CL-02 画面遷移機能

■ 概要

本機能は、業務アプリケーションにおける典型的な画面遷移処理方式を提供する。

画面(Form)の切り替えによる画面遷移については FormForwarder クラスを使用する。開発者は、FormForwarder クラスが提供する簡易な API を呼び出すだけで、実装スタイルを変えることなく複数の画面遷移方式パターンを利用可能である。パネル(Control)の切り替えによる画面遷移については、ContentPlaceHolder クラスを使用する。

また、画面遷移処理の実装時に問題になる、画面間の依存関係の疎結合化や、画面間でのデータの引き継ぎを簡単に実施できる等の仕組みを提供する。

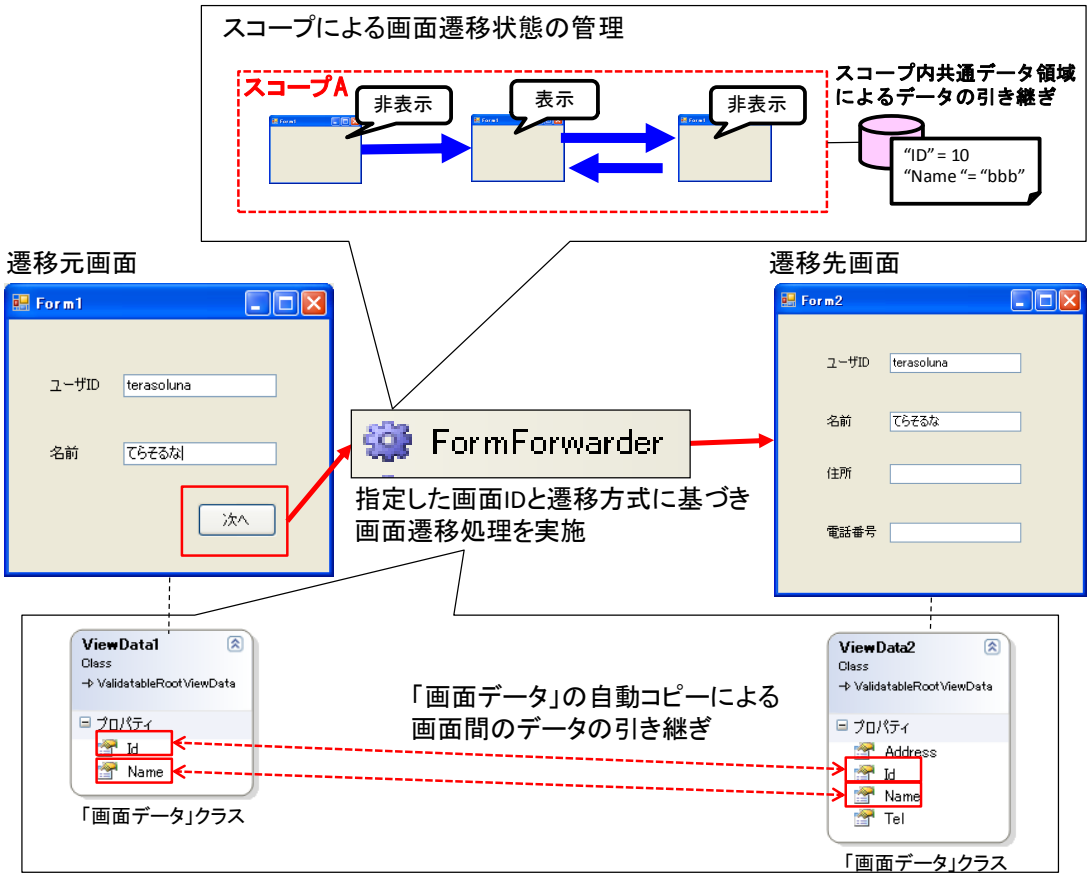

図 1 FormForwarder クラスの動作概念図

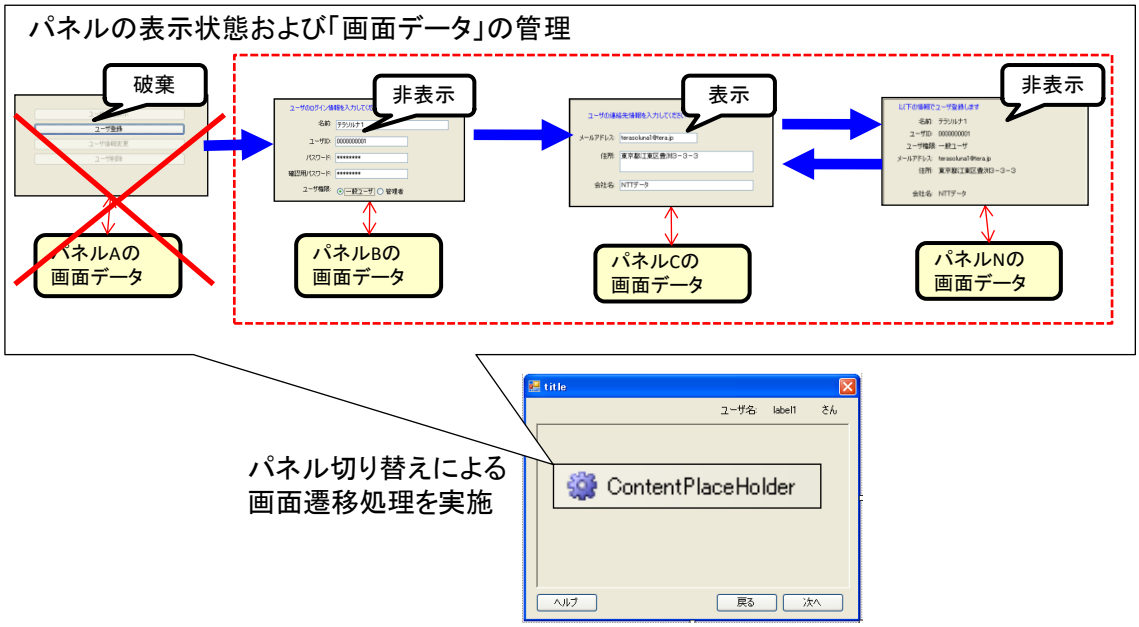

図 2 ContentPlaceHolder の動作概念図

➢ 典型的な画面遷移方式の実現

Windows Forms における画面遷移処理は、.NET の標準的な実装パターンとしては、画面(Form)クラスが持つ表示メソッド(Show,ShowDialog)、非表示メソッド(Hide)、クローズメソッド(Close)等を組み合わせて実現する。また、遷移先画面を閉じ遷移元画面へ戻る処理については、遷移先画面の FormClosed イベントまたは VisibleChanged イベント等を使用して実現する。

こういった組み合わせは業務に依存することなく共通的な処理パターンであり、再利用可能なコンポーネントとして提供可能である。FormForwarder クラスは、表 1 の典型的な画面遷移方式のパターンを提供し、開発者は、業務要件に応じて実現したい画面遷移方式を選択することができる。

表 1 FormForwarder が実現する画面遷移方式のパターン

画面の表示種別	画面遷移方式	説明
モードレス	ShowModeless	画面をモードレスで表示する。
	ShowAsChild	遷移元画面を親画面としてモードレスで表示する。
モーダル	ShowWindowModal	遷移元画面を親画面としてモーダルダイアログで表示する。 親画面のみ操作ができなくなる。
	ShowApplicationModal	アプリケーションモーダルダイアログで表示する。 画面表示中は、同じアプリケーション上の他の全ての画面が操作不可となる。
スコープ内画面遷移	ShowAndHide	遷移先画面を表示(Show)し、 現在の画面を非表示(Hide)にする。 遷移先画面を非表示または閉じると遷移元画面を再表示する。後述するスコープ内でのみ利用可能である。

また、画面遷移を画面(Form)の切り替えでなく、画面上に配置したパネルを切り替える画面遷移方式のパターンも存在する。このパターンについては、ContentPlaceHolder クラスが提供する画面遷移メソッドによりサポートする。

表 2 ContentPlaceHolder が実現するパネル切り替えのパターン(提供機能)

メソッド名	説明
ShowAndCloseContent	指定した型のパネルを表示(Show)し、現在のパネルがあれば破棄(Close)する。
ShowAndHideContent	指定した型のパネルを表示(Show)し、現在のパネルがあれば非表示(Hide)する
ShowWithCloseAllContents	インスタンス管理中のパネルを全て破棄(Close)して、指定した型のパネルを表示(Show)する。
CloseAllContents	インスタンス管理中のパネルを全て破棄(Close)する。

➢ 「スコープ」による画面遷移状態の管理

「入力画面 1」→「入力画面 2」→・・・→「確認画面」といったウィザード形式や、業務ごとに「○○検索画面」→「○○更新画面」→「○○更新確認画面」といったパターン化された画面が存在するなど、業務的に意味のある単位で一連の画面遷移が存在する。

FormForwarder による画面遷移において、この一連の画面遷移を「スコープ」として定義し、スコープごとに画面の表示状態や、インスタンスを管理する。

例えば、画面遷移方式が「ShowAndHide」の場合、インスタンスを生成済の画面であれば再表示するのみでよいが、スコープ内にインスタンスが存在しない場合はインスタンス生成(new)してから Form.Show()したり、遷移先画面が Form.Close()(または Form.Hide())されたら自動的に遷移元画面を再表示(Form.Show())したりすることが可能である。これは、スコープの存在により各画面状態が管理されているためである。

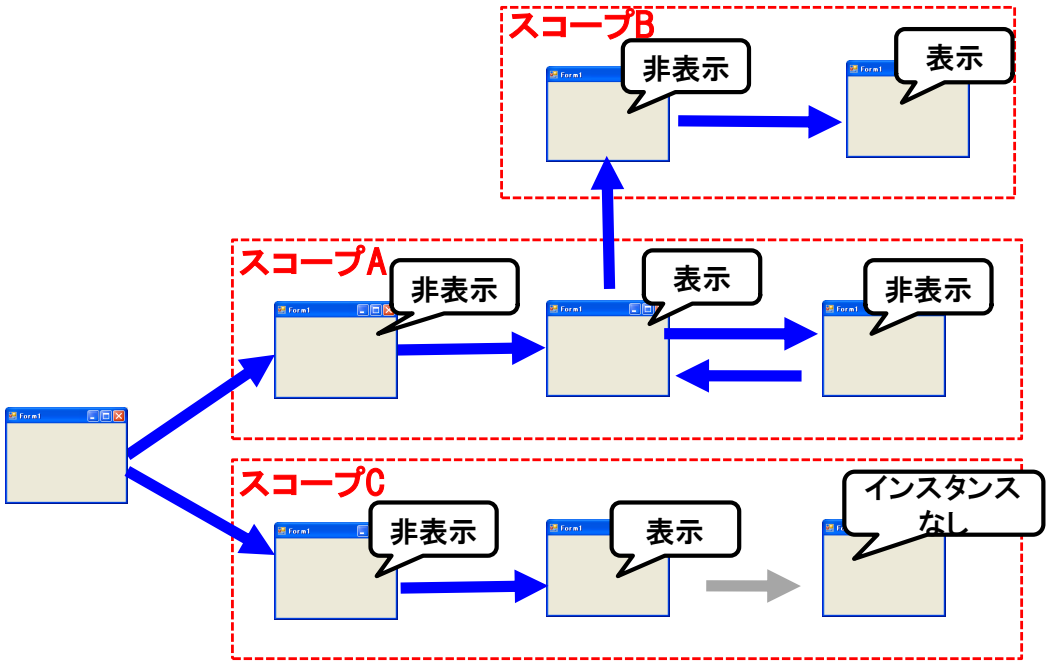

図 3 スコープによる画面表示状態管理

スコープは、FormForwardGroupScope クラスのインスタンスにより実現し、スコープのインスタンスごとに遷移状態を管理する。このため、同一の画面を異なるスコープで同時に複数起動してもそれぞれ独立したスコープとして状態管理されるので、互いに干渉することなく画面遷移を実施することができる。

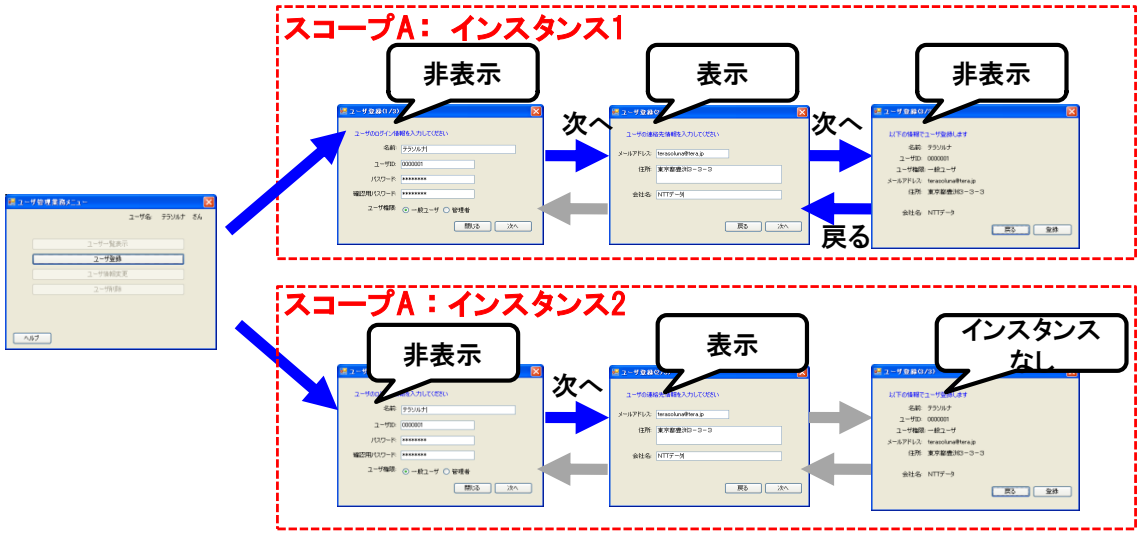

図 4 スコープの各インスタンスでの画面表示状態管理

また、画面の非表示(Form.Hide)を使った画面遷移など、表示画面以外にも生存する画面のインスタンスが存在する。通常、メモリリークを防ぐため、開発者は適宜明示的にそれらのインスタンスを破棄(Form.Close/Form.Dispose)する必要がある。
本機能では、スコープ内の画面のインスタンスを管理することで、スコープ内の画面を全て自動的にクローズする仕組みを提供している。
例えば、スコープの開始画面がクローズされると、連鎖的に遷移関係のある画面がクローズされ、最終的にスコープ内の画面が全てクローズし、スコープが消滅する。

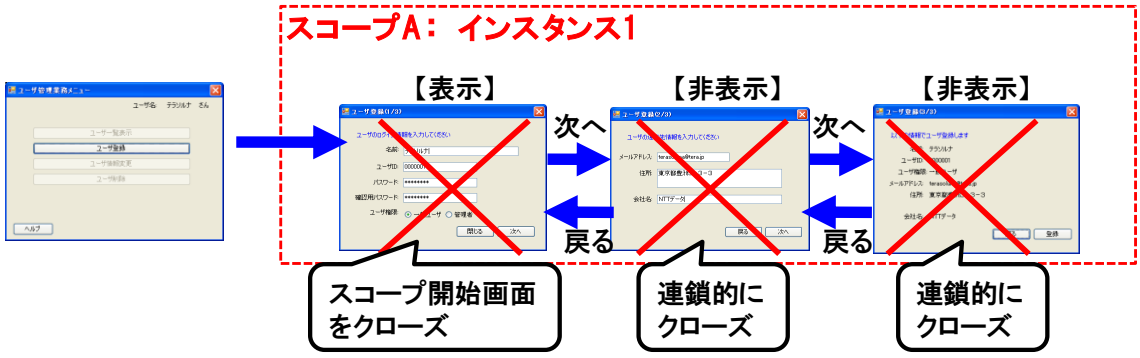

図 5 スコープによる画面インスタンス管理

この時、対象スコープ外の画面については、スコープ内の画面と遷移関係がある画面であったとしても連鎖的にクローズしない。

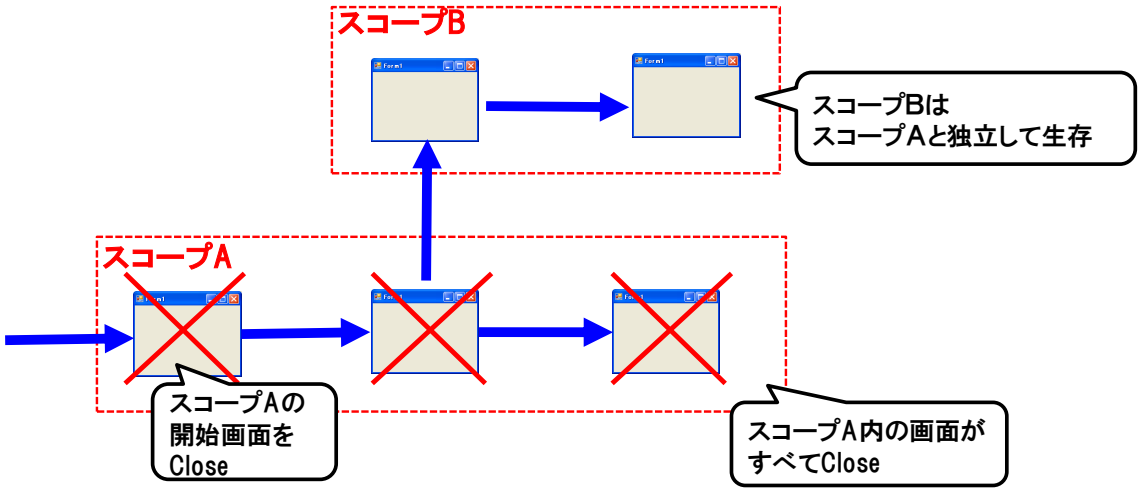

図 6 各スコープの画面インスタンス管理(独立したスコープ間)

スコープは、入れ子関係を実現することも可能である。この場合は、異なるスコープであっても外側のスコープと連鎖して内側のスコープの画面がクローズし、スコープが消滅する。

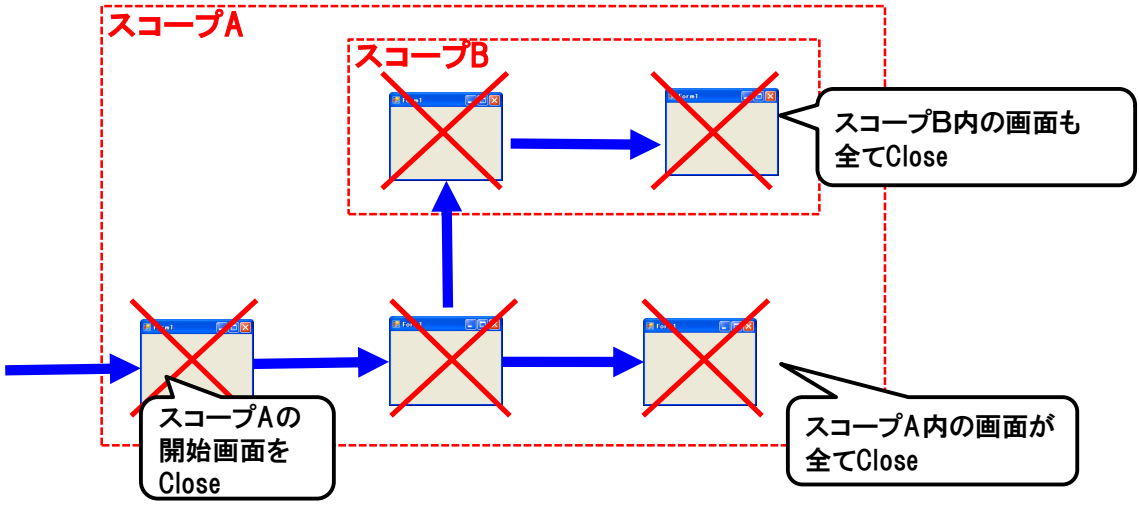

図 7 各スコープの画面インスタンス管理(入れ子関係のスコープ間)

なお、本機能では、スコープの開始画面でなくても任意の画面で現在のスコープ内の全画面をクローズしスコープを消滅させたり、スコープ内の任意の画面からスコープ開始画面へ戻すことができるユーティリティメソッド(FormForwardGroupUtility クラスのメソッド)を提供している。

➢ 画面間のデータの引き継ぎ

画面間のデータを引き継ぐ必要がある場合、以下の方式で実現することとする。

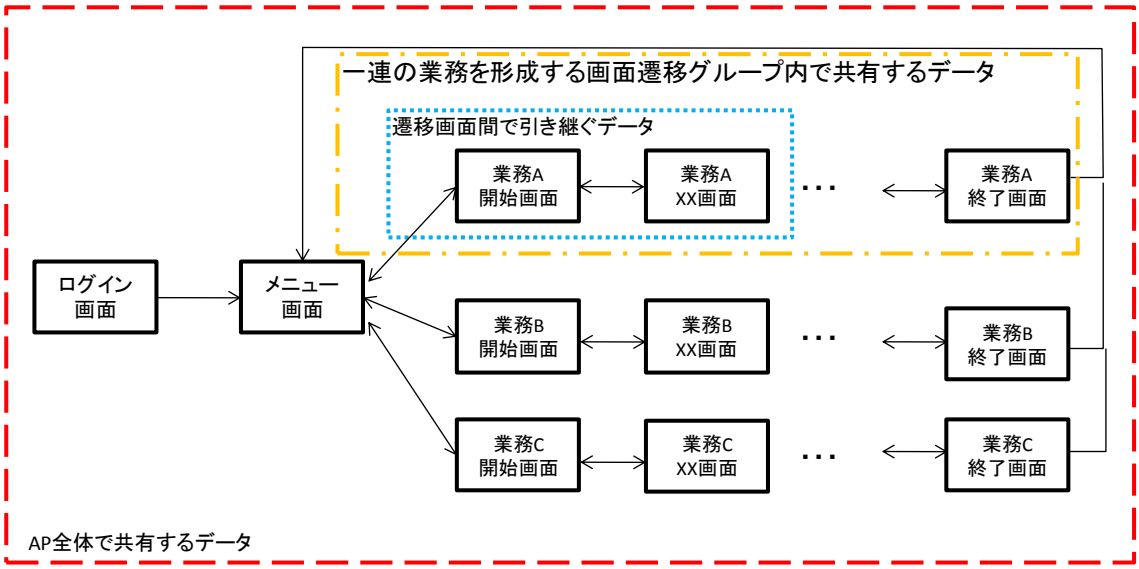

図 8 画面間でのデータの引き継ぎ

表 3 FormForwarder と ContentPlaceHolder におけるデータの引き継ぎ方法

データの種別	FormForwarder	ContentPlaceHolder
AP 全体で共有するデータ	Singleton 等、開発者が実装する共通データクラス	
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	スコープ共通データ領域	ContentPlaceHolder にインスタンス管理された「画面データ」
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	「データコピー機能」による「画面データ」のコピー ※FormForwarder による自動コピー	「データコピー機能」による「画面データ」のコピー ※開発者による明示的な実装

- 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つしかなくどこからでもアクセス可能な static な共通データクラスを開発者が実装する。
  - ➢ 各開発プロジェクトの要件に合わせてクラスを実装することを想定し、TERASOLUNA フレームワークの機能としては提供しない。
  - ➢ アプリケーション起動中は、常にデータが生存するため、一旦格納したデータは、null でセットし直すなど開発者が責任を持って領域を解放する必要がある。

- 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
  - FormForwarder による画面遷移の場合
    - 「スコープ」を利用し、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。

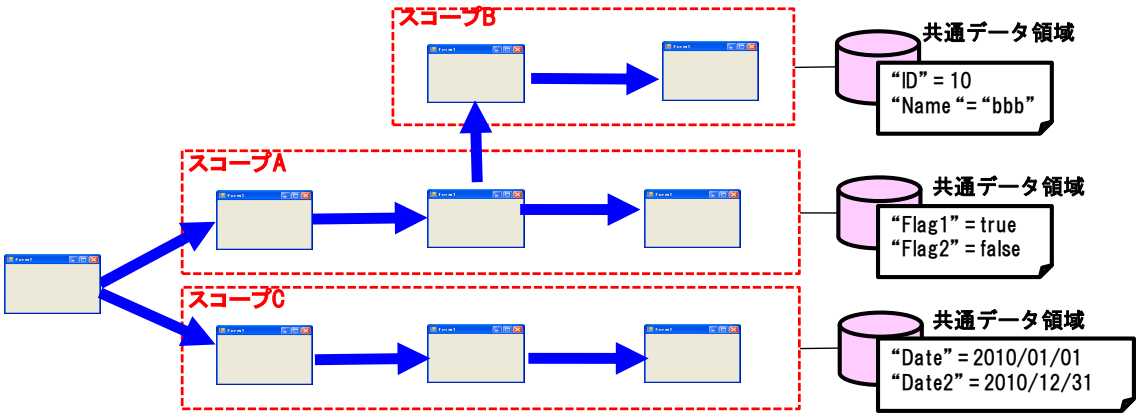

図 9 スコープ内共通データ領域

- 入れ子関係のスコープでも共通データ領域はそれぞれ独立して管理される。

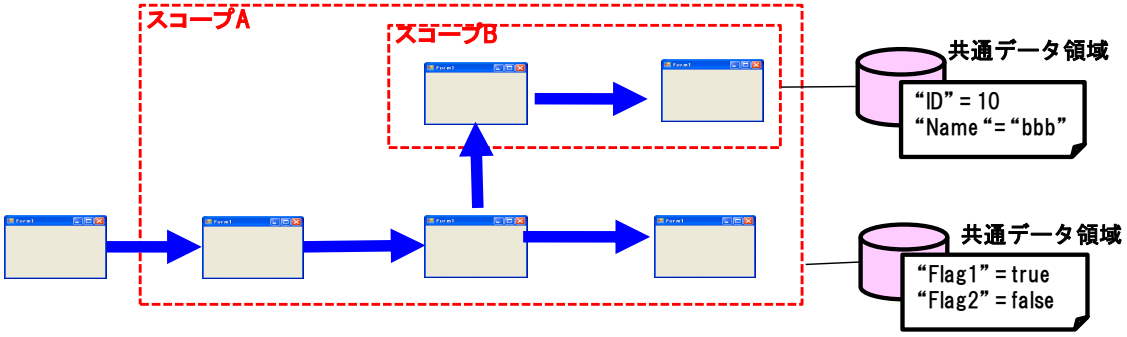

図 10 入れ子関係のスコープにおける共通データ領域

- スコープはインスタンスごとに区別されるため、同じ業務の画面が異なるスコープ間で複数同時に起動されていても、スコープ内の共通データ領域は別のワークスペースとして管理される。
- スコープ内の画面が全てクローズされると、スコープが消滅し自動的にスコープ内共通データ領域も破棄される。これにより、開発者が共通データ領域を明示的に消す必要はない。[1]

[1] ただし、共通データ領域に格納されていたデータが別の強参照を持っていれば、そのデータについては当然破棄されない。

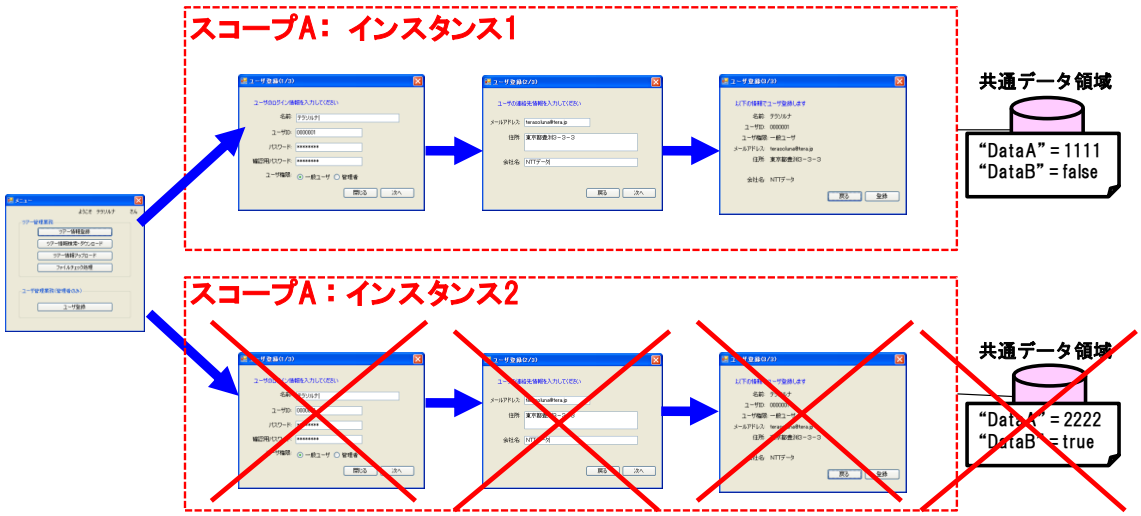

図 11 スコープ共通データ領域の破棄

- ContentPlaceHolder による画面遷移の場合
  - ContentPlaceHolder を使用する場合、パネルごとに「画面データ（ルート）」クラスを保持する。
  - ContentPlaceHolder がインスタンス管理中のパネルであれば、任意の場所でパネルおよび保持する「画面データ」を取得可能である。これを利用して、一連の画面遷移グループで「画面データ」を共有する。
  - ContentPlaceHolder インスタンスごとに独立してパネルの管理が実施されるため、FormForwarder の「スコープ」と同様、同じ型のパネルが異なる ContenPlaceHolder 間で複数同時に起動されていても、共有されるデータは別のワークスペースとして管理される。
  - ContentPlaceHolder は次のパネルへ遷移する時に、現在のパネルや、インスタンス管理中の全パネルのインスタンス管理を終了する機能を提供する。
    インスタンス管理を終了することで、パネルと保持する「画面データ」は自動的に破棄される。これにより FormForwarder の「スコープ」と同様、一連の画面遷移グループで共有するデータを開発者が明示的に削除する必要はない。

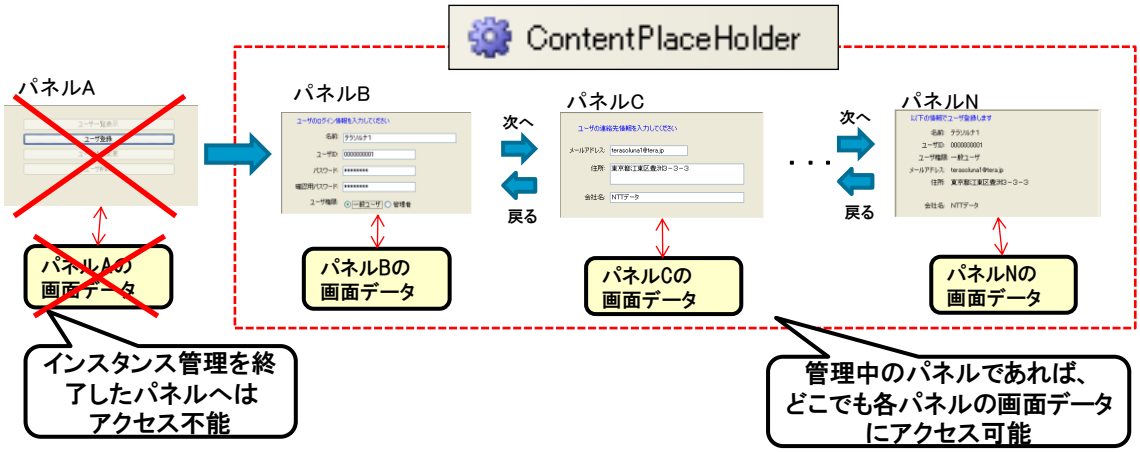

図 12 ContentPlaceHolder によりインスタンス管理されたパネルと「画面データ」

- 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
  - FormForwarder による画面遷移の場合
    - 「CM-04 データコピー機能」を利用し、画面遷移時に遷移元画面と遷移先画面間での「画面データ」の自動コピーが可能である。
    - FormForwarder のプロパティに、各「画面データ」クラスのプロパティパスを設定することで、設定情報に基づき自動的にコピーが実行される。
    - 双方向の画面遷移について自動コピー機能を実現している。なお、遷移方向によって、コピー処理の設定は区別されている。
      ・遷移元画面から遷移先画面へ遷移する時
      ・遷移先画面から遷移元画面へ戻る時
    - 画面がクローズされれば、(他に強参照がなければ)画面が保持していた「画面データ」も自動的に破棄されるので、開発者が明示的に削除する必要はない。

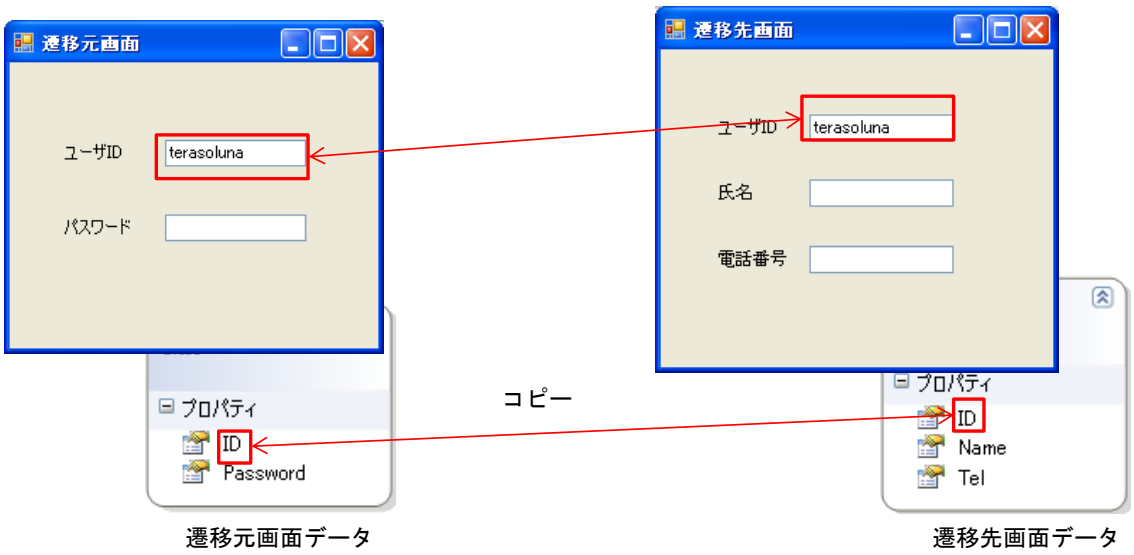

図 13 FormForwarder による「画面データ」の自動コピー

  - ContentPlaceHolder による画面遷移の場合
    - FormForwarder と同様に、「CM-04 データコピー機能」を利用し、画面遷移時に遷移元パネルと遷移先パネル間での「画面データ」のコピーが可能である。
    - ただし、FormForwarder のような自動コピー機能はなく、パネル表示時に実行されるメソッド(IContentControl.HandleShowing メソッド)内で、開発者が DataCopyManager クラスを使用して明示的にコピー処理を実装する。
    - IContentControl.HandleShowing メソッドは遷移方向による処理の区別はなく、パネル表示時に条件によらず呼び出されるため、遷移元画面によって処理を分岐させたい場合などには開発者による実装が必要である。

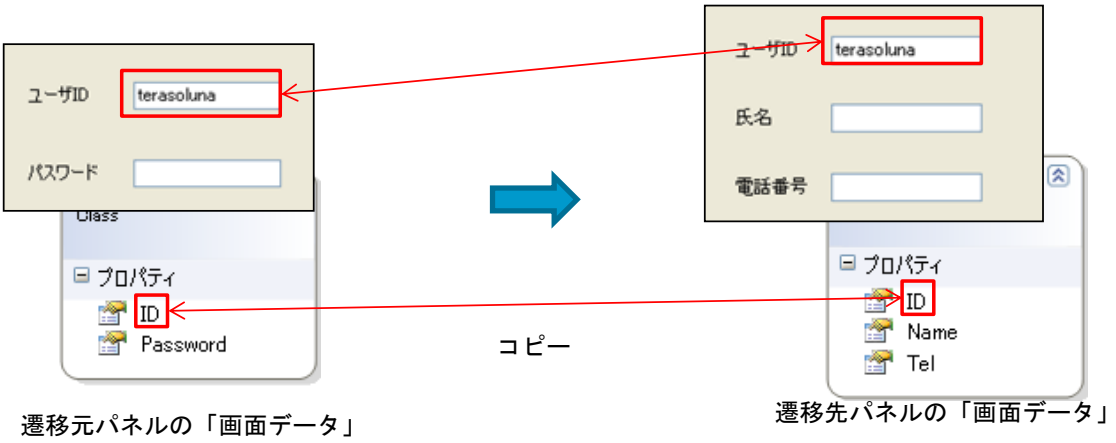

図 14 ContentPlaceHolder による「画面データ」のコピー

- 画面間の依存関係の疎結合化

  . NET の標準的な実装パターンでは、遷移先画面クラスを new 演算子でインスタンス生成し Show()メソッド等を実行することで画面遷移処理を実施する。このため、遷移元の画面クラスのソースコードには遷移先画面クラスを直接呼び出すコードが存在し、遷移先画面クラスがないとビルドできないといった強い依存関係が発生する。

  チーム開発を行う場合、各開発者が並行して製造を実施する。遷移元画面の開発者と遷移先画面の開発者が異なる場合も多く、製造・単体試験時に、遷移先画面が未完成の状態でプログラムをテストする際に、画面間に依存関係があることが問題になることが多い。

  そこで、本機能では遷移先画面のクラスが存在しなくてもビルドやプログラムの実行が可能なように、画面間の依存関係を疎にする。

  画面を一意に識別する文字列(「画面 ID」)を定義し、FormForwarder クラスのプロパティに遷移先の画面 ID を指定する。文字列による指定であるため遷移先画面のクラスが存在しなくてもビルドが可能である。なお、画面クラスに対応する画面 ID を定義する標準的な方法として、画面クラスに対して ScreenId 属性を付与する。

  プログラム実行時には、指定した画面 ID に一致する画面クラスをアセンブリより検索し、画面遷移を実施する。

  これにより、製造時に本当の画面が完成していなくてもモックの画面に切り替えてテストするといったことも可能になる。

  また、大規模な開発プロジェクトでは、複数の業者による並行開発も想定される。このような場合、業者ごとに Visual Studio のソリューションやプロジェクトを分割し、アプリケーションが複数のアセンブリで構成される。

  前述の通り、画面間の依存関係を疎結合化することで、別の業者が開発したアセンブリを参照しなくても並行開発が可能になる。

  しかし、各業者が作成したアセンブリを結合してアプリケーションを実行する際、別のアセンブリにある画面クラスも検索対象とする必要がある。このとき、もし画面がまだ、アプリケーションドメインにロードされていないアセンブリに存在する場合、遷移先画面を識別することができない。そこで、本機能は、「CM-01 アプリケーション起動・終了機能」により、アプリケーション起動時の初期化処理で、実行アプリケーションの実行フォルダ上に存在するアセンブリを全てロードし、アプリケーションの全ての画面 ID(ScreenId 属性)を取得し、全ての画面を特定できるようにする。なお、アセンブリのロード処理は、初期画面の起動が遅くならないようフレームワークによりバックグランドで遅延実行される。

■ FormForwarder の使用方法

◆ 概要

FormForwarder は、Component 継承クラスであり、画面に貼り付け可能な部品である。また、Visual Studio のデザイナ上で画面遷移処理を呼び出すための設定が可能である。

画面クラスのイベントハンドラメソッドより FormForwarder を呼び出すことで、プロパティで設定した定義に基づき、画面遷移を実行し、定められたタイミングで各種イベントが発生する仕組みになっている。

業務開発者が FormForwarder による画面遷移処理を実装する主な手順は以下の通りである。

① 「画面データ」クラスを実装する
② 画面を実装する。
③ FormForwarder をツールボックスから画面へ貼り付ける。
④ FormForwarder のプロパティを設定する。
⑤ FormForwarder のイベントを実装する。
⑥ ボタンクリックなどのイベントハンドラメソッドで、FormForwarder の Forward メソッドを呼び出し、画面遷移処理を実行する。

以下に、FormForwarder を利用して業務処理の呼び出しを行う作業手順を示す。

◆ 実装方法

①「画面データ」クラスの実装

「画面データ」クラスは画面間での「画面データ」コピーに利用される。

「画面データ」の実装方法は、「CL-01 画面データ機能」の機能説明書を参照のこと。

以下に、「画面データ」クラスの実装例を示す。

```
[DefaultRuleset("RS01")]
public class SC_B01_01_01ViewData : 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; }
}
```

リスト 1 「画面データ」クラスの実装例

②画面の実装

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。

画面クラスの実装時には開発者が守らなければならないルールがある。テンプレートは、これら

のルールに従ってひな形を生成するので、開発者による実装の負担はかなり少なくなっている。
テンプレートを使った画面クラスの作成方法と実装ルールについては、「CL-01 画面データ機能」の機能説明書を参照すること。
このうち、とくに画面遷移機能の利用に関して注意すべきルールのみ、記述しておく。

- 「画面データクラス(ルート)」の型を持った「ViewData」という名前のプロパティを定義する。
  - 本機能や「CL-03 イベント処理実行機能」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」として自動的に取得するためである。
- 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。
  - 本機能が、遷移元画面から遷移先画面へ「画面データ」をコピーするには、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。

また、画面 ID の指定による画面遷移を実現するため、以下のルールも守る必要がある。

- 対象の画面クラスに ScreenId 属性を付与し、画面 ID を指定する。

以下に、画面クラスの実装例を示す。

```
    //ScreenId属性で画面IDを設定
    [ScreenId("CalcView")]
    public partial class CalcView : Form
    {
        // 画面データクラス（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)
        {
            calcViewDataBindingSource.DataSource = ViewData;
        }
・・・
    }
```

リスト 2 画面クラスの実装例

③画面への FormForwarder の追加
Visual Studio のデザイナ上で FormForwarder を遷移元画面に貼り付ける。

④FormForwarder のプロパティの設定
FormForwarder のプロパティには、遷移先画面の指定や、画面遷移処理方式の選択、画面間のデータコピー等、プロパティエディタで設定する。
設定するプロパティの詳細については、後述の「FormForwarder のプロパティ」を参照のこと。

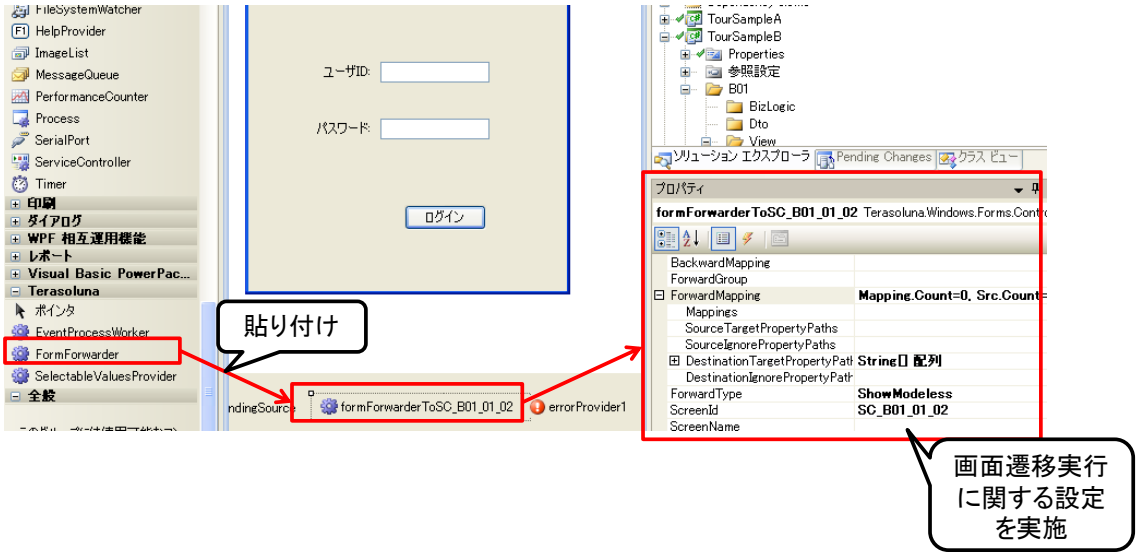

図 15 FormForwarder の追加およびプロパティの設定

⑤FormForwarder のイベントハンドラの実装
遷移先画面の表示/クローズ(または非表示)時に、それぞれイベントが発生するので、ハンドリングしたいポイントで、イベントハンドラメソッドを実装する。
各種イベントの詳細については、後述の「FormForwarder のイベント」を参照のこと。

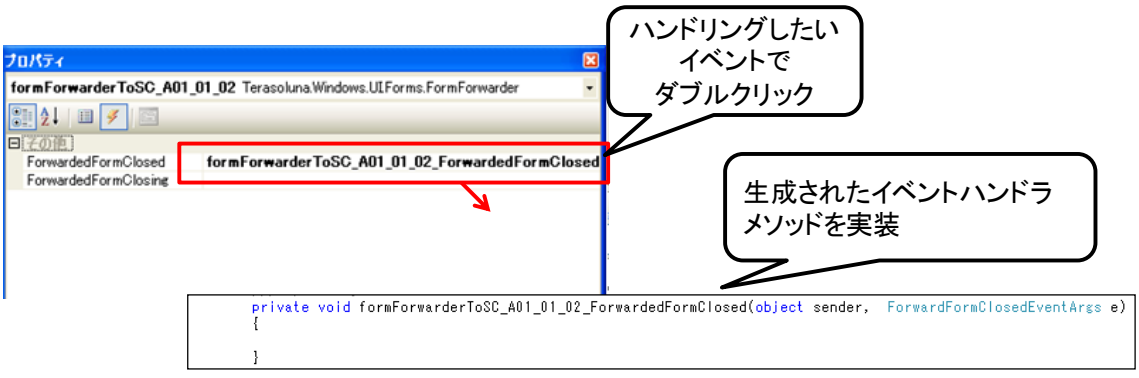

図 16 FormForwarder のイベントハンドラ実装

⑥画面遷移処理の実行

画面遷移処理を実行するには、遷移処理を実施するボタンのクリックなイベントで、遷移元画面で FormForwarder インスタンスの Forward メソッドを呼び出す。

```
// クリックイベントでFormForwarderのForwardメソッドを呼び出し。
private void btn1_Click(void sender, EventArgs e)
{
    formForwarder1.Forward();
}
```

リスト 3 画面遷移処理

◆ FormForwarder のプロパティ

FormForwarder の各プロパティについて表 1 に示す。

表 4 FormForwarder のプロパティ

項番	プロパティ名	必須	説明
1	BackwardMapping		遷移先画面から遷移元画面へのデータコピーのマッピングを設定する。設定方法については、表 6 を参照のこと。
2	ForwardGroup		「スコープ」を一意に特定する任意の文字列を設定する。 同一スコープ内のでの遷移では、全て同じ文字列を設定する。
3	ForwardMapping		遷移元画面から遷移先画面へのデータコピーのマッピングを設定する。設定方法については、表 6 を参照のこと。
4	ForwardType	○	画面遷移の処理方式を指定する。 選択可能な設定値は、表 5 を参照のこと。
5	ScreenId	○	遷移先画面を識別する ID。各画面に ScreenId 属性を用いて付与したもので、この ID から表示する画面を特定する。
6	ScreenName		同一スコープ内において、同一 ScreenId の画面を複数起動することはできないが、異なる ScreenName を指定することで、別インスタンスとして同一画面を複数起動させることが可能になる。

ForwardType プロパティには、本機能が実装する画面遷移方式パターンを選択することができる。選択可能な値を表 5 に示す。

表 5 ForwardType プロパティの設定値

設定値	説明	実現方法の概要
ShowModeless	画面をモードレスで表示する。	遷移先画面を Form.Show()メソッドで表示する。
ShowAsChild	遷移元画面を親画面(owner)としてモードレスで表示する。	遷移先画面を Form.Show(IWin32Window)メソッドで表示する。この時、引数の owner を遷移元画面とする。
ShowWindowModal	遷移元画面を親画面としてモーダルダイアログで表示する。 遷移先画面を表示中は、親画面のみが操作不可。	・遷移先画面を Form.Show(IWin32Window) メソッドで表示する。この時、引数の owner を遷移元画面とする。また、遷移元画面の Enable プロパティを false にして無効化する。 ・遷移先画面が閉じられたら(Form.Close()または Form.Hide()メソッドが実行されたら)、遷移元画面の Enable プロパティを true に戻すようにイベントハンドリングする。
ShowApplicationModal	アプリケーションモーダルダイアログで表示する。 画面表示中は、同じアプリケーション上の他の全ての画面が操作不可となる。	遷移先画面の Form.ShowDialog(IWin32Window)メソッドで表示する。この時、引数の owner を遷移元画面とする。
ShowAndHide	現在の画面を非表示(Hide)にし、遷移先画面を表示(Show)する。 現在の画面を非表示または閉じると遷移元画面を再表示する。	・ShowAsChild を利用して、Form.Show(IWin32Window)メソッドで遷移先画面を表示する。 また、遷移元画面を Form.Hide()メソッドで非表示にする。 ・遷移先画面が閉じられたら(Form.Close()または Form.Hide()されたら)、遷移元画面を Form.Show()し、再表示するようイベントハンドリングする。

.NET の標準機能でモーダルダイアログを実現するには、Form.ShowDialog メソッドを利用する。本機能では、「ShowApplicationModal」として同等の機能を提供している。
「ShowApplicationModal」の場合、表示された画面以外のアプリケーションの全画面が操作不可となる。このため、別の業務画面を同時に操作することが不可能となる。

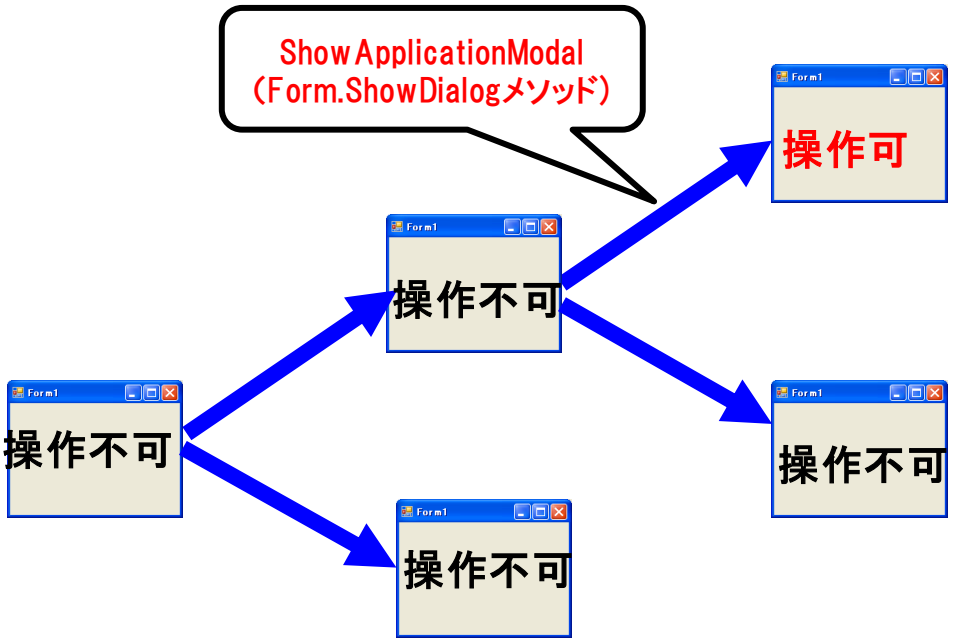

図 17 ShowApplicationModal の遷移イメージ

そこで、本機能では、「ShowWindowModal」という画面遷移方式を提供している。
ShowWindowModal は、Form.Show(IWin32Window)メソッドを使用して、親画面に対する擬似的なモーダルを再現し、モーダルダイアログ表示中も別の業務画面を同時に操作可能である。

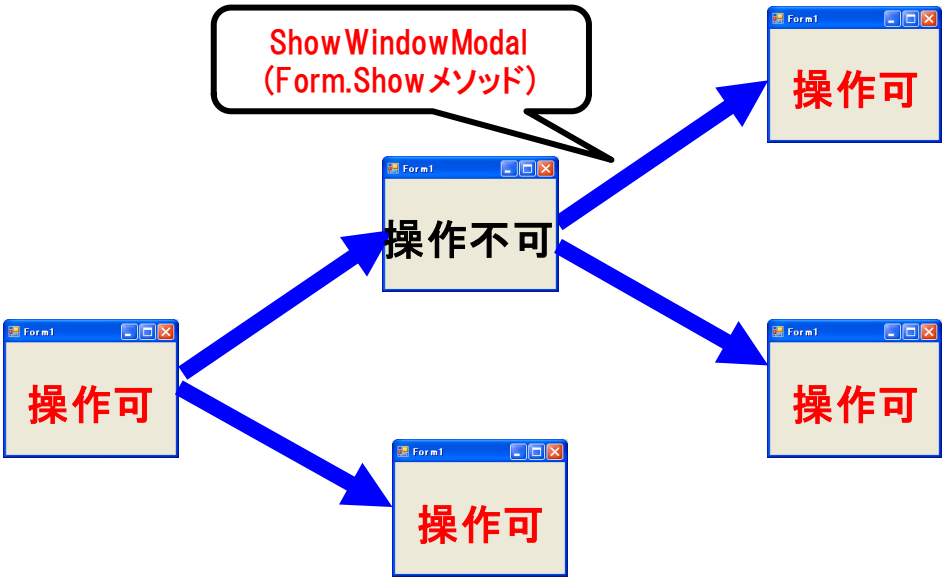

図 18 ShowWindowModal の遷移イメージ

また、「ShowAndHide」は、次の画面へ画面遷移すると遷移元画面は消え、遷移先画面を閉じると遷移元画面が再表示されるといった具合に、Web ブラウザアプリケーションのような業務の一連の流れに合わせた画面遷移パターンを実現する。
遷移先画面から遷移元へ戻る際は、遷移先画面の Form.FormClosed イベントや Form.VisibleChanged イベントで自動的に遷移元画面を再表示(Form.Show())するように本機能内部でイベントハンドラを登録することで機能を実現している。

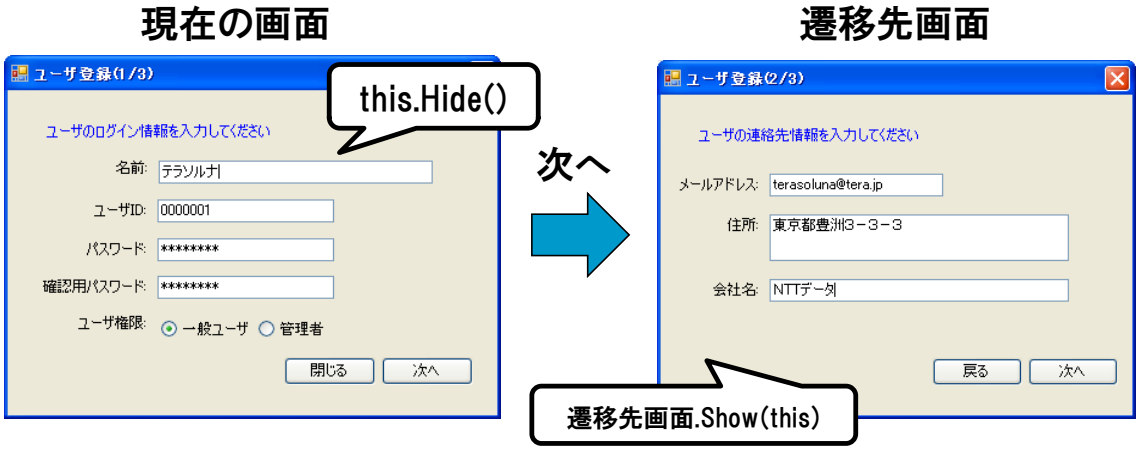

図 19 ShowAndHide を使った順方向の画面遷移(遷移先画面へ進む)

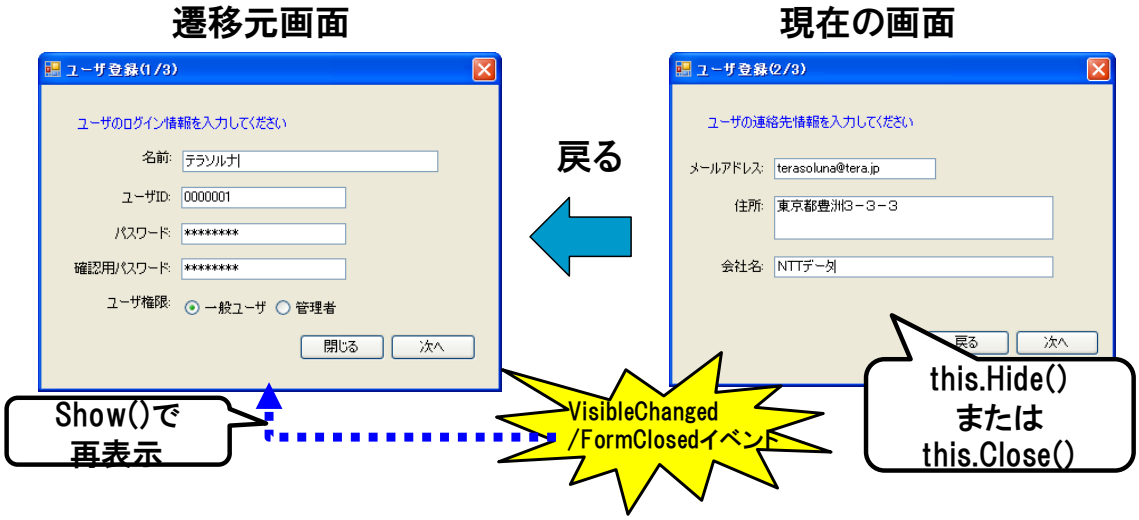

図 20 ShowAndHide を使った逆方向の画面遷移(遷移先画面から戻る)

なお、モーダル表示されている画面(「ShowWindowModal」か「ShowApplicationModal」で遷移した画面)からは ShowAndHide による遷移はできないようになっているので注意が必要である。これは、モーダル表示した画面を Form.Hide()するとモーダル状態が解除されてしまうという問題を避けるためである。

◆「画面データ」の自動コピー

本機能では、「CM-04 データコピー機能」を利用し、画面遷移時に遷移元画面と遷移先画面間の「画面データ」の自動コピーが可能である。
「画面データ」の自動コピーのマッピング設定は、遷移方向により2つの設定がある。

- 遷移元画面⇒遷移先画面に遷移時のコピー
  - FormForwarder.Forward メソッドの呼び出しによる順方向の遷移
  - 遷移元画面⇒遷移先画面のコピーは、無条件に実行する
  - ForwardMapping プロパティでマッピングの設定をする

- 遷移先画面⇒遷移元画面に遷移時のコピー
  - 遷移先画面のクローズ（または非表示）による逆方向の遷移
  - 遷移先画面⇒遷移元画面のコピーは、遷移先画面を閉じたときに返却される System.Windows.Forms.DialogResult 列挙体の値が、「OK」または「Yes」のときのみ実行される。
  - BackwardMapping プロパティでマッピングの設定をする

ForwardMapping プロパティおよび BackwardMapping プロパティには、表 6 に示すプロパティを設定する。マッピング設定は、「CM-04 データコピー機能」の IMappingInfo の設定に基づく。
自動コピーの仕様や、プロパティパスの記述方法などの設定方法の詳細については、「CM-04 データコピー機能」の機能説明書を参照のこと。

表 6 ForwardMapping/BackwardMapping プロパティの詳細

項番	プロパティ名	説明
1	Mappings	「CM-04 データコピー機能」の標準機能により自動コピーされないケースについて、コピー元とコピー先のプロパティパスのマッピングを設定する。
2	SourceTargetPropertyPaths	自動コピー対象とするコピー元の「画面データ」クラスのプロパティパスを設定する。
3	SourceIgnorePropertyPaths	自動コピー対象外とする、コピー元の「画面データ」クラスのプロパティパスを設定する。
4	DestinationTargetProperty Paths	自動コピー対象とするコピー先の「画面データ」のプロパティを設定する。
5	DestinationIgnoreProperty Paths	自動コピー対象外とする、コピー先の「画面データ」クラスのプロパティパスを設定する。

これらのマッピング設定に基づき、「CM-04 データコピー機能」が実行される。しかし、画面遷移機能においては、以下の固有の仕様があるので注意すること。

- 画面遷移機能におけるデータコピーでは、「CM-04 データコピー機能」が標準で実施する暗黙的な自動コピーは実施せず、コピーする対象のプロパティパスを明示する必要がある。その

ため、コピー対象のプロパティパスは全て、DestinationTargetPropertyPaths プロパティに指定する必要がある。「画面データ」の暗黙的な自動コピーを実施しない理由は、異なる画面間では、「画面データ」のプロパティが同名でも異なる意味のデータが格納される可能性が高く、コピーによる誤動作を防ぐ必要があるためである。

以下に、DestinationTargetPropertyPaths プロパティおよび、Mappings プロパティの設定イメージを図 18、図 19 に示す。

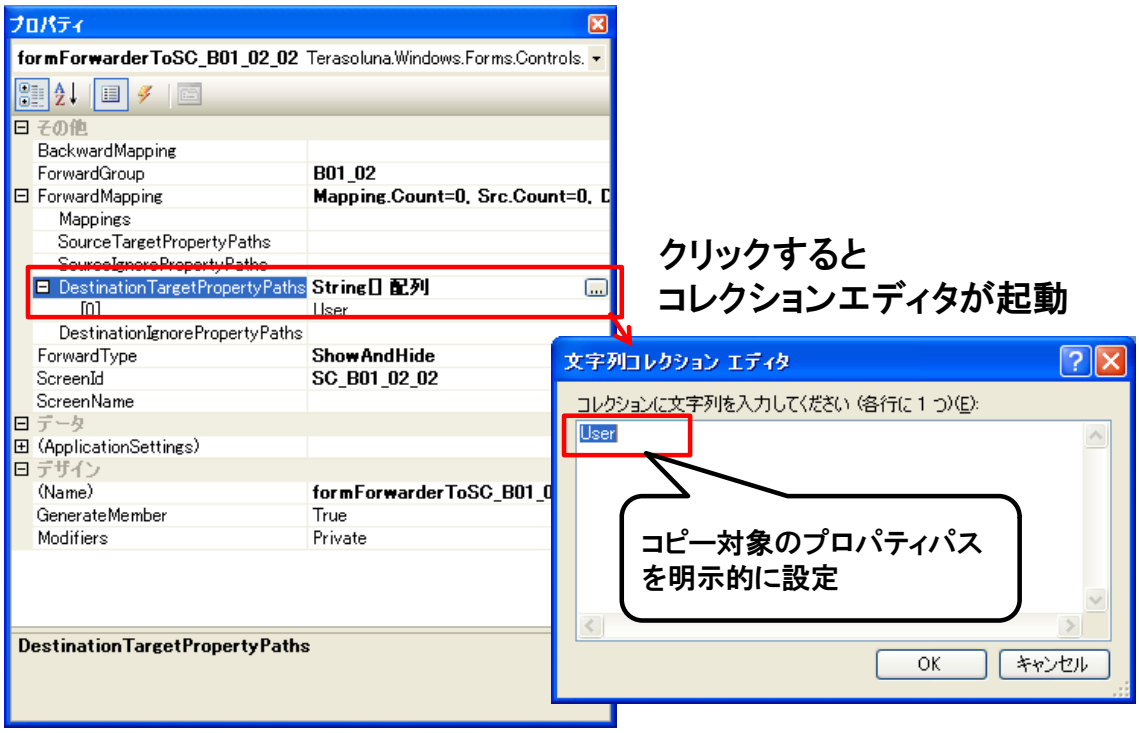

図 21 DestinationTargetPropertyPaths プロパティの設定イメージ

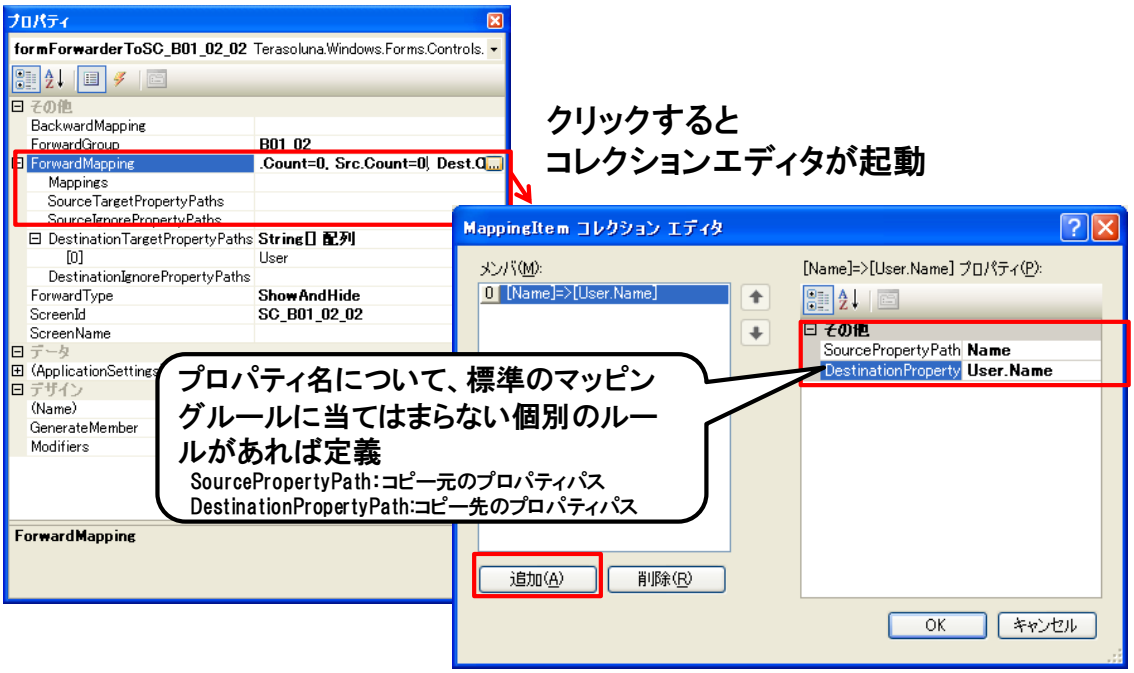

図 22 Mappings プロパティの設定イメージ

以下に、遷移元の「画面データ」から遷移先の「画面データ」への、データコピーの例を示す。

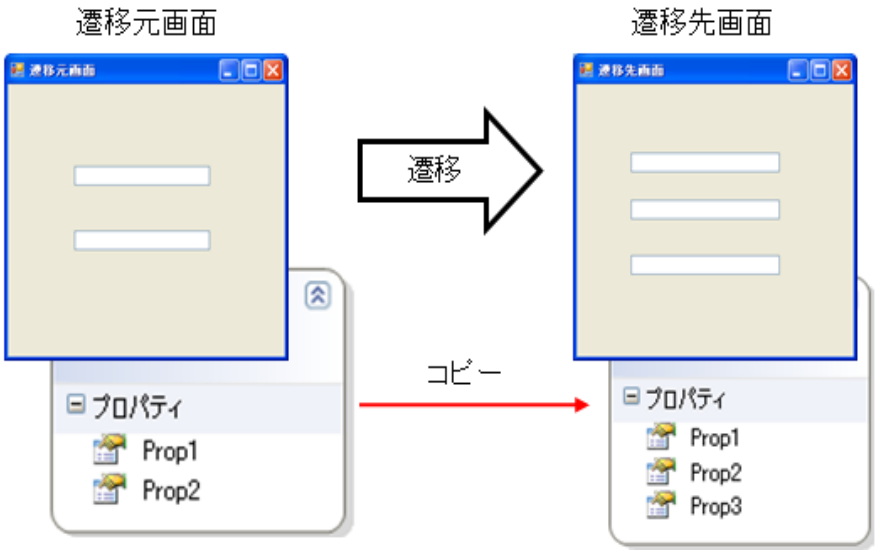

図 23 データコピーのサンプル遷移図

ForwardMapping プロパティ内の、各プロパティ設定例を示す(4 つのプロパティ名末尾の「PropertyPaths」は省略)。

表 7 ForwardMapping プロパティの設定例

項番	Mappings	Source Target	Source Ignore	Destination Target	Destination Ignore
1					
2				Prop1	
3		Prop1		Prop1 Prop2 Prop3	
4			Prop1	Prop1 Prop2	Prop2
5	[Prop1]=>[Prop3]			Prop1 Prop2 Prop3	
6	[Prop1]=>[Prop3]			Prop1 Prop2	
7	[Prop1]=>[Prop3]			Prop2 Prop3	
8	[Prop1]=>[Prop3]	Prop1	Prop2	Prop1 Prop2 Prop3	Prop1

表 7　ForwardMapping プロパティの設定例のそれぞれの設定における、コピー結果（対応する遷移元のプロパティ）を示す。標準のコピーと異なり、コピー対象となるプロパティは DestinationTargetPropertyPaths プロパティに指定したもののみであることに注意する。

表 8　表 5 の設定における、データコピーの結果（対応するプロパティ）

項番	遷移先「画面データ」のプロパティ			解説
	Prop1	Prop2	Prop3	
1	何もコピーされない			DestinationTarget を指定していないので、何もコピーされない（標準のデータコピー機能と異なる点）
2	Prop1			DestinationTarget に指定したもののみコピーされる
3	Prop1			DestinationTarget に指定したもののうち、SourceTarget に指定したもののみコピーされる
4	何もコピーされない			DestinationTarget に含まれるものでも、Ignore に指定したものはコピーされない
5	Prop1	Prop2	Prop1	DestinationTarget による自動コピーに加えて、Mappings で定義されたコピーが実行される
6	Prop1	Prop2		Mapppings の指定も、DestinationTarget に含まれるものに対して有効となる
7		Prop2	Prop1	
8			Prop1	DestinationTarget に含まれるものを起点として、各プロパティの設定からコピー対象が決定する

◆ FormForwarder のイベント

FormForwarder のイベントを表 9 に示す。
イベントに対するイベントハンドラメソッドは、FormForwarder を張り付けた画面で記述する。

表 9 FormForwarder のイベント

項番	イベント	説明
1	ForwardFormLoading	遷移先画面を表示する直前に発生するイベント。
2	ForwardFormClosed	遷移先画面が閉じられるか、非表示になった際に発生するイベント。

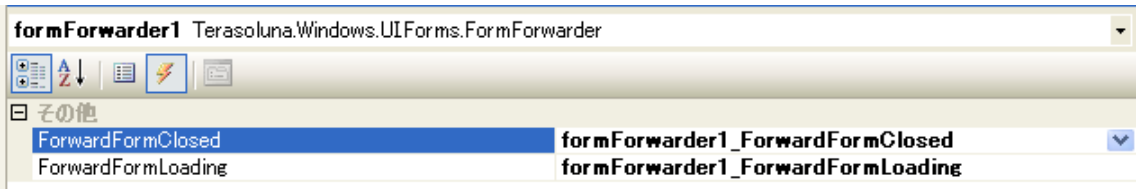

図 24 FormForwarder のイベント設定画面

各イベントハンドラの第 2 引数として ForwardFormLoadingEventArgs、ForwardFormClosedEventArgs を指定する。これらは、それぞれのイベントデータを保持するクラスであり、遷移先のフォーム、「画面データ」、DialogResult（フォームのダイアログ結果）等が格納されている。表 10、表 11 にプロパティの説明を示す。

表 10 ForwardFormLoadingEventArgs のプロパティ

項番	プロパティ	説明
1	public FormForwardContext Context { get; }	画面遷移に関する設定情報を取得する。FormForwarder のプロパティで設定した内容をほぼ同等の情報を取得できる
2	public Form ForwardForm { get; }	遷移先画面を取得する。
3	public object ViewData { get; }	遷移先画面の「画面データ」を取得する。

表 11 ForwardFormClosedEventArgs のプロパティ

項番	プロパティ	説明
1	public CloseReason CloseReason { get; }	遷移先画面が閉じられた場合に、Form.FormClosed イベントで発生した CloseReason を取得する。 遷移先画面を非表示にした場合は、CloseReason.None を返却する。
2	public FormState CloseState { get; }	遷移先画面の状態を取得する。遷移先画面を閉じた場合は、FormState.Closed を返却する。 遷移先画面を非表示にした場合は、FormState.Hidden を返却する。
3	public FormForwardContext Context { get; }	画面遷移に関する設定情報を取得する。 FormForwarder のプロパティで設定した内容と同等の情報を取得できる。
4	public DialogResult DialogResult { get; }	遷移先画面で設定された DialogResult プロパティの値を取得する。
5	public string FormForwardCloseReason { get; }	CloseReason プロパティだけでは特定できない詳細な画面が閉じられた（非表示された）原因がある場合に返却する。 標準では、以下の値を返却するケースが存在する。 “HideForNextShowAndHide” - ShowAndHide で、次の画面へ遷移するために非表示になった場合 “HideForBackToStart” - FormForwardUtility.BackToStartFormWithHide メソッドを使用して、スコープ開始画面へ遷移したために、非表示になった場合
6	public Form ForwardForm { get; }	遷移先画面を取得する。
7	public FormState ShowState { get; }	Forward メソッド実行により遷移先画面表示時に使用した画面遷移方式を取得する。 ShowingModeless - ShowModelessで遷移 ShowingAsChild - ShowAsChildで遷移 （ShowAndHideもShowAsChildを利用し遷移する） ShowingWindowModal - ShowWindowModalで遷移 ShowingApplicationModal - ShowApplicationModalで遷移
8	public object ViewData { get; }	遷移先画面の「画面データ」を取得する。

ForwardFormClosedEventArgs.DialogResult プロパティは、遷移先の画面に設定された DialogResult の値である。画面が閉じられたときに、呼び出し元画面での業務処理の結果を判定するために使用できる。

Form.ShowDialog メソッドによりモーダル表示した画面で、図 25 のように、DialogResult プロパティに DialogResult.None 以外の値が設定されたボタンを押下すると、画面が非表示(Form.Hide)になり、呼び出し元の画面に制御が戻り Form.ShowDialog メソッドの戻り値として結果を取得することができる。

本機能では、ForwardType が「ShowApplicationModal」のみ、Form.ShowDialog メソッドと同様の動作をする。

また、入力値検証やビジネスロジックの実行に成功した時のみ画面遷移させる等、ボタン押下時に、処理の結果によって DialogResult を設定するかどうか判定するといった場合は、

ボタンの DialogResult プロパティを設定するかわりに、ボタンクリック時のイベントハンドラで Form.DialogResult プロパティを設定する処理を実装する。

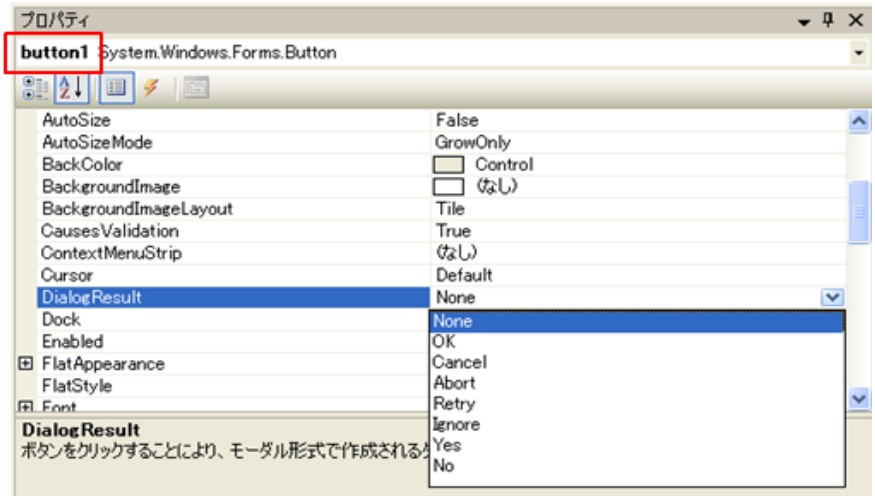

図 25 Button コントロールの DialogResult プロパティ

一方、「ShowModeless」や「ShowWindowModal」など、Form.Show メソッドを使って実装している画面遷移方式では、ボタンに DialogResult プロパティを設定する方法では画面は非表示にならない。

そこで、ボタンクリック時のイベントハンドラで DialogResult プロパティを設定する方法で対応する。

このとき、「ShowApplicationModal」と違い、コード上で、DialogResult を設定しても、画面は非表示にはならないので、Form.Hide または Form.Close メソッドにより画面を明示的に閉じる必要がある。

以下に、遷移先画面で、DialogResult プロパティを設定する実装例を示す。

```
///OKボタンクリック時の例
private void okButton_Click(object sender, EventArgs e)
{
    ///イベント処理の実行
    EventProcessResult result = a01_01_02_C01EventProcessWorker.RunWorker();
    if (result.IsSuccess)
    {
        ///入力値検証処理が失敗した場合は、DialogResult.OKにしないように、
        ///イベント成功時のみDialogResult.OKにする
        this.DialogResult = DialogResult.OK;
        ///ShowApplicationModal以外の場合は、HideまたはCloseメソッドが必要
        ///（ShowApplicationModalの場合は不要）
        Close();
    }
}
```

リスト 4 遷移先画面クラスのコードで DialogResult プロパティ値を設定する実装例

また、以下に、遷移元画面での ForwardFormClosed イベントの実装例を示す。

```
///遷移先画面がOKボタン押下で閉じられた時に、
///遷移先画面で入力したデータをもとに、DataGridViewにデータを追加する
private void formForwarderToSC_A01_01_02_ForwardedFormClosed(object sender,
        ForwardFormClosedEventArgs e)
{
    if (e.DialogResult == DialogResult.OK)
    {
        ///画面データ(DataGridViewの行に相当)を１行追加
        SC_A01_01_01Conductor newConductor = ViewData.ConductorList.AddNew();
        ///
        ///画面データの自動コピー機能でのコピーが難しいためコードで実施する例
        MappingInfo mapingInfo = new MappingInfo();
        ///プロパティ名が一致しない場合のマッピング設定の例
        mapingInfo.AddMapping("ConductorName", "Name");
        ///遷移先画面の画面データを、追加した１行にディープコピー
        DataCopyManager.Copy(e.ViewData, newConductor, mapingInfo);
    }
}
```

リスト 5 遷移元画面クラスのコードでの ForwardFormClosed イベントの実装例

◆ スコープの利用

スコープは全ての画面遷移方式で利用可能である。「ShowAndHide」は画面の状態管理が必要であるため、スコープの利用が必須である。他の画面遷移方式についてはスコープの利用は必須ではない。
スコープを新たに開始するには、FormForwarder で以下の設定をする必要がある。

- ForwardGroup プロパティに、初めて値を設定するか、直前の画面遷移時の ForwardGroup プロパティと異なる新たな値を指定する。
- ForwardType プロパティに ShowModless または ShowAsChild を設定する。

なお、アプリケーション起動時の初期画面は、FormForwarder を利用しないためスコープに所属できないので注意すること。

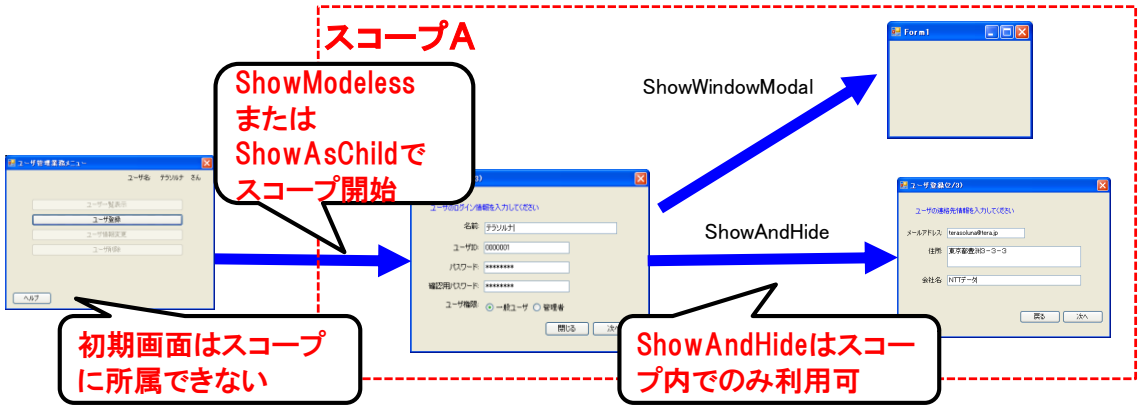

図 26 スコープ利用時の画面遷移方式の制約

現在の画面がスコープに所属していない場合は、「ShowModeless」または「ShowAsChild」のどちらでスコープを開始した場合でもスコープの管理状態は同じになる。しかし、現在の画面がすでにあるスコープに所属している場合、スコープの管理状態は異なってくる。
「ShowModeless」で遷移した場合、現在のスコープと、新たに開始したスコープは独立した関係になる。

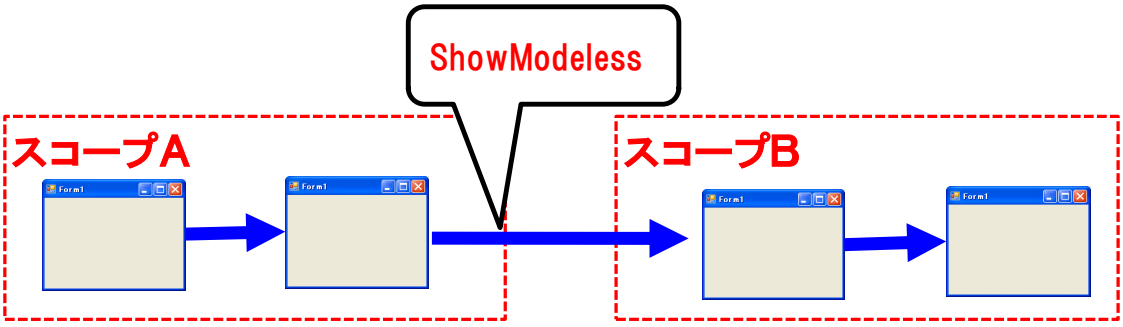

図 27 ShowModeless でスコープを開始した場合(スコープは独立した関係)

一方、「ShowAsChild」で遷移した場合、現在のスコープと新たに開始されたスコープの間で入れ子の関係が発生する。入れ子の関係の場合は、親側のスコープが消滅すると子側のスコープも連鎖的に消滅する。

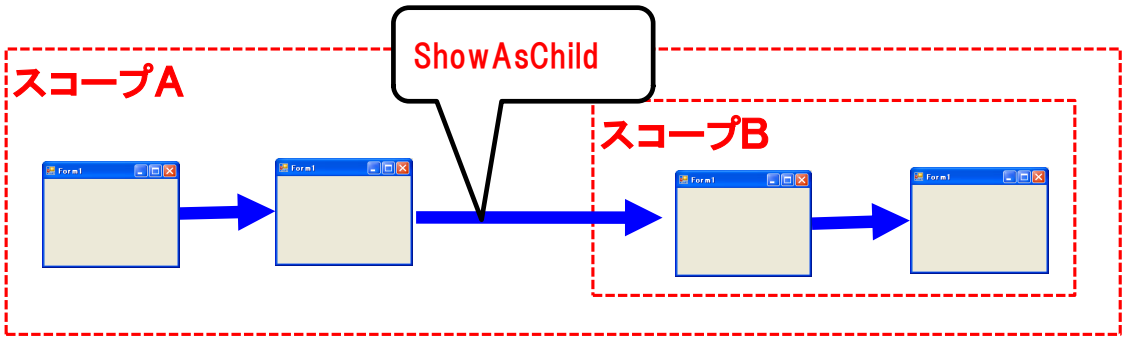

図 28 ShowAsChild でスコープを開始した場合(スコープは入れ子関係)

また、現在のスコープを終了しスコープ外の画面に遷移するには、以下のいずれかの方法を選択する。

- 前述のとおり、FormForwarder で(入れ子ではない)新たなスコープを開始して画面遷移する。
- ForwardGroup プロパティに値を指定せず、かつ ForwardType プロパティを「ShowModeless」設定して遷移する。

ただし、スコープ外へ遷移してもスコープやスコープ内の画面が破棄されるわけでない。
スコープを破棄するためには、スコープ内に生存する全画面をクローズ(破棄)する必要がある。これには、スコープの開始画面をクローズすることで連鎖的に全画面をクローズするか、後述の FormForwardGroupUtility クラスが提供するユーティリティメソッドを利用する。

◆ スコープ内で利用可能なユーティリティメソッド

スコープ利用時には、FormFowardGroupUtility クラスが提供するユーティリティメソッドにより、任意の画面からスコープ開始画面へ戻ったり、スコープ内の全画面をクローズしてスコープを消滅させたりすることができる。

表 12 FormForwardGroupUtility クラスのメソッド

項番	メソッド	第 1 引数	説明
1	public static Form BackToStartFormWithClose(Form current);	現在の画面	スコープ内の画面を全てクローズ(Close)して、スコープ開始画面を再表示(Show)する。 ユーザの入力状態が破棄されるため、再度同じ業務を新規に実施する場合に利用する。
2	public static Form BackToStartFormWithHide(Form current);	現在の画面	スコープ開始画面以外のスコープ内の画面を全て非表示(Hide)にして、スコープ開始画面を再表示(Show)する。 ユーザの入力状態を保持したまま開始画面に戻りたい場合に利用する。
3	public static void CloseAll(Form current);	現在の画面	現在の画面が所属するスコープ内の全画面をクローズし、スコープを消滅させる。 一連の業務完了時やエラー時にメニュー画面に戻る場合など、現在のスコープを終了し、スコープ外の画面に遷移または戻る場合に利用する。

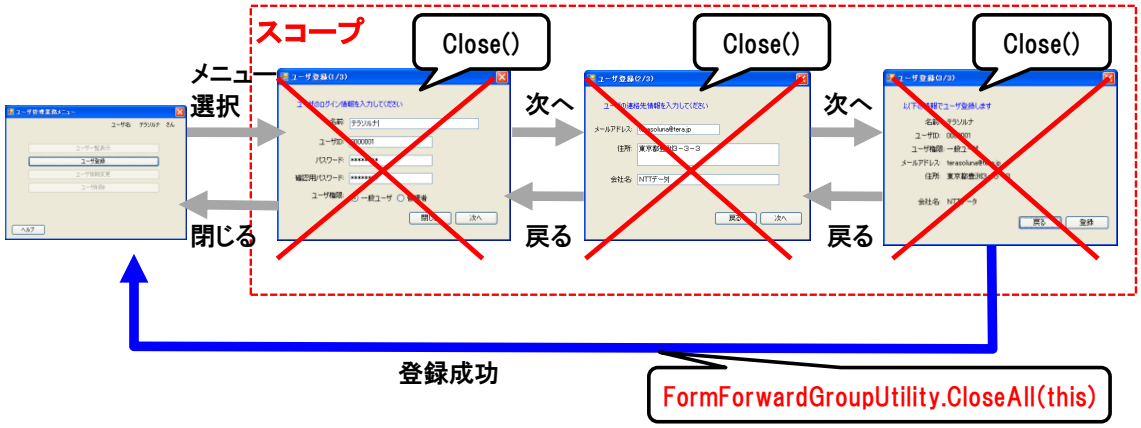

図 29 FormForwardGroupUtility.CloaseAll メソッドの動作イメージ

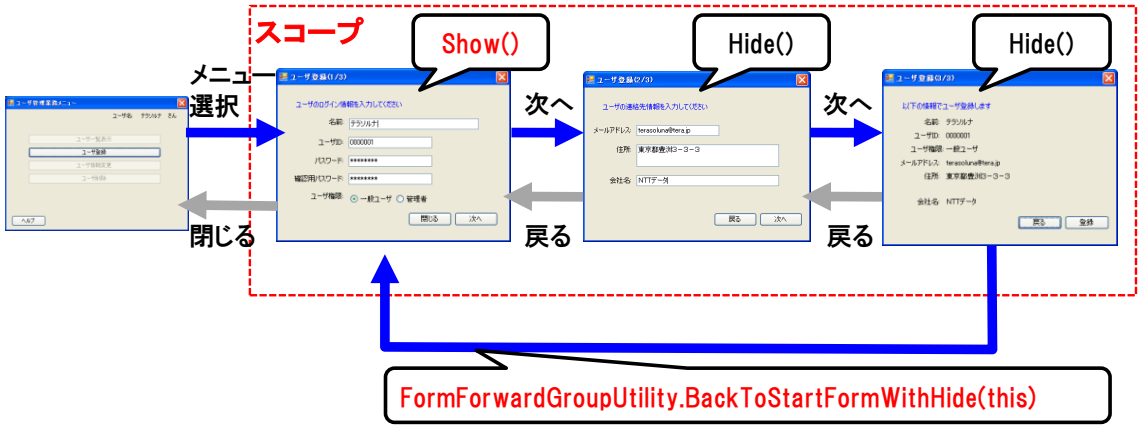

図 30 FormForwardGroupUtility. BackToStartFormWithHide メソッドの動作イメージ

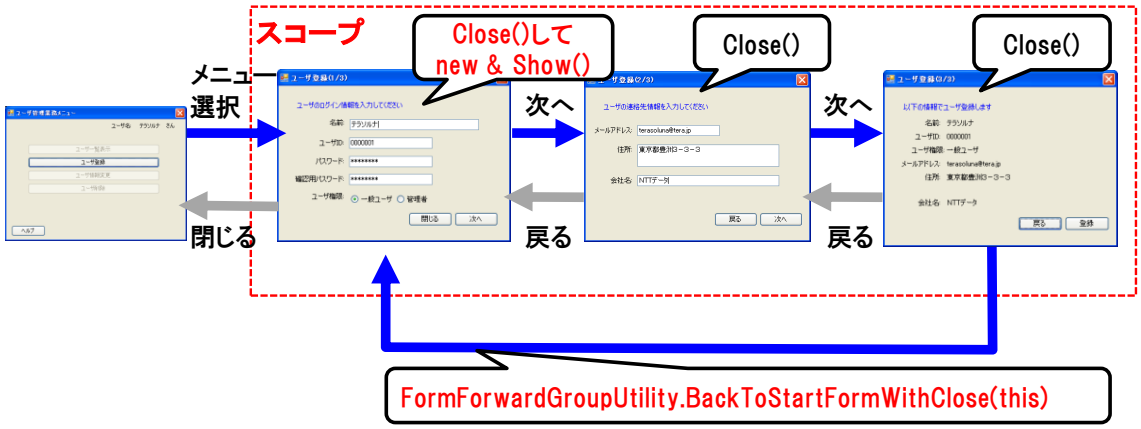

図 31 FormForwardGroupUtility. BackToStartFormWithClose メソッドの動作イメージ

以下に、FormForwardGroupUtility の利用例を示す。

```
//thisは、現在の画面クラスを指す

//スコープ上の画面を全てクローズ
FormForwardGroupUtility.CloseAll(this);

//全てHideして開始画面を再表示（Show）して戻る場合
 FormForwardGroupUtility.BackToStartFormWithHide(this);

 //全てCloseして開始画面を再作成（new & Show）して戻る場合
 FormForwardGroupUtility.BackToStartFormWithClose(this);
```

リスト 6 FormForwardGroupUtility クラスのメソッドの利用例

◆ スコープ内共通データ領域の利用

スコープ内共通データ領域は、スコープ(FormForwardGroupScope クラス)の Item プロパティ(インデクサ)により実現する。
現在のスコープを取得するには、FormFrowardManager.GetScope メソッドを使用する。通常、GetScope メソッドの引数には、現在の画面(通常、画面クラスのコードで記述するため"this"になる)クラスを渡す。
以下に、スコープ内共通データ領域を使用した、データの格納・取得例を示す。

```
//thisは、現在の画面クラスを指す

//スコープ内共通データ領域への格納
FormForwardManager.GetScope(this)["key"] = "スコープ内共通データ"

//スコープ内共通データ領域からの取得(object型へ返却されるためas演算子でキャスト)
string value = FormForwardManager.GetScope(this)["key"] as string;
```

リスト 7 スコープ内共通データ領域へのデータの格納・取得例

なお、「"StartFormShowConetxt"」という文字列は、フレームワーク内部でキーとして利用されている予約語のため、キーとしては利用できないので注意する。

◆ 設定情報を保持した画面の再表示

画面遷移の過程で、非表示（Form.Hide）にした画面を再表示したいケースがある。この際、Form.Show メソッド等を利用すると、FormForwarder の管理外となり、これまでの設定情報は失われてしまう。FormForwarder の設定情報を保持したまま画面を再表示する方法として、FormForwardManager.ReShow メソッドを使用する。

表 13 FormForwardManager.ReShow メソッド

項番	メソッド	第 1 引数	説明
1	public static void ReShow(Form targetForm);	再表示する画面	FormForwarder で表示後、非表示（Hide）にした画面を再表示する。 遷移時の FormForwarder の設定情報を保持したまま、画面を再表示する。

Form.Show メソッドによる再表示と、FormFowardManager.ReShow メソッドによる再表示の違いを以下に示す。Form.Show メソッドで再表示すると、FormForwarder の設定情報は失われ、データコピーや Closed イベントは発生しなくなる。

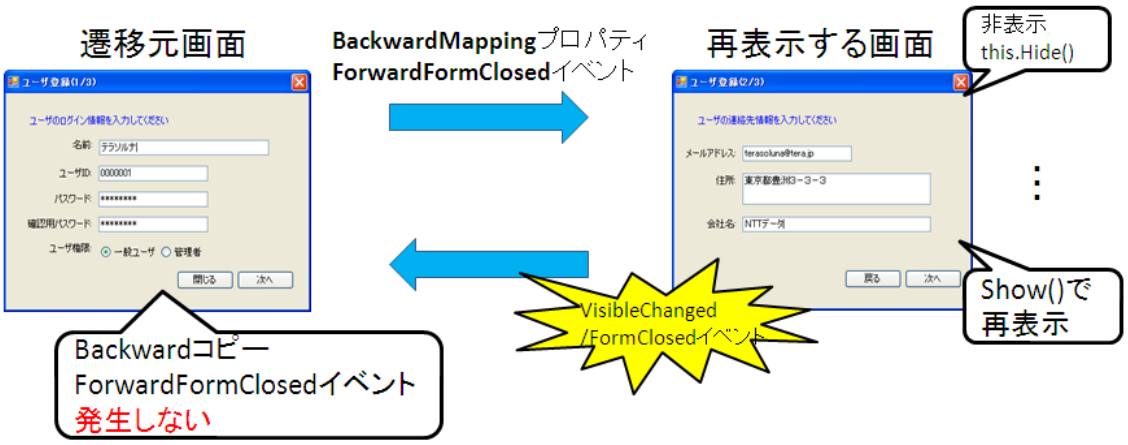

図 32 Show メソッドによる再表示の動作イメージ

一方、FormFowardManager.ReShow メソッドを使用すると、FormForwarder の設定情報を保持したまま再表示でき、データコピーや FowardFormClosed イベントは正常に発生する。

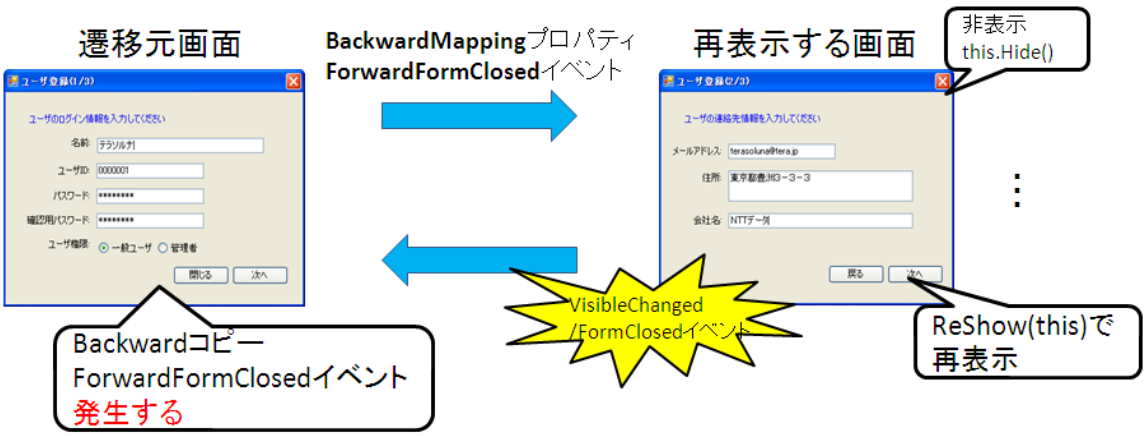

図 33 FormForwardManager.ReShow メソッドによる再表示の動作イメージ

以下に、FormForwardManager.ReShow メソッドの利用例を示す。
この例では、遷移元画面の FormFowardClosed イベントで ReShow メソッドを呼び出すことで、ShowAndHide の動作のように遷移先画面を閉じると元の画面を再表示する処理を実現している。

```
///遷移先画面が閉じられた時に、ReShowメソッドで自分自身を再表示する
private void formForwarderToSC_A01_01_02_ForwardedFormClosed(object sender,
            ForwardFormClosedEventArgs e)
{
    FormForwardManager.ReShow(this);
}
```

リスト 8 FormForwardManager.ReShow メソッドの利用例

なお、ReShow メソッドは、FormForwarder を使用して表示した画面のうち以下のものに対して実行することができる。

- 非表示状態(Hide)の画面
- 「ShowModeless」か「ShowAsChild」の遷移方式で表示された画面

よって、現在表示中の画面や、FormForwarder を使用せずに表示した画面(アプリケーション起動時の表示する初期画面など)、FormForwarder の上記以外の遷移方式で表示された画面に対しては ReShow を実行することができないので注意すること。

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

FormForwarder を使用するには、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。

- Terasoluna.Windows.Forms.FormForward.FormForwardExtension クラス
  - FormForwarder による画面遷移機能に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>
          <!-- 画面遷移機能の設定 -->
          <add type="Terasoluna.Windows.Forms.FormForward.FormForwardExtension,
                    Terasoluna.Windows.Forms" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 9 TerasolunaFramework.config の記述例

◆ FW 起動構成ファイル(TerasolunaBootstrap.config)の設定

FormForwarder を使用する場合、複数のアセンブリから構成されるアプリケーションでも ScreenId 属性に合致する画面クラスを検索できるよう、「CM-01 アプリケーション起動・終了機能」により、AP 起動時に、実行ファイルが存在するディレクトリに存在するアセンブリを全てロードする初期化処理クラス(LoadAssembliesFromDirectoryInitializer クラス)を実行する。
初期化処理クラスの設定は、FW 起動構成ファイル(TerasolunaBootstrap.config)で実施する。詳細は、「CM-01 アプリケーション起動・終了機能」の機能説明書を参照のこと。

以下に、TerasolunaBootstrap.config の設定例を示す。
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>
    <typeAliases>
      <!-- エイリアスの定義 -->
      ・・・
      <typeAlias alias="IInitializer[]" type="Terasoluna.Initialization.IInitializer[],
                        Terasoluna" />
      <typeAlias alias="IInitializer" type="Terasoluna.Initialization.IInitializer,
                        Terasoluna" />
      <typeAlias alias="SequenceInitializer" type="Terasoluna.Initialization.SequenceInitializer,
                        Terasoluna" />
    <containers>
      <container>
        <types>
          <!-- フレームワーク起動時に実行される初期化処理実施(Intializer)クラスの設定-->
          <!-- IInitilizerインタフェースのデフォルトDI定義
                SequenceInitializerクラスが、Intializersに登録したIIntializerクラスに呼び出す -->
          <type type="IInitializer" mapTo="SequenceInitializer">
            <lifetime type="singleton" />
            <typeConfig>
              <property name="Initializers" propertyType ="IInitializer[]">
                <array>
                  <dependency name=" LoadAssembliesFromDirectoryInitializer "/>
                </array>
              </property>
            </typeConfig>
          </type>
          <!-- LoadAssembliesFromDirectoryInitializer の実行設定-->
          <type type="IInitializer" name="LoadAssembliesFromDirectoryInitializer"
                  mapTo="Terasoluna.Windows.Initializers.LoadAssembliesFromDirectoryInitializer,
                        Terasoluna.Windows" >
            <lifetime type="singleton" />
          </type>
          ・・・
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 10 TerasolunaBootstrap.config の記述例

■ ContentPlaceHolder の使用方法

◆ 概要

ContentPlaceHolder は、FormForwarder と同様、画面に貼り付け可能な部品である。
業務開発者が、ContentPlaceHolder による画面遷移処理を実装する主な手順は以下の通りである。

① 「画面データ」クラスを実装する。
② パネル共通の業務インタフェースを定義する。
③ パネルを実装する。各パネル固有のバインド設定やイベント処理の実行など実装する。
④ パネルを配置する画面を作成し ContentPlaceHolder をツールボックスから画面へ貼り付ける。
⑤ タイトル名の取得やパネル外に配置された共通のボタン押下時の業務処理の呼び出しなど画面で発生する処理を実装する。

以下に、ContentPlaceHolder を利用して業務処理の呼び出しを行う作業手順を示す。

◆ 実装方法

①「画面データ」クラスの実装

TERASOLUNA フレームワークでは、通常、画面(Form)クラスと「画面データ(ルート) 」クラスが1対1になるように設計する。しかし、パネル切り替えの場合は、パネルと「画面データ(ルート) 」が1対1になるように設計する。
この時、大元の画面は ContentPlaceHolder を介して現在表示中のパネルに対応する「画面データ(ルート)」を動的に取得するように実装する。(画面の実装例については後述する)

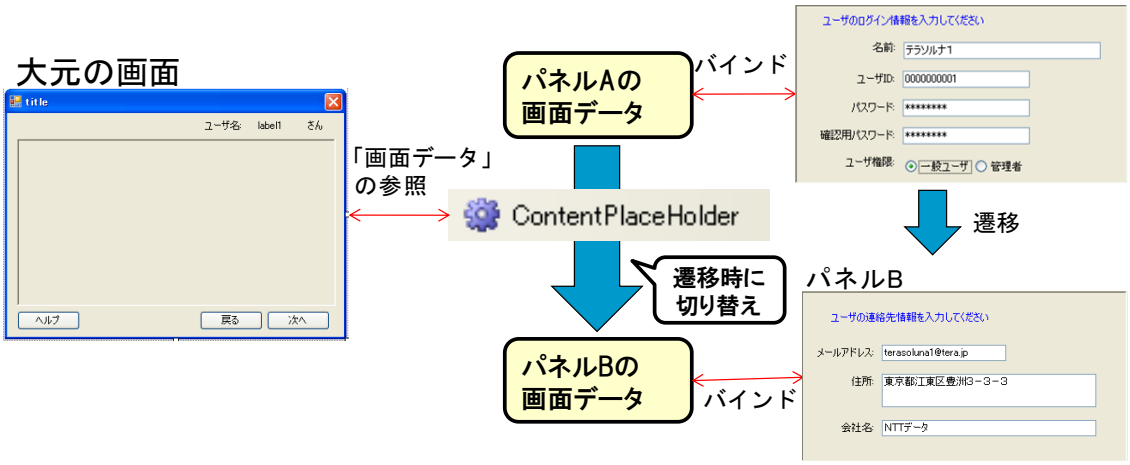

図 34 ContentPlaceHolder 使用時の「画面データ」の切り替えイメージ

「画面データ(ルート)」クラスの作成単位がパネルであること以外は、通常の「画面データ」と全く同じ実装方法である。実装方法については、「CL-01 画面データ機能」の機能説明書を参照のこと。

```
[DefaultRuleset("RS01")]
public class SC_B01_01_01ViewData : 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; }
}
```

リスト 11 「画面データ」クラスの実装例（再掲）

②パネル共通の業務インタフェースの定義

「画面データ」の動的な切り替え以外にも、表示されるパネルによって大元の画面にあるボタン押下時の挙動やタイトルの表示などを動的に切り替える必要がある。

そこで、業務要件に応じてパネルに必要な業務処理をインタフェースとして定義する。

大元の画面は、ContentPlaceHolder クラス経由で表示中のパネルを取得し、当業務インタフェースを介してメソッドやプロパティを操作することで、現在表示中のパネル固有の業務処理を実施する（画面の実装例については後述する）。

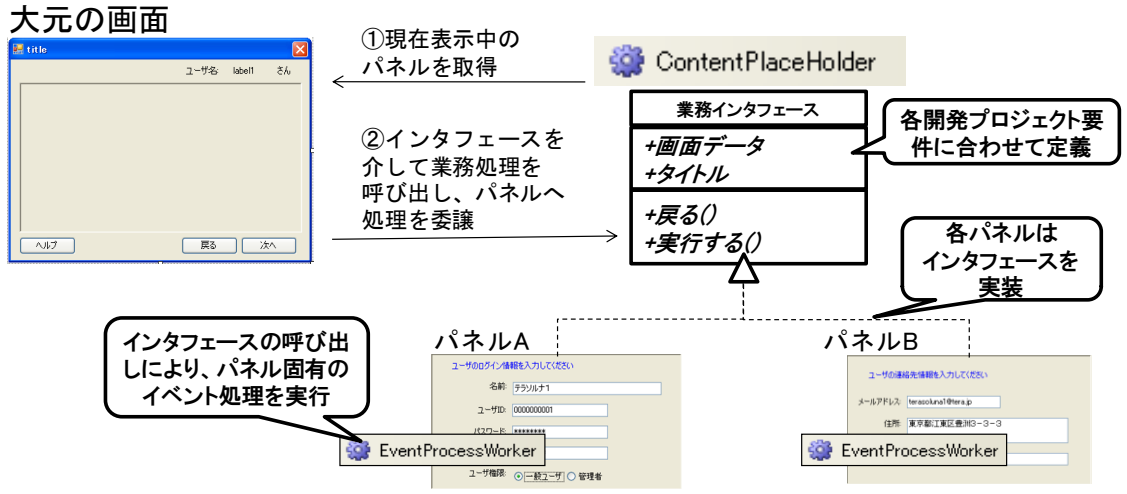

図 35 パネルが実装する業務インタフェースのイメージ

以下に、業務インタフェースの記述例を示す。

このとき、大元の画面は、表示中のパネルの「画面データ」を取得するのに業務インタフェースを介して取得するため（後述の画面の実装例を参照のこと）、「画面データ」のプロパティを定義する。

「画面データ」はパネルによって切り替わるため、object 型やインタフェース等で定義する。

また、タイトル、ボタンに表示する文字列やボタンの有効/無効にかかわるプロパティ、業務処理を実施するメソッドなど各プロジェクトの業務要件に合わせて定義する。

```
public interface ISC_B01_03_01MainContent
{
    /// パネルに対する画面データ（ルート）
    object ViewData { get; }

    /// 画面のタイトル
    string Title { get; }

    /// 実行ボタンに表示する文字列
    string ExecuteButtonCaption { get; }

    /// 実行ボタンを有効にするか
    bool ExecutionEnabled { get; }

    /// 戻るボタンを有効にするか
    bool BackEnabled { get; }

    ・・・..

    /// 実行ボタン押下による処理を実行する
    void Execute();

    /// 戻るボタン押下による処理を実行する
    void Back();
}
```

リスト 12 パネル共通の業務インタフェースの例

③パネルの実装

パネルを作成する際には、TERASOLUNA フレームワークが提供する「Panel コンテント コントロール」テンプレートを使用する。

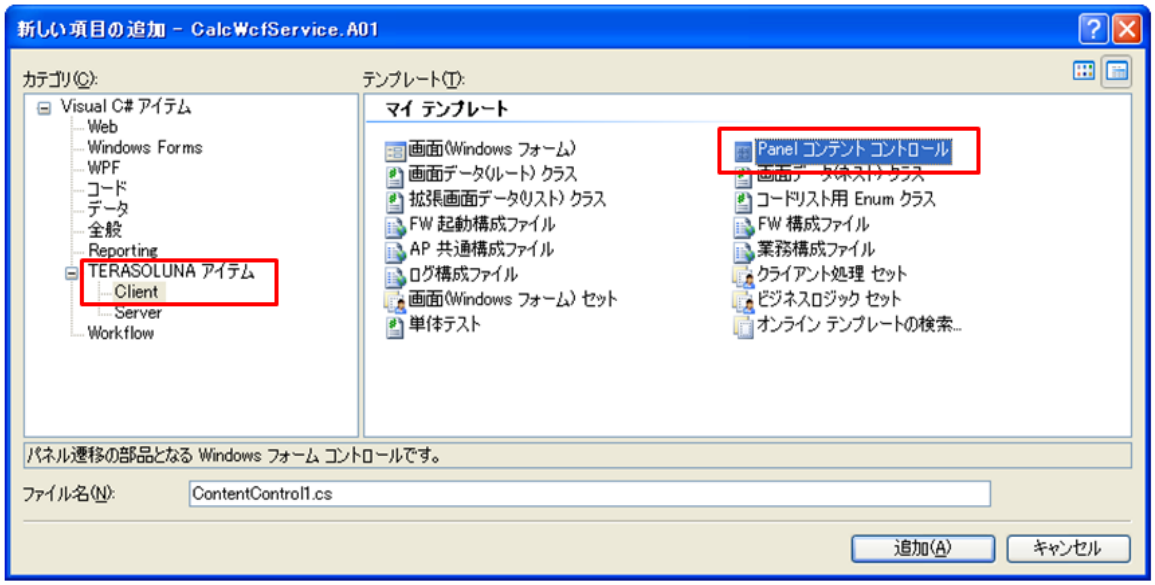

図 36 「Panel コンテントコントロール」の追加

テンプレートを使用して作成することで、パネルのコードのひな形が追加された Control 継承クラスが生成される。
パネルの実装時には開発者は以下のルールを守り実装する。ただし、テンプレートが、これらのルールに従ったひな形を生成するので、開発者による実装の負担はかなり少なくなっている。

- IContentControl インタフェースを実装する。
  各パネルは、IContentControl が定義する、以下のメンバを実装する必要がある。

表 14 IContentControl インタフェースのメンバ

メンバ	説明
ContentPlaceHolder ContentPlaceHolder { get; set; }	当該パネルを管理する ContentPlaceHolder を保持するプロパティ。ContentPlaceHolder の画面遷移メソッド（後述）を使用してパネルを表示すると、新しく表示されたパネルに自動的に設定される。
void HandleHidden(object sender, ControlEventArgs e);	当該パネルが非表示状態になった際に実施する処理を実装するメソッド。パネルが非表示状態になると実行される。
void HandleShowing(object sender, ControlEventArgs e);	当該パネルが表示状態になった際に実施する処理を実装するメソッド。パネルが表示状態になると実行される。

  また、この時 ContentPlaceHolder プロパティには DefaultValue 属性を引数 null で付与すること。
  ContentPlaceHolder プロパティには、自動的にパネルを管理する ContentPlaceHolder オブジェクトが格納される。ところが、DefaultValue 属性を付与しない場合は、Visual Studio のデザイナによって ContentPlaceHolder プロパティに null をセットするコードが生成される。これにより、Visual Studio のデザイナ上でパネルを画面に貼り付けた場合、画面の初期表示時に ContentPlaceHolder プロパティの値を null で上書きしてしまい、うまく動作しなくなってしまうための対処策である。

- 実際の「画面データクラス（ルート）」の型を持ったプロパティを定義する。この時、パネルのコンストラクタで「画面データ」のインスタンスを生成しないようにすること。
  その代わり、「画面データ」プロパティの get アクセサで、「画面データ」が null であればインスタンスを生成するように実装して、インスタンス生成処理を遅延させるようにする。
  これは、「画面データ」のインスタンスをコンストラクタで生成してしまうと、Visual Studio のデザイナ上でパネルを画面に貼り付けた場合、TERASOLUNA フレームワークのデザインモードで「画面データ」のインスタンス生成処理が実行されエラーになってしまうの防ぐ対処策である。
  パネルには、バインド対象の「画面データ」を使用して、UI コントロールとの双方向データバインドを実施する。双方向バインド設定は、通常の画面と同じ手順である。詳細は「CL-01 画面データ機能」の機能説明書を参照のこと。

  また、前述のパネル共通の業務インタフェースを実装し、コントロール固有の処理を実装す

る。このとき、パネルに対するイベント処理を実装するには、パネルに直接 EventProcessWorker を貼り付けて、イベント処理の設定を実施すること。
イベント処理の実装方法も通常の画面と同じ手順である。詳細は、「CL-03 イベント処理実行機能」の機能説明書を参照のこと。
なお、パネル切り替えによる画面遷移を実行したい場合には、ContentPlaceHolder クラスが提供する以下の画面遷移メソッドを使用する。

表 15 ContentPlaceHolder クラスが提供する画面遷移メソッド

メソッド	引数	説明
public void ShowAndCloseContent(Type contentType)	遷移先パネルの型	指定した型のパネルを表示(Show)し、現在のパネルがあればインスタンス管理を終了し破棄(Close)する。
public void ShowAndHideContent(Type contentType)	遷移先パネルの型	指定した型のパネルを表示(Show)し、現在のパネルがあれば、インスタンス管理は継続したまま非表示(Hide)にする。
public void ShowWithCloseAllContents(Type contentType)	遷移先パネルの型	インスタンス管理中の全てのパネルをインスタンス管理終了／破棄(Close)した上で、指定した型のパネルのみを表示(Show)する。
public void CloseAllContents()	-	インスタンス管理中の全てのパネルをインスタンス管理終了／破棄(Close)する。

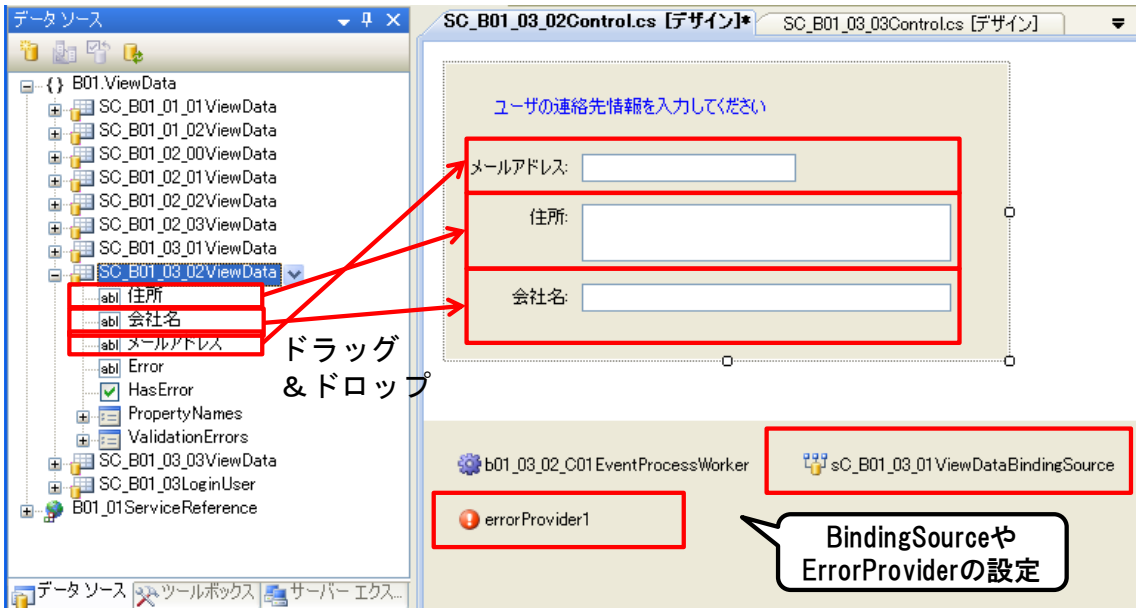

図 37 パネルに対する UI コントロールの貼り付けと「画面データ」とのバインド設定

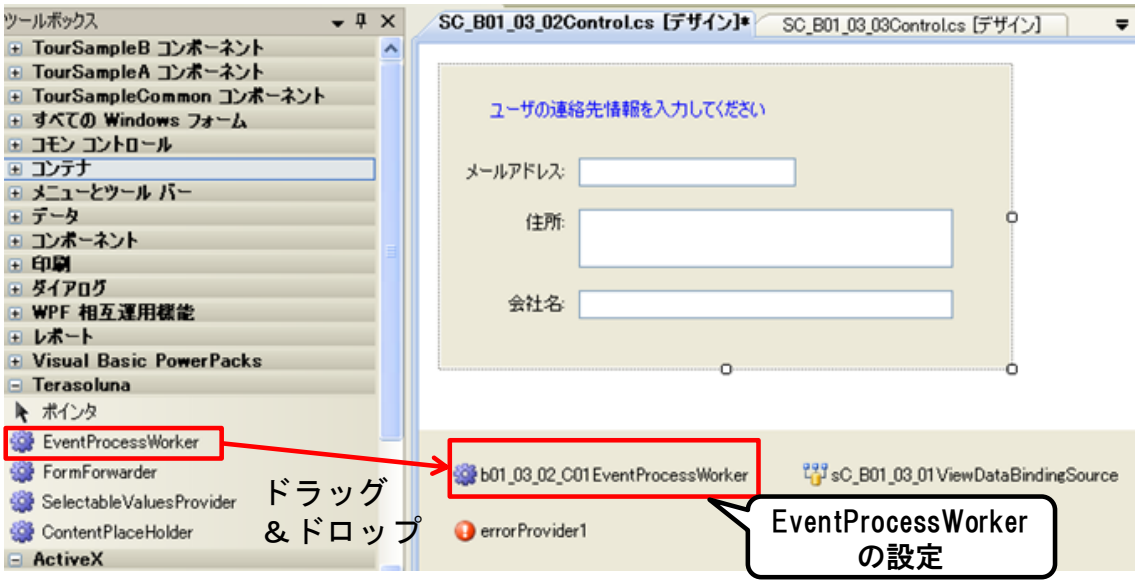

図 38 パネルに対する EventProcessWorker の貼り付け

- 通常の「画面データ」のバインド設定と同様に、パネルの Load イベントで、BindingSource.DataSource プロパティに「画面データ」をセットする。

以下に、パネルの実装例を示す。

```
public partial class SC_B01_03_02Control : UserControl, ISC_B01_03_01MainContent, IContentControl
{
    /// <summary>
    /// パネルに対する画面データ（ルート）
    /// </summary>
    private SC_B01_03_02ViewData _viewData;

    ///デザイナでパネルを貼り付ける場合を考慮して、
    ///画面データのインスタンス生成処理が実行されないように、
    ///ViewDataプロパティへの初回アクセス時に画面データのインスタンスを作成
    public SC_B01_03_02ViewData ViewData
    {
        get
        {
            if (_viewData == null)
            {
                _viewData = CreateViewData();
            }
            return _viewData;
        }
    }
    /// 画面データの生成
    protected virtual SC_B01_03_02ViewData CreateViewData()
    {
        return ValidatableViewDataManager.CreateViewData<SC_B01_03_02ViewData>();
    }

    public SC_B01_03_02Control()
    {
        InitializeComponent();
    }

   /// 以下、パネル共通の業務インタフェースの実装
    #region ISC_B01_03_01MainContent メンバ

    /// パネルに対する画面データ（ルート）
    object ISC_B01_03_01MainContent.ViewData
    {
        get { return this.ViewData; }
    }

    /// 画面のタイトル
    public string Title
    {
        get { return "ユーザ登録(2/3)"; }
    }

    /// 実行ボタンに表示する文字列
    public string ExecuteButtonCaption
    {
        get { return "次へ"; }
    }
・・・
```

```
・・・
    /// 実行ボタンを有効にするか
    public bool ExecutionEnabled
    {
        get { return true; }
    }
    /// 戻るボタンを有効にするか
    public bool BackEnabled
    {
        get { return true; }
    }

・・・
    /// 実行ボタン押下による処理を実行する
    public void Execute()
    {
        EventProcessResult result = b01_03_02_C01EventProcessWorker.RunWorker();
        if (result.IsSuccess)
        {
            ContentPlaceHolder.ShowAndHideContent(typeof(SC_B01_03_03Control));
        }
    }

    /// 戻るボタン押下による処理を実行する
    public void Back()
    {
        ContentPlaceHolder.ShowAndHideContent(typeof(SC_B01_03_01Control));
    }

    #endregion

   /// 以下、IContentControlインタフェースの実装
    #region IContentControl メンバ

    //デザイナがContentPlaceHolderにnullをセットするコードを生成しないよう
    //DefaultValue(null)を付与する
    [DefaultValue(null)]
    public ContentPlaceHolder ContentPlaceHolder { get; set; }

    public void HandleShowing(object sender, ControlEventArgs e)
    {
    }
・・・
    public void HandleHidden(object sender, ControlEventArgs e)
    {
    }
    #endregion

    ///通常の画面と同様、パネルのロード時に「画面データ」（ルート）を
    ///BindingSource.DataSourceプロパティにセットする
    private void SC_B01_03_02Control_Load(object sender, EventArgs e)
    {
        sC_B01_03_02ViewDataBindingSource.DataSource = ViewData;
    }
}
```

リスト 13　パネルの実装例

④画面の作成と ContentPlaceHolder の貼り付け

パネルを配置する画面クラスを作成し、ContentPlaceHolder を貼り付ける。

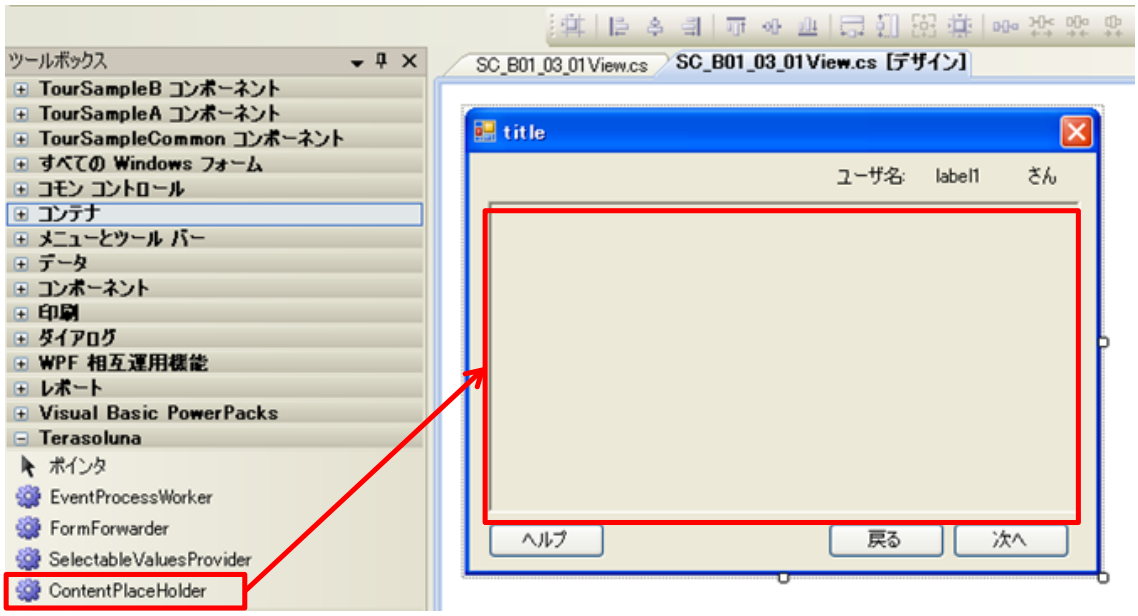

図 39 ContentPlaceHolder の貼り付け

⑤画面で発生する処理の実装

通常の画面クラスの作成ルール(「CL-01 画面データ機能」の機能説明書を参照)をベースに、以下のパネル切り替え固有のルールに従い実装する。

- 通常の画面と同様のルールにより、「ViewData」という名前のプロパティを定義する必要がある。
  ただし、パネル切り替えの場合、通常の画面とは違い画面自身は「画面データ(ルート)」のインスタンス生成処理は実施しない。その代わり、「ViewData」プロパティの get アクセサの中で、現在表示中のパネルの保持する「画面データ(ルート)」を取得する。
  ContentPlaceHolder.CurrentControl プロパティで現在表示されているパネルを取得後、as 演算子により業務インタフェースにキャストして、業務インタフェースに定義した「画面データ」のプロパティを使用して「画面データ(ルート)」を動的に取得することができる。
  実際の「画面データ(ルート)」の型は表示されるパネルによって切り替わるため、object 型や汎用的なインタフェースで定義する。

- 画面のコンストラクタで、最初のパネルを表示する処理を記述する。
  遷移元の画面(Form)から当該画面に遷移時、FormForwarder による「画面データ」の自動コピー処理を実施するケースがあることを考慮し、
  ContentPlaceHolder.ShowAndHideContent メソッドを使用して、画面のコンストラクタで最初のパネルを表示する。これにより、画面の「ViewData」プロパティに最初のパネルの「画面データ(ルート)」のインスタンスがセットされ、取得可能な状態になるので、FormForwarder によるコピー処理を正常に実行することができる。

- パネル外に配置された「実行」ボタンや「戻る」ボタンなどの画面共通のボタン押下時に対する業務処理を、ボタンクリック時のイベント等で実装する。この時、イベントハンドラには、パネル固有の業務処理を実装せず、業務インタフェースのメソッドを呼び出すようにする。
  ContentPlaceHolder.CurrentControl プロパティで現在のパネルを取得後、as 演算子により業務インタフェースにキャストして、業務インタフェースに定義したメソッドやプロパティを呼び出し、現在表示中のパネルに処理を委譲する。

- タイトル表示やボタンの有効化など、パネル切り替え時の画面描画処理は、ContentPlaceHolder.ControlAdded イベント内に実装する。
  ContentPlaceHolder.ControlAdded イベントは、遷移先パネルを表示した時（つまり、遷移先パネルを ContentPlaceHolder へ 追加した時）に発生するイベントである。
  イベントハンドラでは、ContentPlaceHolder.CurrentControl プロパティで現在のパネルを取得後、as 演算子により業務インタフェースにキャストして、業務インタフェースに定義したメソッドやプロパティを呼び出し、現在表示中のパネルに処理を委譲する。

以下に、画面の実装例を示す。

```
[ScreenId("SC_B01_03_01")]
public partial class SC_B01_03_01View : TourSampleViewBase
{
    /// 画面データ
    public object ViewData
    {
        get
        {
            object viewData = null;
            ///ContentPlaceHolder経由で表示中のパネルに対する画面データを取得
            ISC_B01_03_01MainContent mainContent = contentPlaceHolder.CurrentContentControl
as ISC_B01_03_01MainContent;
            if (mainContent != null)
            {
                viewData = mainContent.ViewData;
            }
            return viewData;
        }
    }

    public SC_B01_03_01View()
    {
        InitializeComponent();
        //最初のパネルを追加
        contentPlaceHolder.ShowAndHideContent(typeof(SC_B01_03_00Control));
    }

・・・
```

```
・・・

    /// 戻るボタン押下時
    private void backButton_Click(object sender, EventArgs e)
    {
        ///ContentPlaceHolder経由で表示中のパネルに処理を委譲
        ISC_B01_03_01MainContent mainContent = contentPlaceHolder.CurrentContentControl as
ISC_B01_03_01MainContent;
        if (mainContent != null)
        {
            mainContent.Back();
        }
    }

    /// 実行ボタン押下時
    private void executeButton_Click(object sender, EventArgs e)
    {
        ///ContentPlaceHolder経由で表示中のパネルに処理を委譲
        ISC_B01_03_01MainContent mainContent = contentPlaceHolder.CurrentContentControl as
ISC_B01_03_01MainContent;
        if (mainContent != null)
        {
            mainContent.Execute();
        }
    }

     /// ContentPlaceHolderにパネルが追加された時のイベント
     private void contentPlaceHolder_ControlAdded(object sender, ControlEventArgs e)
     {
        ///追加されたパネルを取得
        ISC_B01_03_01MainContent mainContent = e.Control as ISC_B01_03_01MainContent;
        if (mainContent != null)
        {
            ///実行ボタンの文字列を設定
            executeButton.Text = mainContent.ExecuteButtonCaption;
            ///実行ボタンを表示するかを設定
            executeButton.Visible = mainContent.ExecutionEnabled;
            ///戻るボタンを表示するかを設定
            backButton.Visible = mainContent.BackEnabled;
            ///タイトルを設定
            this.Text = mainContent.Title;
        }
    }
}
```

リスト 14 画面の実装例

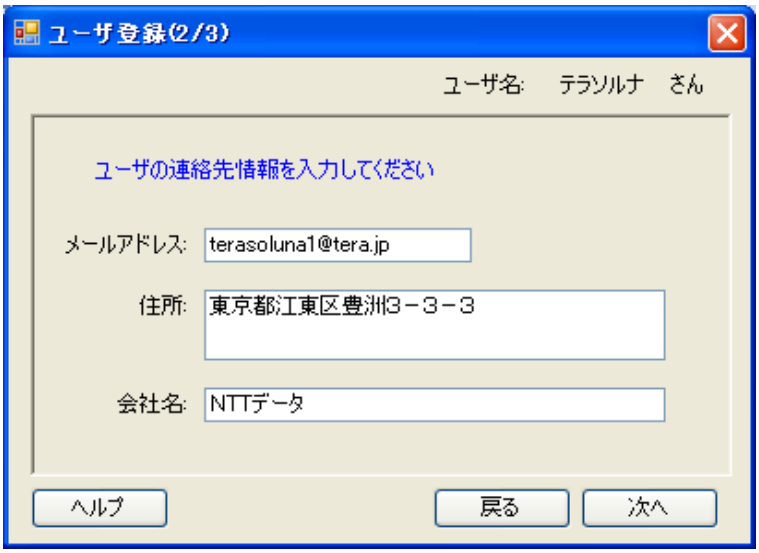

図 40 実行時のイメージ

◆「画面データ」のコピー処理の実装

本機能では、「CM-04 データコピー機能」を利用し、画面遷移時に遷移元パネルと遷移先パネル間の「画面データ」のコピーが可能である。
ただし、FormForwarder のようにプロパティ設定に基づき自動的にコピーする機能はないため、開発者が、IContentControl.HandleShowing メソッドで、遷移時に、遷移元パネルと遷移先パネルの「画面データ」をコピーする処理を実装する必要がある。
遷移元パネルは、ContentPlaceHolder.LastContentControl プロパティで取得可能である。遷移先パネル(現在表示中のパネル)は、ContentPlaceHolder.CurrentContentControl プロパティで取得可能である。
データコピーの実行には、DataCopyManager.Copy メソッドを使用する。DataCopyManager クラスの使用方法は、「CM-04 データコピー機能」を参照のこと。

以下に、データコピーの実装を示す。

```
public void HandleShowing(object sender, ControlEventArgs e)
{
    ///遷移元遷移先パネル間で画面データをコピーして引き継ぐ例
    ISC_B01_03_01MainContent lastContentControl =
        ContentPlaceHolder.LastContentControl as ISC_B01_03_01MainContent;
    ISC_B01_03_01MainContent currentContentControl =
        ContentPlaceHolder.CurrentContentControl as ISC_B01_03_01MainContent;
    DataCopyManager.Copy(lastContentControl.ViewData, currentContentControl.ViewData);
}
```

リスト 15 画面データのコピー処理の実装例

IContentControl.HandleShowing を使用した画面コピーの使用する際の注意点として、HandleShowing メソッドは、パネルが表示されたときに無条件に実行されるため、FormForwarder の画面コピーのように遷移方向を意識しない。

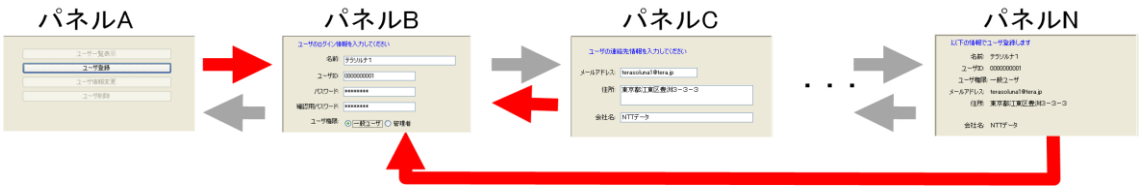

図 41 遷移元が複数あるパネルの例

遷移元パネルによって処理を分岐させたい場合は、遷移元パネルの型を判定する処理を実装する必要がある。以下に実装例を示す。

```
public void HandleShowing(object sender, ControlEventArgs e)
{
    ///遷移元パネルの取得
    ISC_B01_03_01MainContent lastContentControl = ContentPlaceHolder.LastContentControl
                    as ISC_B01_03_01MainContent;
    if (lastContentControl is SC_B01_03_00Control)
    {
        //遷移元がパネルAの場合の処理
    }
    else if (lastContentControl is SC_B01_03_02Control)
    {
        //遷移元がパネルCの場合の処理
    }
    else if (lastContentControl is SC_B01_03_02Control)
    {
        //遷移元がパネルNの場合の処理
    }
}
```

リスト 16 遷移元パネルによって処理を分岐する実装例

◆ ContentPlaceHolder によりインスタンス管理されたパネルの取得

ContentPlaceHolder がインスタンス管理しているパネルで」あれば、

ContentPlaceHolder.GetContentControl メソッドにより任意の場所でパネルオブジェクトを取得することができる。これにより、パネルが保持する「画面データ（ルート）」も取得できるため、FormForwarder における「スコープ」内共通データ領域のように、一連の業務の画面遷移グループ間で共有するデータとして使用することができる。

以下に ContentPlaceHolder.GetContentControl メソッドの使用例を示す。
メソッドの型パラメータには、取得するパネルの型を指定する。なお、指定した型のパネルが ContentPlaceHolder で管理されていない場合は、null が返却される。

```
///ContentHolderが管理するパネルの画面データを取得し、現在のパネルの画面データへコピー
SC_B01_03_01Control sc_B01_03_01Control =
                        ContentPlaceHolder.GetContentControl<SC_B01_03_01Control>();
SC_B01_03_01ViewData sc_B01_03_01ViewData = sc_B01_03_01Control.ViewData;
```

リスト 17　ContentPlaceHolder.GetContentControl メソッドの使用例

■ 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。

◆ FormForwarder による「スコープ」を使った「画面データ」の引き継ぎ

ウィザード形式の画面や、「○○検索画面」→「○○修正画面」→「○○入力確認画面」といった一連の画面遷移がパターン化されたアプリケーションでは、同一業務処理の画面を進んだり戻ったりしながら、最後の画面で一度にビジネスロジックを実行するといった形式になる。このため、前画面の入力データを引き継いでいく必要がある。
このようなケースでは、「スコープ」を使用し、スコープ内共通データ領域に、画面間で引き継ぎたいデータを格納するとよい。

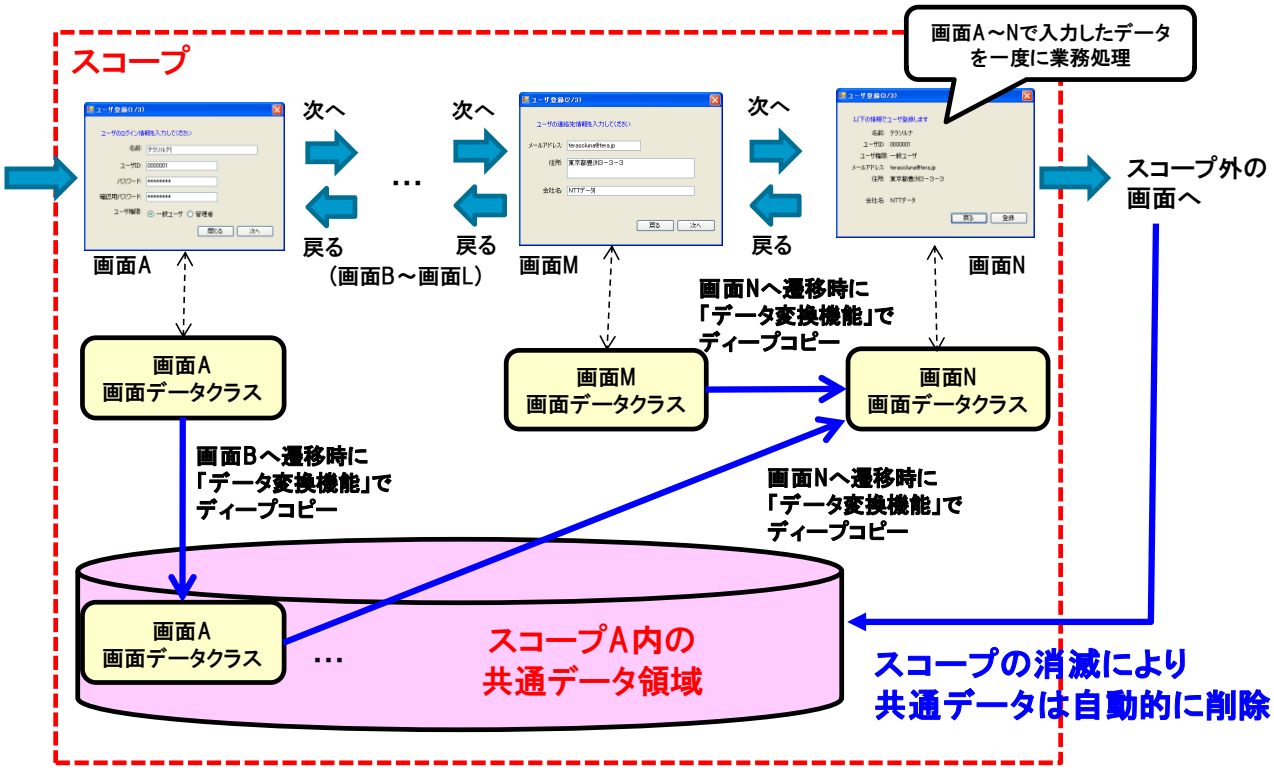

図 42 スコープを使った「画面データ」の引き継ぎイメージ

まず、スコープ内最終画面の2つ前までの画面遷移では(画面 A～L での実装に相当)、FormForwarder.ForwardFormLoading イベントを使用して、次画面へ遷移する際に現在の「画面データ」をディープコピーして複製を作成し、スコープ内共通データ領域に格納するように実装する。
ここでディープコピーしている理由は、もし参照渡しをしてしまうとオブジェクトに対して別の画面の処理で開発者が「画面データ」のプロパティを更新してしまい元の画面に対して予期せぬ画面更新が実行され誤動作が起こることを防ぐためである。
誤動作の心配がない場合や、他の画面で修正したものを元の画面表示に反映したい場

合などは、参照渡しを採用してもよい。
なお、ディープコピー処理には、「CM-04 データコピー機能」の DataCopyManager クラスの Copy メソッドを利用する。
以下に、実装例を示す。

```
private void formForwarderToSC_B01_02_02_ForwardFormLoading(object sender,
            ForwardFormLoadingEventArgs e)
{
      SC_B01_02_01ViewData sC_B01_02_01ViewData =
            ValidatableViewDataManager.CreateViewData<SC_B01_02_01ViewData>();
      //ディープコピーしてスコープ内共通データ領域に格納
      DataCopyManager.Copy(ViewData, sC_B01_02_01ViewData);
      FormForwardManager.GetScope(this)["SC_B01_02_01ViewData"] = sC_B01_02_01ViewData;
}
```

リスト 18「画面データ」を引き継ぐための実装例

次に、スコープ内最終画面への画面遷移では(画面 M での実装に相当)、FormForwarder.ForwardFormLoading イベントを使用して、次画面へ遷移する際にスコープ内共通データ領域に格納したすべての「画面データ」(画面 A～L に相当)を、最終画面の「画面データ」へディープコピーする。また、現在の画面(画面 M に相当)は、直接、最終画面へディープコピーする。通常、遷移元画面と遷移先画面間でのデータの引き継ぎは、「画面データ」の自動コピー機能を使用するため、FormForwarder.ForwardMapping.DestinationTargetPropertyPaths プロパティにコピー対象の項目を明示しなければならない。
しかし、このようなケースでは、「画面データ」のほぼ全ての項目を引き継ぐ必要があるので、DestinationTargetPropertyPaths プロパティを一つ一つ指定するのは大変である。そこで、暗黙的な自動コピーをさせるため、DataCopyManager.Copy メソッドを使ってコピーするとよい。

```
private void formForwarderToSC_B01_02_03_ForwardFormLoading(object sender,
ForwardFormLoadingEventArgs e)
{
      //画面データをスコープ内共通データ領域から取得
      object sC_B01_02_01ViewData =
                  FormForwardManager.GetScope(this)["SC_B01_02_01ViewData"];
      if (sC_B01_02_01ViewData != null)
      {
            //次画面の画面データへディープコピー
            DataCopyManager.Copy(sC_B01_02_01ViewData, e.ViewData);
      }
            //本画面データを最終画面の画面データへディープコピーする
            //この場合は、画面データの自動コピー機能を機能は使用せず、
            //DataCopyManager.Copyメソッドを呼び出すことで暗黙的なに自動コピーをする
            DataCopyManager.Copy(ViewData, e.ViewData);
}
```

リスト 19 最終画面へ遷移時に「画面データ」を引き継ぐための実装例

スコープ最終画面では、業務処理成功時に、FormForwardGroupUtility クラスが提供する CloseAll メソッドを使用して、スコープを終了する。これにより、スコープ内共通データ領域も自動的に破棄される。CloseAll メソッドの代わりに、BackToStartFormWithClose メソッドを使用して、一旦スコープを終了した後に、開始画面を再作成してスコープを再開始することもできる。
また、業務エラー時など、BackToStartFormWithHide メソッドで、スコープを終了せずに、スコープ内の画面を Hide し開始画面を再表示することで、ユーザ入力状態を保持したまま、開始画面へ遷移することもできる。

以下に、スコープ最終画面での業務処理完了時の実装例を示す。

```
/// ボタン押下時のイベント処理完了時
private void b01_02_03_C01EventProcessWorker_Completed(object sender,
                        Terasoluna.Windows.Forms.Events.EventProcCompletedEventArgs e)
{
    if (e.Result.IsSuccess)
    {
        //スコープ上の画面を全てクローズ
        FormForwardGroupUtility.CloseAll(this);
        //必要に応じてスコープ外の画面への遷移処理を実装

        //全てCloseして開始画面を再作成（new & Show）して戻る場合は、こちらを使用
        //FormForwardGroupUtility.BackToStartFormWithClose(this);
    }
    else if (e.Result.IsAnyBizLogicError || e.Result.IsAnyValidationError)
    {
        //業務エラー時には、スコープ上の画面開始画面へ戻る
        //全てHideして開始画面を再表示（Show）して戻る場合
        FormForwardGroupUtility.BackToStartFormWithHide(this);
    }
}
```

リスト 20 スコープ最終画面での業務処理（イベント処理）完了時の実装例

◆ ContentPlaceHolder を使った「画面データ」の引き継ぎ

前述の FormForwarder を使ったウィザード形式の画面や、「○○検索画面」→「○○修正画面」→「○○入力確認画面」といった一連の画面遷移がパターン化されたアプリケーションを、ContentPlaceHolder によりパネル切り替えで実装する場合、ContentPlaceHolder がインスタンス管理するパネルの「画面データ」を使用して、画面間でのデータの引き継ぎを実施する。

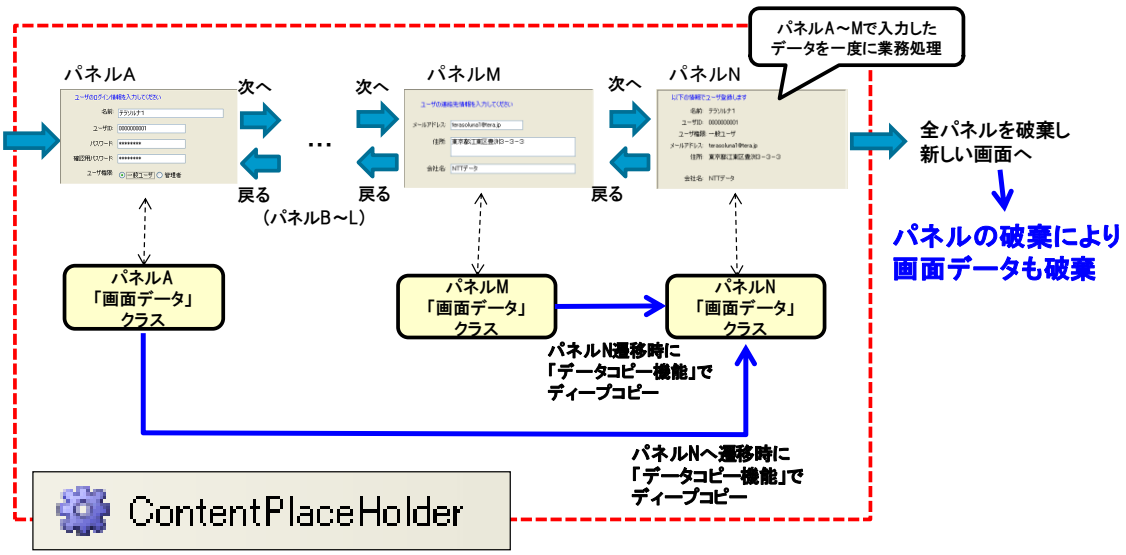

図 43 ContentPlaceHolder による画面の引き継ぎイメージ

最終パネル（パネル N）で、IContentControl.HandleShowing メソッドを使用してパネル表示する際、ContentPlaceHolder.GetContentControl メソッドで任意のパネルを取得後、パネルの「画面データ」プロパティを取得し、現在の「画面データ」へディープすることで、一連の業務の画面遷移グループで共有されるデータを取得する。
以下に、実装例を示す。

```
public void HandleShowing(object sender, ControlEventArgs e)
{
    ///ContentHolderが管理するパネルの画面データを取得し、現在のパネルの画面データへコピー
    SC_B01_03_01Control sc_B01_03_01Control =
ContentPlaceHolder.GetContentControl<SC_B01_03_01Control>();
    SC_B01_03_02Control sc_B01_03_02Control =
ContentPlaceHolder.GetContentControl<SC_B01_03_02Control>();
    DataCopyManager.Copy(sc_B01_03_01Control.ViewData, ViewData);
    DataCopyManager.Copy(sc_B01_03_02Control.ViewData, ViewData);
}
```

リスト 21 最後のパネルへ遷移時に「画面データ」を引き継ぐための実装例

また、最終パネルでは、業務処理成功時に、
ContentPlaceHolder.ShowWithCloseAllContents メソッドを使用して、インスタンス管理中の全パネルを破棄し、次のパネルへ遷移する。これにより、全ての「画面データ」も自動的に破棄される。
また、業務エラー時など、ContentPlaceHolder.ShowAndHideContent メソッドを使用して入力開始時のパネルを再表示することで、ユーザ入力状態を保持したまま、開始パネルへ遷移することもできる。
以下に、最終パネルでの業務処理完了時の実装例を示す。

```
/// 登録ボタン押下時のイベント処理完了時
private void b01_03_03_C01EventProcessWorker_Completed(object sender,
EventProcCompletedEventArgs e)
{
    if (e.Result.IsSuccess)
    {
        //業務処理成功時にContentHolder上のパネルを全てクローズし、別のパネルへ遷移
        ContentPlaceHolder.ShowWithCloseAllContents(typeof(SC_B01_03_00Control));
    }
    else if (e.Result.IsAnyBizLogicError || e.Result.IsAnyValidationError)
    {
        //業務エラー時には、入力開始時のパネルへ戻る例
        ContentPlaceHolder.ShowAndHideContent(typeof(SC_B01_03_01Control));
    }
}
```

リスト 22 最終パネルでの業務処理(イベント処理)完了時の実装例

◆ パネル切り替えにおけるヘッダー・フッター部分の実装

パネル切り替えを利用する画面では、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。なお、パネル切り替えの「画面データ」の設計の考え方については、「CL-01 画面データ機能」の「TIPS」も併せて参照すること。
以下に、ヘッダー部分にログインユーザ情報をラベルに出力する例を示す。

まず、ヘッダー部分の表示項目を保持する「画面データ(ネスト)」を取得するプロパティを業務共通インタフェースに定義し、実際の「画面データ」取得処理を各パネルに実装する。

```
public interface ISC_B01_03_01MainContent
{
・・・
    /// ログインユーザを保持する画面データ（ネスト）クラス
    /// メインのパネルの外のヘッダー領域に表示する
    SC_B01_03LoginUser LoginUser { get; }
・・・
}
```

リスト 23 業務インタフェースの記述例

```
public partial class SC_B01_03_01Control : UserControl, ISC_B01_03_01MainContent,
            IContentControl
{
    /// 画面データ
    public SC_B01_03_01ViewData ViewData
    {
      …
    }
…
    /// ログインユーザを保持する画面データ（ネスト）クラスを返却
    public SC_B01_03LoginUser LoginUser
    {
        get { return this.ViewData.LoginUser; }
    }
… }
```

リスト 24 パネルの実装例

なお、上記実装例では、各パネルの「画面データ(ルート)」に「画面データ(ネスト)」を定義する設計を想定しているが、パネル切り替え時にヘッダー情報が変わらない場合は、ContentPlaceHolder.GetContentControl メソッドを使用して、遷移元の「画面データ」を取得して返却するように実装すれば、「画面データ(ネスト)」の設計を省略することも可能である。

次に、大元の画面またはヘッダーやフッター部分を実現する画面部品に、対象の「画面データ(ネスト)」を使用して UI コントロールとのバインド設定を実施する。

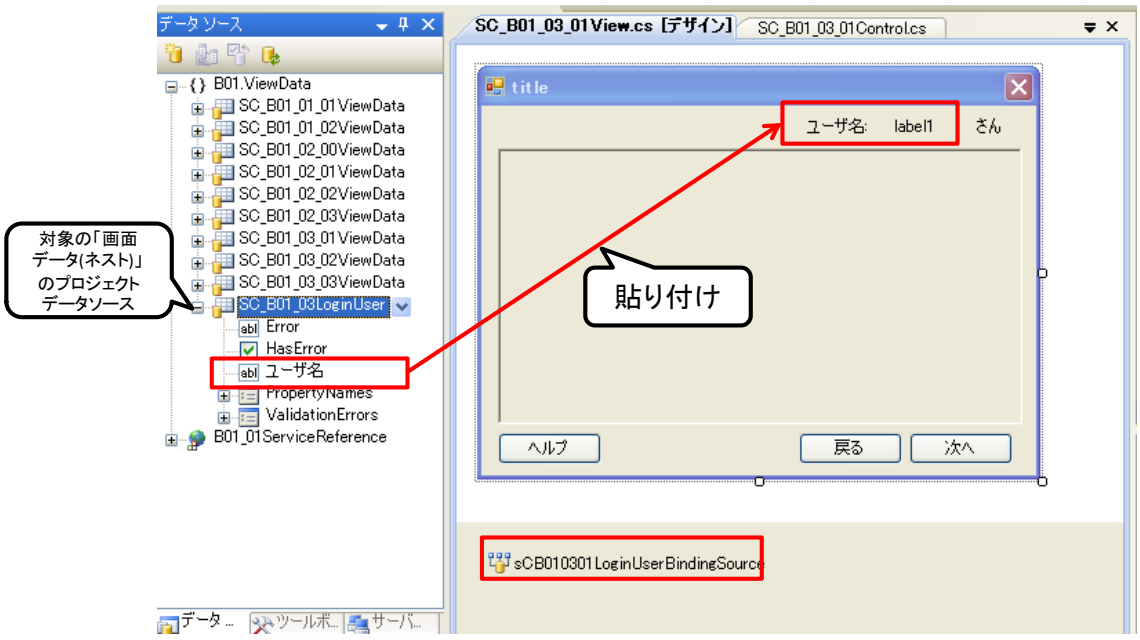

図 44 ヘッダー部分の双方向バインド設定の例

最後に、大元の画面クラスで、ContentPlaceHolder.Add イベントや Form.Load イベント[2]等で、BindingSource の DataSource プロパティにパネルに対する「画面データ」をセットする。

```
///パネルを配置する画面
[ScreenId("SC_B01_03_01")]
public partial class SC_B01_03_01View：TourSampleViewBase
{
        /// ContentPlaceHolderに、パネルが追加された時のイベント
        private void contentPlaceHolder_ControlAdded(object sender, ControlEventArgs e)
        {
            ///追加されたパネルを取得
            ISC_B01_03_01MainContent mainContent = e.Control as
ISC_B01_03_01MainContent;
            if (mainContent != null)
            {
                …
                ///ヘッダー部分とバインド設定するBindingSource.DataSourceプロパティを設定
                sCB010301LoginUserBindingSource.DataSource = mainContent.LoginUser;
            }
    }
}
```

リスト 25 パネルを配置する画面の実装例

[2] Form.Load イベントで実施する場合は、画面ロード時のみ実行されるので、ヘッダーがパネルによって切り替わらない固定的な表示をする場合に使用する。このような場合は、全てのパネルに対して「画面データ(ネスト)」を設計しなくても、最初のパネルに対する「画面データ」のみ設計すればよい。

■ 内部構成

◆ クラス図

FormForwarder による画面遷移機能の主要クラスについてクラス図を図 45 に示す。

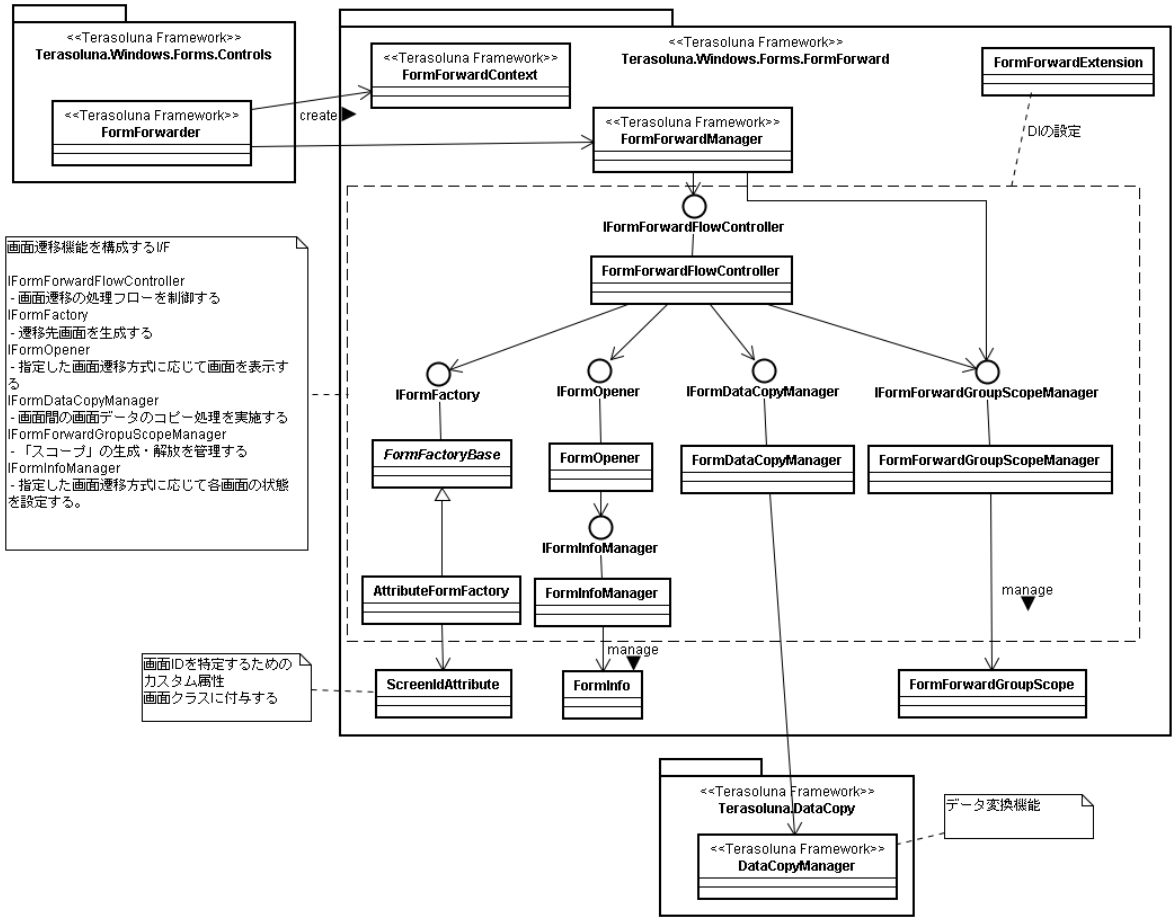

図 45 クラス図

◆ シーケンス図

FormForwarder により、遷移元画面から遷移先画面へ遷移する(順方向の画面遷移)時のシーケンス図を図 46 に示す。

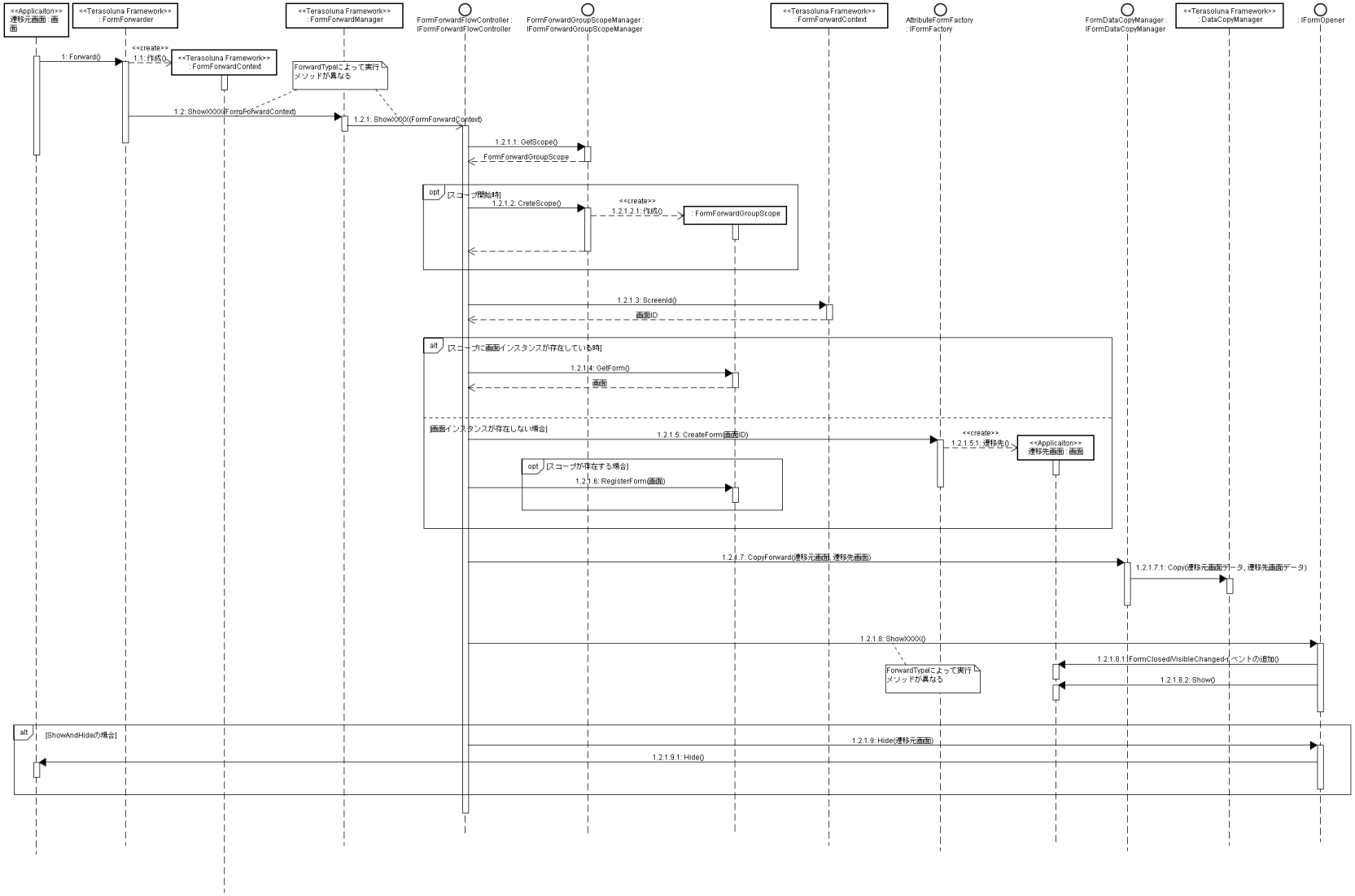

図 46 シーケンス図

◆ 構成クラス

本機能を構成するクラスを以下に示す。

表 16 構成クラス一覧

項番	クラス名	説明
Terasoluna.Windows.Forms.Controls 名前空間		
1	FormForwarder	画面切り替えによる遷移機能を提供するコンポーネント。
2	ContentPlaceHolder	パネル切り替えによる遷移機能を提供するクラス。
3	IContentControl	ContentPlaceHolder に追加可能なパネルが実装すべきインタフェース。
Terasoluna.Windows.Forms.FormForward 名前空間		
4	FormForwardManager	画面切り替えによる遷移機能のエントリクラス。FormForwarder から呼び出される。
5	FormForwardContext	画面遷移処理の設定情報を保持するクラス
6	FormForwardType 列挙体	画面表示のモーダルタイプを保持する列挙体。ForwardType プロパティで設定する。
7	ForwardFormLoadingEventArgs	FormFrowarder.FormLoading イベント時の EventArgs
8	ForwardFormClosedEventArgs	FormForwarder.FormClosed イベント時の EventArgs
9	IFormForwardFlowController	画面遷移機能の一連の処理を管理するインタフェース。
10	FormForwardFlowController	IFormForwardFlowController のデフォルト実装クラス。
11	IFormFactory	遷移先画面を生成するインタフェース。
12	FormFactoryBase	IFormFactory の基底クラス。
13	AttributeFormFactory	IFormFactory インタフェースのデフォルト実装クラス。画面クラスに付与された ScreenId 属性をもとに、遷移先画面を識別する
14	ScreenIdAttribute	画面 ID を識別するためのカスタム属性。
15	NamingRuleFormFactory	IFormFactory インタフェース実装クラス。画面クラスの命名規則ををに基づき遷移先画面クラスを識別する。
16	ViewInputAttribute	遷移元画面から遷移先画面への「画面データ」コピーに関する設定を実施するカスタム属性。
17	ViewInputUtility	ViewInput 属性を処理するためのユーティリティクラス。

18	IFormOpener	遷移先画面を表示するインタフェース。
19	FormOpener	IFormOpener インタフェースのデフォルト実装クラス。
20	FormOpenerFormClosedEventArgs	FormOpener で発生するイベントの EventArgs
21	IFormDataCopyManager	「画面データ」間のデータコピーを行うインタフェース。
22	FormDataCopyManager	IFormDataCopyManager インタフェースのデフォルト実装クラス。ForwardMapping／BackwardMapping の設定値に従って、データコピーを行う。
23	FormDataCopyManagerCopyingToViewInput	FormDataCopyManager の派生クラス。ForwardMapping に加えて、遷移先「画面データ」に付与される ViewInput 属性値に従って、データコピーを行う。
24	FormInfo	各画面で実施した画面遷移に関する情報を格納する。
25	FormState	画面遷移状態を識別する列挙体。
26	IFormInfoManager	各画面の FormInfo の取得、更新を行うインタフェース。
27	FormInfoManager	IFormInfoManager インタフェースのデフォルト実装クラス。
28	FormInfoUtility	FormInfo を扱うためのユーティリティクラス。
29	FormForwardEventReporter	FormForwarder のイベントを通知するクラス。
30	FormForwardGroupScope	画面遷移のスコープを表すクラス。
31	IFormForwardGroupScopeManager	スコープを管理するインタフェース
32	FormForwardGroupScopeManager	IFormForwardGroupScopeManager インタフェースのデフォルト実装クラス。
33	FormForwardGroupUtility	スコープを操作するためのユーティリティクラス。
34	ScreenKey	画面インスタンスを識別するキー情報を表すクラス。
35	FormForwardExtension	画面遷移機能のデフォルトの UnityContainerExtension。

■ 拡張ポイント

遷移先画面の生成や画面の表示、スコープの管理など、FormForwarder による画面遷移機能を構成するインタフェース（IFormForwardFlowController、IFormFactory、IFormOpener、IFormDataCopyManager、IFormForwardGroupScopeManager、IFormInfoManager）のデフォルト実装クラスは、FormForwardExtension クラスで DI 設定されて有効化されている。本機能を拡張する場合、各インタフェースを実装し、FormForwadExtension 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。

なお、遷移先画面を生成する機能、「画面データ」のコピーを行う機能についてには、デフォルトで設定されている実装クラスとは異なるの処理方式をとる実装クラスが提供されているため、プロジェクトの要件に応じて利用することができる。

- IFormFactory インタフェース
  - デフォルトでは AttributeFormFactory クラスが設定されている。
    AtrributeFormFactory クラスでは、画面クラスに付与された ScreenId 属性をもとに生成すべき遷移先画面を決定する。
  - 他の IFormFactory インタフェースの実装クラスとして、NamingRuleFormFactory クラスが提供されている。NamingRuleFormFactory は、画面クラスのクラス名から遷移先画面を識別しインスタンスを生成する。指定された画面 ID から画面クラスの命名規則に基づき、画面クラスを特定する。この場合、画面クラスへの ScreenId 属性の付与は不要である。
- IFormDataCopyManager インタフェース
  - デフォルトでは FormDataCopyManager クラスが設定されている。
    FormDataCopyManager では、
    FormForwarder.ForwardMapping/DestinationTargetPropertyPaths プロパティでコピー対象のプロパティパスを明記することで自動コピーする。
  - 他の IFormDataCopyManager インタフェースの実装クラスとして FormDataCopyManagerCopyingToViewInput クラスが提供されている。
    FormDataCopyManagerCopyingToViewInput は、
    遷移先の「画面データ」クラスに付与した ViewInput 属性で記述したプロパティパスをコピー対象とする。属性を利用することで、プロパティエディタに設定作業の煩雑化が解消される場合に利用するとよい。
    なお、ViewInput 属性は遷移元から遷移先へのコピー（Forward）のみで有効であり、遷移先から遷移元へのコピー（Backward）は標準の FormDataCopyManager の動作と同様である。

以下に UnityContainerExtension 継承クラスの実装例を示す。
ここでは、デフォルトで設定されている画面遷移機能を NamingRuleFormFactory、FormDataCopyManagerCopyingToViewInput に差し替えた例を示す。

```
public class SampleFormForwardExtension : UnityContainerExtension
{
    protected override void Initialize()
    {
        Container.RegisterType<FormForwardManager>(
            new ContainerControlledLifetimeManager());
        Container.RegisterType<IFormForwardFlowController, FormForwardFlowController>(
            new ContainerControlledLifetimeManager());
        //IFormFactoryの実装クラスを変更
        //FormNameTemplateプロパティに画面クラスの命名規約を指定
        //この場合、「Terasoluna.TourSample.Client.View.(画面ID)」のクラスを取得する
        Container.RegisterType<IFormFactory, NamingRuleFormFactory>(
            new ContainerControlledLifetimeManager(),
            new InjectionProperty("FormNameTemplate",
                                  "Terasoluna.TourSample.Client.View.{0}"));
        Container.RegisterType<IFormOpener, FormOpener>(
            new ContainerControlledLifetimeManager());
        Container.RegisterType<IFormInfoManager, FormInfoManager>(
            new ContainerControlledLifetimeManager());
        //IFormDataCopyManagerの実装クラスを変更
        Container.RegisterType<IFormDataCopyManager,
                    FormDataCopyManagerCopyingToViewInput>(
                new ContainerControlledLifetimeManager());
        Container.RegisterType<IFormForwardGroupScopeManager,
                    FormForwardGroupScopeManager>(
                new ContainerControlledLifetimeManager());
    }
}
```

リスト 26 UnityContainerExtension 継承クラスの実装例

また、FormForwardExtension クラスの代わりに FW 構成ファイル(TerasolunaFramework.config)に設定することも可能である。
以下に TerasolunaFramework.config の記述例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
       <extensions>
          <!-- 画面遷移機能の設定 -->
          <add type="Terasoluna.TourSample.Client.Common.FormForward,
                    TourSampleCommon" />
       </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 27 TerasolunaFramework.config の記述例

なお、FormDataCopyManagerCopyingToViewInput クラスについて、自動コピーの基本ルールは以下の通りである。

- 遷移先「画面データ」の ViewInput 属性値で指定されているプロパティはコピー対象となる。ここに指定されていないプロパティはコピーされない。
- さらに、ForwardMapping/DestinationTargetPropertyPaths プロパティが指定されている場合、ViewInput 属性で設定したプロパティパスとの積集合がコピー対象となる。
- プロパティ名や階層が異なるといったデフォルトのマッピングルールでコピーされないものについて、Mappings プロパティでマッピング定義をすることで自動コピーされる。
- その他、SourceTarget(Ignore)PropertyPaths、TargetIgnorePropertyPaths を使用して、詳細なルール設定もできる。

ViewInput 属性は、遷移先画面の「画面データ」クラスに付与し、コピー対象のプロパティパスを指定する。遷移先「画面データ」クラスの実装例を以下に示す。

```
[ViewInput("Prop1")]
[ViewInput("Prop2")]
[ViewInput("Prop3")]
public class SampleViewData : ValidatableRootViewData
{
    [DisplayName("Prop1")]
    public virtual string Prop1 { get; set; }

    [DisplayName("Prop2")]
    public virtual string Prop2 { get; set; }

    [DisplayName("Prop3")]
    public virtual string Prop3 { get; set; }
}
```

リスト 28 ViewInput 属性を付与した「画面データ」クラス

図 47 のケースを例に、ViewInput 属性によるデータコピーの挙動を示す。FowardMapping プロパティの各プロパティ設定例とコピー結果を、表 17、表 18 に示す（4 つのプロパティ名末尾の「PropertyPaths」は省略）。

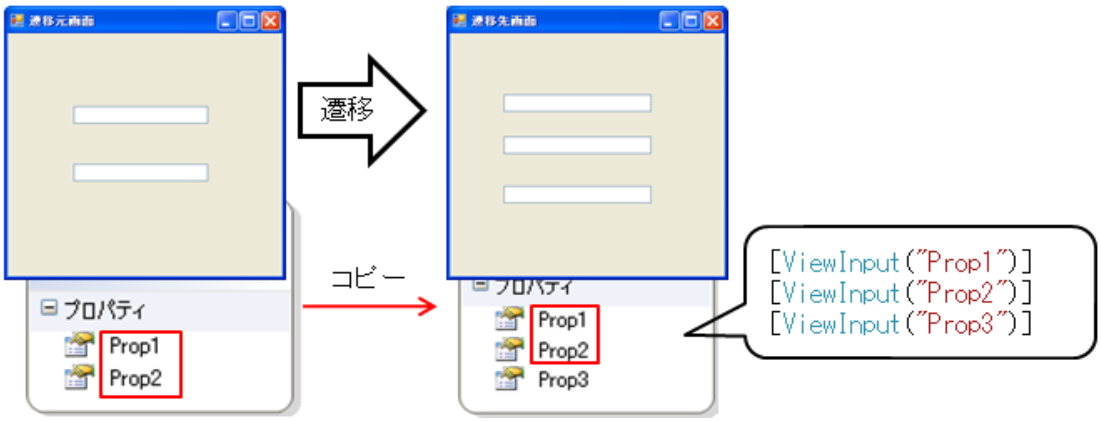

図 47 ViewInput 属性を使用した「画面データ」のコピー

表 17 ForwardMapping プロパティの設定例

項番	Mappings	Source Target	Source Ignore	Destination Target	Destination Ignore
1					
2		Prop1			
3			Prop1		
4				Prop1	
5	[Prop1]=>[Prop3]				
6	[Prop1]=>[Prop3]				Prop1

表 18 表 13 の設定における、データコピーの結果(ViewInput 属性あり)

項番	遷移先「画面データ」のプロパティ			解説
	Prop1	Prop2	Prop3	
1	Prop1	Prop2		ViewInput 属性値のものがコピー対象となる (画面遷移機能の標準ルールと異なり、DestinationTarget に指定していなくてもコピーされる)
2	Prop1			ViewInput 属性値のもののうち、SourceTarget に指定したもののみコピーされる
3		Prop2		ViewInput 属性値のものでも、Ignore に指定したものはコピーされない
4	Prop1			ViewInput 属性値のもののうち、DestinationTarget に指定したもののみコピーされる
5	Prop1	Prop2	Prop1	ViewInput 属性による自動コピーに加えて、Mappings で定義されたコピーが実行される
6		Prop2	Prop1	

また、前述の通り ViewInput 属性を用いたコピーは、遷移元画面から遷移先画面へのコピー(Forward コピー)のみ有効であり、逆(Backward コピー)は、画面遷移機能標準のコピールールと同様である。よって、遷移元画面の「画面データ」には ViewInput 属性は不要である。

■ 関連機能

- CL-01 画面データ機能
- CM-04 データコピー機能
- CM-02 インスタンス管理機能
- CM-01 アプリケーション起動・終了機能