装

画面データ(ネスト)のリストデータクラスである、ValidatableNestedViewDataList<T>クラスは、System.ComponentModel.BindingList<T>クラスを継承している。
BindingList<T>クラスは、System.ComponentModel.IBindingList インタフェースが持つ、DataGridView において単一列に対する行の並び替えやグリッド内のデータ検索に関する処理が未実装になっている。下記の MSDN ライブラリでは、BindigList<T>を継承し、並び替えや検索機能を実現する方法が記述されている。

- カスタムデータバインド
  - http://msdn.microsoft.com/ja-jp/library/ms993236.aspx

- 舞台裏で: .NET Framework 2.0 における Windows フォーム データ バインドの改良点 (第 2 部)
  - http://msdn.microsoft.com/ja-jp/library/aa480736.aspx

実装には、TERASOLUNA フレームワークが提供する、「拡張画面データ(リスト) クラス」アイテムテンプレートを使用する。これにより、並び替えや検索可能な ValidatableNestedViewDataList<T>継承クラスのひな形が生成されるので、実装の負担を大幅に軽減することが可能となる。

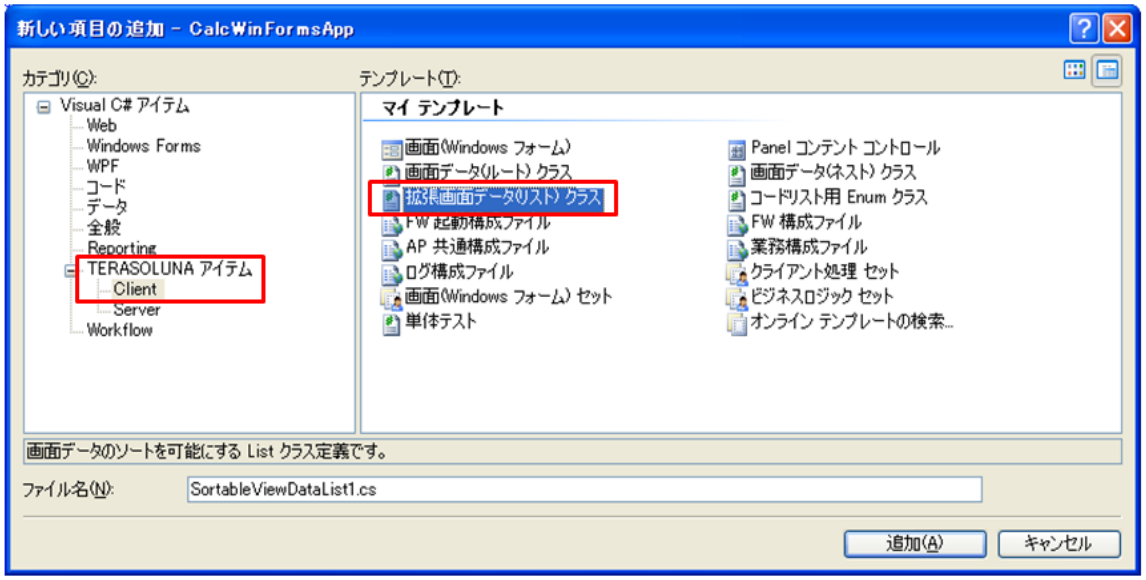

図 42「拡張画面データ(リスト) クラス」の追加

具体的な実装方法は、上記 MSDN ライブラリで紹介されている BindingList<T>の拡張方法と同様である。BindingList<T>の代わりに、ValidatableNestedViewDataList<T>を継承し、行の並び替えやグリッド内のデータ検索に関するメソッドをオーバライドし実装する。
以下に、並び替え可能な ValidatableNestedViewDataList<T>拡張クラスの実装サンプルを示す。ソートに関する virtual メンバ(SupportsSortingCore プロパティ、ApplySortCore メソッド、IsSortCore プロパティ、SortPropertyCore プロパティ、SortDirectionCore プロパティ)をオーバライドし実装している。さらに、並べ替えを解除したい場合には、RemoveSortCore メソッドを実装すること。

```
    /// ValidatableNestViewDataList<T>を継承
    public class SortableNestedViewDataList<T> : ValidatableNestedViewDataList<T>
        where T : Terasoluna.Windows.ViewModel.Validation.ValidatableNestedViewData
    {
        private ListSortDirection sortDirectionValue;
        private PropertyDescriptor sortPropertyValue;
        private bool isSortedCore = false;

        //ソートグリフ（逆三角形のマーク「▼」）を表示させるためには実装が必要
        protected override bool IsSortedCore
        {
            get
            {
                return isSortedCore;
            }
        }
        protected override PropertyDescriptor SortPropertyCore
        {
            get
            {
                return sortPropertyValue;
            }
        }
        protected override ListSortDirection SortDirectionCore
        {
            get
            {
                return sortDirectionValue;
            }
        }
        ///ソートをサポートしているかをtrueで返却
        protected override bool SupportsSortingCore
        {
            get
            {
                return true;
            }
        }
        ///  ソート処理の実装
        protected override void ApplySortCore(PropertyDescriptor prop, ListSortDirection direction)
        {
            //SortPropertyValue と SortDirectionValue を設定
            sortPropertyValue = prop;
            sortDirectionValue = direction;

            //対象プロパティでソート
            List<T> items = this.Items as List<T>;
            if (items != null)
            {
                items.Sort(new PropertyComparer<T>(prop, direction));
                isSortedCore = true;
            }
            else
            {
                isSortedCore = false;
            }
            //リスト全体が変更されたことを通知する
            OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
        }
・・・
```

```
・・・
        protected override void RemoveSortCore()
        {
            //並べ替えを取り消すときは、最低でも並べ替え状態を false に設定する
            isSortedCore = false;
            //TODO:実際に並べ替えを取り消す処理の実装
        }
    }

    class PropertyComparer<T> : IComparer<T>
    {
        private PropertyDescriptor prop;
        private ListSortDirection direction;
        public PropertyComparer(PropertyDescriptor prop, ListSortDirection direction)
        {
            this.prop = prop;
            this.direction = direction;
        }
        public int Compare(T x, T y)
        {
            object xPropValue = prop.GetValue(x);
            object yPropValue = prop.GetValue(y);
            if (direction == ListSortDirection.Ascending)
            {
                return Comparer.Default.Compare(xPropValue, yPropValue);
            }
            else
            {
                return Comparer.Default.Compare(yPropValue, xPropValue);
            }
        }
    }
```

リスト 13 並べ替え可能な ValidatableNestedViewDataList<T>継承クラス

また、バインド対象の画面データ（ネスト）のリストデータの型を、作成した ValidatableNestedViewDataList<T>継承クラスに変更する。

```
    public class SC_A01_02_01ViewData : ValidatableRootViewData
    {
・・・
        /// 元のリストデータの定義
        //public virtual ValidatableNestedViewDataList<SC_A01_02_01Tour> TourList { get; set; }

        /// 並べ替え可能なリストデータに変更した例
        public virtual SortableNestedViewDataList<SC_A01_02_01Tour> TourList { get; set; }

    }
```

リスト 14「画面データ」クラスの修正例

上記の「画面データ」クラスと双方向バインドした DataGridView は、以下のように各カラムのヘッダをクリックすることで、対象カラムに注目した行の並び替えを実施することができる。

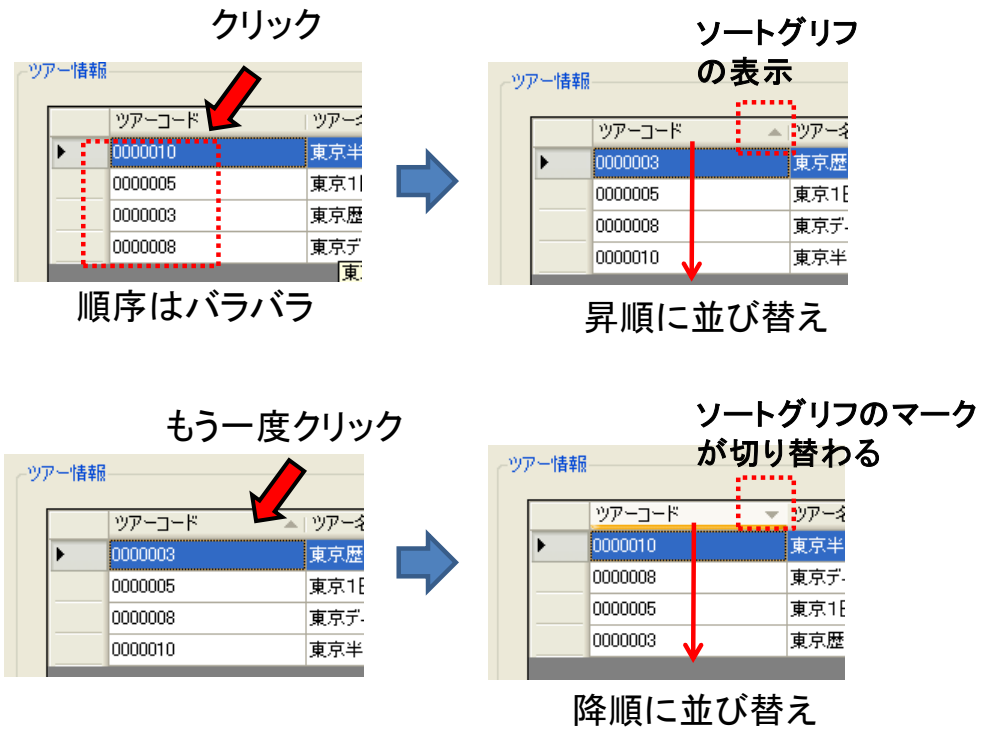

図 43 DataGridView の行の並び替え

並び替え処理の呼び出しをプログラムで記述する場合は、BindingSource.Sort プロパティを使用する。以下に、実装例を示す。
Sort プロパティに指定可能な文字列は、MSDN のドキュメント等[10]を参照のこと。

```
//ツアーコードを降順で並び替える
tourListBindingSource.Sort = "TourCode DESC";
```

リスト 15 BindingSource.Sort プロパティの使用例

並び替え処理を実施する際の注意点として、バインドされているデータは並び替わっても、DataGridView のセルの背景色などのセルスタイル(DataGridViewCellStyle)は一緒に並び替えられない。また、データの並べ替え直後に OnListChanged メソッドが実行され「ListChangedType.Reset」が通知されるため、セルスタイルがリセットされてしまう。

[10] BindingSource.Sort プロパティ
http://msdn.microsoft.com/ja-jp/library/system.windows.forms.bindingsource.sort.aspx

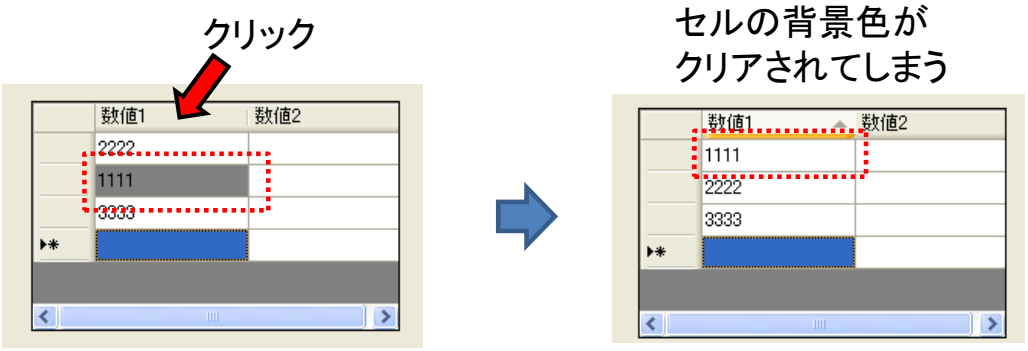

図 44 並べ替え時の DataGridView のセルスタイル

このため、並べ替え時にセルスタイルを保持する必要がある場合は、DataGridView.Sorted イベントで、CellStyle を再設定する。以下に、Sorted イベントの実装例を示す。

```
/// <summary>
/// DataGridViewのソート時
/// </summary>
private void tDataGridView1_Sorted(object sender, EventArgs e)
{
        foreach (DataGridViewRow row in tDataGridView1.Rows)
        {
                foreach (DataGridViewCell cell in row.Cells)
                {
                        //RefreshCellStyleメソッドの中でセルのスタイルを再設定
                        RefreshCellStyle(cell);
                }
        }
}
```

リスト 16 DataGridView.Sorted イベントの実装例

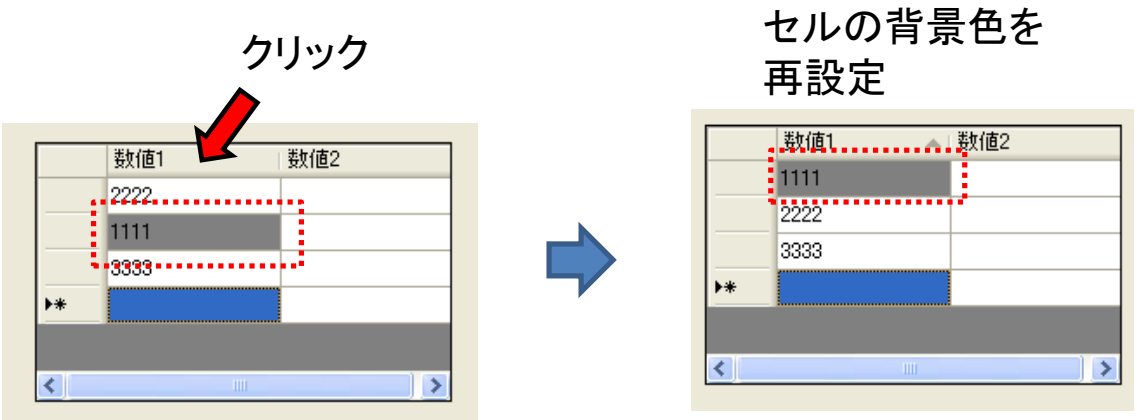

図 45 Sorted イベントを実装した場合の並べ替え時のセルスタイル

なお、「画面データ」(ネスト)のリストに対して、BindingList<T>の virtual メンバを実装しただけでは、単一列の並び替え処理しか実現できない。より高度な機能を実現したい場合には、System.CompnentModel.IBindingListView インタフェースを実装することで、複数列に対する並び替え機能や、フィルタ機能を実現することができる。
IBindingListView インタフェースについては、以下の MSDN のドキュメントを参照のこと。

- IBindingListView インタフェース
  - http://msdn.microsoft.com/ja-jp/library/system.componentmodel.ibindinglistview.aspxp

◆ カスタムコントロールの双方向バインディング

(1) 単純データバインド可能なカスタムコントロール(画面部品)の作成

単純バインドを実現させるためには、以下の実装を実施すること。詳細は、MSDN のドキュメント[11]を参照すること。

- 対象のカスタムコントロールクラスに DefaultBindingProperty 属性を付与
  - パラメータにバインド対象のプロパティ名を記述
  - 「データソースウィンドウ」からの貼り付けの際、指定したプロパティで自動的にバインド設定される。
- バインド対象のプロパティに Bindable 属性を付与
  - パラメータを true にする。
- 「*PropertyName*Changed パターン」[12]の実装
  - 新しく定義したバインド可能なプロパティ名に"Changed"を連結した名前を持つイベントを定義し、そのプロパティに変更があった場合に、そのイベントを発生させるように実装する。

単純データバインドの1例として、画面データの string 型プロパティとカスタムコントロールをバインドするケースがある。

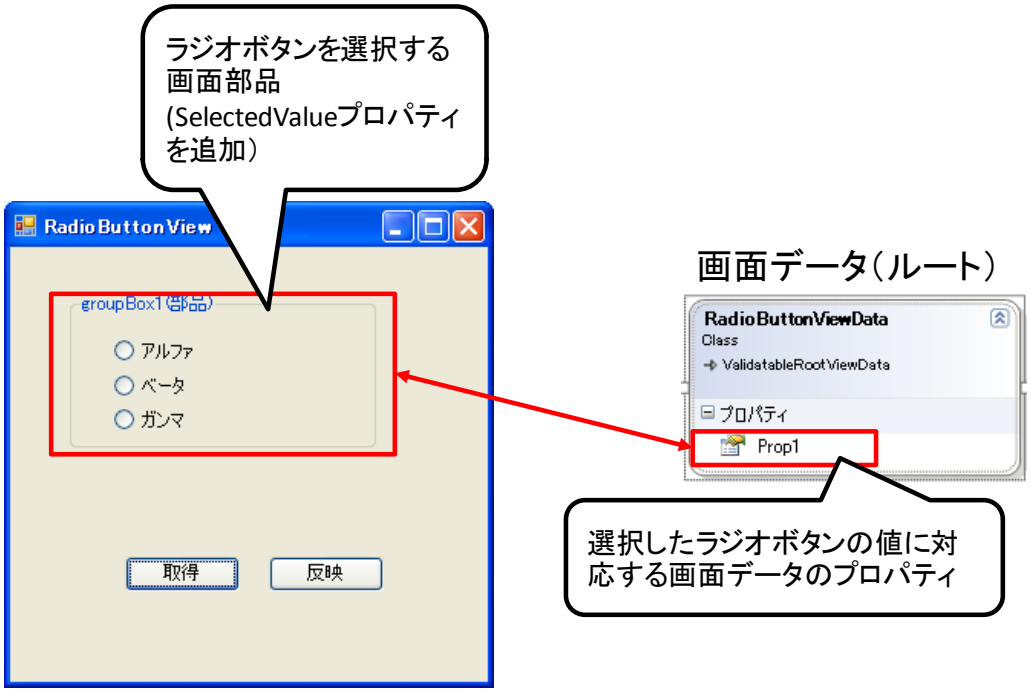

図 46 カスタムコントロールのイメージ

---

[11] チュートリアル:単純データ バインディングをサポートするユーザーコントロールの作成(MSDN)
http://msdn.microsoft.com/ja-jp/library/ms171926.aspx

[12] 方法:PropertyNameChanged パターンを適用する(MSDN)
http://msdn.microsoft.com/ja-jp/library/ms229615.aspx

この場合、UI コントロール側のバインド対象のプロパティとして、string 型、または int,enum、bool といった値型のプロパティを新規に追加する。以下に、図 46 に示したラジオボタンのグループを部品化したコントロールの例を示す。

```
[DefaultBindingProperty("SelectedValue")]
public partial class UserControl1 : UserControl
{
    private const string Alpha = "Alpha";
    private const string Beta = "Beta";
    private const string Gamma = "Gamma";

    private string selectedValue = null;
    public UserControl1()
    {
        InitializeComponent();
    }

    public event EventHandler SelectedValueChanged;

    [Bindable(true)]
    public string SelectedValue
    {
        get
        {
            return selectedValue;
        }
        set
        {
            if (!string.Equals(selectedValue, value, StringComparison.Ordinal))
            {
                selectedValue = value;
                switch (selectedValue)
                {
                    case Alpha :
                        alphaRadioButton.Checked = true;
                        break;
                    case Beta:
                        betaRadioButton.Checked = true;
                        break;
                    case Gamma :
                        gammaRadioButton.Checked = true;
                        break;
                    default:
                        break;
                }
                OnSelectedValueChanged(EventArgs.Empty);
            }

        }
    }
・・・
```

DefautlBindingProperty 属性の付与

*PropertyName*Changed パターンの実装
この例では
“SelectedValue” +” Changed”

単純データバインド対象のプロパティ
Bindable 属性を付与

現在のプロパティ値と異なる値がセットされようとしているのかをチェック

プロパティ変更時には、
*PropertyName*Changed イベントを発生させる

```
・・・

        protected virtual void OnSelectedValueChanged(EventArgs e)
        {
            EventHandler handler = SelectedValueChanged;
            if (handler != null)
            {
                handler(this, e);
            }
        }

        private void alphaRadioButton_CheckedChanged(object sender, EventArgs e)
        {
            if (alphaRadioButton.Checked)
            {
                SelectedValue = Alpha;
            }
        }

        private void betaRadioButton_CheckedChanged(object sender, EventArgs e)
        {
            if (betaRadioButton.Checked)
            {
                SelectedValue = Beta;
            }
        }
```

*PropertyName*Changed イベントの発生

ラジオボタンがチェックされた時もプロパティ値が変更し *PropertyName*Changed イベントが発生

リスト 17 カスタムコントロールの実装例

作成したカスタムコントロールと画面データのバインド設定は、通常の画面部品と同様に、バインド設定を行う。

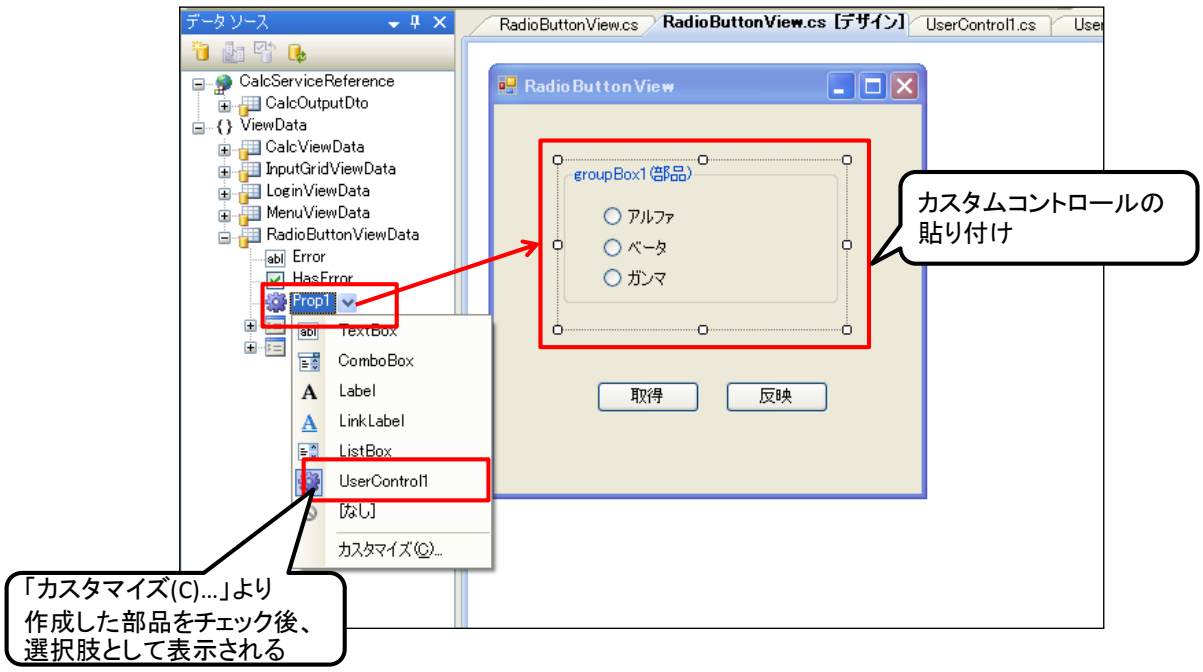

図 47 「データソースウィンドウ」でのカスタムコントロールの貼り付けるケース

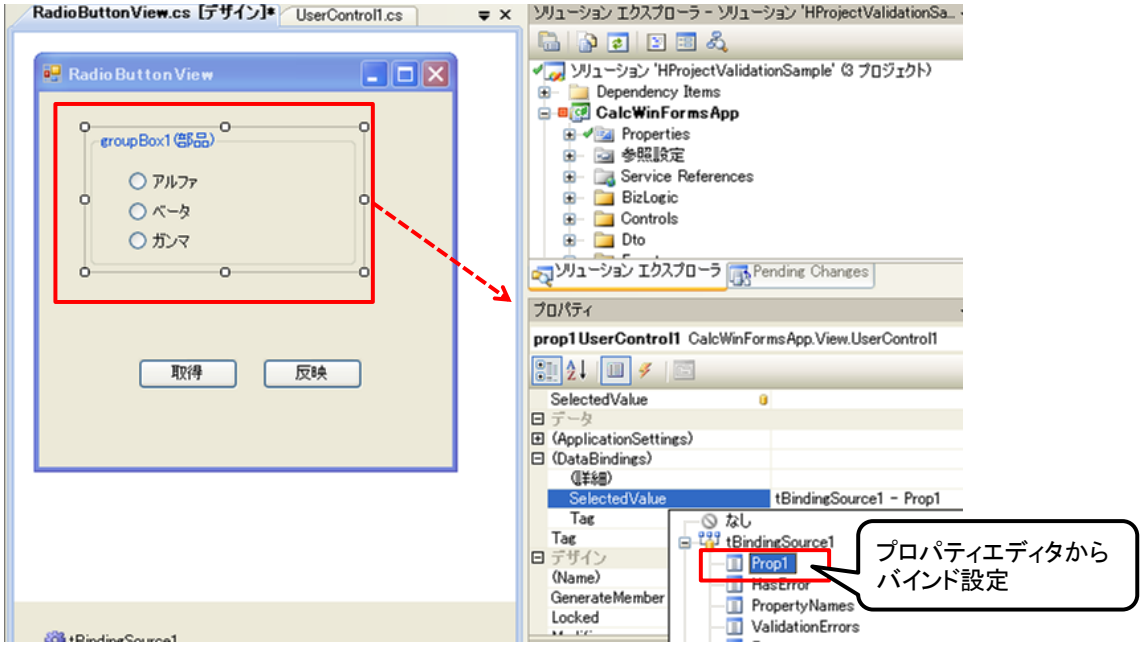

図 48 ツールボックスから張り付けたカスタムコントロールを後からバインド設定するケース

単純データバインドのもう1つの例として、「画面データ(ネスト)」にバインドされたカスタムコントロールを利用するケースがある。
「画面データ」の構造化が進むと、共通「画面データ」クラスに対応する共通画面部品を作成するケースが出てくる。

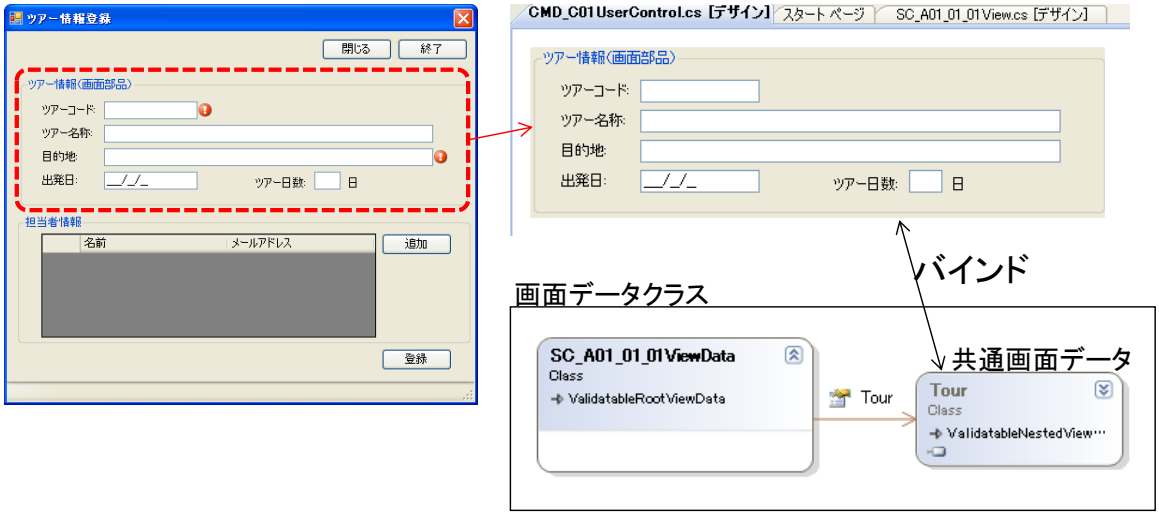

図 49 カスタムコントロールのイメージ

このようなケースの場合、カスタムコントロールのデザイナ画面上で、「画面データ(ネスト)」とカスタムコントロールの各 UI コントロールのバインド設定、ErrorProvider の設定を行う。

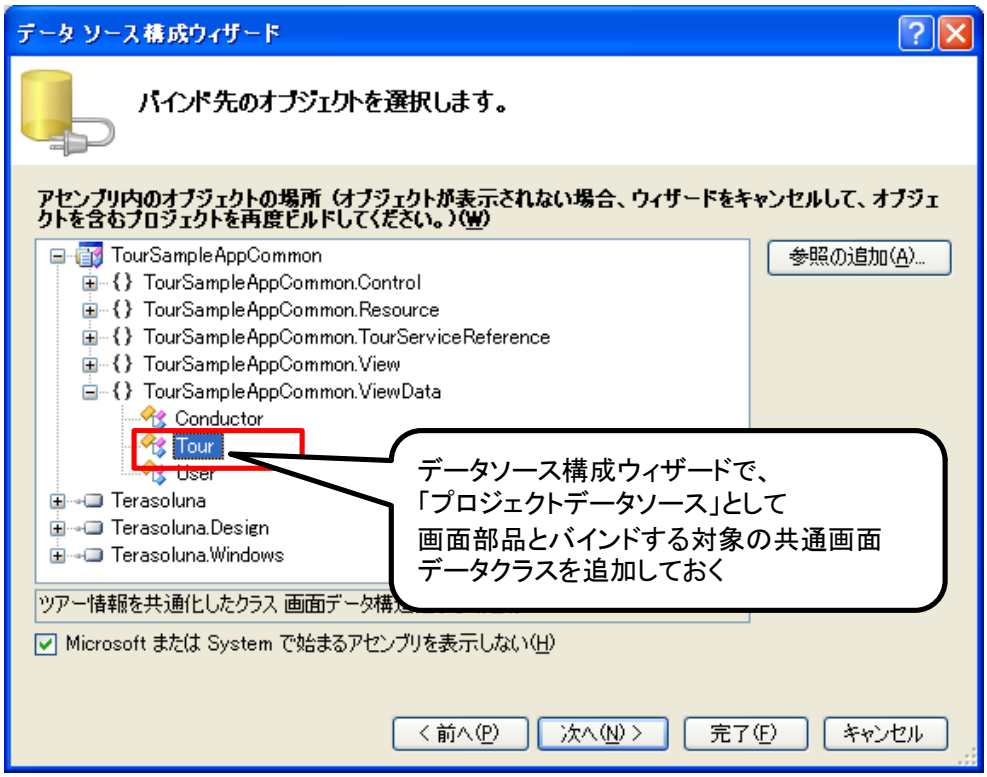

図 50 データソース構成ウィザード

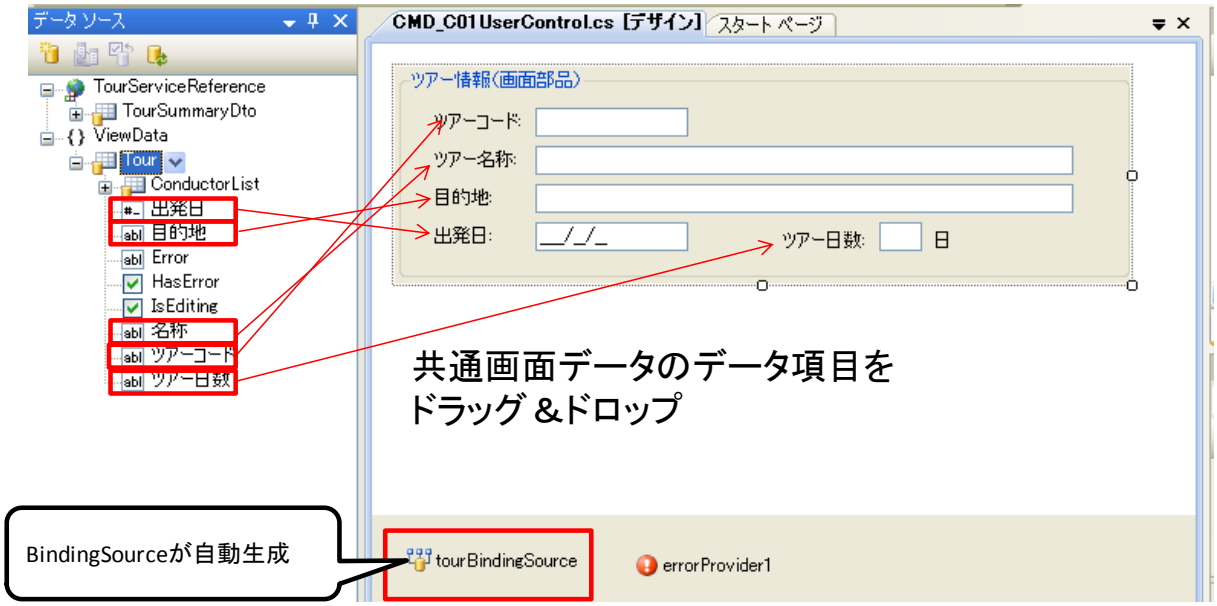

図 51 カスタムコントロールにおけるバインド設定

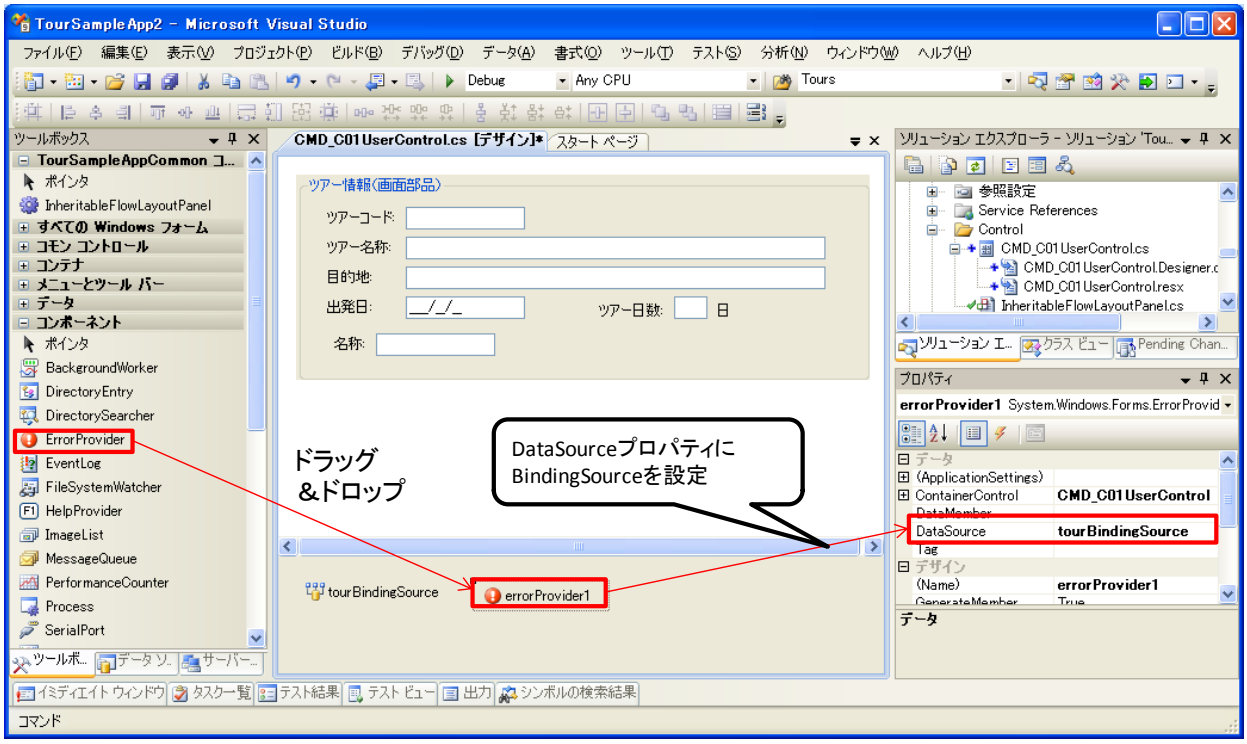

図 52 カスタムコントロールの ErrorProvider の貼り付け

また、バインド対象のプロパティに対して、Bindable 属性の付与や *PropertyName*Changed パターンといった単純データバインドに必要な実装を行う[13]。
以下に実装例を示す。

```
public partial class CMD_C01UserControl : UserControl
{
    public CMD_C01UserControl()
    {
        InitializeComponent();
    }

    [Bindable(true)]
    public virtual Tour Tour
    {
        get
        {
            return tourBindingSource.DataSource as Tour;
        }
        set
        {
            if (tourBindingSource.DataSource == value)
            {
                return;
            }
            if (value == null)
            {
                tourBindingSource.DataSource = typeof(Tour);
            } else {
                tourBindingSource.DataSource = value;
            }
            OnTourChanged(EventArgs.Empty);
        }
    }

    public event EventHandler TourChanged;

    protected virtual void OnTourChanged(EventArgs e)
    {
        EventHandler handler = TourChanged;
        if (handler != null)
        {
            handler(this, e);
        }
    }
}
```

バインド対象の画面データ(ネスト)型のプロパティ
Bindable 属性を付与

プロパティ変更時には、
*PropertyName*Changed イベント
を発生させる

*PropertyName*Changed パターンの実装
この例では“Tour” +” Changed”

リスト 18 カスタムコントロールの実装例

[13] データソースウィンドウによる UI コントロールの貼り付けでは双方向バインドの設定が実施できない画面部品なので、DefaultBindingProperty 属性の付与は不要。

次に、「画面データ(ネスト)」とバインド設定されたカスタムコントロールを画面に張り付け、張り付けたカスタムコントロールの「画面データ(ネスト)」に対応するプロパティと「画面データ(ルート)」の対応するプロパティの間で双方向データバインドを設定する。
このとき、「画面データ」クラスとのバインドの設定は、(方法 1)や(方法2)の「データソースウィンドウ」を使った貼り付けでは実施できないため、(方法3)のプロパティエディタを使ったバインド設定をする必要がある。

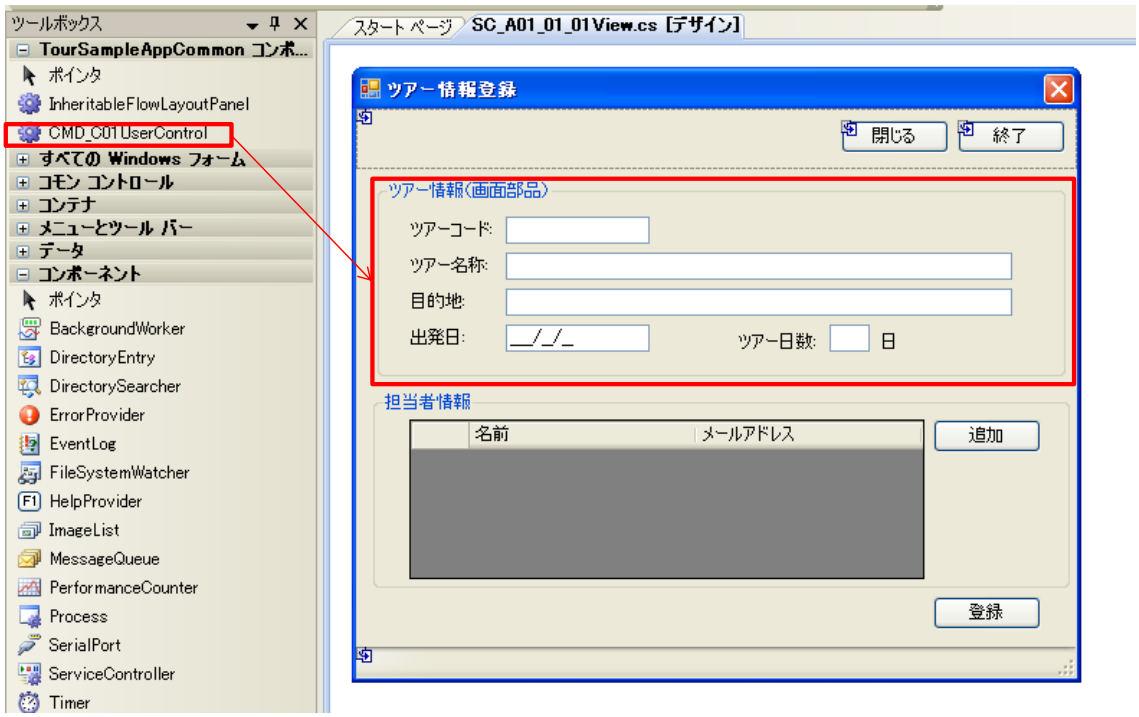

図 53 カスタムコントロールの貼り付け

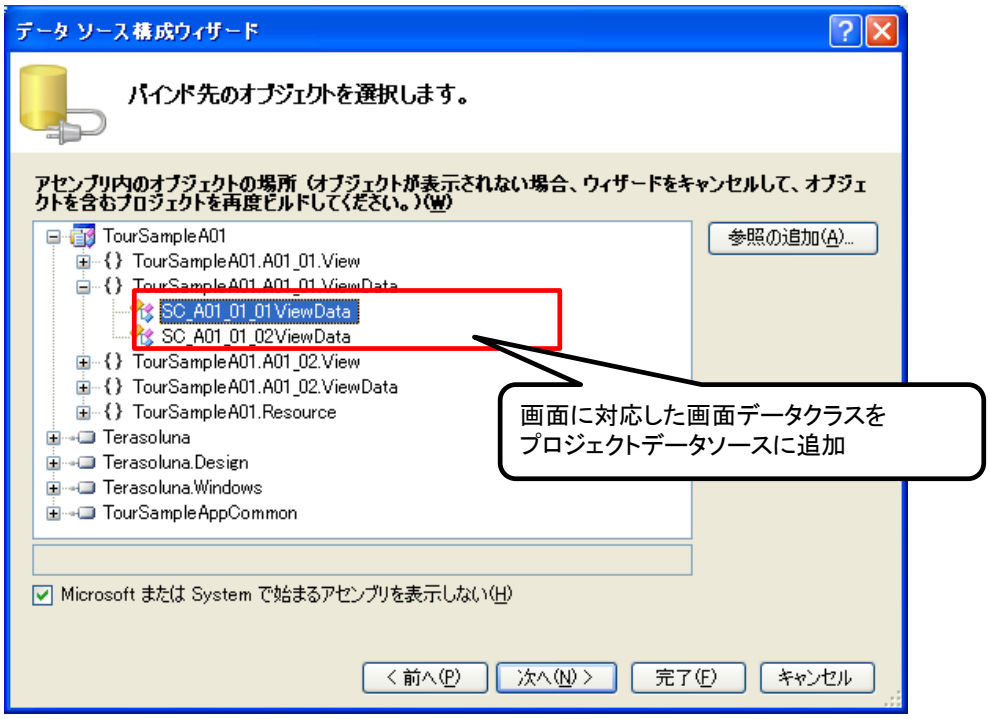

図 54 データソース構成ウィザード

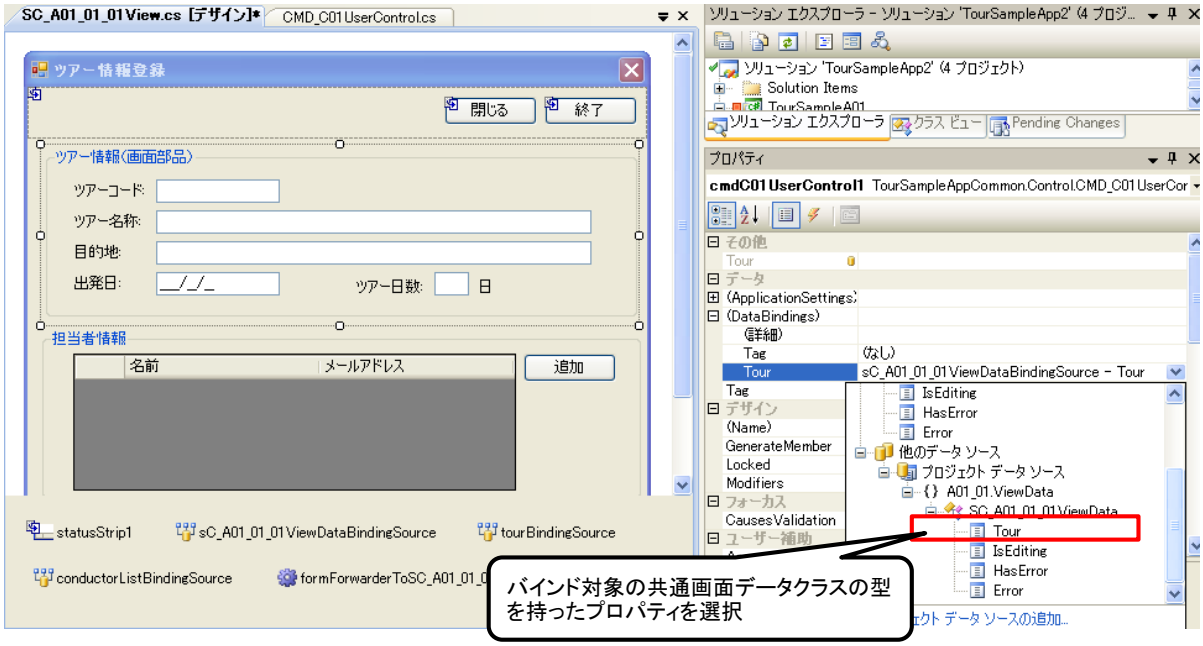

図 55 カスタムコントロールを張り付けた画面クラスにおけるバインド設定

(2) 複合データバインド可能な画面部品(カスタムコントロール)の作成

DataGridView のようなテーブルデータを扱うカスタムコントロールでは複合データバインドの作成、ComboBox のようなコードリストより選択表示するといったカスタムコントロールでは検索データバインドの作成が必要である。詳細は以下の MSDN のドキュメントを参照し実装すること。

- 「チュートリアル ： 複合データ バインディングをサポートするユーザーコントロールの作成」
  - http://msdn.microsoft.com/ja-jp/library/ms233813.aspx
- 「チュートリアル ： 検索データ バインドをサポートするユーザーコントロールの作成」:
  - http://msdn.microsoft.com/ja-jp/library/ms233787.aspx

◆ UI コントロールのエラーメッセージの表示方法の変更

ErrorProvider を使用せずに、各 UI コントロールに独自のエラー表示処理を実現する場合、以下の実装を行う。

- TextBox など単純データバインドしているコントロールの場合
  双方向バインディングの完了時イベントである、BindingSource.BindingComplete イベントのイベントハンドラ内で、バインド時のエラー状態をチェックし、コントロールに対する独自のエラー表示処理を実装する。
  以下に、実装した例を示す。
  この例では、入力チェックエラー時に、対象コントロールの背景色を変更し、ツールチップでエラーメッセージを表示するように BindingSource を拡張している。

```
public class TBindingSource : BindingSource
{
    private ToolTip toolTip;

    public TBindingSource() : base()
    {
        Initialize();
    }

    public TBindingSource(IContainer container) : base(container)
    {
        Initialize();
    }

    public TBindingSource(object dataSource, string dataMember)
                                                    : base(dataSource, dataMember)
    {
        Initialize();
    }

    private void Initialize()
    {
        this.toolTip = new ToolTip();
        //エラー表示するイベントハンドラの挿入
        this.BindingComplete +=
            new BindingCompleteEventHandler(TBindingSource_BindingComplete);
    }

・・・
```

```
・・・
        //エラー表示処理の実装
        protected virtual void TBindingSource_BindingComplete
                                        (object sender, BindingCompleteEventArgs e)
        {
            if (e.Binding.Control is UserControl)
            {
                ///部品などユーザコントロールのときには処理対象外
                return;
            }
            if (e.BindingCompleteState != BindingCompleteState.Success)
            {
                //背景色とTooltipへのメッセージ表示
                e.Binding.Control.BackColor = Color.Aqua;
                string errorText = e.ErrorText;
                if (!string.IsNullOrEmpty(errorText))
                {
                    toolTip.SetToolTip(e.Binding.Control, errorText);
                }
            }
            else
            {
                //成功時は元に戻す
                e.Binding.Control.BackColor = Color.White;
            }

        }
    }
```

リスト 19　BindingSource.BindingComplete イベントの実装例

標準の BindingSource を使用する代わりに、拡張 BindingSource を使って「画面データ」クラスとコントロールをバインディングし、アプリケーションを実行すると、エラー表示は以下のようになる。

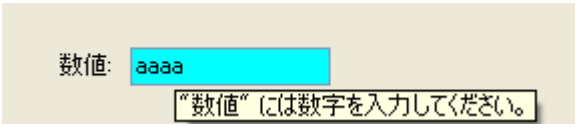

図 56 エラー表示の例

- DataGridView の場合

DataGridView で、各セルに独自のエラー表示処理を実施するには、セルのエラー時に発生する DataGridView.CellErrorTextNeeded イベントで、エラー表示処理を実装する。
以下に、実装例を示す。
この例では、入力チェックエラー時に、対象セルの背景色を変更し、ツールチップでエラーメッセージを表示するように拡張している。

```
    public class TDataGridView : DataGridView
    {
        public TDataGridView()
            : base()
        {
            //エラー表示するイベントハンドラの挿入
            this.CellErrorTextNeeded +=
new DataGridViewCellErrorTextNeededEventHandler(TDataGridView_CellErrorTextNeeded);
            //エラーアイコンを非表示設定にする
            this.ShowCellErrors = false;
        }

        //エラー表示処理の実装
        protected virtual void TDataGridView_CellErrorTextNeeded
                                (object sender, DataGridViewCellErrorTextNeededEventArgs e)
        {
            if (!string.IsNullOrEmpty(e.ErrorText))
            {
                //背景色の変更とTooltipの表示
                this[e.ColumnIndex, e.RowIndex].Style.BackColor = Color.Aqua;
                this[e.ColumnIndex, e.RowIndex].ToolTipText = e.ErrorText;
            }
            else
            {
                //正常な場合はもとに戻す
                this[e.ColumnIndex, e.RowIndex].Style.BackColor = Color.White;
                this[e.ColumnIndex, e.RowIndex].ToolTipText = null;
            }
        }
    }
```

リスト 20 DataGridView.CellErrorTextNeeded イベントの実装例

アプリケーションを実行すると、エラー表示は以下のようになる。

図 57 エラー表示の例

● ダイアログ等にエラーを表示する場合

イベント処理実行時の入力値検証エラーでエラーダイアログに、エラーメッセージを列挙して全て表示を実施したい場合、「CL-03 イベント処理実行機能」の、「入力値検証」フェーズでエラーメッセージをダイアログ表示するように拡張する。

詳細は、「CL-03 イベント処理実行機能」の機能説明書を参照すること。

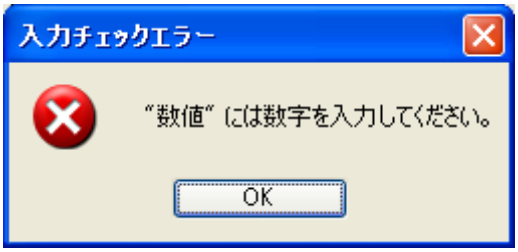

図 58 機能拡張したエラーダイアログの動作イメージ

◆ 複雑な画面レイアウトと画面データの設計

リッチクライアント AP の場合、画面遷移処理の実現方法として、画面(Form)の表示(Show)、クローズ/非表示(Close/Hide)を組み合わせる方法以外に、タブ切り替えや、パネル切り替えにより、1つの画面内で遷移と同様の動作を実現することがある。各開発プロジェクトは、このような複雑な画面のレイアウト構造に合わせて画面データの階層構造を工夫する必要がある。

ここでは、複雑な画面レイアウトに対する「画面データ」クラスの基本的な設計指針を示す。これを参考に各プロジェクトの画面遷移処理パターンに合わせて適切な「画面データ」クラスの設計指針を検討するとよい。

(1) タブ切り替えによる画面

画面項目が多く、1画面内に表示しきれないような場合、TabControl を使って、タブを切り替えることがある。タブ切り替えによる画面データの設計には、以下の 2 つの方式がある。

➢ タブごとにイベント処理が独立しているケース

タブごとに「画面データ(ルート)」を対応させるため、1 画面に複数の「画面データ(ルート)」が存在する。各タブは、通常の画面クラスと同じように独立して、設計／開発する。

ただし、TERASOLUNA フレームワークでは、通常、画面と「画面データ(ルート)」が1対1に対応するように設計するため、大元の画面が保持する「画面データ(ルート)」のプロパティは、現在表示しているタブの「画面データ(ルート)」を返すように実装する必要がある。

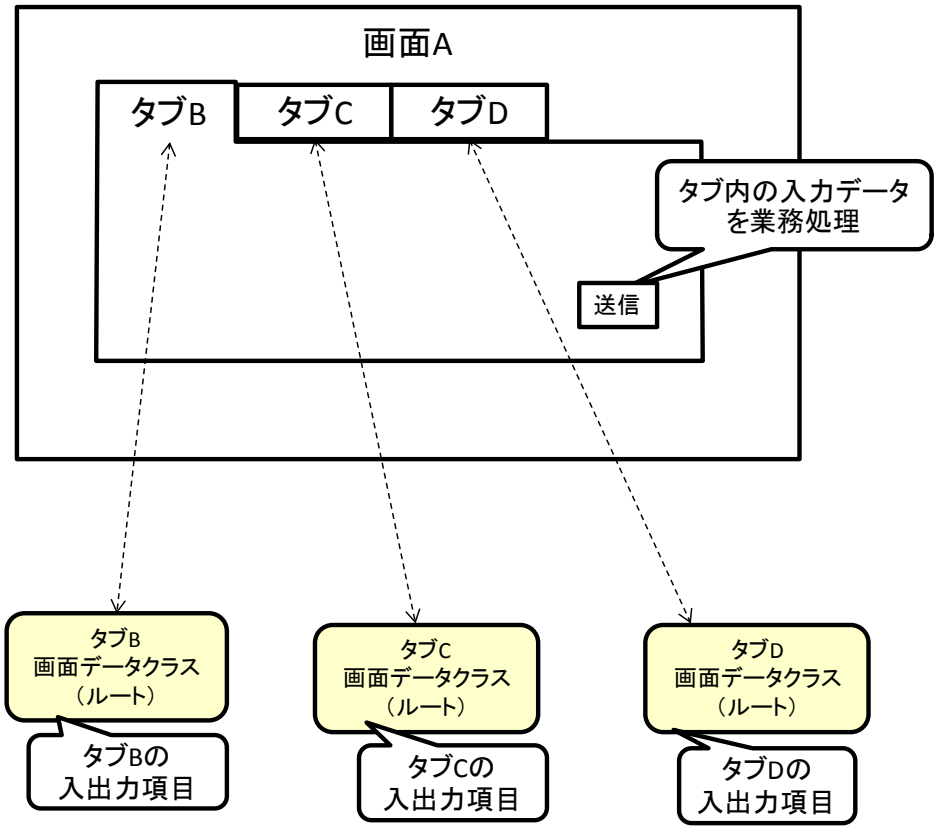

図 59 タブごとにイベントが独立している場合の「画面データ」クラスの設計

なお、大元の「画面データ（ルート）」の切り替えは、タブの表示状態が切り替わるタイミング（TabControl.SelectedIndexChanged イベント）で実施するとよい。以下に実装の例を示す。

```
[ScreenId("SC_A01View")]
public partial class SC_A01View : Form
{
    public object ViewData { get; set; }

    public SC_A01View()
    {
        InitializeComponent();
    }

    private void SC_A01View_Load(object sender, EventArgs e)
    {
        //初期画面表示時には、SelectedIndexChangedイベントが呼ばれないため
        RefreshTabPage();
    }

    private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
    {
        //タブが切り替わるたびにViewDataを切り替え
        RefreshTabPage();
    }

    private void RefreshTabPage()
    {
        TabPage tab = tabControl1.SelectedTab;
        SC_A01ContentControl a01 = tab.Controls[0] as SC_A01ContentControl;
        if (a01 != null)
        {
            ViewData = a01.ViewData;
        }

        SC_A02ContentControl a02 = tab.Controls[0] as SC_A02ContentControl;
        if (a02 != null)
        {
            ViewData = a02.ViewData;
        }
    }
}
```

リスト 21 Tab 利用時の「画面データ」切り替えの実装例

➢ タブ全体でイベント処理が共有されるケース

各タブの外側にボタンが配置される場合には、イベント発生時に、タブ全体のデータを扱う必要がある。この場合、画面クラスと「画面データ(ルート)」を 1 対 1 で対応させ、かつタブごとに、「画面データ(ネスト)」を作成する。

また、タブの外側にボタンを配置しつつ、内部のタブごとにもボタンが配置されるというハイブリッド型のケースもある。この場合、1つのタブ内の入力データを業務処理する場合は、対象の「画面データ(ネスト)」のみ入力値検証を実施するように RulesetMapping 属性を定義する。また、画面全体の入力データを業務処理する場合は、全ての画面データ(ネスト)を入力チェックするように RulesetMapping 属性を定義する。

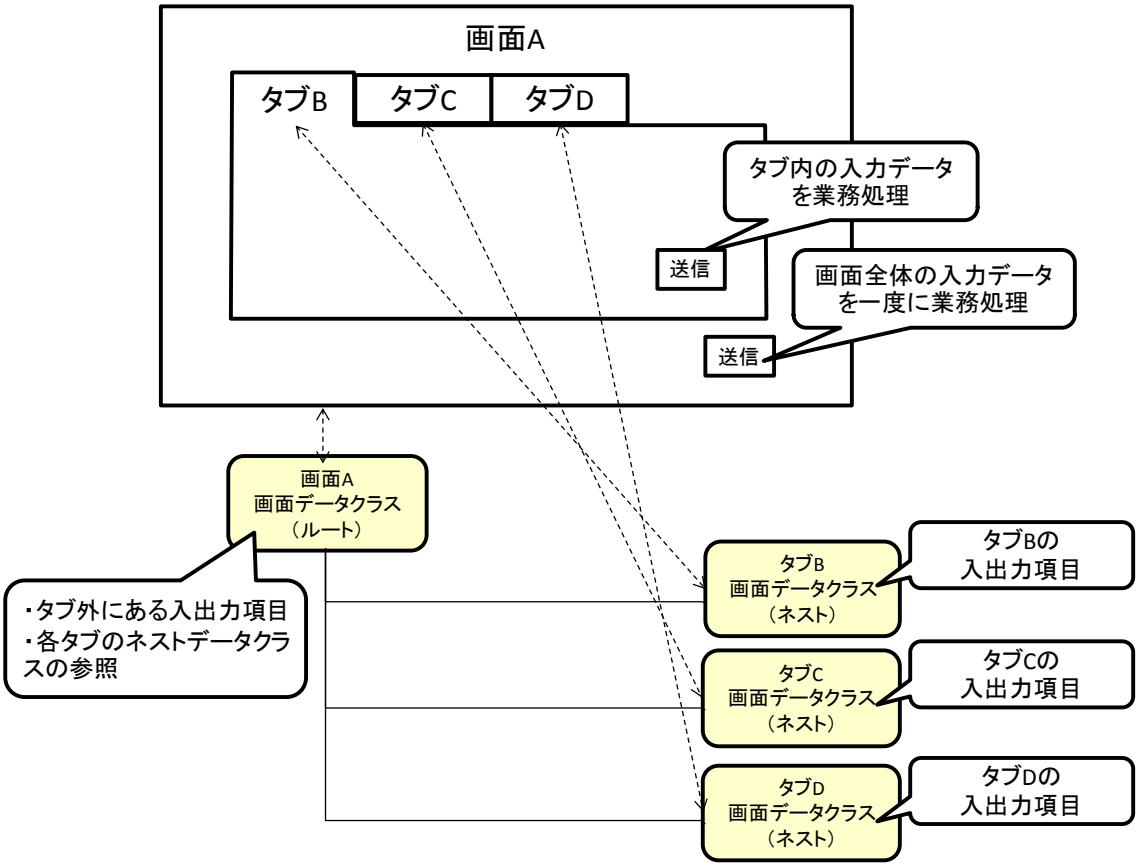

図 60 タブ全体でイベント処理が共有される場合の「画面データ」クラスの設計

(2) パネル切り替えによる画面

パネル切り替えによる画面遷移を行う場合、「CL-02 画面遷移機能」が提供する ContentPlaceHolder クラスを使用する。ContentPlaceHolder の使用方法の詳細は、「CL-02 画面遷移機能」の機能説明書を参照すること。

ContatePlaceHolder を使用する場合、パネルごとに「画面データ（ルート）」を作成する。パネルに固有のイベント処理の場合は、各パネルの保持する「画面データ（ルート）」を処理する。

また、画面全体の入力データを一度に処理する場合には、一括処理を実行するパネルに対応する「画面データ」（図 52 のパネル D に対応する画面データ）に、その他パネルの「画面データ」をいったん全てコピーするか、イベント処理実行時の BuiltRequestData イベント等で、各パネルに対応する「画面データ」の値を DTO に直接コピーすることで、全てのパネルのデータを一か所に集約させるよう実装する。

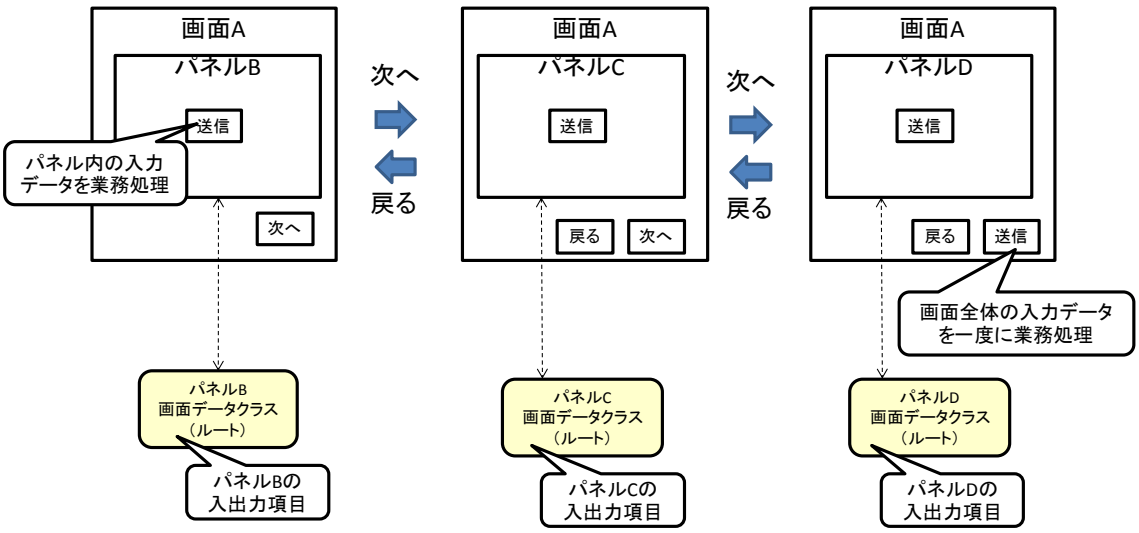

図 61 パネル切り替え場合の「画面データ」クラスの設計

また、パネル切り替えでは、ヘッダーやフッターといった画面のメインとなるパネルの外側にも共通の入出力項目が配置される場合があり、複雑な画面になると、ヘッダーやフッター部分が切り替わることもある。
このような場合、パネルの画面データに対して、ヘッダーやフッター部分の項目を「画面データ（ネスト）」として設計するとよい。ContentPlaceHolder を経由して、「画面データ（ネスト）」を大元の画面やヘッダーやフッター部分を実現する画面部品へ渡し、双方向バインドの設定をする。
この時、ヘッダーやフッターの項目が変化しない場合は、ヘッダー、フッターを表示する最初のパネルに対してのみ「画面データ（ネスト）」を設計してバインド設定することで、次のパネル以降では、ヘッダーやフッターの項目が変化するパネルまで、「画面データ（ネスト）」の設計や画面データのコピー処理を省略するといった設計も検討できる。

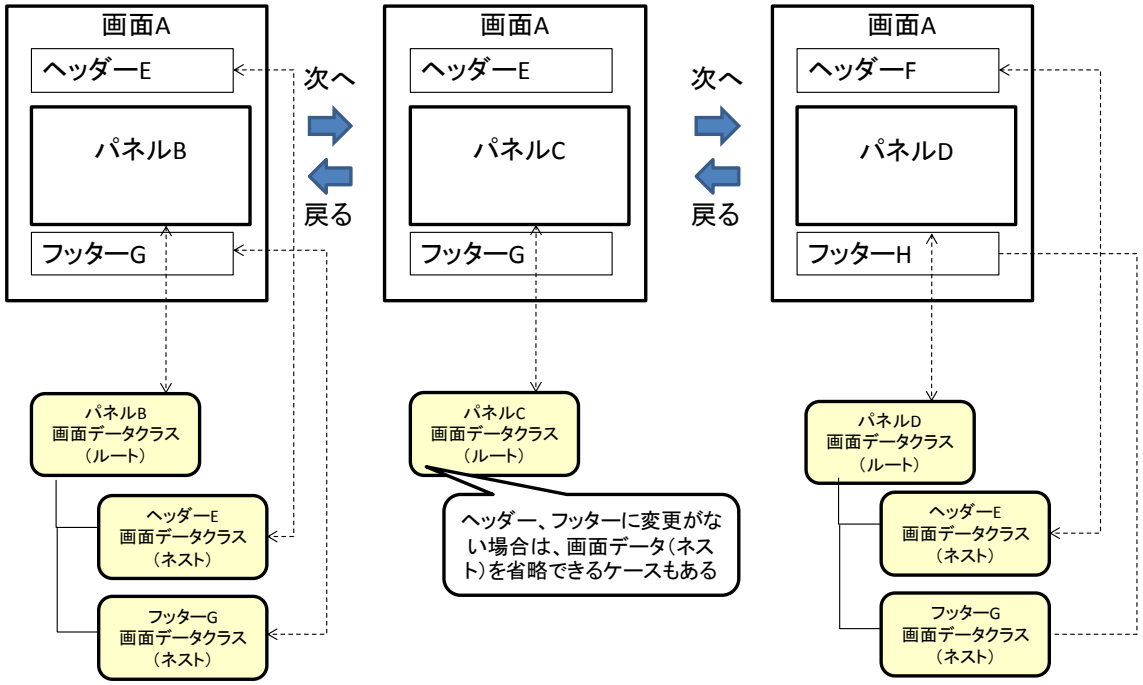

図 62 パネル切り替えでヘッダー・フッタ部分がある画面での「画面データ」の設計

■ 内部構成

◆ クラス図

本機能の主要クラスについて、クラス図を以下に示す。

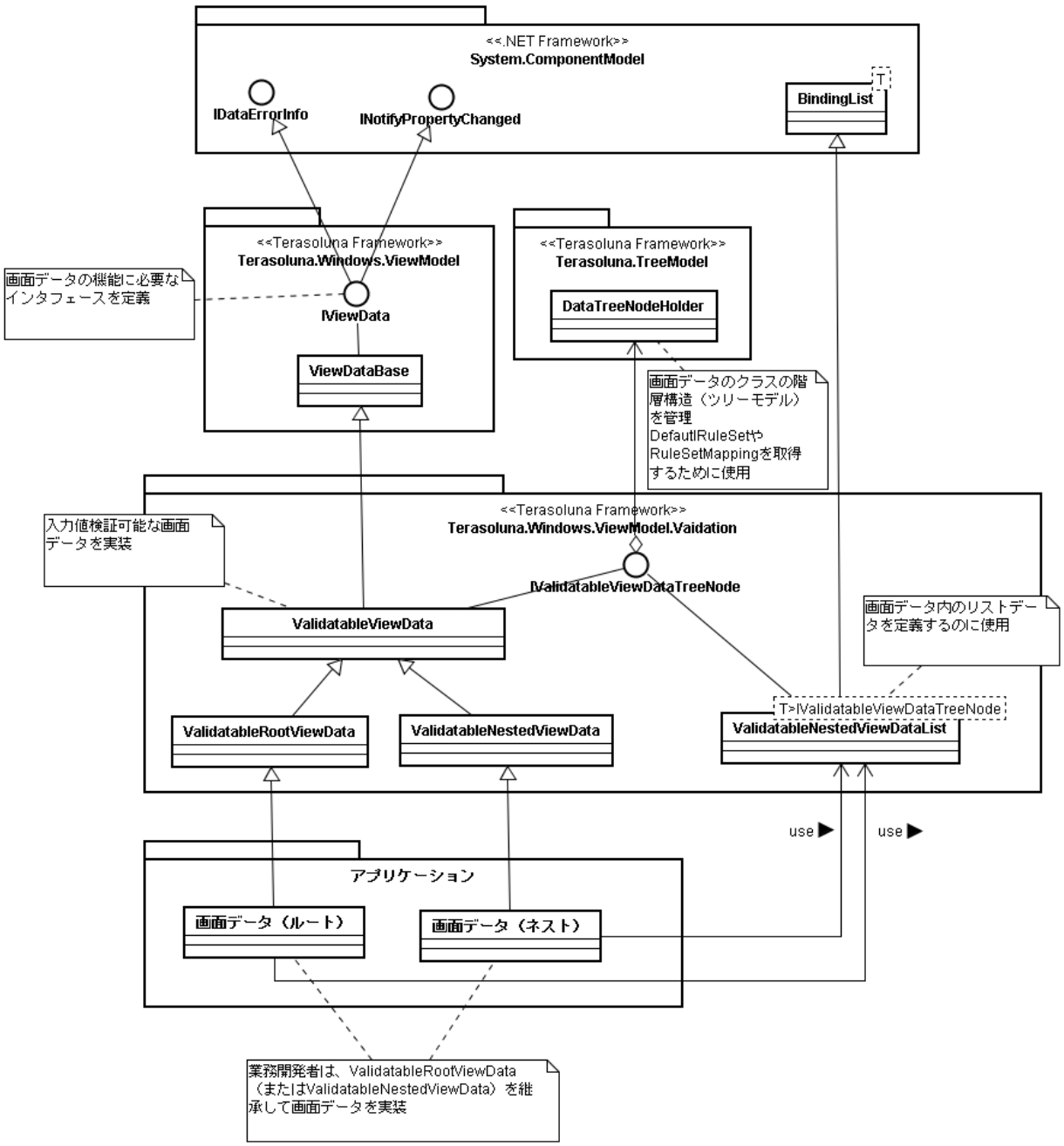

図 63 クラス図 1

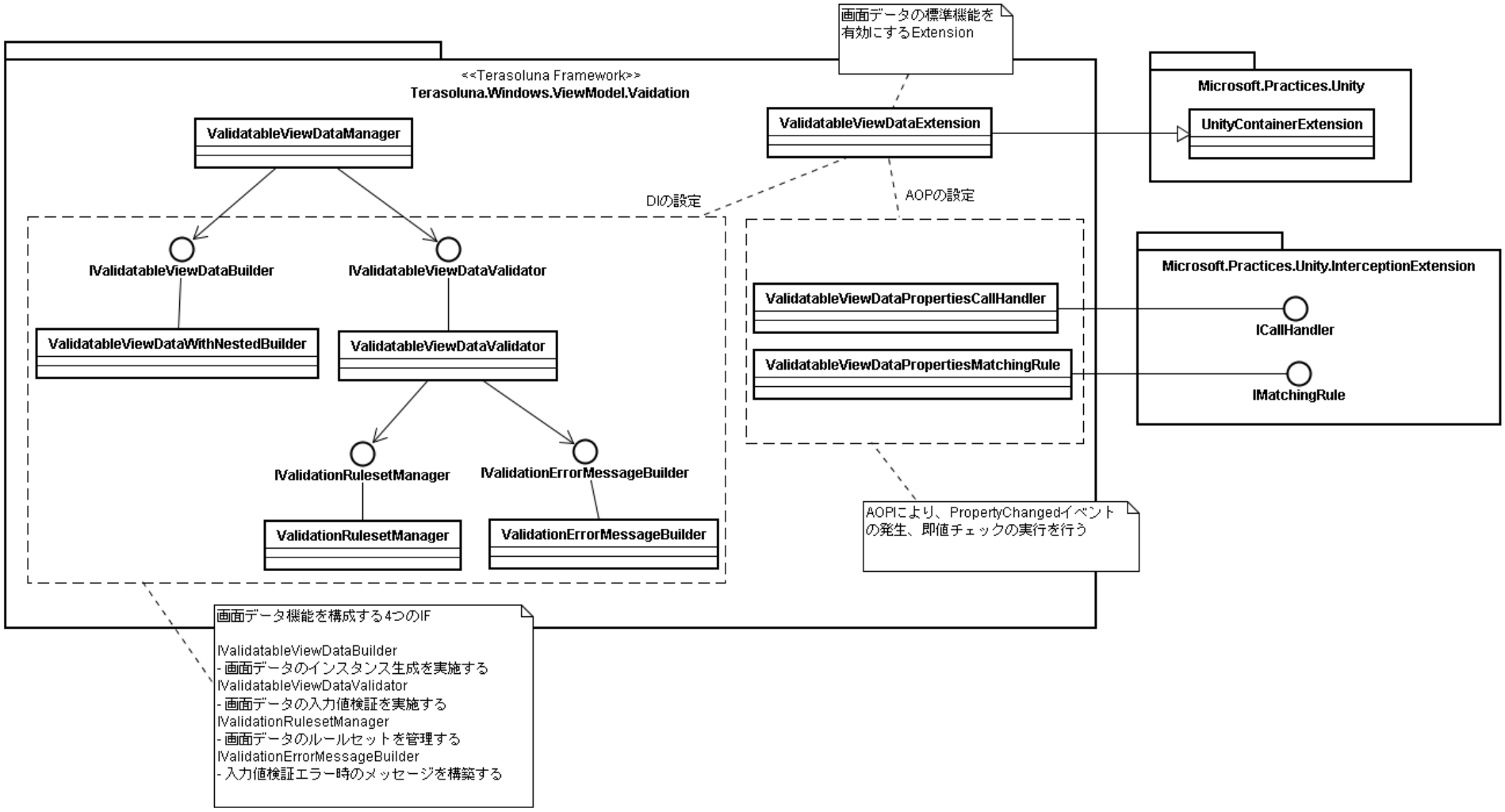

図 64 クラス図2

◆ シーケンス図

即値チェック時のシーケンス図を図 65、業務処理時の入力値検証処理のシーケンス図を図 66 に示す。

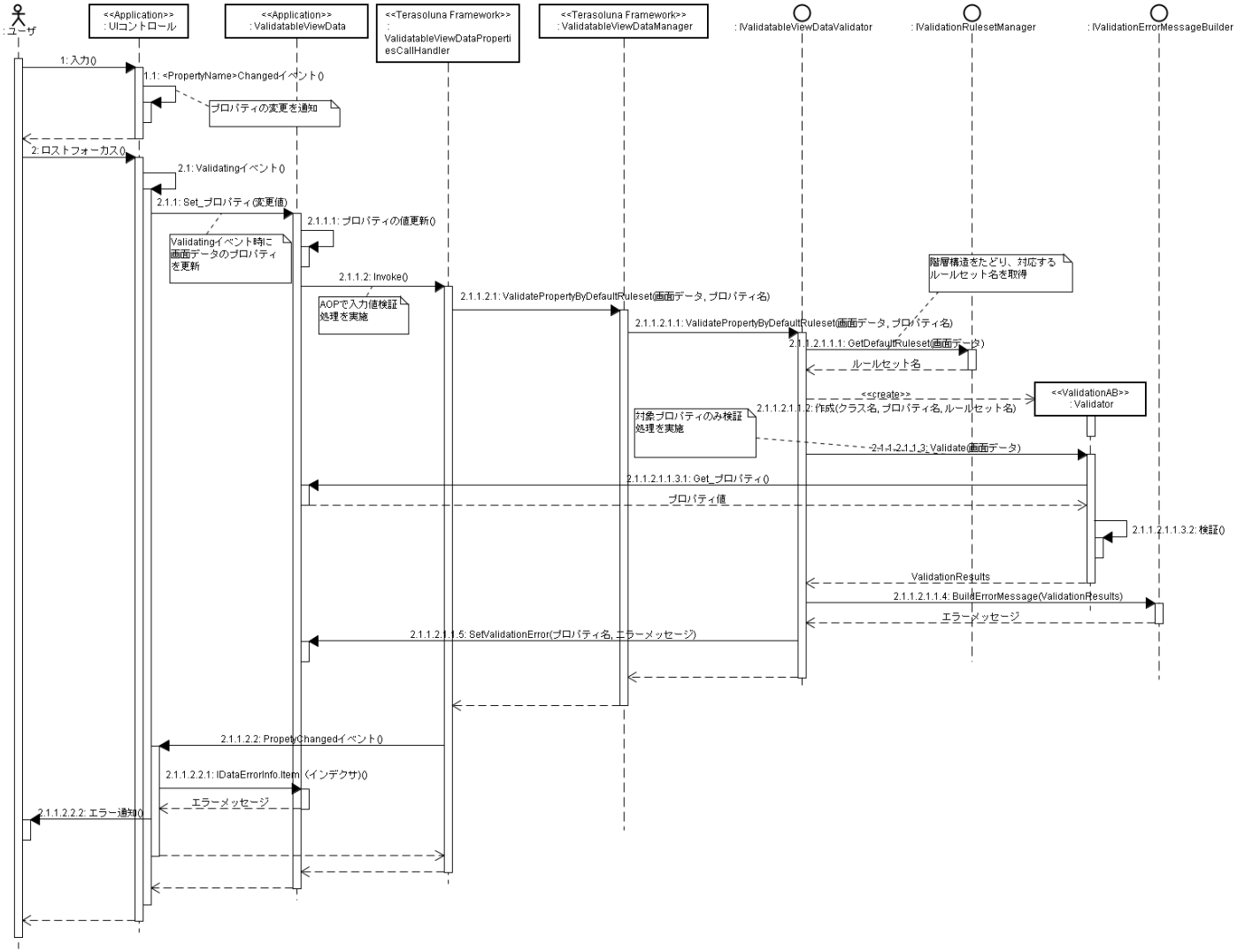

図 65 即値チェック時のシーケンス図

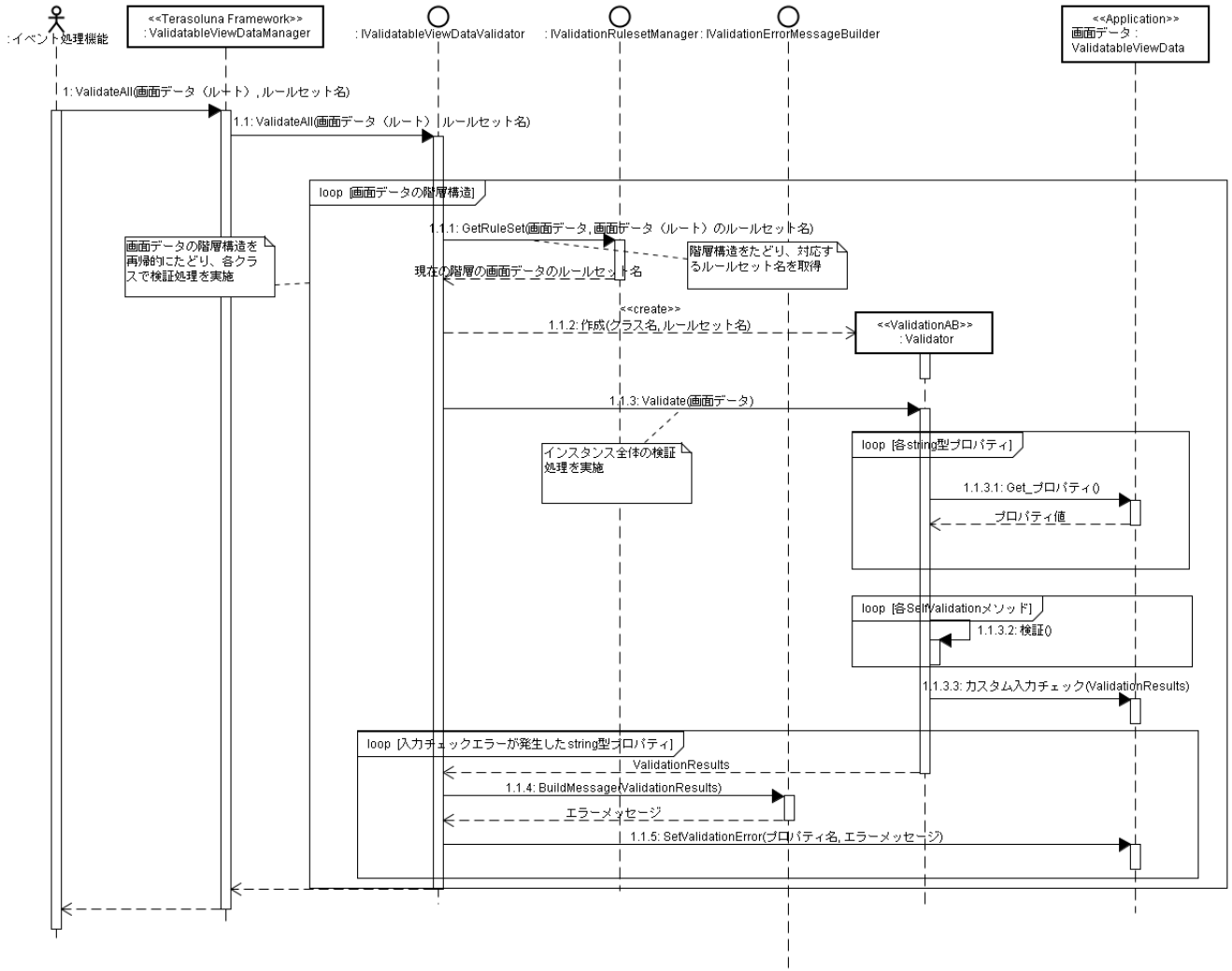

図 66 業務処理時の入力値検証処理のシーケンス図

◆ 構成クラス

本機能を構成するクラスを以下に示す。

表 10 構成クラス一覧

項番	クラス名	説明
Terasoluna.Windows.ViewModel 名前空間		
1	IViewData	「画面データ」が共通的に実装するべきインタフェース。
2	ViewDataBase	「画面データ」の基底クラス。
Terasoluna.ViewModel.Validation 名前空間		
3	IValidatableViewDataTreeNode	階層化されたデータ構造を表現可能な「画面データ」クラスのインタフェース。
4	ValidatableViewData	入力値検証可能な「画面データ」クラスを表す規定クラス。 IValidatableViewDataTreeNode インタフェースを実装し、階層構造のノードを表す。 後述の ValidatableRootViewData クラスや ValidatableNestedViewData クラスの基本となる抽象クラス。
5	ValidatableRootViewData	「画面データ(ルート)」の基底クラス。
6	ValidatableNestedViewData	「画面データ(ネスト)」の基底クラス。
7	ValidatableNestedViewDataList<T>	「画面データ(ネスト)」のリストデータを表すクラス。
8	ValidatableViewDataManager	「画面データ」の生成や入力値検証を実施する static クラス。
9	IValidatableViewDataBuilder	画面データのインスタンスを生成するインタフェース。 ValidatableViewDataManager クラスは、左記インタフェースの実装クラスを利用し、「画面データ」クラスのインスタンスを生成する。
10	ValidatableViewDataBuilder	IValidatableViewDataBuilder の実装クラス。1 つの「画面データ」クラスのインスタンスを生成する基本的な処理を実施する。
11	ValidatableViewDataWith NestedBuilder	IValidatableViewDataBuidler のデフォルト実装クラス。「画面データ」クラスのプロパティにネストされた画面データが存在する場合、その「画面データ」クラスもインスタンス生成し、プロパティに自動セットする。
12	IValidatableViewDataValidator	「画面データ」の入力値検証を実施するインタフェース。 ValidatableViewDataManager クラスは、左記インタフェースの実装クラスを利用し、「画面データ」の入力値

項番	クラス名	説明
		検証を実施する。
13	ValidatableViewDataValidator	IValidatableViewDataValidator のデフォルト実装クラス。
14	IValidationRulesetManager	「画面データ」のルールセットを管理するインタフェース。 ValidatableViewDataValidator クラスは、左記インタフェースの実装クラスを利用し、ルールセットを取得するする。
15	ValidationRulesetManager	IValidationRulesetManager のデフォルト実装クラス
16	IValidationErrorMessageBuilder	入力値検証エラー時のエラーメッセージ生成するインタフェース。 ValidatableViewDataValidator クラスは、左記インタフェースの実装クラスを利用し、エラーメッセージを生成する。
17	ValidationErrorMessageBuilder	IValidationErrorMessageBuilder のデフォルト実装クラス。
18	ValidatableViewData PropertiesCallHandler	プロパティのアクセサメソッドが呼ばれた時に処理を挟み込むインターセプタ。Set アクセサが呼ばれた直後、即値チェックを実行後、INotifyPropertyChanged インタフェースの PropertyChanged イベントを発生させる。
19	ValidatableViewData PropertiesMachingRule	ValidatableViewDataValidationCallHandler を実施する対象を特定するマッチングルールを実装する。
20	DefaultRulesetAttribute	即値チェックで使用するルールセット名を設定するカスタム属性。
21	RulesetMappingAttribute	「画面データ(ルート)」と「画面データ(ネスト)」のルールセットの対応関係を定義するためのカスタム属性。
22	ValidatableViewDataExtension	本機能を利用するためデフォルトで提供されている UnityContainerExtension 継承クラス

■ 拡張ポイント

画面データの生成、入力値検証、エラーメッセージの生成など、本機能の基本要素を構成するインタフェース（IValidatableViewDataBuilder、IValidatableViewDataValidator、IValidationRulesetManager、IValidationErrorMessageBuilder）のデフォルト実装クラスは、ValidatableViewDataExtension クラスで DI 設定されている。
本機能を拡張する場合、各インタフェースの実装クラスを作成し、ValidatableViewDataExtension クラスを参考に、UnityContainerExtension 継承クラスを作成し、DI 設定を差し替える。
以下に UnityContainerExtension 継承クラスの実装例を示す。

```
public class SampleValidatableViewDataExtension : UnityContainerExtension
{
    protected override void Initialize()
    {
・・・
        // IValidationErrorMessageBuilderの実装クラスを差し替えた例
        Container.RegisterType<IValidationErrorMessageBuilder,
                                SampleValidationErrorMessageBuilder>(
            new ContainerControlledLifetimeManager());
    }
}
```

リスト 22　UnityContainerExtension 継承クラスの実装例

また、ValidatableViewDataExtension クラスの代わりに FW 構成ファイル（TerasolunaFramework.config）に設定する。
以下に TerasolunaFramework.config の記述例を示す。

```
<configuration>
  <unity>
    <containers>
      <container>
        <extensions>
・・・
          <add type="TourSampleAppCommon.Validation.SampleValidatableViewDataExtension,
                    TourSampleAppCommon" />
        </extensions>
      </container>
    </containers>
    </containers>
  </unity>
</configuration>
```

リスト 23　TerasolunaFramework.config の記述例

■ 関連機能

- CL-03 イベント処理実行機能
- CM-05 入力値検証機能
- CM-02 インスタンス管理機能