CL-03 イベント処理実行機能

■ 概要

Windows アプリケーションにおける業務処理呼び出しの処理フローは、入力値検証、画面データ⇔ビジネスロジック入出力 DTO の変換、ビジネスロジックの実行など実行順序や処理内容は定形的である(この業務処理呼び出しの一連の流れを「イベント処理」と呼ぶ)。本機能ではこの「イベント処理」を Visual Studio のデザイナ上で簡単に設定し、実行できる仕組みを提供する。また、開発コストの高い非同期処理を容易に行えるようにする。

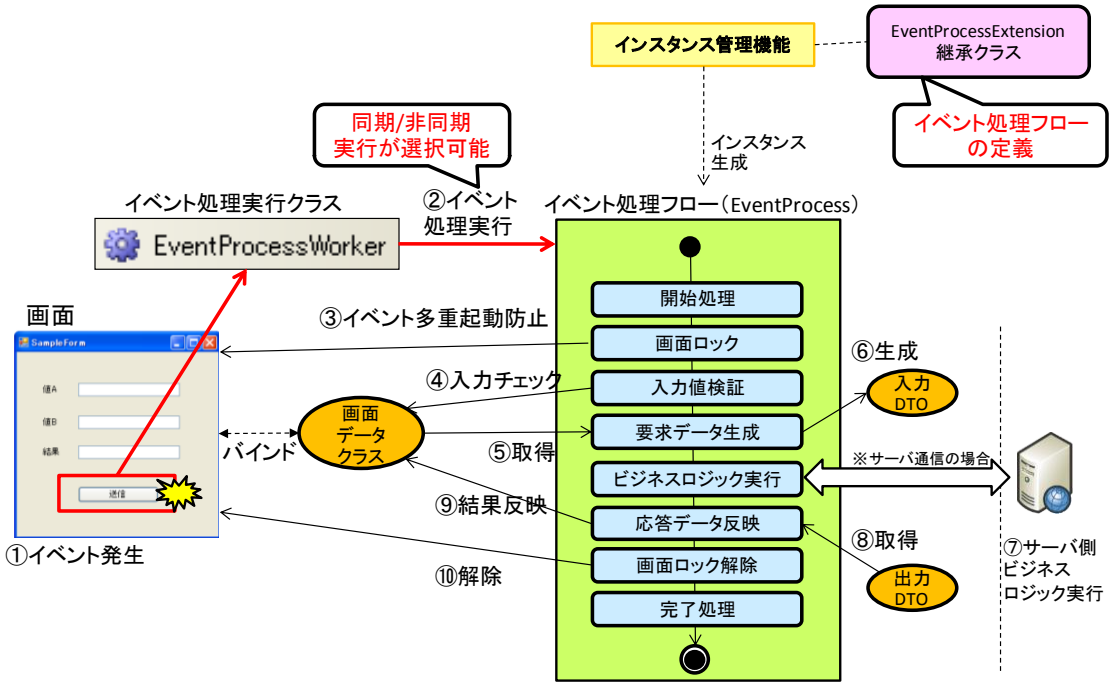


図 1 イベント処理実行機能の動作概念図

➢ イベント処理のフロー制御

図 1 のように、EventProcessWorker クラスを用いることで、入力値検証、画面データ⇔ビジネスロジック入出力 DTO)の変換、ビジネスロジック実行といった一連の「イベント処理」を実行することが可能となる。
イベント処理フローを構成する各フェーズは、以下のとおりである。

①開始処理
イベント開始時の前処理を実施する。標準機能では何も実施しない。

②画面ロック
ボタンのダブルクリックによるデータの二重登録などの誤動作を防止するため、イベント発生元の画面全体または指定した画面コントロールを無効化(Enable=false)したり、進捗ダイアログをモーダル表示したりすることで、ユーザがイベント発生元画面操作できないように自動的にする。

③入力値検証

「CL-01 画面データ機能」により実装した「画面データ」クラスを 「CM-05 入力値検証機能」により検証し、画面の入力項目に誤りがないかチェックする。
入力値検証エラー発生時には、ダイアログを表示して、イベント処理を中断する。また、「画面データ」クラスは UI コントロールと双方向バインドの設定をしているため、各画面項目に ErrorProvider を使ってリアルタイムでエラーメッセージを表示することができる。

④要求データ生成
ビジネスロジックの入力 DTO を生成し、「CM-04 データコピー機能」により一定のマッピングルールをもとに「画面データ」クラスの値を自動的に型変換しコピーする。

⑤ビジネスロジック実行
④要求データ生成で作成した入力 DTO をインプットとして、クライアントのビジネスロジックを実行する。ビジネスロジッククラスは、POCO(Plain Old CLR Object)で実装可能である。
また、「CL-04 サーバ通信機能」によりサーバビジネスロジックの実行が可能である。サーバとの通信は通常 WCF(Windows Communication Foundation)を利用する。接続先が Web サービスで提供されていれば、Visual Studio が提供する「サービス参照の追加」を利用し、WSDL(Web Services Description Language)より生成された WCF クライアントを自動的に呼び出し、サーバビジネスロジックを実行することができる。

⑥応答データ反映
「CM-04 データコピー機能」により、一定のマッピングルールをもとにビジネスロジックの出力 DTO の値を「画面データ」クラスに自動的に型変換しコピーする。このとき、「画面データ」クラスは画面クラス上の UI コントロールと双方向バインドしているため、即時に、ビジネスロジックの結果が画面へ表示される。

⑦画面ロック解除
②で実施した画面ロックを解除する。

⑧完了処理
イベント完了時の後処理を実施する。デフォルトでは、キャンセル時や業務エラー発生時のダイアログ表示などを実施する。

処理フローの各フェーズが進むにつれて、イベント(.NET の event)が発生する。イベントの発生タイミングは、図 2 の通り、イベント処理フローを構成する要素となる各フェーズの前後や進捗状況(プログレスバーなどに利用)の通知時である。
EventProcessWorker クラスは、Component 継承クラスであり、画面に貼り付け可能な部品であるため、業務開発者は各画面クラスで EventProcessWorker のイベントに対するイベントハンドラを実装することができ、確認ダイアログの表示など業務個別処理を追加することができる。

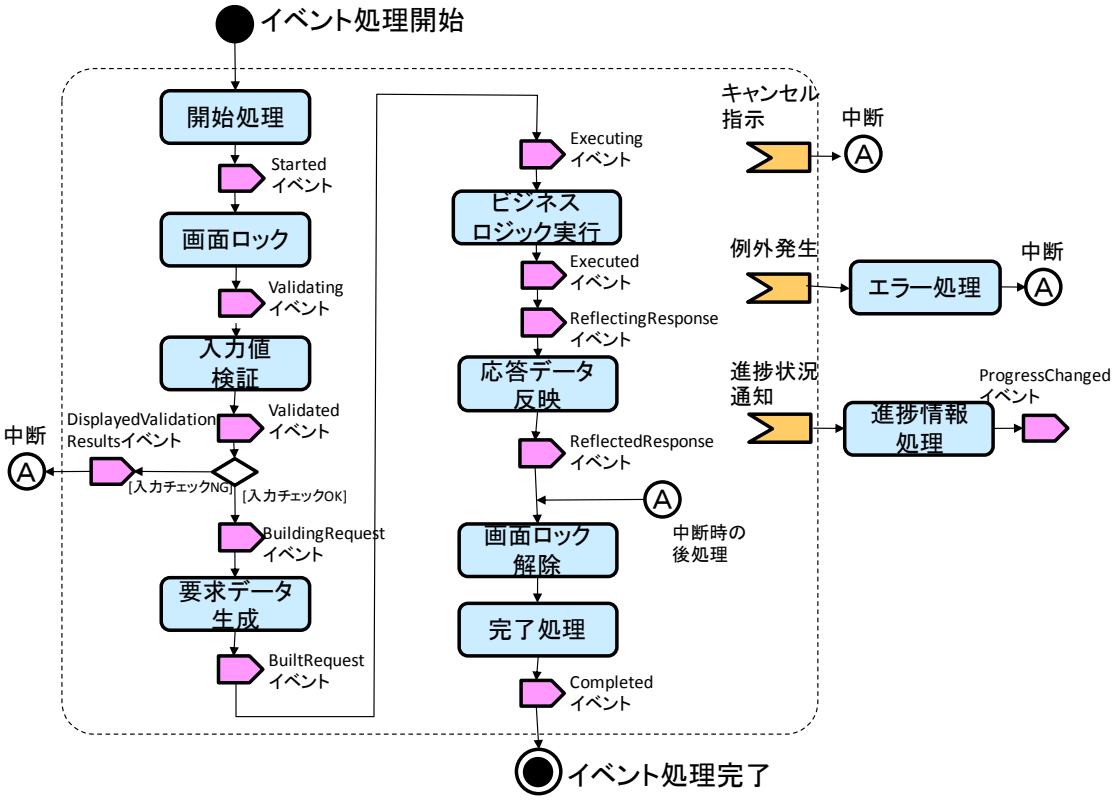


図 2 処理フローとイベント発生タイミング

➢ マルチスレッド制御

本機能は、イベント処理の同期実行と非同期実行が選択可能である。、例えば、サーバとのファイル送受信処理など長時間かかる処理についてバックグランドで実行させたいといった要求や、ユーザが画面上でボタンクリック後、バックグラウンドでビジネスロジックを実行させて処理中も画面を自由に操作できるようにしたいといった要求がある場合は、非同期で実行させる。

同期実行処理の場合はすべて UI スレッド(メインスレッド)で処理を実行するが、非同期実行処理の場合、ビジネスロジック処理を非 UI スレッド(タスクスレッド)で実行し、画面クラスのイベントハンドラメソッド等で実施する画面操作処理は、UI スレッド(メインスレッド)で実行するように実装する必要がある

通常、開発者は実行されるスレッドがどちらのスレッドであるかを意識して処理を実装しなければならないが、本機能は、「CM-03 スレッド制御機能」を共通基盤として利用しており、こういったマルチスレッドにかかわる煩雑な処理を隠蔽し、開発者が EventProcessWorker の呼び出しメソッドを切り替えるだけで、同期/非同期処理を切り替えることが可能である。

また、非同期処理実行時は、ユーザによるボタンクリックなどの契機で UI スレッドからキャンセル指示するケースがある。本機能では、ビジネスロジックを実行するスレッドと別にイベント処理フローを実行するスレッドを起動して、イベント処理フローを実行するスレッドがビジネスロジックの非同期実行処理の処理結果を取得できるまで待機する。

これにより、UI スレッドが待機することがなくなるため、画面の操作を継続することができ、ユーザがキャンセルボタンをクリックしてキャンセル指示するといった実装を実現できる。

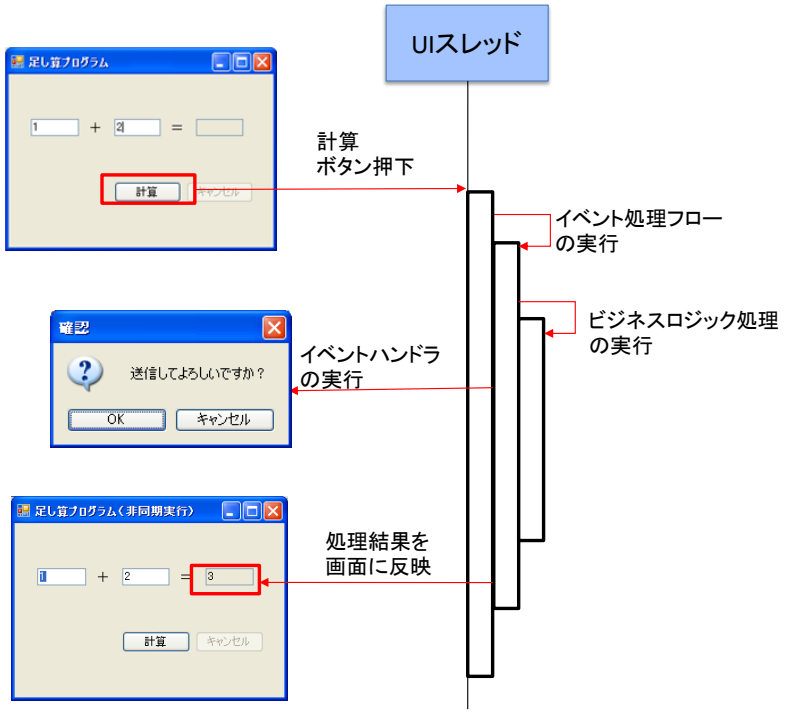


図 3 イベント処理の同期実行のイメージ

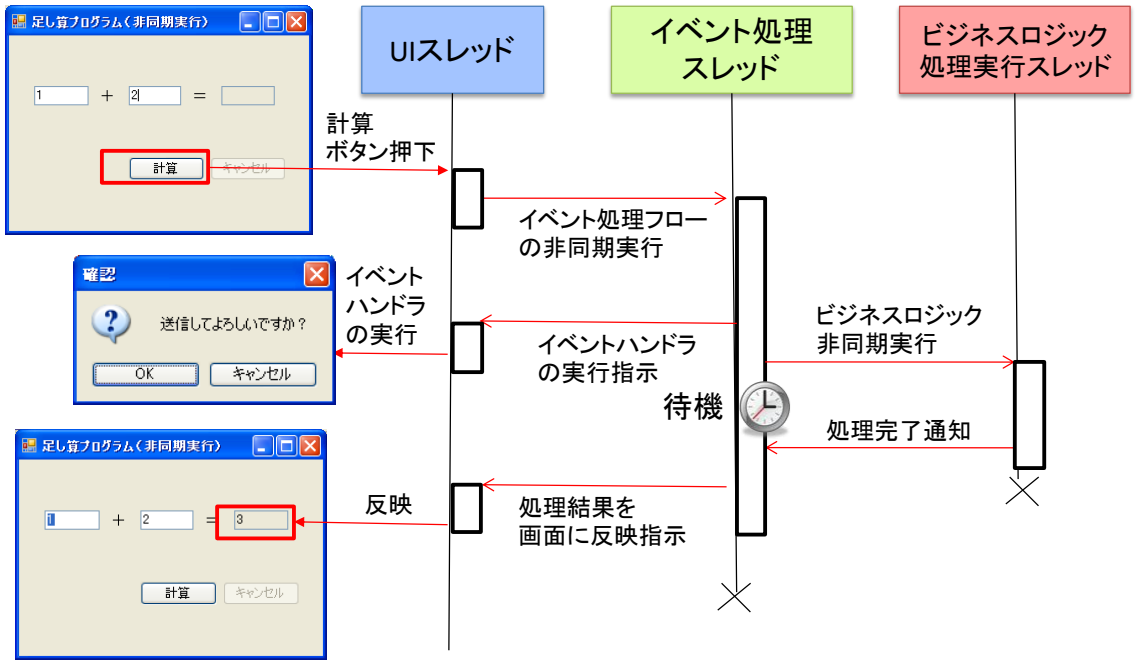


図 4 イベント処理の非同期実行のイメージ

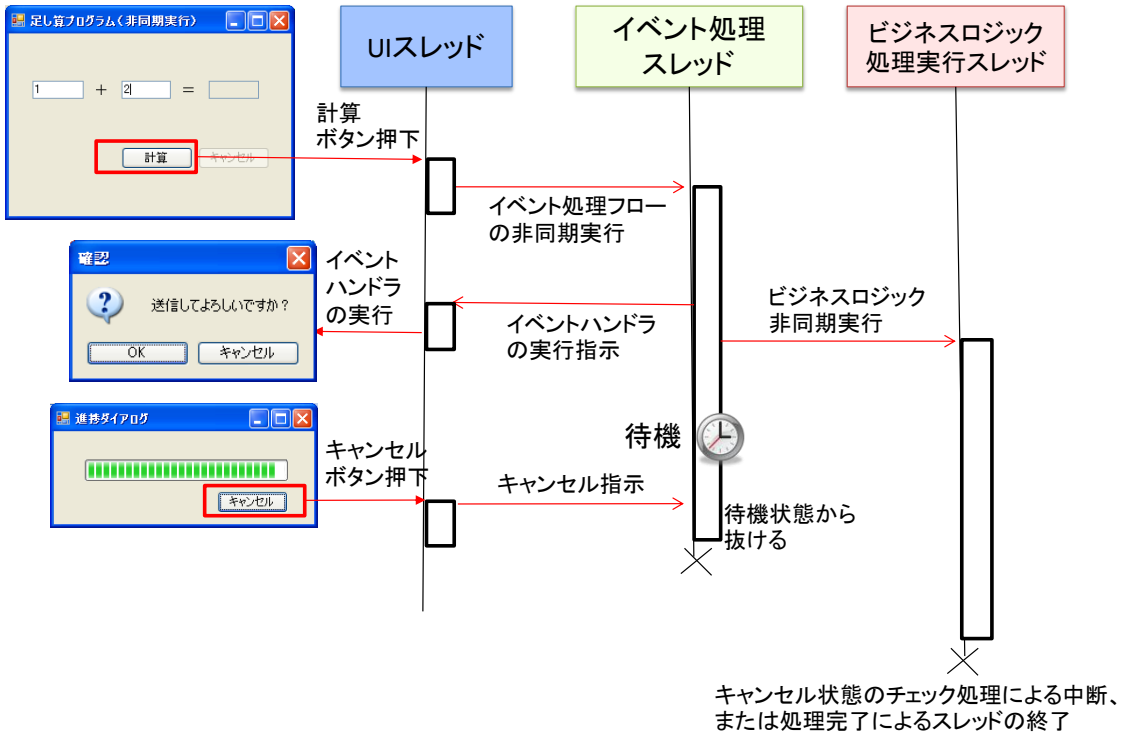


図 5 イベント処理の非同期実行時のキャンセル処理のイメージ

➢ イベント処理の進捗状況通知

本機能は、イベント処理フローが進むに従って、各フェーズの前後でイベント処理の進捗率をイベントで通知する。このとき、アーキテクトまたは業務共通開発者が、各フェーズの進捗率の範囲(各フェーズの開始時および終了時に対応する進捗率)や進捗率が進む速度(何 ms ごとに何%進捗率を増加するか)を定義することができる。これにより、イベント処理の進捗率をプログレスバーに表示するといった処理を簡単に実装できる。

また、ビジネスロジック内でイベントを通知するメソッドを定期的に呼び出すことでビジネスロジックの実行状況についてより細かい情報を通知することもできる。

➢ イベント処理フローのカスタマイズ

本機能は、典型的な処理パターンを実現するイベント処理フローを標準実装として提供しているが、開発プロジェクトの特色に合わせて、アーキテクトがイベント処理フローの構成要素を組み替えて拡張することができる。図 6 のように、イベント処理を構成する要素ごとに実装するべきインタフェースが定義されている。イベント処理フロー(EventProcess クラス)は、要素ごとにプロパティが用意されており、各種インタフェースを実装すれば容易にイベント処理の実施内容を切り替えることができる。イベント処理フローの定義は、UnityContainerExtension の継承クラスである EventProcessExtension を継承して実装する。アーキテクトが Extension クラスに登録しておいたイベント処理フローは、業務開発者が Visual Studio を使ってイベント処理を実装する際に、EventProcessWorker のプロパティで設定可能なイベント処理名としてプルダウン表示されて選択できるようになる。

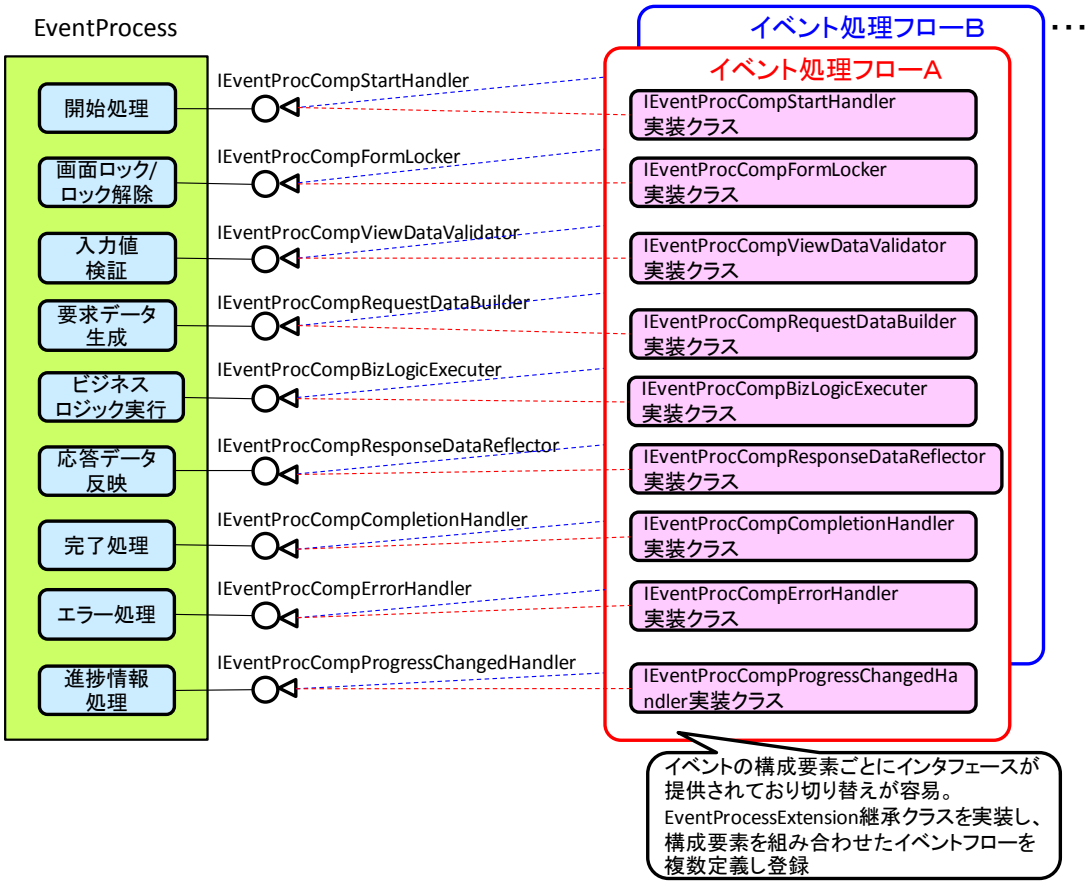


図 6 イベント処理フローの定義

➢ ビジネスロジッククラスのインスタンス生成管理

ビジネスロジック実行フェーズにおいて「CM-02 インスタンス管理機能」によりインスタンス生成する。これにより、ビジネスロジッククラスから UnityContainer で管理している各クラスのインスタンスを自由に取得したり、DI で外部からオブジェクトを注入したりすることができる。

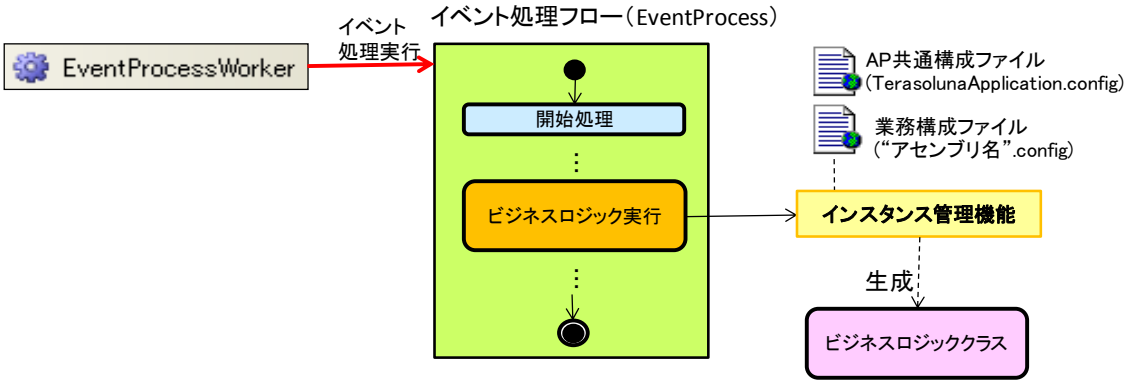


図 7 ビジネスロジックのインスタンス生成管理

➢ Visual Studio における入力支援

EventProcessWorker は、イベント処理の実行に必要な設定を Visual Studio のプロパティエディタで設定する。
本機能はデザイナ機能を拡張し、プロパティに設定する値の候補を表示し選択できるようにするなど、プロパティ設定作業の負担や入力ミスを防ぐために以下のような入力支援機能を提供している。

- TERASOLUNA フレームワークにより標準提供されたイベント処理フロー(EventProcessName プロパティ)や進捗率の計算方法(ProgressSetName プロパティ)の選択可能な値を、プルダウン表示する。また、アーキテクトが定義を追加したものを表示／選択することも可能である。
- 画面データ⇔ビジネスロジック入出力データ(DTO)のコピーに関して、マッピング設定等の入力が容易なエディタを提供する。
- 実行するビジネスロジッククラスや WCF クライアントについて、プロジェクト内および参照されるアセンブリ内のクラスを候補としてツリービュー表示し、選択可能とする。

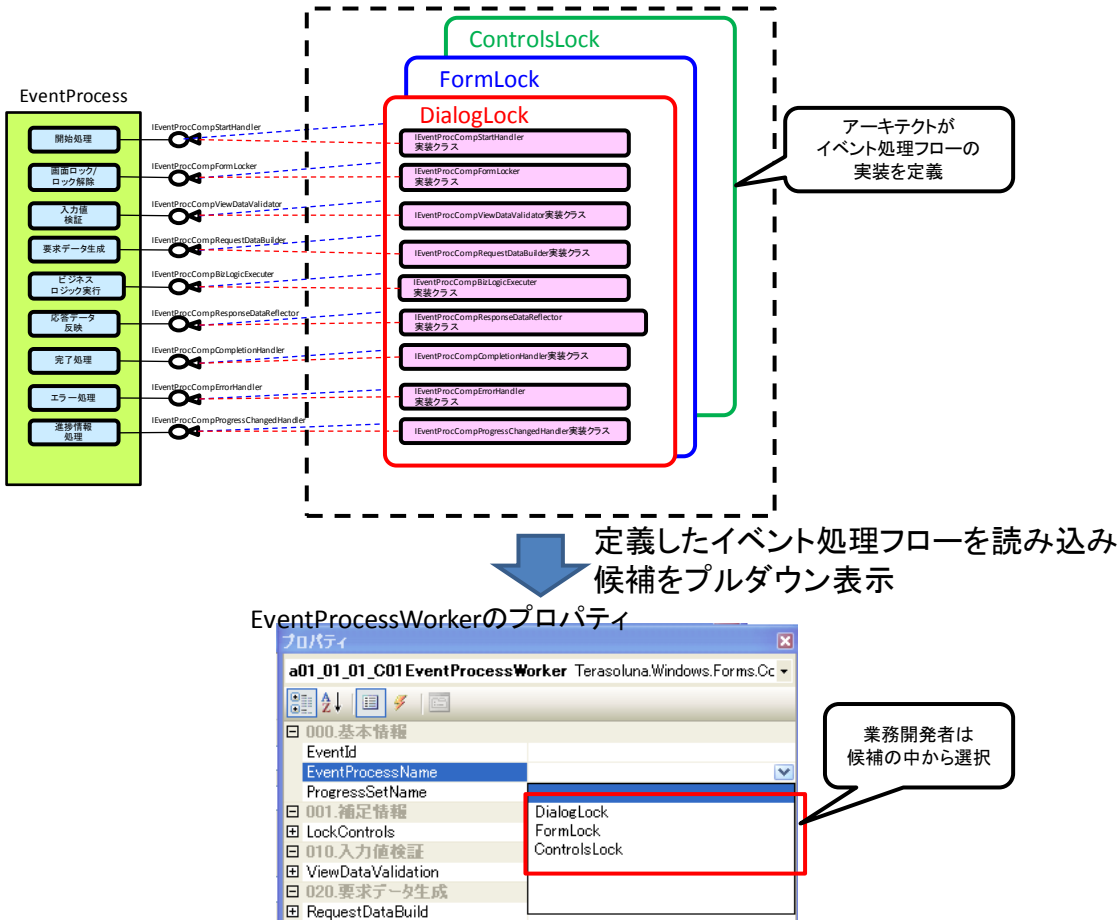


図 8 Visual Studio の入力支援機能の一例(イベント処理フローの設定)

■ 使用方法

◆ 概要

EventProcessWorker 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。このため、Visual Studio のデザイナ上でイベント処理を呼び出すための設定が可能である。

画面クラスのボタンクリック等のイベントハンドラ内から EventProcessWorker を実行することで、EventProcesWorker のプロパティで設定した定義に基づき、処理が順番に実行され、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。

業務開発者がイベント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 EventProcessWorker をツールボックスから画面へ貼り付ける。
④ EventProcessWorker のプロパティを設定する
⑤ EventProcessWorker のイベント処理を実装する。
⑥ ボタンクリックなどのイベントハンドラ内から、EventProccessWorker の RunWorker メソッド（または RunWorkerAsync メソッド）を呼び出す。

以下に、EventProcessWorker を利用したイベント処理の実装手順を示す。

◆ 実装方法

① 「画面データ」クラスの作成

「画面データ」クラスは、画面の入力値等を保持するクラスであり、入力値検証や DTO とのデータコピーに利用される。

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利用することで、「画面データ」クラスのひな形が生成される。

「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アイテ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、開発者の実装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、とくに本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。

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 EventProcessWorker の追加

EventProcessWorker をツールボックスから、対象の画面へドラッグ＆ドロップする。

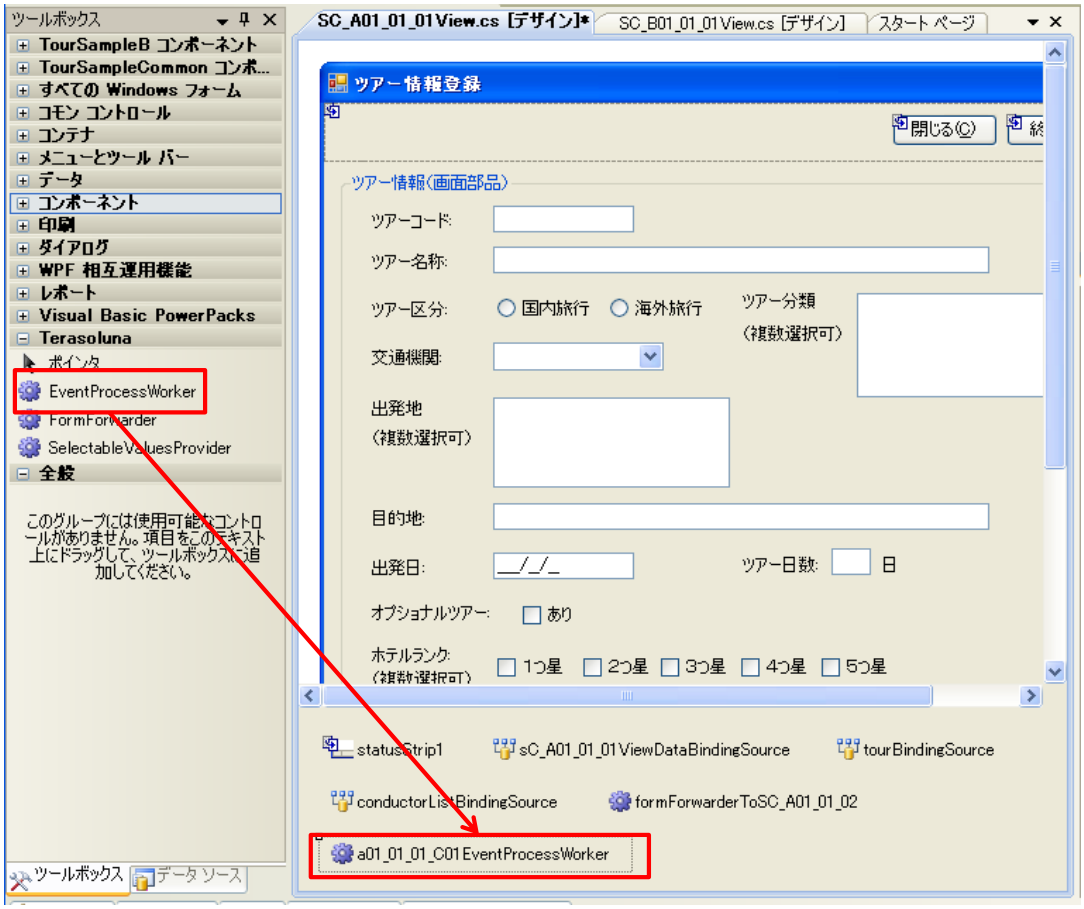


図 9　EventProcessWorker の追加

④ EventProcessWorker のプロパティ設定

EventProcessWorker のプロパティには、イベント処理の各フェーズで実施する処理をする。設定するプロパティの詳細については、後述の「EventProcessWorker のプロパティ」を参照のこと。

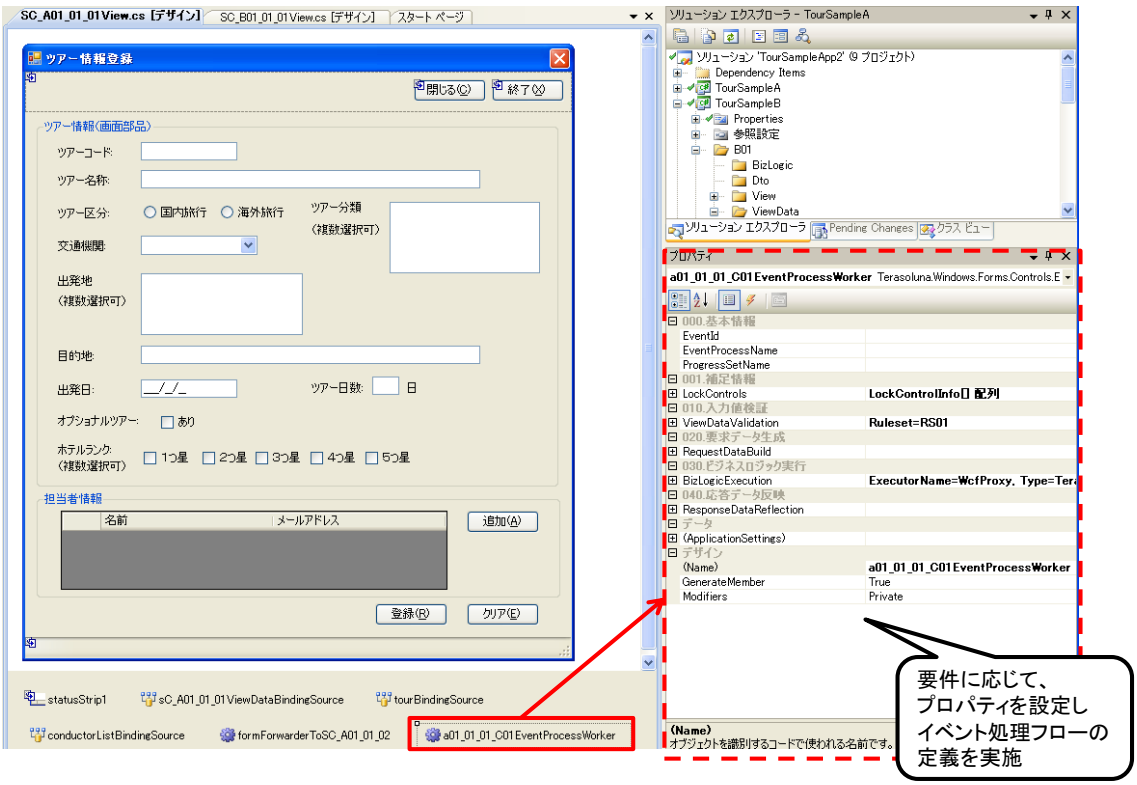


図 10　EventProcessWorker のプロパティ設定

⑤ EventProcessWorker のイベントハンドラの実装

イベント処理フローの各フェーズの処理前後やエラー発生時などで、定期的に各種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「EventProcessWorker のイベント」を参照のこと。

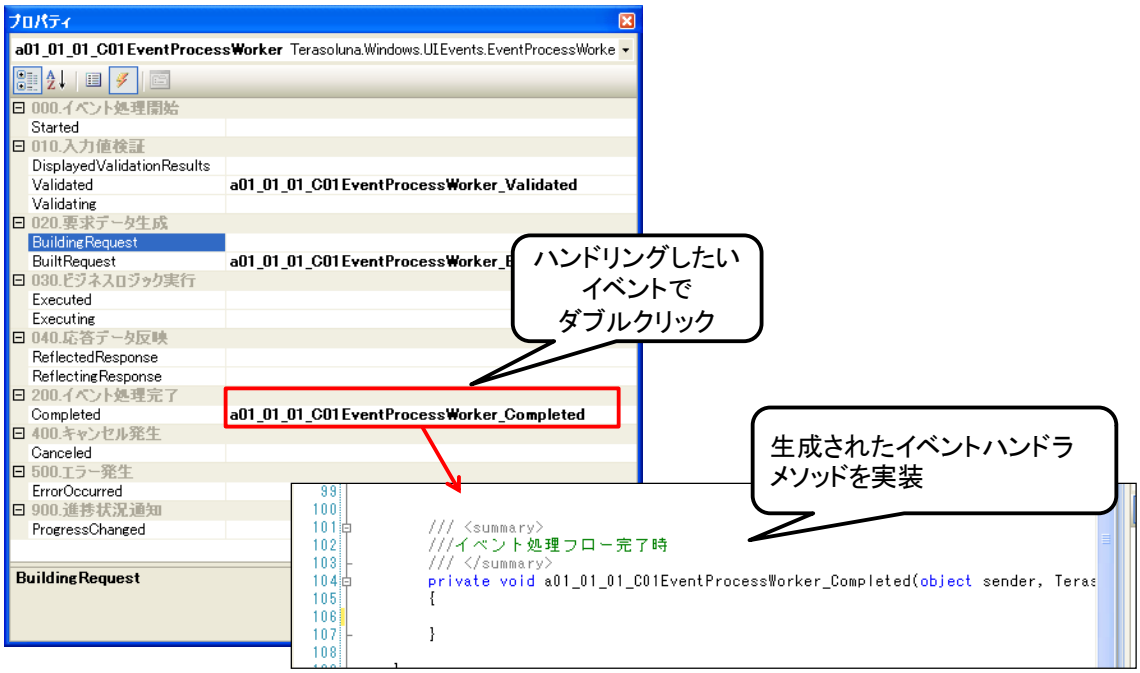


図 11 EventProcessWorker のイベントハンドラ実装

⑥ EventProcessWorker のイベント処理開始メソッドの実行

ボタンクリックのイベントハンドラなどから、EventProcessWorker の RunWorker メソッド、または RunWorkerAsync メソッドを実行することで、プロパティ設定したとおりにイベント処理フローが実行される。

イベント処理を同期実行したい場合は、RunWorker メソッドを実行する。非同期で実行したい場合は、RunWorkerAsync メソッドを実行する。
RunWorker メソッドを実行した場合、戻り値にイベント処理の実行結果(EventProcessResult オブジェクト)が格納される。以下に RunWorker メソッドの実行例を示す。

```
public partial class SC_B01_01_01View : Form
{
・・・
        private void loginButton_Click(object sender, EventArgs e)
        {
            ///イベント処理実行
            EventProcessResult result =   b01_01_01_C01EventProcessWorker.RunWorker();

            if (result.IsSuccess)
            {
                ///成功時は、FormForwarderを使って画面遷移
                formForwarderToSC_B01_01_02.Forward();
            }
        }
}
```

リスト 3 イベント処理の同期実行の記述例（RunWorker メソッド）

また、RunWorkerAsync メソッドを実行した場合、メソッドの戻り値はイベント処理の実行結果を表すものではなく、イベント処理の非同期実行を正常に受け付けたかどうかを表す bool 値になる。通常、戻り値は true を返すが、すでに EventProcessWorker がイベント処理実行中の場合はイベント処理を受け付けることができないため false を返す。
イベント処理の実行結果は、EventProcessWorker の Completed イベントのイベントハンドラを実装し、メソッド引数である EventProcCompletedEventArgs オブジェクトの Result プロパティより取得する。以下に、RunWorkerAsync メソッドの実行例を示す。

なお、EventProcessResult の値については、後述の「イベント処理結果の取得」を参照のこと。

```
public partial class SC_A01_01_01View : Form
{
・・・
    private void registTourButton_Click(object sender, EventArgs e)
    {
            ///イベント処理実行
           a01_01_01_C01EventProcessWorker.RunWorkerAsync();
    }
・・・

    /// <summary>
    ///イベント処理フロー完了時
    /// </summary>
    private void a01_01_01_C01EventProcessWorker_Completed(object sender,
                                    Terasoluna.Windows.UI.Events.EventProcCompletedEventArgs e)
    {
        if (e.Result.IsSuccess)
        {
            ///成功時は、MessageBoxを表示
            MessageBox.Show(this, ResourceCommon.INFO_COMMON_0001);
        }
    }
}
```

リスト 4 イベント処理の非同期実行の記述例（RunWorkerAsync メソッド）

◆ EventProcessWorker のプロパティ

EventProcessWorker で定義されているプロパティの一覧を示す。

表 1　EventProcessWorker のプロパティ概要

項番	項目名	説明	関連する処理フェーズ
1	EventProcessName	実行したいイベント処理フローを、事前に定義したイベントフローパターンの中から、要件に合わせて選択する。	イベント処理全般
2	LockControls	イベント実行時にロックしたいボタン等の UI コントロールを指定する。EventProcessName を「ControlsLock」で選択した場合にのみ有効となる。	画面ロック
3	ProgressSetName	プログレスバーに通知するイベント処理の進捗率の計算方法を設定する。	イベント処理全般
4	ViewDataValidation	入力値検証処理の設定をする。	入力値検証
5	RequestDataBuild	「画面データ」クラスから入力 DTO（ビジネスロジックの入力データクラス（要求データ））へのデータコピー時のマッピング定義を設定する。	要求データ生成
6	BizLogicExecution	実行したいビジネスロジッククラスおよびメソッドを指定する。クライアント処理を伴わないサーバ通信の場合には、WCF プロキシのクラスおよびメソッドを指定する。	ビジネスロジック実行
7	ResponseDataReflection	出力 DTO(ビジネスロジックの出力データクラス（応答データ）)から「画面データ」クラスへのデータコピー時のマッピング定義を設定する。	応答データ反映

各プロパティの詳細な設定方法について、以下で説明する。

✧ イベント処理フローの種別設定（EventProcessName プロパティ）

EventProcessName プロパティには、事前定義されたイベント処理フローの名前がプルダウン表示されるので、業務開発者は要件にあったものを選択する。

デフォルトでは、画面ロックの方式ごとに3種類のイベント処理フローが提供されているが、アーキテクトまたは業務共通開発者は、プロジェクトの特性に合わせてイベント処理フローの構成要素を組み替えて、イベント処理フローのパターンを Extension クラスに追加定義することができる。

デフォルト提供さている EventProcessName の設定値を表 2 に示す。また、各設定値について画面ロック時の業務画面イメージを示す。

表 2 標準機能における EventProcessName プロパティの設定可能値

項番	設定値	説明	デフォルト値
1	ControlsLock	指定した UI コントロールのみを無効化するためのイベント処理フロー。LockControls プロパティ（後述の説明参照）で指定したイベント発生元画面上の UI コントロールを無効化する。 非同期実行(RunWorkerAsync メソッド)時に、イベント発生元画面上のキャンセルボタンのみを有効化するといった細かい制御や、イベント発生元画面を自由に操作できるなど操作性の高い画面の作成が可能である。	○
2	DialogLock	非同期実行（RunWorkerAsync メソッド）専用のイベント処理フロー。 ダイアログ上には、進捗率計算／通知機能が組み込まれたプログレスバーとイベント処理のキャンセル指示（ユーザキャンセル）が可能なキャンセルボタンが標準で提供されている。	
3	FormLock	イベント発生元画面を無効(Enable=False)にして画面をロックするイベント処理フロー。同期実行（RunWorker メソッド）で利用する。	

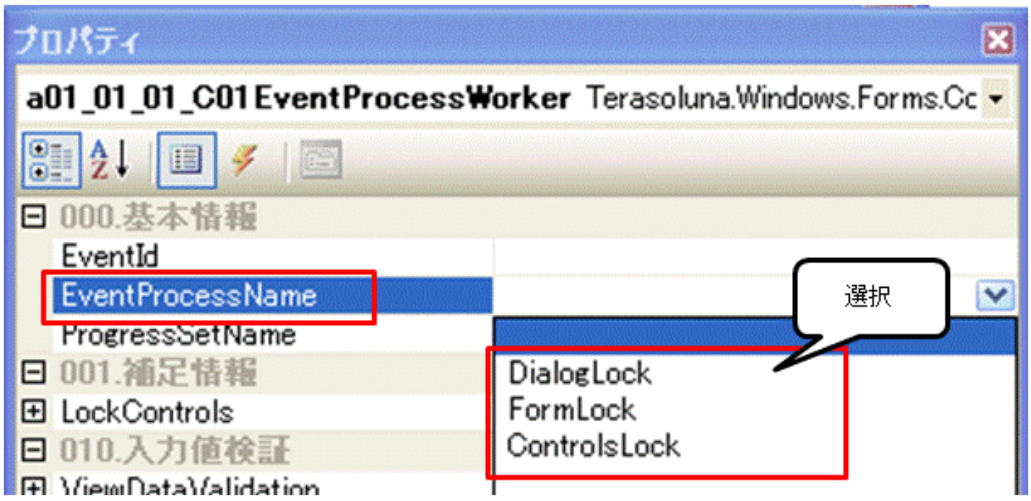


図 12 EventProcessName プロパティの設定手順

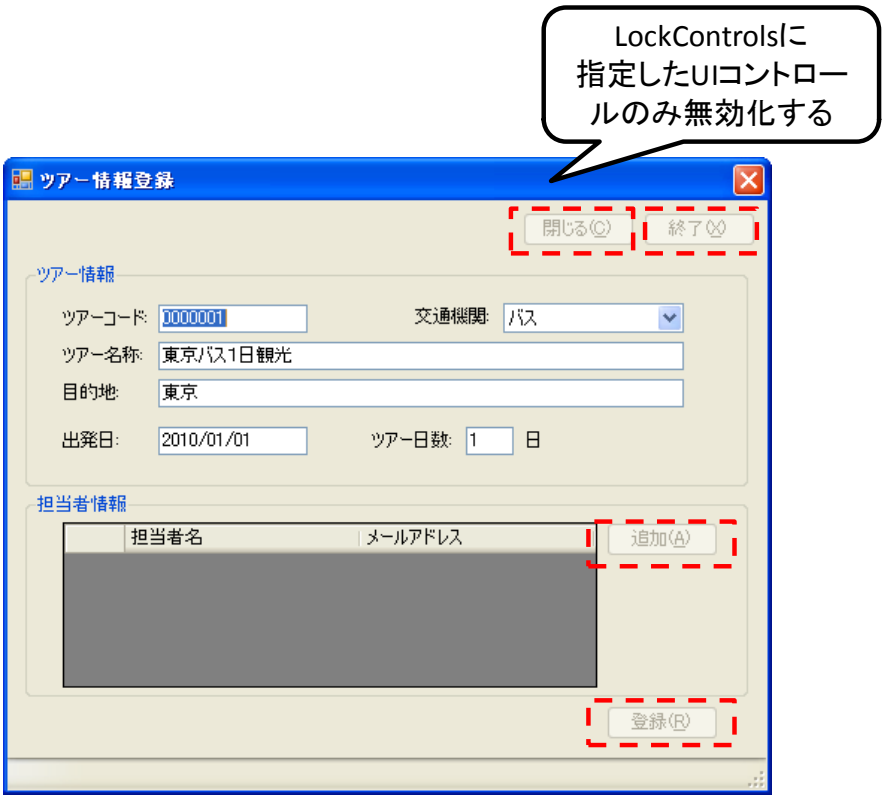


図 13 ControlsLock の画面ロック状態のイメージ

図 14 DialogLock の画面ロック状態のイメージ

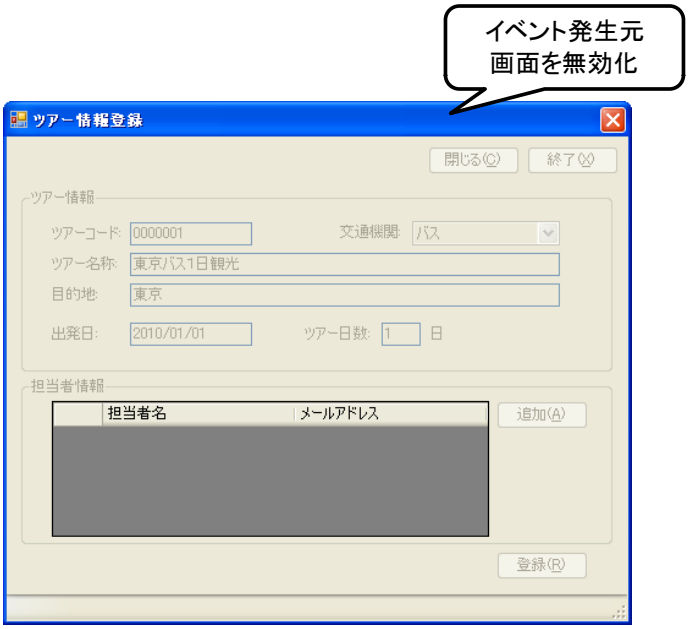


図 15 FormLock の画面ロック状態のイメージ

また、EventProcessName プロパティの値として、「ControlsLock」を選択した場合は、LockControls プロパティにイベント処理実行中に無効化(Enable=false)したい UI コントロールのプロパティ名を列挙する。

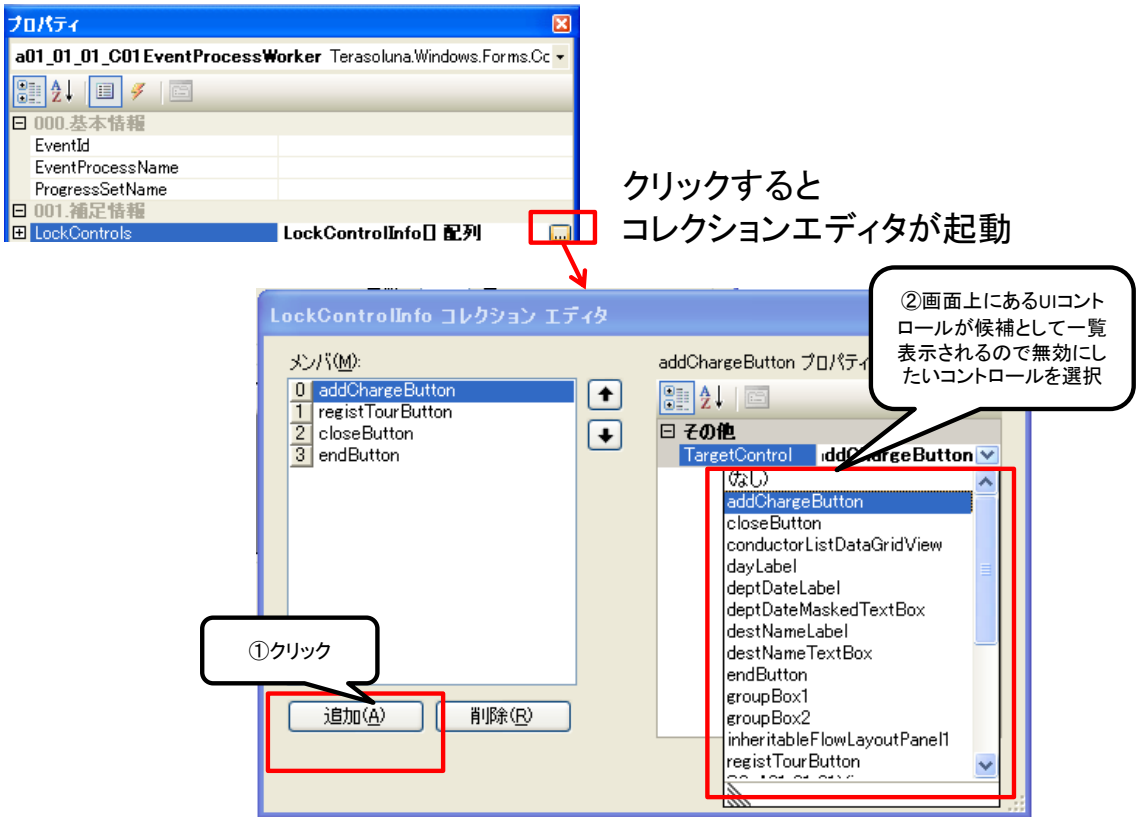


図 16 LockControls プロパティの設定手順

✧ 進捗率の計算方法の設定(ProgressSetName プロパティ)

イベント処理が進むと本プロパティの設定をもとに進捗率を自動計算し、プログレスバーに表示することができる。

進捗率の計算方法は、TERASOLUNA フレームワークにより標準提供されているが、イベント処理の Extension クラスで追加定義することも可能であるできる。

進捗率の定義としては、イベント処理フローの各フェーズの進捗率の範囲(フェーズ開始時と完了時に対応する進捗率)、進捗率を上げる速度(何 ms ごとに何%進捗率を増加させるか)などがある。

業務開発者は、ProgressSetName プロパティに、事前に定義された進捗率の計算方法の一覧がプルダウン表示されるので、業務要件に合ったものを選択する。

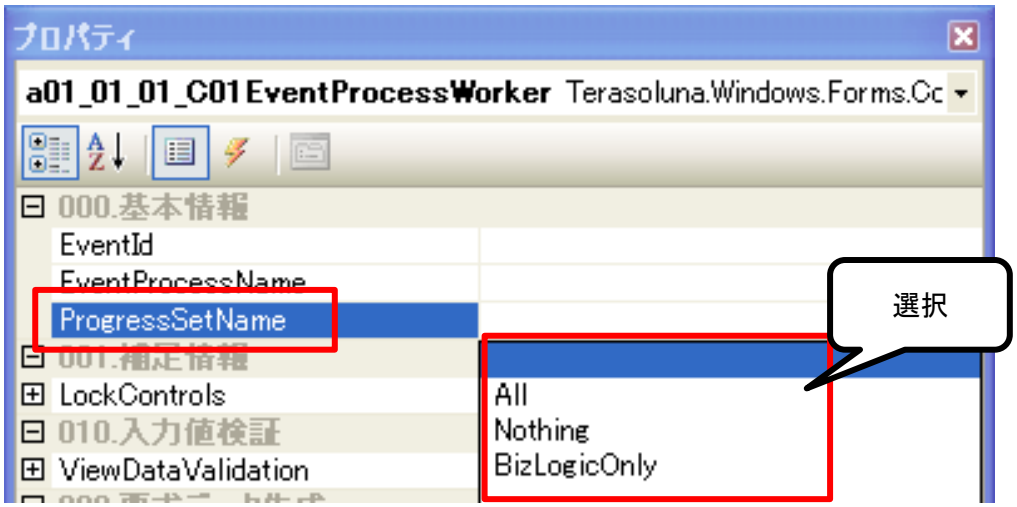


図 17 ProgressSetName プロパティの設定手順

標準機能における ProgressSetName の値を以下に示す。

表 3 標準機能における ProgressSetName プロパティの設定可能値

項番	設定値	説明	デフォルト値
1	Nothing	進捗率を自動計算しない。	○
2	All	イベント処理の初めから終わりまでの進捗率を計算する。 各フェーズの進捗率の範囲は以下の通りである。 ・画面ロック(0～10%) ・入力チェック(10～20%) ・要求データ生成(20～30%) ・ビジネスロジック実行(30～90%) ビジネスロジック実行中は、100ms ごとに 1%ずつ進捗率が上がり、範囲の限界値に達すると次のフェーズに移るまでその値で止まる。 ・応答データ反映(90～100%)	
3	BizLogicOnly	ビジネスロジック実行フェーズのみの進捗率を計算する。 ・ビジネスロジック実行(0～100%) 100ms ごとに 1%ずつ進捗率が上がり、100%に達すると止まる。	

参考に、ProgressSetName プロパティが「All」の場合の、プログレスバーの動作イメージを示す。

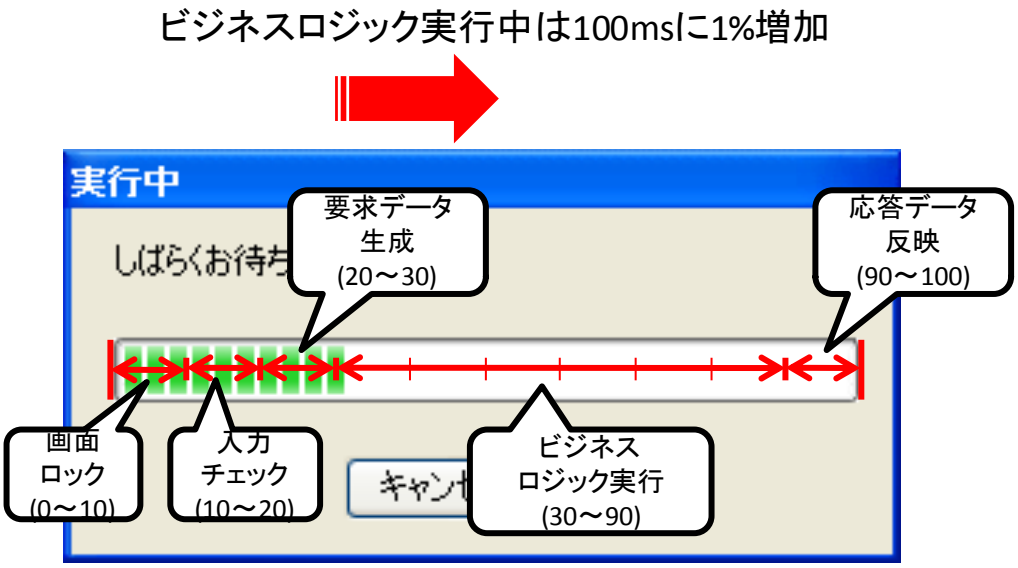


図 18　設定値 Default を設定した場合のプログレスバーの動作イメージ

なお、フレームワークで自動計算する進捗率は、実際のビジネスロジックの進捗率ではない。もし、実際のイベント処理フローの実行状況に合わせて厳密に進捗率を計算し通知したい場合は、イベント処理(ビジネスロジック)の中で進捗状況を通知する処理を実装する必要がある。実装方法については、後述の「ビジネスロジックでの厳密な進捗率」を参照のこと。

✧ 入力値検証処理の設定(ViewDataValidation プロパティ)

入力値検証処理のフェーズでは、「CL-01 画面データ機能」を利用して作成した「画面データ」を、「CM-05 入力値検証機能」により提供されるバリデータで入力項目に誤りがないかチェックする。
ViewDataValidation プロパティでは、イベント処理時にどの入力値検証処理を実施するかルールセット名を設定する。ViewDataValidation プロパティの詳細設定項目およびプロパティの設定手順について、表 4、図 19 に示す。

なお、「画面データ」クラスの作成方法については、「CL-01 画面データ機能」を、「画面データ」クラスに実装する入力値検証処理の作成方法については「CM-05 入力値検証機能」をそれぞれ参照のこと。

表 4 標準機能における ViewDataValidation プロパティの詳細設定項目

項番	プロパティ名	説明
1	ViewDataValidation /RuleSet	イベント処理時に実行するルールセット名を設定する。 未設定時は、入力値検証処理を実施しない。 ルールセット名については、「CM-05 入力値検証機能」の機能説明書を参照。

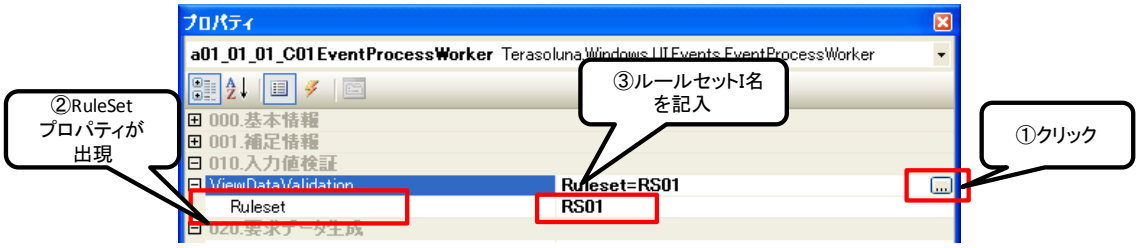


図 19 ViewDataValidation/RuleSet プロパティの設定手順

✧ 要求データ生成の設定(RequestDataBuild プロパティ)

要求データ生成のフェーズでは、「CM-04 データコピー機能」により一定のマッピングルールをもとに「画面データ」クラスの値を自動的に型変換し、入力 DTO へコピーする。標準のマッピングルールに基づき自動コピーすればよい場合は、設定不要である。
標準のマッピングルールでは対応できない個別のマッピングルールがあれば RequestDataBuild プロパティで設定する。RequestDataBuild プロパティの詳細設定項目とプロパティエディタでの設定手順について、以下に示す。
なお、RequestDataBuild/Mapping プロパティで設定する個別のマッピングルールは、「CM-04 データコピー機能」の MappingInfo クラスの API に基づいている。
MappingInfo クラスの詳細は、「CM-04 データコピー機能」の機能説明書を参照のこと。

表 5 標準機能における RequestDataBuild プロパティの詳細設定項目

項番	プロパティ名	説明
-	RequestDataBuild/Mapping	-
1	Mappings	「CM-04 データコピー機能」の標準ルールにより「画面データ」クラスのプロパティからビジネスロジックの入力 DTO のプロパティへ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	DestinationTargetPropertyPaths	自動コピー対象とする、コピー先のビジネスロジッククラスの入力 DTO のプロパティパスを設定する。
5	DestinationTargetPropertyPaths	自動コピー対象外とする、コピー先のビジネスロジッククラスの入力 DTO データクラスのプロパティパスを設定する。

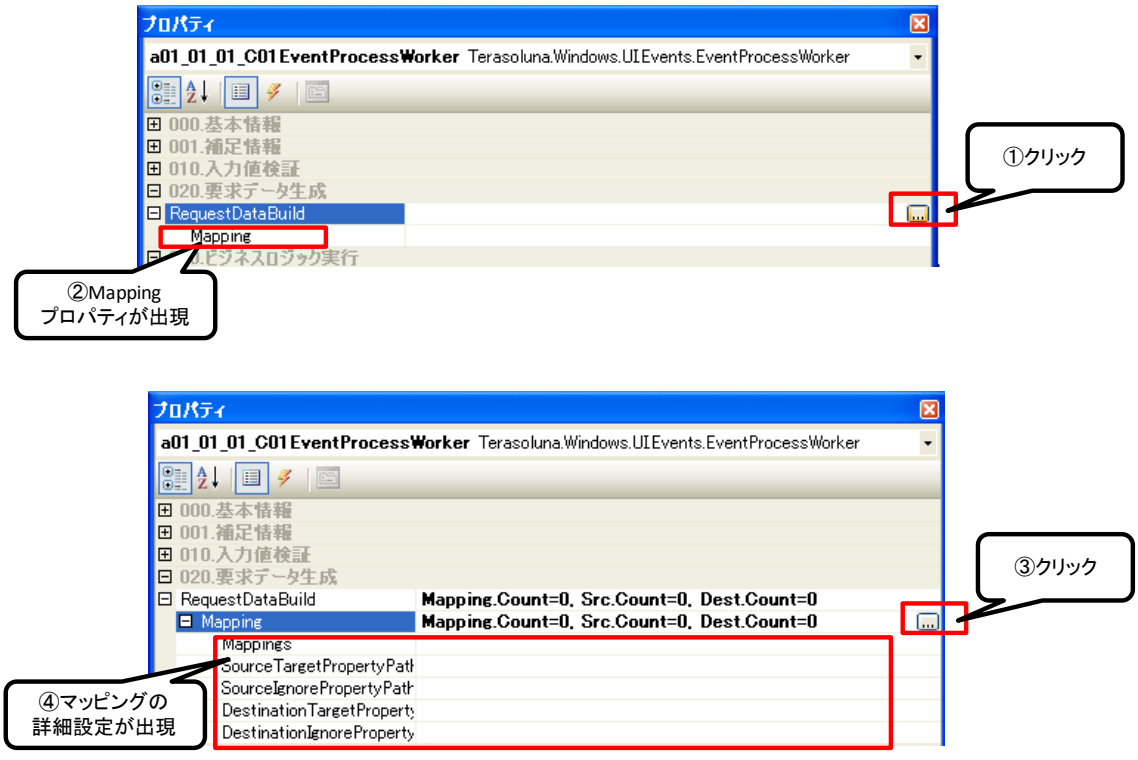


図 20 RequestDataBuild/Mapping プロパティの設定手順

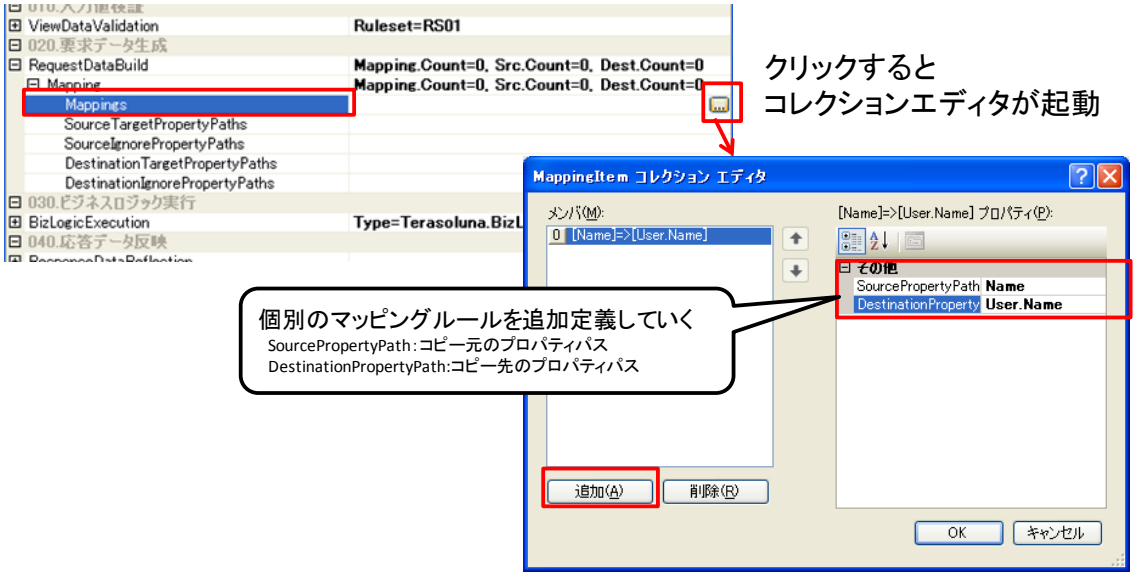


図 21 RequestDataBuild/Mapping/Mappings プロパティの設定手順

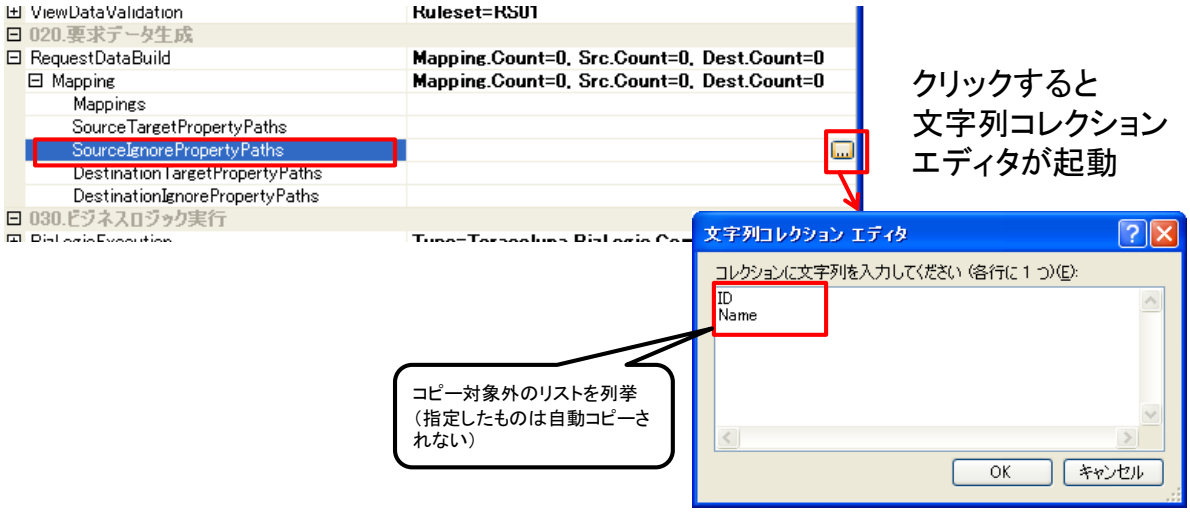


図 22 RequestDataBuild/Mapping/SourceIgnorePropertyPaths プロパティの設定手順

(RequestDataBuild/Mapping/DestinationIgnorePropertyPaths プロパティも同様)

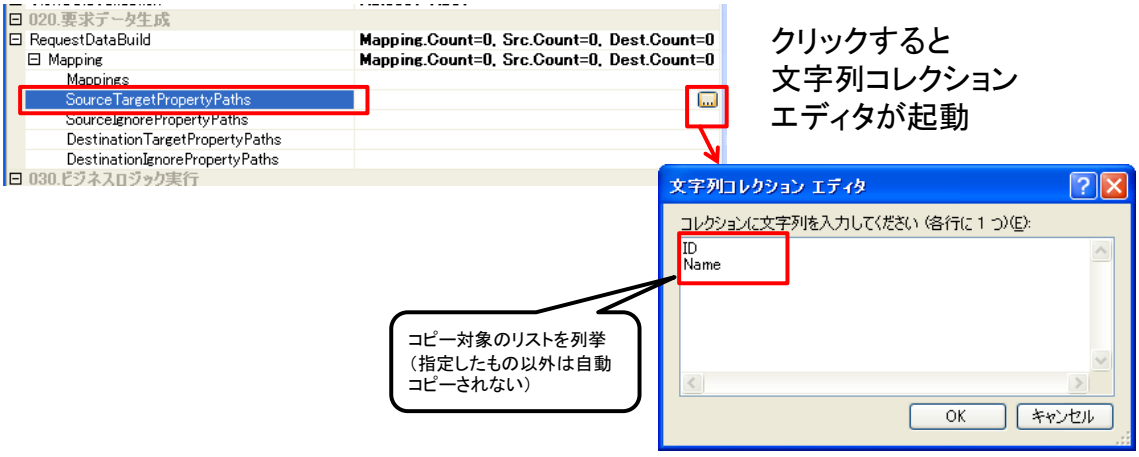


図 23 RequestDataBuild/Mapping/SourceTargetPropertyPaths プロパティの設定手順

(RequestDataBuild/Mapping/DestinationTargetPropertyPaths プロパティも同様)

なお、コピー処理のルールとして、コピー先となる入力 DTO には「既定の（引数をとらない）コンストラクタ」が必須であることに注意する。これは、TERASOLUNA フレームワークが、イベント処理フローにおいて、入力 DTO インスタンスを自動生成しているためである。既定のコンストラクタは、コンストラクタが 1 つも定義されないクラスの場合はコンパイル時に自動生成されるが、コンストラクタを持つ定義が 1 つでもあるクラスの場合は自動生成されない。よって、引数を含むコンストラクタが含まれる DTO をコピー先に指定する場合、既定のコンストラクタを明示的に定義する必要がある。詳細は「CM-04 データコピー機能」を参照のこと。

✧ ビジネスロジッククラスの実装と BizLogicExecution プロパティの設定

ビジネスロジック実行フェーズでは、「要求データ生成」フェーズで作成した入力 DTO をインプットとしてビジネスロジックを実行する。

ビジネスロジックは、実装すべきインタフェース等の制約はなく、実行するメソッドの引数が 1 つだけを持つというルールに従いさえすればどんなクラスでも実行可能である。これにより、ビジネスロジックは POCO で実装可能である。また、WCF クライアント（WSDL より生成したプロキシコード）を呼び出すことでサーバビジネスロジックを実行することも可能である。

実行するビジネスロジッククラスや WCF クライアントの設定は、BizLogicExecution プロパティで設定する。BizLogicExecution プロパティの詳細設定項目を表 6 に示す。

なお、リッチクライアント AP では、デバイスの制御やサーバ送信前のファイルの形式チェックなど、クライアント側での処理を伴うサーバ通信処理も存在する。このような場合は、クライアントのビジネスロジッククラスを実装し、その中で WCF クライアントを呼び出すように実装する。

表 6 標準機能における BizLogicExecution プロパティの詳細設定項目

項番	プロパティ名	説明
1	BizLogicExecution/BizLogic	実行するビジネスロジッククラスを設定する。 専用のエディタ「ビジネスロジック情報設定画面」で設定する。 未設定時は、ビジネスロジックは実行しない。
1-1	ExecutorName （ビジネスロジック種別）	以下のビジネスロジックの実行手段を選択する。 「ClientBizLogic」:クライアントビジネスロジックを実行する。 「WcfProxy」：WCF クライアント(ClientBase 継承クラス)を実行する。
1-2	BizLogicType （ビジネスロジッククラス）	【ExecutorName が「ClientBizLogic」の場合】 ビジネスロジッククラスの型を設定する。 【ExecutorName が「WcfProxy」の場合】 WCF クライアント(ClientBase 継承クラス)の型を設定する。 ※FW の入力支援機能により、ソリューション内で参照されるアセンブリから、クラス名をツリー表示し選択できる。
1-3	BizLogicName （定義名）	【ExecutorName が「ClientBizLogic」の場合】 通常は設定不要であるが、構成ファイルの DI 設定で定義したビジネスロジックを実行したい場合に使用する。実行するクラスを定義した/configuration/unity/containers/container/types/type 要素の name 属性の値を設定する。 【ExecutorName が「WcfProxy」の場合】 アプリケーション構成ファイル(App.config)で、WCF のクライアント側設定の endpoint 名を設定する。 ※FW の入力支援機能により、候補があれば自動的に表示する。
1-4	MethodName （メソッド名）	【ExecutorName が「ClientBizLogic」の場合】 ビジネスロジッククラスのメソッドを設定する。 【ExecutorName が「WcfProxy」の場合】 WCF クライアント（ClientBase 継承クラス)のメソッドを設定する。 ※FW の入力支援機能により、候補があれば自動的に表示する。

以下に、ビジネスロジックの実行パターンごとに、ビジネスロジッククラスの実装手順やBizLogicExecution プロパティの設定手順を説明する。
サーバ通信に関する実装手順については、「CL-04 サーバ通信機能」も合わせて参照すること。

① サーバ通信を伴わないクライアント処理の場合
クライアントビジネスロジックを作成し、EventProcessWorker.BizLogicExecution プロパティに、作成したビジネスロジッククラスとメソッド名を指定する。
以下に、クライアントビジネスロジックの実装例と、EventProcessWorker のプロパティ設定手順を示す。

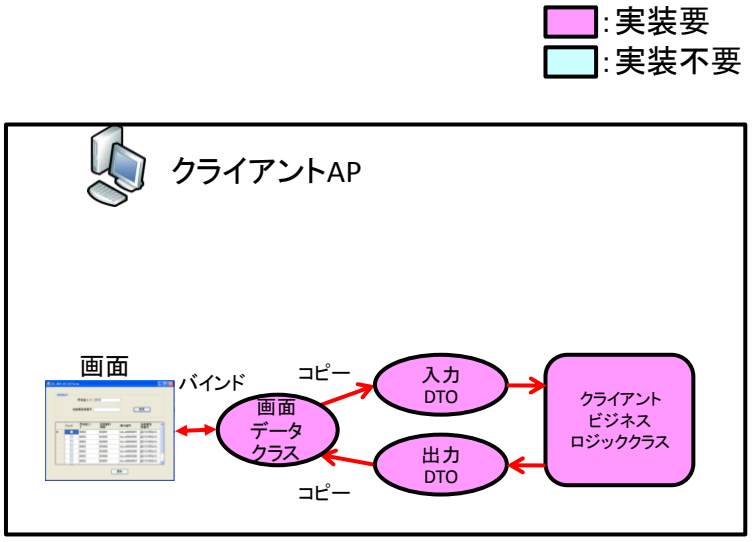


図 24 サーバ通信を伴わないクライアント処理の実装イメージ

```
public class A01_04_01_C01BizLogic
{
    public void Execute(A01_04_01_C01InputDto inputDto)
    {
        ///ファイルをウィルスチェック処理する例
        VirusCheckResult virusResult =
                        VirusChecker.CheckFile(inputDto.FilePath);
        ・・・
    }
}
```

リスト 5 クライアントビジネスロジックの記述例

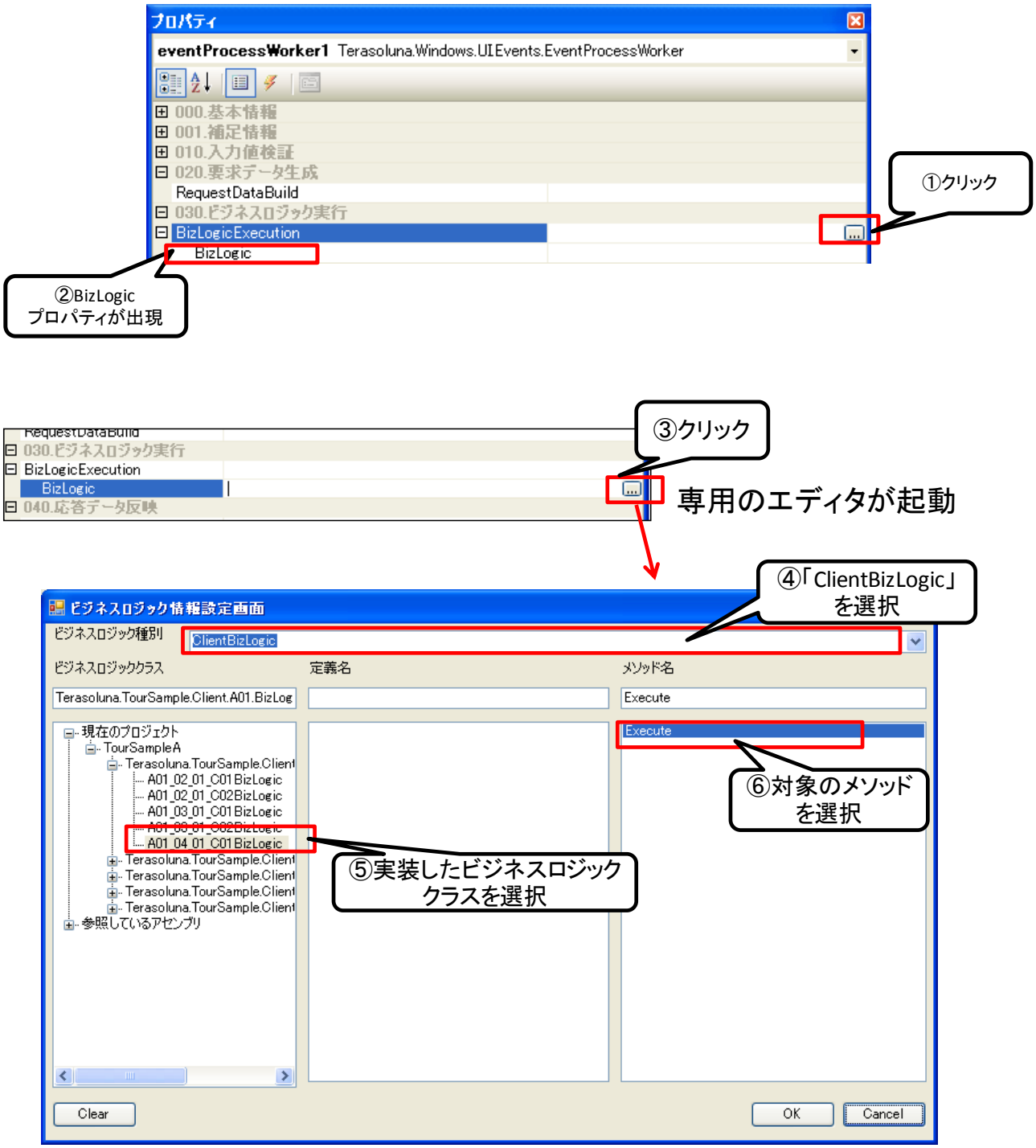


図 25 BizLogicExecution のプロパティ設定手順

② クライアント処理を伴わないサーバ通信処理の場合
WCF クライアントを直接呼び出すため、開発者はクライアントビジネスロジッククラスを実装する必要はない。WCF クライアントの生成方法は「CL-04 サーバ通信機能」の機能説明書を参照のこと。
EventProcessWorker の BizLogicExecution プロパティに、実行対象の WCF クライアントのクラス名およびメソッド名を指定する。

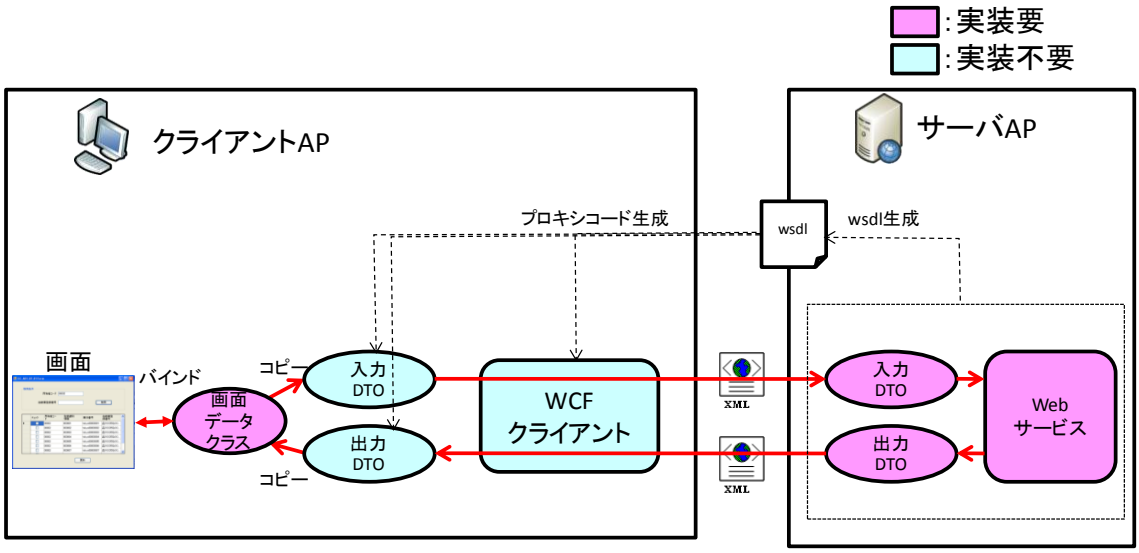


図 26 クライアント処理を伴わないサーバ通信処理の実装イメージ

以下に、EventProcessWorker のプロパティ設定手順を示す。

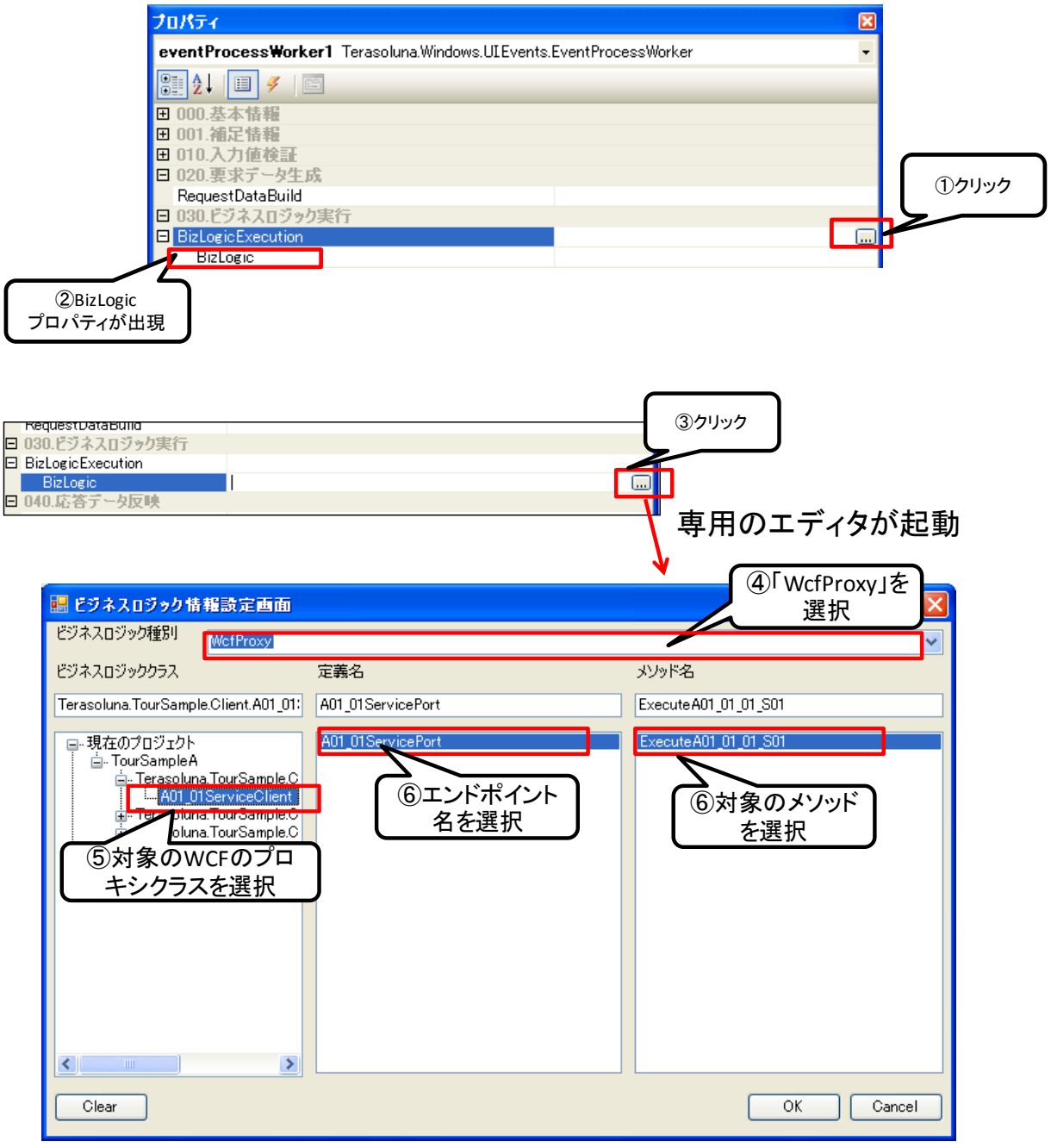
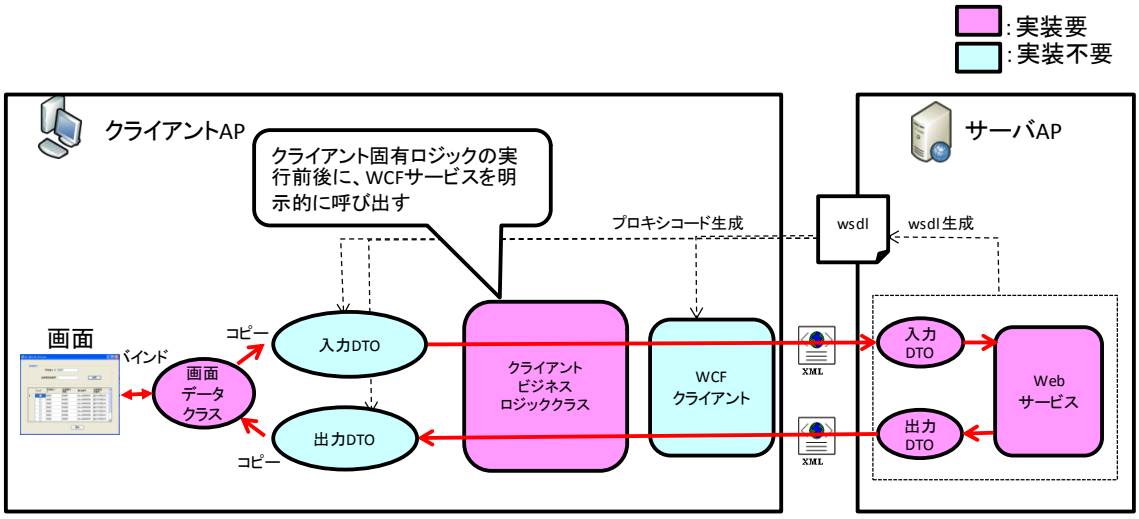


図 27 BizLogicExecution のプロパティ設定手順

③ クライアント処理を伴うサーバ通信処理の場合

クライアントビジネスロジッククラスを実装し、その中で、WCF クライアントのメソッドを呼び出す。プロパティの設定方法は、サーバ通信を伴わないクライアント処理と同様で、EventProcessWorker のプロパティに作成したビジネスロジッククラスとメソッド名を指定する。(図 25 を参照)



※入出力 DTO は別途クライアントビジネスロジッククラス専用の DTO を別で実装するケースもある

図 28 クライアントビジネスロジックを伴うサーバ通信処理の実装イメージ

クライアントビジネスロジックにおける WCF クライアント呼び出し方法は、一般的な WCF を使った開発と同様である。ただし、「CL-04 サーバ通信機能」の接続先「サーバアドレスの一括置換」機能を利用する場合は、CommunicationManager クラスを使ってエンドポイントアドレスを取得するよう実装する。CommunicationManager クラスの使用方法については、「CL-04 サーバ通信機能」の機能説明書を参照すること。
以下に、クライアントビジネスロジッククラスの実装例を示す。

```
    public class A01_02_01_C01BizLogic
    {
        /// App.configに記述されたエンドポイント名
        private const string EndpointName = "A01_02ServicePort";

        /// <summary>
        /// UnityContainerを取得・設定する。
        /// </summary>
        [Dependency]
        public IUnityContainer Container { get; set; }

        public A01_02_01_C01OutputDto Execute(A01_02_01_S01InputDto inputDto)
        {
            ・・・
            A01_02_01_S01OutputDto outputDto = null;
            A01_02ServiceClient client = null;
            try
            {
                ///エンドポイントアドレスの取得
                ///WCF接続先変更機能により、接続先サーバアドレスを一律置換してくれる。
                EndpointAddress address =
CommunicationManager.GetReplacedEndpointAddress(Container, EndpointName);
                //WCFサービスの実行
                client = new A01_02ServiceClient(EndpointName, address);
                outputDto = client.ExecuteA01_02_01_S01(inputDto);
                client.Close();
            }
            catch (System.Exception)
            {
                if (client != null)
                {
                    client.Abort();
                }
                throw;
            }
            return AfterCommunicate(outputDto);          }
・・・
    }
```

リスト 6 WCF クライアントの呼び出しを伴うクライアントビジネスロジックの記述例

✧ 応答データ反映方法の設定(ResponseDataReflection プロパティ)

応答データ反映フェーズは、「要求データ生成」フェーズと同様に、「CM-04 データコピー機能」により一定のマッピングルールをもとに出力 DTO の値を自動的に型変換し、「画面データ」クラスへコピーする。

「要求データ生成」(RequestDataBuild プロパティ)と同様、標準のマッピングルールに基づき自動コピーすればよい場合は、設定不要である。

標準のマッピングルールでは対応できない個別のマッピングルールがあれば ResponseDataReflection プロパティで設定する。

ResponseDataReflection プロパティの詳細設定項目について以下に示す。プロパティエディタでの設定手順のイメージは、「要求データ生成」(RequestDataBuild プロパティ)と同じため省略する。

表 7 標準機能における ResponseDataReflection プロパティの詳細設定項目

項番	プロパティ名		説明
-	ResponseDataReflection/ Mapping		-
1		Mappings	「CM-04 データコピー機能」の標準ルールによりビジネスロジックの出力 DTO のプロパティから「画面データ」クラスのプロパティへ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		SourceTargetProperty Paths	自動コピー対象とする、コピー元のビジネスロジックの出力 DTO のプロパティパスを設定する。
3		SourceIgnoreProperty Paths	自動コピー対象外とする、コピー元のビジネスロジックの出力 DTO のプロパティパスを設定する。
4		DestinationTargetPro pertyPaths	自動コピー対象とする、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。
5		DestinationTargetPro pertyPaths	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。

◆ EventProcessWorker のイベント

一連の「イベント処理」フローの途中で個別の処理を実行したい場合には、各ポイント（イベント）で個別の処理を差し込むことが可能である。EventProcessWorker から実行可能なイベントの一覧を以下に示す。

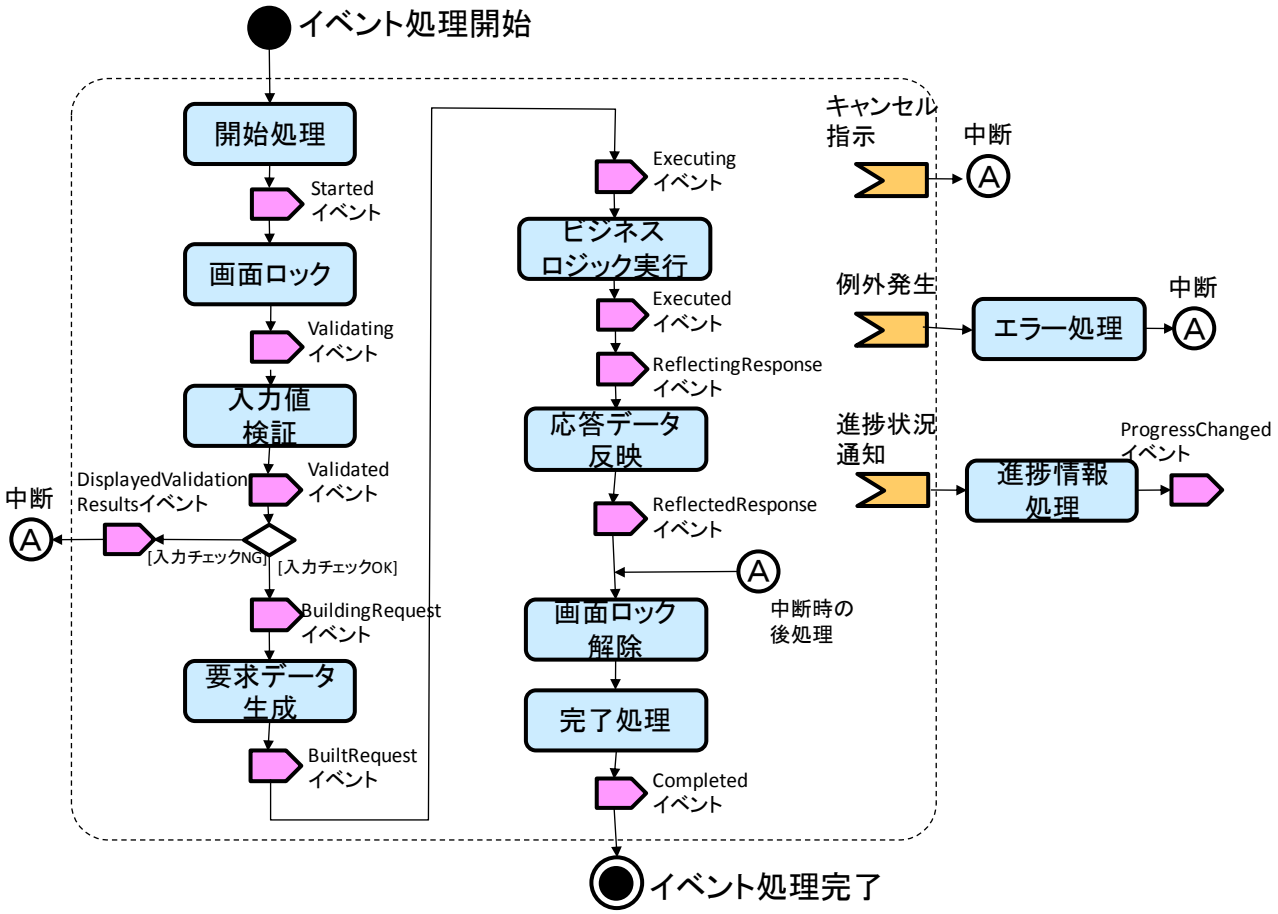


図 29 処理フローとイベント発生タイミング（再掲）

表 8　EventProcessWorker のイベント一覧

項番	イベント名	説明
1	Started	イベント処理開始時に発生するイベント
2	Validating	入力値検証処理開始直前に発生するイベント
3	Validated	入力値検証処理完了時に発生するイベント
4	DisplayedValidationResults	入力値検証処理において、エラーが発生した場合に発生するイベント
5	BuildingRequest	要求データ生成開始直前に発生するイベント
6	BuiltRequest	要求データ生成完了時に発生するイベント
7	Executing	ビジネスロジック実行開始直前に発生するイベント
8	Executed	ビジネスロジック実行完了時に発生するイベント
9	ReflectingResponse	応答データ反映開始直前に発生するイベント
10	ReflectedResponse	応答データ反映完了時に発生するイベント
11	Completed	イベント処理完了時に発生するイベント
12	ProgressChanged	イベント処理の進捗状況を通知するイベント

✧ Started イベント

Started イベントは、開始処理完了時に発生するイベントである。イベントハンドラの引数である EventProcEventArgs クラスで定義されたプロパティクラスで定義されたプロパティを以下に示す。

表 9　EventProcEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。ReadOnly プロパティ。

✧ Validating イベント

Validating イベントは、入力値検証処理開始直前に発生するイベントである。イベントハンドラの引数である ViewDataValidatingEventArgs クラスで定義されたプロパティを以下に示す。

表 10 ViewDataValidatingEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public bool Cancel { get; set; }	「イベントキャンセル」用指示フラグ。 デフォルトでは false でに設定されているが、true に変更すると、入力値検証処理を実行せずに即座にイベントが中断される。
3	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。

✧ Validated イベント

Validated イベントは、入力値検証処理完了時に発生するイベントである。イベントハンドラの引数である ViewDataValidatedEventArgs クラスで定義されたプロパティを以下に示す。

表 11 ViewDataValidatedEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public bool IsValid { get; set; }	入力値検証の処理結果。 正常の場合は true。エラーの場合は false。 フレームワークによる検証結果を上書きすることが可能であり、値を変更した場合、その変更した値を踏まえて、その後の処理を実施する。
3	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。

✧ DisplayedValidationResults イベント

入力値検証処理エラー時に、エラー情報の表示用に発生するイベントである。イベントハンドラの引数である ViewDataValidationEventArgs クラスで定義されたプロパティを以下に示す。

表 12 ViewDataValidationEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。

✧ BuildingRequest イベント

BuildingRequest イベントは、要求データ生成開始直前に発生するイベントである。イベントハンドラの引数である RequestDataBuildingEventArgs クラスで定義されたプロパティを以下に示す。

表 13 RequestDataBuildingEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public bool Cancel { get; set; }	「イベントキャンセル」用指示フラグ。 デフォルトでは false でに設定されているが、true に変更すると、要求データ生成処理を実行せずに即座にイベントが中断される。
3	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。

✧ BuiltRequest イベント

BuiltRequest イベントは、要求データ生成完了時に発生するイベントである。イベントハンドラの引数である RequestDataBuiltEventArgs クラスで定義されたプロパティを以下に示す。

表 14 RequestDataBuiltEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。
3	public object RequestData { get; set; }	要求データオブジェクト（ビジネスロジックの入力 DTO）。 ビジネスロジックに渡す要求データオブジェクトの値を変更したり、要求データオブジェクト自体を変更することも可能である。

✧ Executing イベント

Executing イベントは、ビジネスロジック実行開始直前に発生するイベントである。イベントハンドラの引数である BizLogicExecutingEventArgs クラスで定義されたプロパティを以下に示す。

表 15　BizLogicExecutingEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public bool Cancel { get; set; }	「イベントキャンセル」用指示フラグ。 デフォルトでは false でに設定されているが、true に変更すると、ビジネスロジック処理を実行せずに即座にイベントが中断される。
3	public object RequestData { get; }	要求データオブジェクト（ビジネスロジックの入力 DTO）。 ReadOnly プロパティ。

✧ Executed イベント

Executed イベントは、ビジネスロジック実行完了時に発生するイベントである。イベントハンドラの引数である BizLogicExecutedEventArgs クラスで定義されたプロパティを以下に示す。
Executed イベントの注意事項として、ビジネスロジックを非同期実行中にユーザ操作によりキャンセルされた場合、イベント処理フローを進行するスレッドと、ビジネスロジックおよび Executed イベントを実行するスレッドは別のスレッドであるため、Completed イベント実行後にビジネスロジックが終了し Executed イベントが実行されることがある。

表 16 BizLogicExecutedEventArgs のプロパティ

項番	プロパティ名	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public Exception Error { get; }	ビジネスロジック処理実行中に発生した例外オブジェクト。 ReadOnly プロパティ。
3	public bool NeedsRethrow { get; set; }	ビジネスロジック実行中に例外が発生した場合に、リスローするかのフラグ。 通常、フレームワークがリスロー可否の標準的な判定を実施するが、これを上書きたい場合にイベント内で値を変更する。 リスローする場合は true。しない場合は false。
4	public object RequestData { get; }	要求データオブジェクト（ビジネスロジックの入力 DTO）。 ReadOnly プロパティ。
5	public object ResponseData { get; set; }	応答データオブジェクト（ビジネスロジックの出力 DTO）。 応答データオブジェクトの値を変更したり、応答データオブジェクト自体を変更することも可能である。

✧ ReflectingResponse イベント

ReflectingResponse イベントは、応答データ反映処理開始直前に発生するイベントである。ResponseDataReflectingEventArgs の定義するプロパティを以下に示す。

表 17 ResponseDataReflectingEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public bool Cancel { get; set; }	「イベントキャンセル」用指示フラグ。 デフォルトでは false でに設定されているが、true に変更すると、応答データ反映処理を実行せずに即座にイベントが中断される。
3	public object ResponseData { get; }	応答データオブジェクト（ビジネスロジックの出力 DTO）。 ReadOnly プロパティ。
4	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。

✧ ReflectedResponse イベント

ReflectedResponse イベントは、応答データ反映処理完了時に発生するイベントである。イベントハンドラの引数である ResponseDataReflectedEventArgs クラスで定義されたプロパティを以下に示す。

表 18 ResponseDataReflectedEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	public object ResponseData { get; }	応答データオブジェクト（ビジネスロジックの出力 DTO）。 ReadOnly プロパティ。
3	public object ViewData { get; }	「画面データ」クラスのインスタンス。 ReadOnly プロパティ。

✧ Completed イベント

Completed イベントは、イベント処理完了時に発生するイベントである。イベントハンドラの引数である EventProcCompletedEventArgs クラスで定義されたプロパティの一覧を以下に示す。
なお、イベント処理で例外が発生した場合も、この Compeled イベントは実行される。
注意事項として、Completed イベント内で例外が発生すると最初にエラーとなった原因が消失する可能性がある。Completed イベントでは例外を発生させないように実装すること。

表 19 EventProcCompletedEventArgs のプロパティ

項番	プロパティ	説明
1	public EventProcessContext Context { get; }	イベント処理実行にかかわる制御情報オブジェクト。 ReadOnly プロパティ。
2	Exception Error { get; }	イベント処理実行中に発生した例外オブジェクト。 ReadOnly プロパティ。
3	public EventProcessResult Result { get; }	イベント処理の実行結果オブジェクト。 ReadOnly プロパティ。

✧ ProgressChanged イベント

ProgressChanged イベントはイベント処理の進捗状況を通知する。イベントハンドラの引数 である System.ComponentModel.ProgressChangedEventArgs クラスのプロパティを以下に示す。

表 20 ProgressChangedEventArgs のプロパティ

項番	プロパティ	説明
1	public int ProgressPercentage { get; }	イベント処理の進捗率。 ReadOnly プロパティ。
2	public object UserState { get; }	固有のユーザ状態。本機能では、実行中のフェーズを示すイベント処理の進行状態。 ReadOnly プロパティ。

UserState プロパティには、イベント処理フローのどのフェーズまで進行したかを示す EventProcState クラスの定数値を格納する。以下に EventProcState クラスの定数値について示す。

表 21 EventProcState クラスの定数値

項番	変数名	説明
1	NotStart	イベント処理実行前の状態
2	Start	イベント処理実行開始時の状態
3	ViewDataValidation	入力値検証フェーズ実行中の状態
4	RequestDataBuild	要求データ生成フェーズ実行中の状態
5	BizLogicExecution	ビジネスロジック実行中の状態
6	ResponseDataReflection	応答データ生成フェーズ実行中の状態
7	Completion	イベント処理実行正常終了の状態

また、以下にイベント処理フローと EventProcState の対応関係を示す。
なお、「画面ロック解除～完了処理」間のイベントは、イベント処理の正常終了、中断に関わらず必ず実施され、イベント処理成功時（Completion）または中断時の状態が保持される。

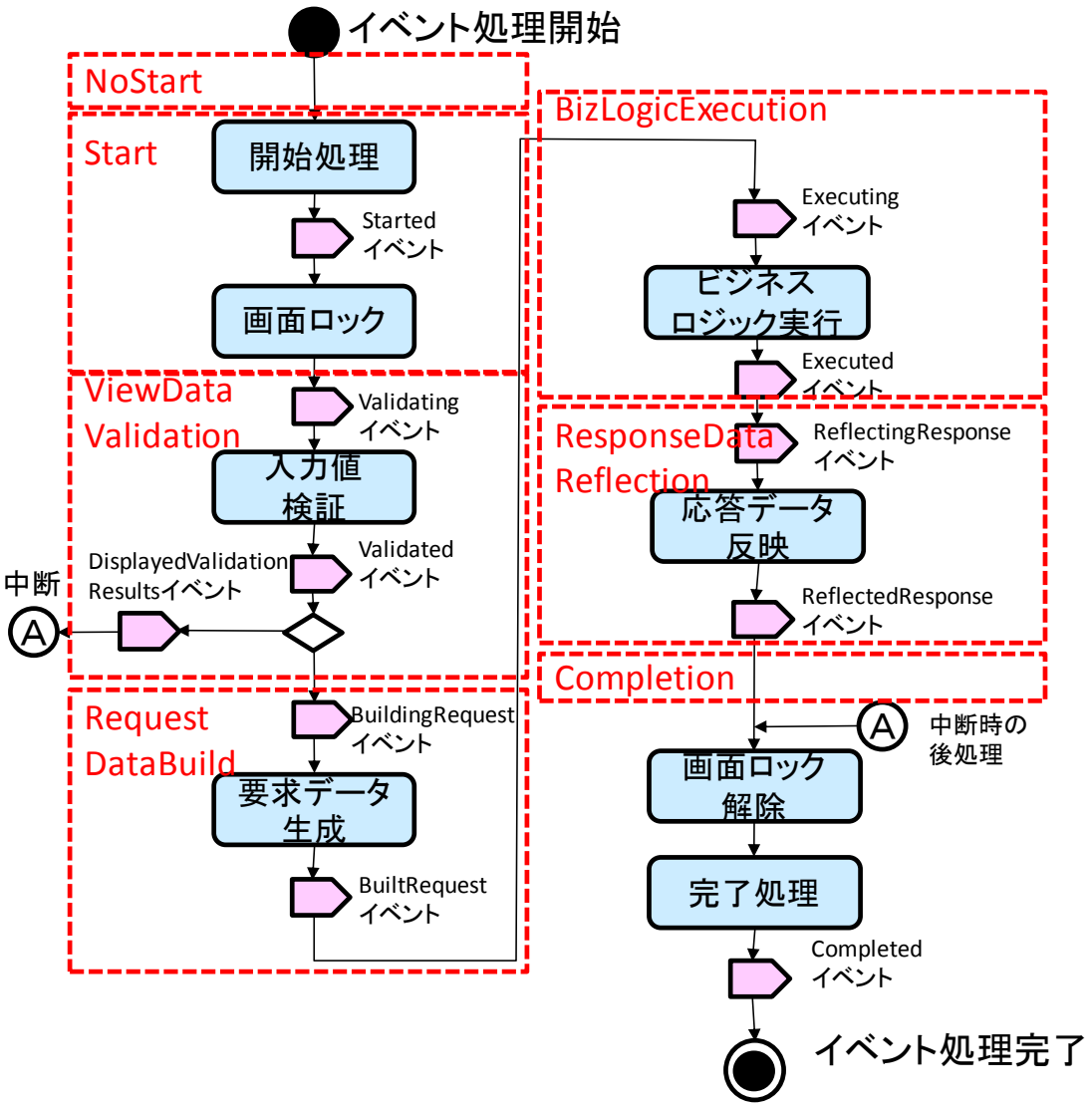


図 30 イベント処理フローと EventProcState の対応関係

◆ イベント処理結果の取得

RunWorker メソッドの戻り値や Completed イベントの引数の Result プロパティから、EventProcessResult クラスのインスタンスとして一連のイベント処理の結果を取得することができる。EventProcessResult の定義するプロパティの一覧を以下に示す。

表 22　EventProcessResult のプロパティ

項番	プロパティ名	説明
1	State	イベント処理がどこまで正常に進行できたかを表す状態文字列。EventProcState クラスの定数値が格納される。（表 21、表 24 を参照）
2	HasResult	イベント処理の結果を格納できているどうかを表す bool 値。ResultType プロパティに値が格納されていれば true を返す。
3	ResultType	イベント処理の結果文字列。EventProcResultTypes クラスの定数値（表 23、表 24 を参照）
4	BizLogicResult	応答データオブジェクト（ビジネスロジックの出力 DTO）。IsSuccess プロパティが true の場合のみ値が格納されている。object 型のプロパティであるため、通常、as 演算子等で 実際の応答データの型にキャストして扱う。
5	BizLogicErrorType	業務エラー種別を表す任意の文字列。業務エラー発生時に格納。（表 26 を参照）
6	Errors	エラー情報(ErrorInfo オブジェクト)のリスト。エラー発生時に格納。（表 26 を参照）
7	IsSuccess	イベント処理が成功したかどうかを表す bool 値。ResultType プロパティの値が「Success」かどうかの判定と等価である。（表 24 を参照）
8	IsCanceled	ユーザのキャンセルボタン押下等の任意のタイミング（「ユーザキャンセル」）で、イベント処理が中断されたかどうかを表す bool 値。EventProcessWorker の CacelAsync メソッドにより、イベント処理が中断された場合に true を返す。ResultType プロパティの値が「Canceled」かどうかの判定と等価である。（表 24 を参照）
9	IsEventCanceled	イベント処理がイベントハンドラ内の実装により中断（「イベントキャンセル」）されたかどうかを表す bool 値。EventProcessWorker の各フェーズの開始イベント（イベント名に「ing」が付くイベント）中で、イベント引数の Cancel プロパティを true にセットすることでイベント処理が中断された場合に true を返す。ResultType プロパティの値が「EventCanceled」かどうかの判定と等価である。（表 24 を参照）

10	IsAnyValidationError	入力値検証エラーが発生したかどうかを表す bool 値。クライアントまたはサーバで入力値検証エラーが発生した場合に true を返す。ResultType プロパティの値が「ValidationFailure」または「 ServerValidationFailure」かどうかの判定と等価である。（表 24 を参照）
11	IsAnyBizLogicError	業務エラーが発生したかどうかを表す bool 値。クライアントまたはサーバで業務エラーが発生した場合に true を返す。ResultType プロパティの値が「BizLogicFailure」または「ServerBizLogicFailure」かどうかの判定と等価である。（表 24 を参照）
12	IsAnyUnexpectedError	システムエラーが発生したかどうかを表す bool 値。クライアントまたはサーバでシステムエラーが発生した場合に true を返す。ResultType プロパティの値が「UnexpectedFailure」または「ServerUnexpectedFailure」かどうかの判定と等価である。（表 24 を参照）
13	IsAnyServerError	サーバエラーが発生したかどうかを表す bool 値。サーバエラー（入力チェックエラー、業務エラー、システムエラーのいずれか）の場合に true を返す。ResultType プロパティの値が「ServerValidationFailure」、「 ServerBizLogicFailure」、「ServerUnexpectedFailure」のいずれかどうかの判定と等価である。（表 24 を参照）
14	IsAnyCommunicationError	サーバ通信時のエラーが発生したかどうかを表す bool 値。サーバ通信時に何らかのエラー（サーバ通信エラー、サーバ入力チェックエラー、サーバ業務エラー、サーバシステムエラーのいずれか）が返却された場合に true を返す。 ResultType プロパティの値が「CommunicationFailure」、「ServerValidationFailure」、「 ServerBizLogicFailure」、「ServerUnexpectedFailure」のいずれかどうかの判定と等価である。（表 24 を参照）

State プロパティは、表 21 に示す状態文字列を格納しており、イベント処理フローがどこまで進行したかの情報を取得することができる。また、ResultType プロパティは、表 23 に示すイベント処理の結果文字列を格納しており、イベントが成功したか否かやイベント処理が中断した場合の原因を取得することができる。「Is」で始まるプロパティは、該当するエラーの原因に分類されるかを ResultType プロパティの値を基に判定する。
表 24 に、イベント処理の結果に対する State、ResultType、「Is～」プロパティの格納値を示す。

表 23　EventProcResultTypes の定数値

項番	変数名	説明
1	Success	正常終了。
2	Canceled	キャンセルボタン押下等、ユーザ操作による中断（「ユーザキャンセル」）。イベントハンドラ内から EventProcessWorker の CacelAsync メソッドを呼び出す。
3	EventCanceled	EventProcessWorker 実行時に発生する各種前処理イベント（Validating イベント、BuildingRequest イベント、ReflectingResponse イベント、Executing イベント）内での中断（「イベントキャンセル」）。イベントの引数である各 EventArgs クラスの Cancel フラグを true にすることで実現。
4	ValidationFailure	クライアント AP で入力値検証エラー発生。
5	BizLogicFailure	クライアント AP で業務エラー発生。
6	UnexpectedFailure	クライアント AP でシステムエラー発生。
7	CommunicationFailure	サーバ接続エラー(CommunicationException)や通信中のタイムアウト(TimeoutException)といった通信エラーが発生。
8	ServerValidationFailure	サーバ AP で入力値検証エラーが発生。
9	ServerBizLogicFailure	サーバ AP で業務エラーが発生。
10	ServerUnexpectedFailure	サーバ AP でシステムエラーが発生。

表 24 イベント処理結果に対する EventProcessResult プロパティの格納値

項番	処理結果	State	Result Type	IsSuccess	IsCanceled	IsEventCanceled	IsAnyValidationError	IsAnyBizLogicError	IsAnyUnexpectedError	IsAnyServerError	IsAnyCommunicationError
1	開始処理前										
1-1	ユーザキャンセルによる中断	NoStart	Cancel	false	true	false	false	false	false	false	false
2	開始処理～画面ロック処理										
2-1	予期せぬエラー	Start	Unexpected Failure	false	false	false	false	false	true	false	false
2-2	ユーザキャンセル による中断		Cancel	false	true	false	false	false	false	false	false
3	入力値検証処理										
3-1	予期せぬエラー	ViewDataValidation	Unexpected Failure	false	false	false	false	false	true	false	false
3-2	入力値検証エラー		ValidationFailure	false	false	false	true	false	false	false	false
3-3	Validating イベントで中断		EventCancel	false	false	true	false	false	false	false	false
3-4	ユーザキャンセルによる中断		Cancel	false	true	false	false	false	false	false	false
4	要求データ生成処理										
4-1	予期せぬエラー	RequestDataBuild	Unexpected Failure	false	false	false	false	false	true	false	false
4-2	BuildingRequest イベントで中		EventCancel	false	false	true	false	false	false	false	false

表 24 イベント処理結果に対する EventProcessResult プロパティの格納値

項番	処理結果	State	Result Type	IsSuccess	IsCanceled	IsEventCanceled	IsAnyValidationError	IsAnyBizLogicError	IsAnyUnexpectedError	IsAnyServerError	IsAnyCommunicationError
	断										
4-3	ユーザキャンセルによる中断		Cancel	false	true	false	false	false	false	false	false
5	ビジネスロジック実行										
5-1	クライアントで予期せぬエラー発生	BizLogicExecution	UnexpectedFailure	false	false	false	false	false	true	false	false
5-2	Executing イベントで中断		EventCancel	false	false	true	false	false	false	false	false
5-3	ユーザキャンセルによる中断		Cancel	false	true	false	false	false	false	false	false
5-4	クライアントビジネスロジックで業務エラー発生		BizLogicFailure	false	false	false	false	true	false	false	false
5-5	サーバ通信エラー発生		CommunicationFailure	false	false	false	false	false	false	false	true
5-6	サーバで入力値検証エラー発生		ServerValidationFailure	false	false	false	true	false	false	true	true
5-7	サーバビジネスロジックで業務エラー発生		ServerBizLogicFailure	false	false	false	false	true	false	true	true

表 24 イベント処理結果に対する EventProcessResult プロパティの格納値

項番	処理結果	State	Result Type	IsSuccess	IsCanceled	IsEventCanceled	IsAnyValidationError	IsAnyBizLogicError	IsAnyUnexpectedError	IsAnyServerError	IsAnyCommunicationError
5-8	サーバで予期せぬエラーが発生		ServerUnexpectedFailure	false	false	false	false	false	true	true	true
6	応答データ反映										
6-1	予期せぬエラー発生	ResponseDataReflection	UnexpectedFailure	false	false	false	false	false	true	false	false
6-2	ReflectingResponse イベントで中断		EventCancel	false	false	true	false	false	false	false	false
7	イベント処理終了時										
7-1	すべて正常	Completion	Success	true	false	false	false	false	false	false	false
7-2	画面ロック解除または完了処理で予期せぬエラー発生	(※)	(※)	(※)	(※)	(※)	(※)	(※)	(※)	(※)	(※)

(※)画面ロック解除処理と完了処理はイベント処理フローが正常に終了しても失敗した場合も必ず実行されるため、イベント処理が成功または中断された時点の値が格納される。

◆ ビジネスロジッククラスにおける業務エラーの返却

TERASOLUNA フレームワークでは業務エラーを例外として扱っている。
ビジネスロジッククラス内で業務エラーを返却するには、業務エラー専用の例外である BizLogicException クラスをスローする。BizLogicException クラスの詳しい使用方法については、「CL-05 クライアントエラーハンドリング機能」の機能説明書を参照のこと。
以下に、BizLogicException を使った業務エラーの記述例を示す。

```
public class LoginBizLogic
{
    public void Login(LoginInputDto input)
    {
        if (!"terasoluna".Equals(input.UserId, StringComparison.Ordinal)
            || !"password".Equals(input.Password, StringComparison.Ordinal))
        {
            ///業務エラーは、errorTypeをBizLogicExceptionErrorType.BizLogicFailureにして
            ///BizLogicExceptionをスローする
            throw new BizLogicException(
                BizLogicExceptionErrorType.BizLogicFailure,
                CalcResource.ERROR_CALC_MSG002,
                new List<ErrorInfo>()
                {
                    new ErrorInfo("ERROR_CALC_MSG003", null,
                                  CalcResource.ERROR_CALC_MSG003, null)
                });
        }
    }
}
```

リスト 7 BizLogicException を使った業務エラーの記述例

◆ エラーハンドリング

クライアントまたはサーバで発生した、入力値検証エラー、業務エラー、システムエラーについて、システム全体で集約例外処理を実施する。集約例外処理の詳細は、「CL-05 クライアントエラーハンドリング機能」の機能説明書を参照のこと。
入力値検証エラーや業務エラーについては、EventProcessWorker のイベントにより固有のエラー処理を実施することも可能である。

入力値検証エラーが発生した場合には、DisplayedValidationResults イベントや Validted イベントで、固有の入力エラー処理を実施できる。
以下に DisplayedValidationResults イベントの実装例を示す。

```
private void a01_01_01_C01EventProcessWorker_DisplayedValidationResults(
            object sender, Terasoluna.Windows.Forms.Events.ViewDataValidationEventArgs e)
{
    ///画面データから個別にエラー情報を取得
    ValidationErrorInfo errorInfo = ViewData.Tour.GetValidationError("DeptPlaceList");
    if (errorInfo != null)
    {
        ///ErrorProviderが自動表示されないケースのため
        ///例外的にをErrorProviderへエラーメッセージを手動でセットした例
        errorProvider.SetError(
                    deptPlaceListSelectedItemsBindableListBox,
                    errorInfo.Message);
    }
}
```

リスト 8　DisplayedValidationResults イベントの実装例

また、業務エラーが発生した場合には、RunWorker メソッドの戻り値や Completed イベントの引数の Result プロパティからイベント処理結果を取得し、EventProcessResult の BizLogicErrorType プロパティや Errors プロパティを確認することで、エラー情報を使用した固有のエラー処理を実施することも可能がある。各種エラー発生時に格納される値を以下に示す。

表 25　ErrorInfo のプロパティ

項番	プロパティ名	説明
1	ErrorId	エラーID
2	Arguments	メッセージの置換文字列
3	Message	エラーメッセージ
4	RawData	EventInfo の生成元となったエラー情報オブジェクト

表 26 イベント処理結果に対する EventProcessResult プロパティの格納値

エラーの種類	BizLogicErrorType プロパティ	Errors プロパティ(IList<ErrorInfo>)			
		ErrorInfo 要素の各プロパティ			
		ErrorId	Arguments	Message	RawData
クライアント入力値検証エラー	null	EventProcResultTypes.ValidationFailure 定数	null	開発者が設定した入力値検証エラーメッセージ	元になった ValidationResults オブジェクト
クライアント業務エラー	スローした BizLogicException の BizLogicErrorType プロパティと同値	スローした BizLogicException の Errors プロパティと同値			
クライアントシステムエラー	null	null			
サーバ入力値検証エラー	SOAP Fault の Detail 要素内の BizLogicErrorType 要素	SOAP Fault の Detail 要素内の各 ErrorMessage 要素の ErrorId 要素	SOAP Fault の Detail 要素内の各 ErrorMessage 要素の Arguments 要素	SOAP Fault の Detail 要素内の各 ErrorMessage 要素の Messsage 要素	元になった ITerasolunaSoapFaultErrorMessage オブジェクト
サーバビジネスロジックで業務エラー					
サーバシステムエラー【SOAPFault の Detail 要素が解析が可能な場合】					
サーバシステムエラー【SOAPFault の Detail 要素が解析が不可能な場合】	null	null			
サーバ通信エラー（CommunicationException 発生時）	null	“CommunicationUnexpectedFailure”	null	発生した例外のメッセージ	発生した例外オブジェクト
サーバ接続中にタイムアウト(TimeoutException 発生時)	null	“CommunicationTimedOut”	null	発生した例外のメッセージ	発生した例外オブジェクト

以下に Completed イベントでエラー情報を取得した例を示す。

```
private void eventProcessWorker_Completed(
        object sender, Terasoluna.Windows.Forms.Events.EventProcCompletedEventArgs e)
{
    if (e.Result.IsAnyBizLogicError)
    {
        StringBuilder sb = new StringBuilder();
        foreach (ErrorInfo error in e.Result.Errors)
        {
            sb.AppendFormat("{0}:{1}", error.ErrorId, error.Message).AppendLine();
        }
        errorMessageLabel1.Text = sb.ToString();
    }
}
```

リスト 9 Completed イベントでのエラー情報の取得例

◆ イベント処理の中断

➢ 前処理イベント(Validating イベント、BuildingRequest イベント、ReflectingResponse イベント、Executing イベント)での中断(「イベントキャンセル」)

EventProcessWorker の前処理イベントにおいて、各種 EventArgs クラスの Cancel プロパティを true に変更することにより、処理イベント処理を各フェーズの実行直前に即座に中断することができる(「イベントキャンセル」)。
以下に、Cancel プロパティの使用例を示す。
この例では、Executing イベントで、ビジネスロジック実行前にユーザに確認ダイアログを表示して、ダイアログでキャンセルされた場合に Cancel プロパティを true にすることで、イベント処理を中断している。

```
/// ビジネスロジック実行直前のイベント
private void a01_01_01_C01EventProcessWorker_Executing(object sender,
            Terasoluna.Windows.Forms.Events.BizLogicExecutingEventArgs e)
{
        ///「メッセージ通知機能」による確認ダイアログの表示
        bool result = MessageNotifier.ShowConfirmMessage(
                        this, "ツアー情報を登録してもよろしいですか？");
        ///ダイアログでキャンセルされた場合はイベントを中断する
        e.Cancel = !result;
}
```

リスト 10 Executing イベントでの Cancel プロパティの使用例

➢ EventProcessWorker の CancelAsync メソッドによる中断(「ユーザキャンセル」)

イベント処理を RunWorkerAsync メソッドにより非同期実行している場合は、ユーザのキャンセルボタン押下等によるキャンセル指示(「ユーザキャンセル」)により、任意のタイミングでイベント処理の中断指示をすることが可能である。
トリガとなる UI コントロールのイベントハンドラ(例えば、キャンセル用ボタンのクリックイベント)で、EventProcessWorker の CancelAsync メソッドを実行することでイベント処理を中断する。

以下に、EventProcessWorker の CancelAsync メソッドの使用例を示す。

```
/// キャンセルボタン押下
private void cancelButton_Click(object sender, EventArgs e)
{
        ///キャンセル指示
        a01_04_01EventProcessWorker.CancelAsync();
}
```

リスト 11 EventProcessWorker の CancelAsync メソッドの使用例

なお、EventProcessWorker.EventProcessName プロパティを「DialogLock」に設定することで表示される進捗ダイアログにある「キャンセル」ボタンも、EventProcessWorker の CancelAsync メソッドを内部的に利用している。

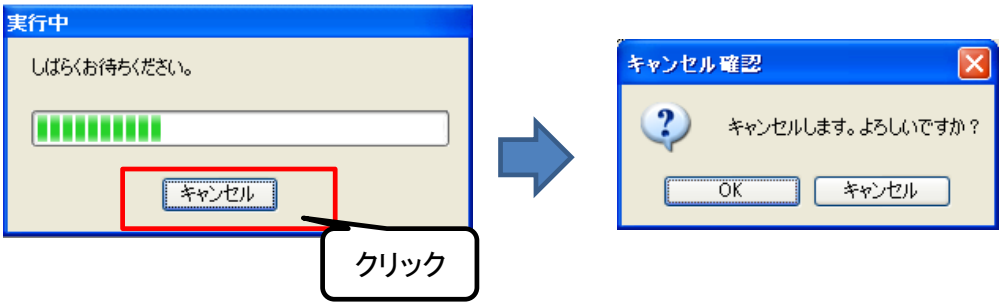


図 31 標準機能の進捗ダイアログによるキャンセル処理

また、「ユーザキャンセル」時には、デフォルトでは「完了処理」フェーズで以下のダイアログを表示する。プロジェクトの要件に合わせて変更したい場合は、イベント処理の定義を拡張した Extension クラスを実装すること。

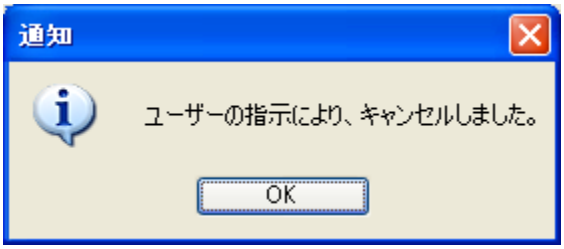


図 32 キャンセル時に標準で表示されるダイアログ

「ユーザキャンセル」時に、ボタンのクリックイベントなどから CancelAsync メソッドを実行しても、ビジネスロジックを実行しているタスクスレッドそのものは終了しない。それほど重くないタスクであれば、ビジネスロジックが終了することによる、タスクスレッドの消滅のを待てばよいが、ロジック内で無限ループを実行している場合や、負荷が高く、即座に処理を停止する必要がある場合などは、キャンセルフラグを定期的にチェックするなどして、「ユーザキャンセル」の指示を検知し、処理を終了するように実装する必要がある。
ビジネスロジッククラスでキャンセル状態を確認するには、
「InvocationScope.Current.GetContext<EventProcessContext>.CancellationPending」でキャンセルフラグをチェックする。値が true であれば「ユーザキャンセル」の指示が出されていることを意味する。

以下に、実装例を示す。

```
private bool Cancelled
{
        get
        {
            ///キャンセルフラグの取得
            return InvocationScope.Current.GetContext<EventProcessContext>().CancellationPending;
        }
}

・・・

///ビジネスロジックの実行
public void Execute(A01_04_01_C01InputDto inputDto)
{
・・・
        ///定期的にキャンセルされたかどうかチェック
        while (!Canceled)
        {
            ・・・
        }
}
```

リスト 12 キャンセルフラグの使用例

◆ 進捗状況の表示

EventProcessWorker.EventProcessName プロパティを「DialogLock」に設定することで表示されるダイアログには、プログレスバーがありイベント処理の進捗状況を表示することができる。
これは、フレームワークにより標準提供されている機能であるが、イベント処理を定義する Extension クラスを作成することで、プロジェクトの要件に合わせた進捗状況の表示処理に変更することも可能である。

また、業務個別でプログレスバー等に進捗状況を表示する場合には、ProgressChanged イベントを使用する。
以下に、ProgressChanged イベントの実装例を示す。
この例では、画面上にある ToolStripProgressBar に対して進捗率を表示する。

```
private void a01_04_01EventProcessWorker_ProgressChanged(
            object sender, ProgressChangedEventArgs e)
{
        ///画面上にあるToolStripProgressBarに対して進捗率を表示
        toolStripProgressBar1.Value = e.ProgressPercentage;
}
```

リスト 13 ProgressChanged イベントの実装例

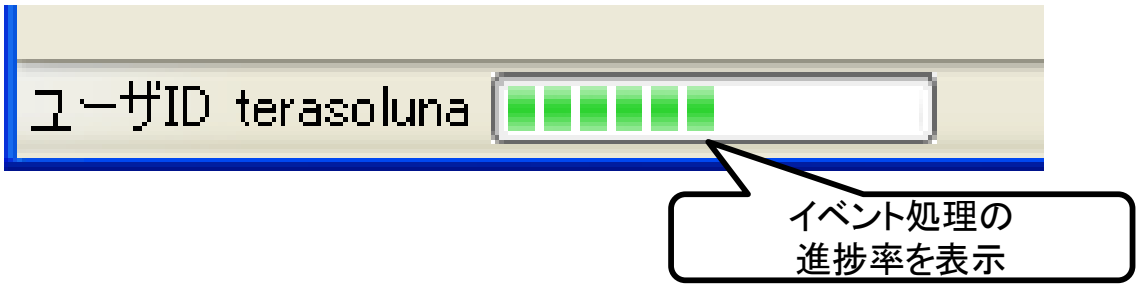


図 33 業務個別でのプログレスバーの表示例

◆ ビジネスロジックでの厳密な進捗率計算の実装

フレームワークが自動的に計算する進捗率は、ビジネスロジックの実行状況を表す厳密な進捗率ではない。厳密に進捗率を計算したい場合は、ビジネスロジッククラスの中で進捗率を計算する処理を実装する必要がある。
この場合は、EventProcesWorker.ProgressSetName プロパティの値を”Nothing”(または未指定)にして、進捗率の自動計算を実施しないようにする。
ビジネスロジックから進捗率を通知するには、
「InvacationScope.Current.GetContext<EvetProcessContext>().ProgressManager」により、進捗状況を管理する IEventProcProgressManager オブジェクトを取得し、進捗率を引数として IEventProcProgressManager.SetProgress メソッドを実行する。
以下に、実装例を示す。

```
/// 進捗率管理クラス
private IEventProcProgressManager progressManager
        = InvocationScope.Current.GetContext<EventProcessContext>().ProgressManager;
・・・
///ストリームをReadした割合を進捗率として通知する
int result = fileStream.Read(buffer, offset, count);
int percentage = (int)(Position * 100 / Length);
progressManager.SetProgress(new ProgressInfo(percentage));
```

リスト 14 進捗状況通知の実装例

なお、ファイルアップロード・ダウンロード時の進捗状況表示のより具体的な実装方法については、「CL-04 サーバ通信機能」の機能説明書を参照のこと。

◆ ビジネスロジックにおける DI の利用

ビジネスロジッククラスは「CM-02 インスタンス管理機能」によりインスタンス管理されているため、DI により外部からオブジェクトを注入することができる。以下に、DI を用いたビジネスロジックの実装例を示す。
この例では、ビジネスロジックから呼ばれるウィルスチェック処理クラス(IVirusChecker インタフェース実装クラス)を、製造時にウィルスチェックライブラリがない場合にモックに差し替えられるようにした例である。IVirusChecker の実装オブジェクトは、AP 共通構成ファイル(TerasolunaApplication.config)または業務構成ファイル(”アセンブリ名”.config)で DI 定義し、設定ファイルで切り替え可能にしている。

```
public class A01_04_01_C01BizLogic
{
    /// 製造時、全ての開発者にウィルスチェックライブラリがない場合などを考慮し、
    /// ウィルスチェック処理をモックでさしえられるようにDIしている例
    [Dependency]
    public IVirusChecker VirusChecker { get; set; }

    /// ビジネスロジックの実行
    public void Execute(FileInfoDto inputDto)
    {
        ///ウィルスチェック
        VirusCheckResult virusResult = VirusChecker.CheckFile(inputDto.FilePath);

        ///エラー情報の格納処理
        List<ErrorInfo> errors = new List<ErrorInfo>();
        if (virusResult.Status != VirusCheckStatus.SUCCESS)
        {
            errors.Add(virusResult.Error);
        }

        ///エラーが存在する場合には業務エラー発生
        if (errors.Count > 0)
        {
            throw new BizLogicException(
                BizLogicExceptionErrorType.BizLogicFailure,
                "",
                errors);
        }
    }
}
```

図 34 ビジネスロジックでの DI の使用例

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <typeAliases>
      <typeAlias alias="IVirusChecker"
                 type="Terasoluna.TourSample.Client.Common.VirusCheck.IVirusChecker,
TourSampleCommon" />
      <typeAlias alias="MockVirusChecker"
                 type="Terasoluna.TourSample.Client.Common.VirusCheck.MockVirusChecker,
TourSampleCommon" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <!-- ウィルスチェック処理をモックに差し替えた例-->
          <type type="IVirusChecker" mapTo="MockVirusChecker">
            ・・・
          </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 15 構成ファイルに記述した DI の使用例

また、実行するビジネスロジッククラスをインタフェースとして実装オブジェクトを外部から注入するなど、実行対象のビジネスロジック自体を構成ファイルで定義したい場合もある。
このような場合、EventProcessWorker.Execution.BizLogic プロパティの「ビジネスロジック情報設定画面」の「定義名」(BizLogicName)で、構成ファイルに定義した type タグの name 属性の値を指定することで、構成ファイルの定義に基づくオブジェクトを実行対象のビジネスロジックとして指定することができる。
以下に、実装例を示す。

```
///ビジネスロジックを実現するインタフェースの例
public interface ILoginBizLogic
{
    void Login(LoginInputDto input);
}

///インタフェースの実装
public class LoginBizLogic : ILoginBizLogic
{
     public void Login(LoginInputDto input)
    {
        ・・・
    }
}
```

リスト 16 ビジネスロジックの実行例

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <typeAliases>
      <typeAlias alias="ILoginBizLogic"
          type="CalcBusinessApplication.BizLogic.ILoginBizLogic, CalcBusinessApplication" />
      <typeAlias alias="LoginBizLogic"
          type="CalcBusinessApplication.BizLogic.LoginBizLogic, CalcBusinessApplication" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <type name="loginBiz" type="ILoginBizLogic" mapTo="LoginBizLogic"/>
        </types>
        <extensions>
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 17 構成ファイルの記述例

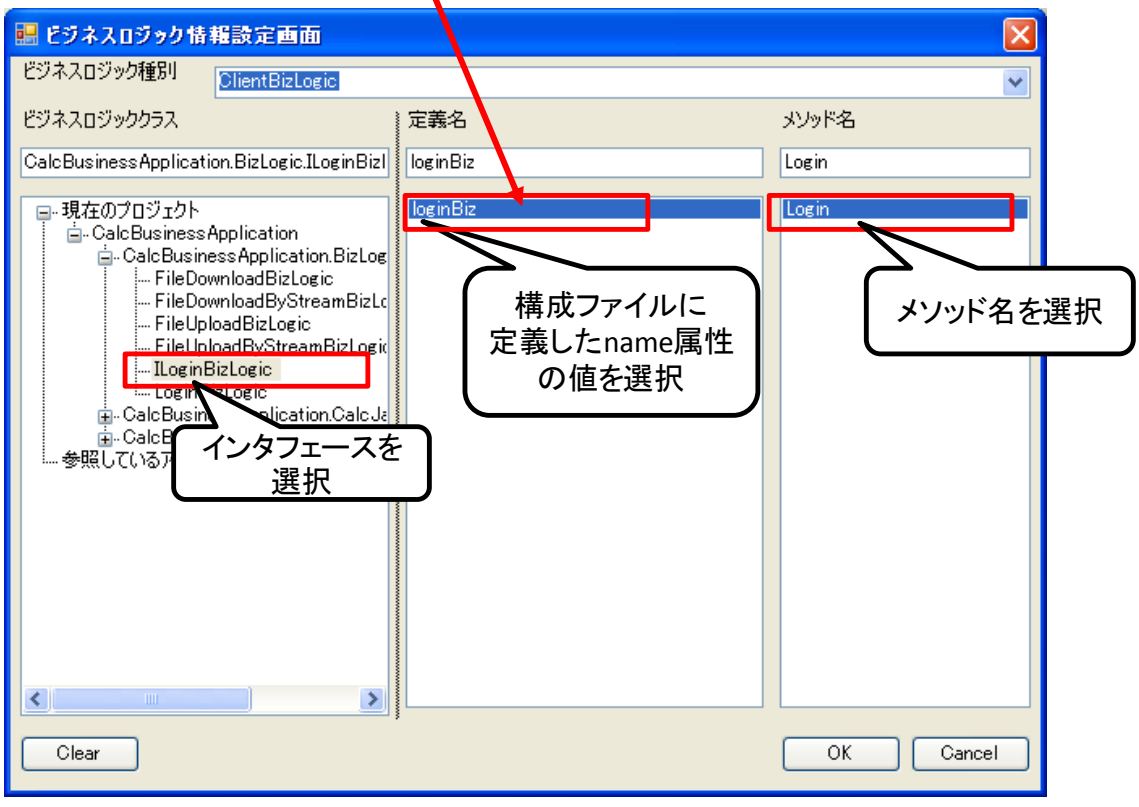


図 35 EventProcessWorker.Execution.BizLogic プロパティの設定例

◆ 構成ファイルの設定

本機能を利用するためには、AP 共通構成ファイル(TerasolunaApplication.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.BizLogic.ClientBizLogicExtension クラス
  - EventProcessWorker の BizLogicExecution プロパティで設定したビジネスロジッククラスを自動実行するためのデフォルトの Extension
- Terasoluna.Windows.ViewModel.Validation.Events.ValidatableEventProcessExtension
  - 標準のイベント処理フローを定義するデフォルトの Extension

また、本機能を利用するためには、以下のフレームワーク機能が必要であるため、アーキテクトは、FW 構成ファイル(TerasolunaFramework.config)および、AP 共通構成ファイル(TerasolunaApplication.config)に、UnityContainerExtesion 継承クラスの設定が必要である。設定方法は、各フレームワーク機能の機能説明書を参照のこと。

- 「CM-03 スレッド制御機能」
- 「CL-01 画面データ機能」
- 「CM-04 データコピー機能」
- 「CL-04 サーバ通信機能」

以下に、TerasolunaFramework.config および TerasolunaApplication.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する場合、下記の設定内容はすでに実装されている。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
・・・
  <unity>
・・・
    <containers>
      <container>
          <!-- 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" />
          <!-- 画面データ機能のデフォルト設定 -->
          <add type="Terasoluna.Windows.ViewModel.Validation.ValidatableViewDataExtension,
                    Terasoluna.Windows.ViewModel.Validation" />
          <!-- データ変換機能のデフォルト設定 -->
          <add type="Terasoluna.Windows.ViewModel.Validation.DataCopy.ValidatableDataCopyExtension,
                     Terasoluna.Windows.ViewModel.Validation" />
          <!-- 画面遷移機能のデフォルト設定 -->
          <add type="Terasoluna.Windows.Forms.FormForward.FormForwardExtension,
                    Terasoluna.Windows.Forms" />
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 18 TerasolunaFramework.config の記述例

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
・・・
  <unity>
    <containers>
      <container>
・・・
        <extensions>
          <!-- AOP機能の設定 -->
          <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" />
          <!-- ビジネスロジック実行機能の設定 -->
          <add type="Terasoluna.Windows.Forms.BizLogic.ClientBizLogicExtension,
                    Terasoluna.Windows.Forms" />
          <!-- イベント処理フロー定義のデフォルト設定 -->
          <add
           type="Terasoluna.Windows.ViewModel.Validation.Events.ValidatableEventProcessExtension,
                    Terasoluna.Windows.ViewModel.Validation" />
          <!—サーバ通信機能のデフォルト設定 -->
          <add type="Terasoluna.ServiceModel.CommunicationExtension,
                    Terasoluna" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 19 TerasolunaApplication.config の記述例

■ TIPS

◆ 画面データと DTO の自動コピーができない場合の対処

データの構造化や共通化が進むと、「画面データ」クラスと DTO の階層構造がずれる場合がある。たいていの場合は MappingInfo による個別のマッピング設定で対応可能であるが、対応できない場合は「画面データ」と DTO 間の自動コピーができないため、BuiltRequest イベントや ReflectedResponse イベントで、「画面データ」と DTO 間のデータコピー処理を個別に実装することが可能である。
以下に、BuiltRequest イベントを使った実装例を示す。
この例では、ルートとなる画面データ(SC_A01_01_01ViewData クラス)と、DTO(TourDto クラス)の階層構造が1階層ずれており、「画面データ(ネスト)」と DTO のプロパティが一致している。

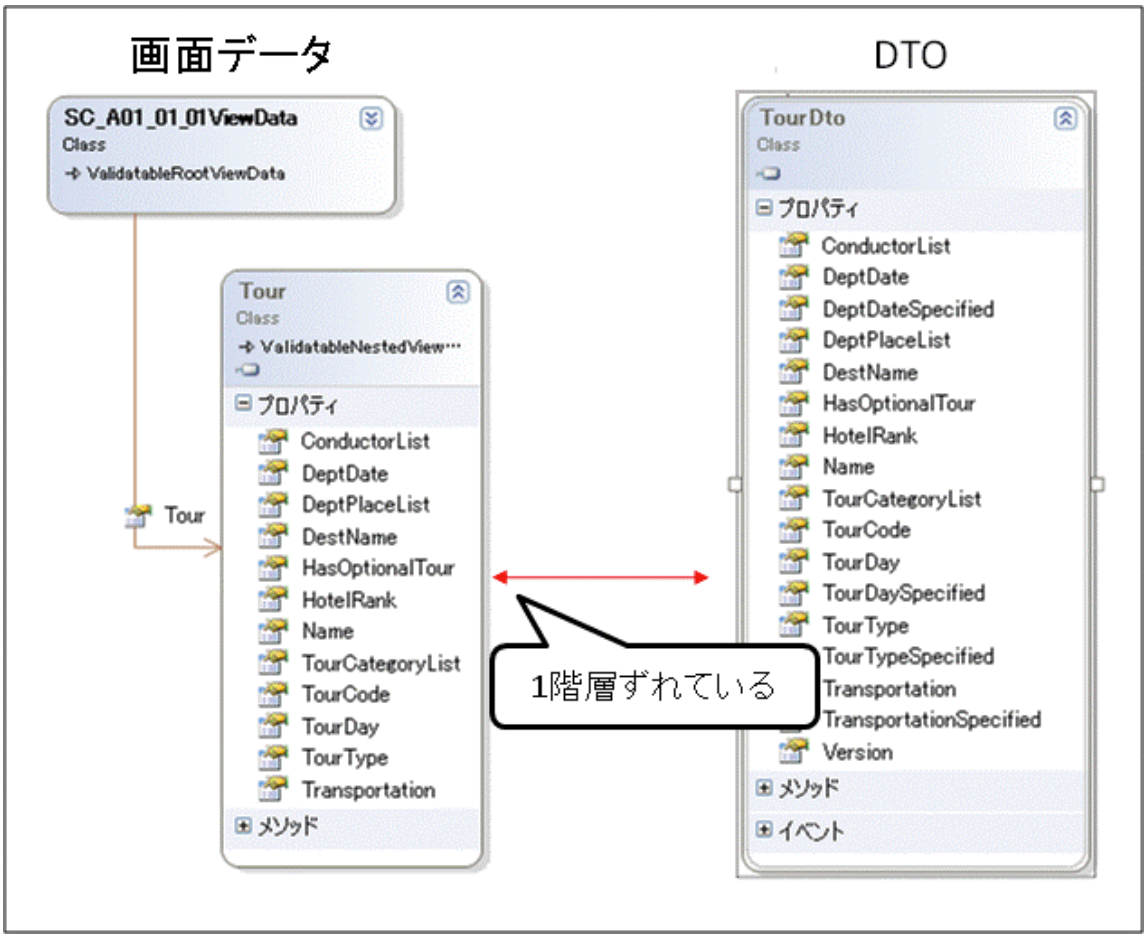


図 36 画面データと DTO のクラス階層がずれた例

クラスのルート階層に対するマッピングは、Visual Studio のデザイナ機能では設定できないため、この例では EventProcessWorker による自動コピーでは、何もコピーされない。このため、BuiltRequest イベント内で、DataCopyManager.Copy メソッドを使用することで、階層ずれのコピー処理を実装する。

```
[ScreenId("SC_A01_01_01")]
public partial class SC_A01_01_01View : SC_Z99_01View
{

    /// 要求データ生成完了時
    private void a01_01_01_C01EventProcessWorker_BuiltRequest(
            object sender, Terasoluna.Windows.Forms.Events.RequestDataBuiltEventArgs e)
    {
        ///階層ずれのデータのコピー処理
        SC_A01_01_01ViewData viewData = e.ViewData as SC_A01_01_01ViewData;
        TourDto dest = e.RequestData as TourDto;
        DataCopyManager.Copy(viewData.Tour, dest);
    }
}
```

リスト 20 BuiltRequest イベントの実装例

◆ ビジネスロジック内での UI 更新処理の実装

UI コントロールに対する処理は、画面クラスのイベントハンドラ内で実装し、ビジネスロジックでは原則 UI コントロールを更新するコードは記述するべきではない。しかし、ビジネスロジックの状況に応じて UI を更新する場合など、やむを得ずビジネスロジック内で UI を更新するコードを記述せざるを得ない場合がある。このような場合、「CM-03 スレッド制御機能」が提供する「非 UI スレッドからの透過的な UI スレッド実行」機能を利用し実装する。この際、以下のルールに従う。

- UI スレッドで実行するビジネスロジッククラスは、UnityContainer で管理されていること
  - EventProcessWorker.Execution.BizLogic プロパティで指定したビジネスロジックのインスタンス、またはそのビジネスロジックから DI されたインスタンスである必要がある。
  - 「CM-03 スレッド制御機能」は、「CM-02 インスタンス管理機能」の AOP をベースに実現されているため UnityContainer で管理されたインスタンスでなければならない。
- SetDefaultInterceptor 属性（または SetInterceptor 属性）が付与されていること
- UI スレッドとしてメソッドに、UICallHandler 属性が付与されていること

以下に、UI 更新処理を実施するビジネスロジックの実装例を示す。

この例では、ビジネスロジックが進むごとに、処理状況を画面上のプログレスバーとラベルに表示するサンプルを示す。
イベント処理の進捗通知は前述の EventProcessWorker.ProgressChanged イベントを利用することで実現可能であるが、この例ではロジック内で進捗率を精緻に計算／通知したいため、ビジネスロジックで直接 UI コントロールを更新する例になっている。

```
///UICallHandler属性を有効にするためにSetDefaultInterceptor属性を付与
[SetDefaultInterceptor(typeof(VirtualMethodInterceptor))]
public class MockVirusChecker : IVirusChecker
{
    protected const string RUNNING = "処理中";
   /// UIを更新するデリゲートが登録されるイベント
    public event EventHandler<ProgressChangedEventArgs> ProgressChanged;
    public VirusCheckResult CheckFile(string filePath)
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(20); ///ダミーの業務処理
           ///UIスレッドで実行したい処理の実行
            OnProgressChanged(new ProgressChangedEventArgs(i, RUNNING));
        }
        ・・・
    }
    /// UIスレッドで動作させる処理についてUICallHandler属性を付与
    [UICallHandler]
    protected virtual void OnProgressChanged(ProgressChangedEventArgs e)
    {
        EventHandler<ProgressChangedEventArgs> handler = ProgressChanged;
        if (handler != null)
        {
            /// イベントハンドラの実行
            handler(this, e);
        }
    }
}
```

リスト 21 UICallHanlder を使った処理の実装例

```
public class A01_04_01_C01BizLogic
{
    ///UnityContainerで管理されたオブジェクト
    [Dependency]
    public IVirusChecker VirusChecker { get; set; }

    public void Execute(A01_04_01_C01InputDto inputDto)
    {
        SC_A01_04_01View view =
            InvocationScope.Current.GetContext<EventProcessContext>().TargetForm
            as SC_A01_04_01View;
        if (view != null)
        {
            ///UIスレッド処理を実施するデリゲートをイベントに登録
            VirusChecker.ProgressChanged += view.VirusCheckProgressChanged;
            ・・・
        }
        ///UI更新を含むビジネスロジックの実行
        VirusCheckResult virusResult = VirusChecker.CheckFile(inputDto.FilePath);
        ・・・
    }
}
```

リスト 22 UI 更新処理を含むクライアントビジネスロジッククラスの実行例

```
[ScreenId("SC_A01_04_01")]
public partial class SC_A01_04_01View : SC_Z99_01View
{
・・・
    /// 実際のUI更新処理するメソッド
    public void VirusCheckProgressChanged(object sender, ProgressChangedEventArgs e)
    {
        virusCheckProgressBar.Value = e.ProgressPercentage;
        virusCheckResultLabel.Text = e.UserState as string;
    }
}
```

リスト 23 UI 更新処理の実装例

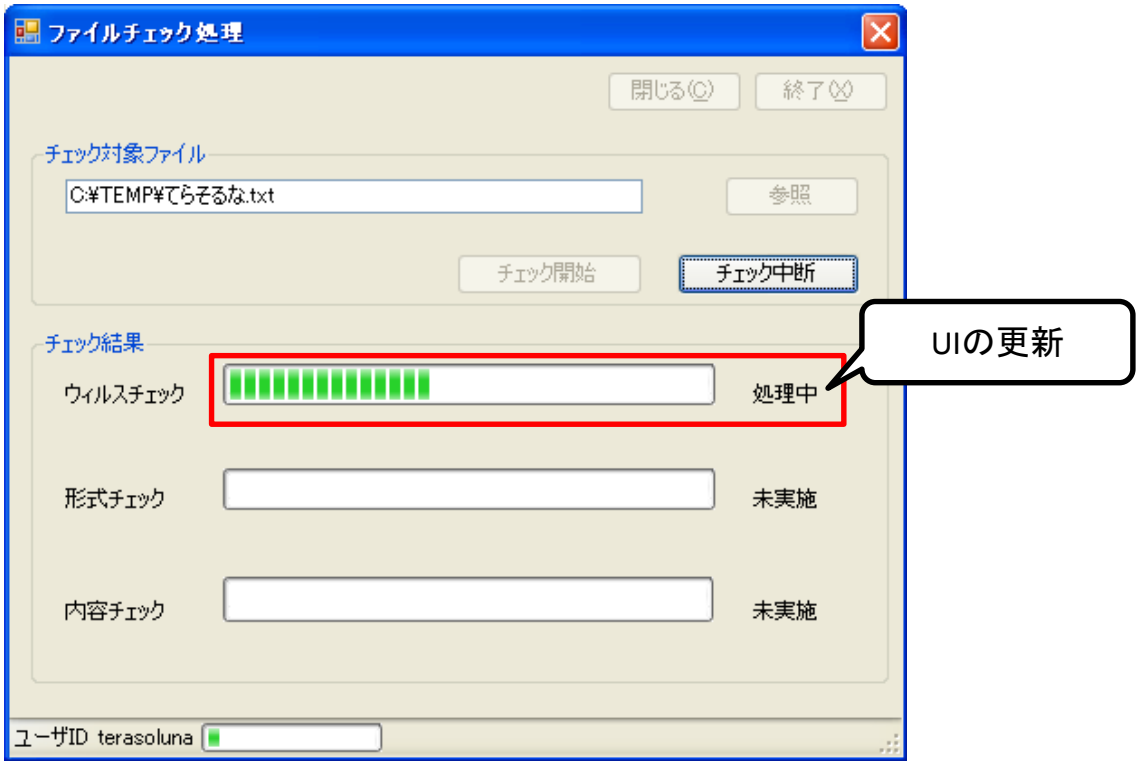


図 37 UI 更新処理の実行結果

◆ 非同期実行時の「入力値検証」と「リクエスト要求データ生成」

「入力値検証」フェーズと「リクエスト要求データ生成」フェーズはそれぞれ UI スレッドで処理されるが、イベント処理を非同期実行した場合、各フェーズ間のイベント処理フローの進行は別スレッドで実施される。
このため、「入力値検証」フェーズ完了後から「リクエスト要求データ生成」フェーズ開始までの間に、ユーザが画面より入力値を変更すると入力値検証が成功しないデータが送信される可能性がある。
この事象に対して厳密な対処を実施する場合は、LockControls プロパティを使用して対象となる入力フィールドをロックすることで回避可能である。
ただし、この事象を発生させるためには「入力値検証」フェーズと「リクエスト要求データ生成」フェーズの間に、ユーザが素早く入力値を変更する必要があり、厳密な対処を実施しなかった場合においても、この事象が発生する可能性は極めて低いと考えられる。開発するシステムに求められる信頼性等要件を考慮して対処策を講ずるかどうかを判断すること。

■ 内部構成

◆ クラス図

本機能の主要なクラスにおけるクラス図を示す。

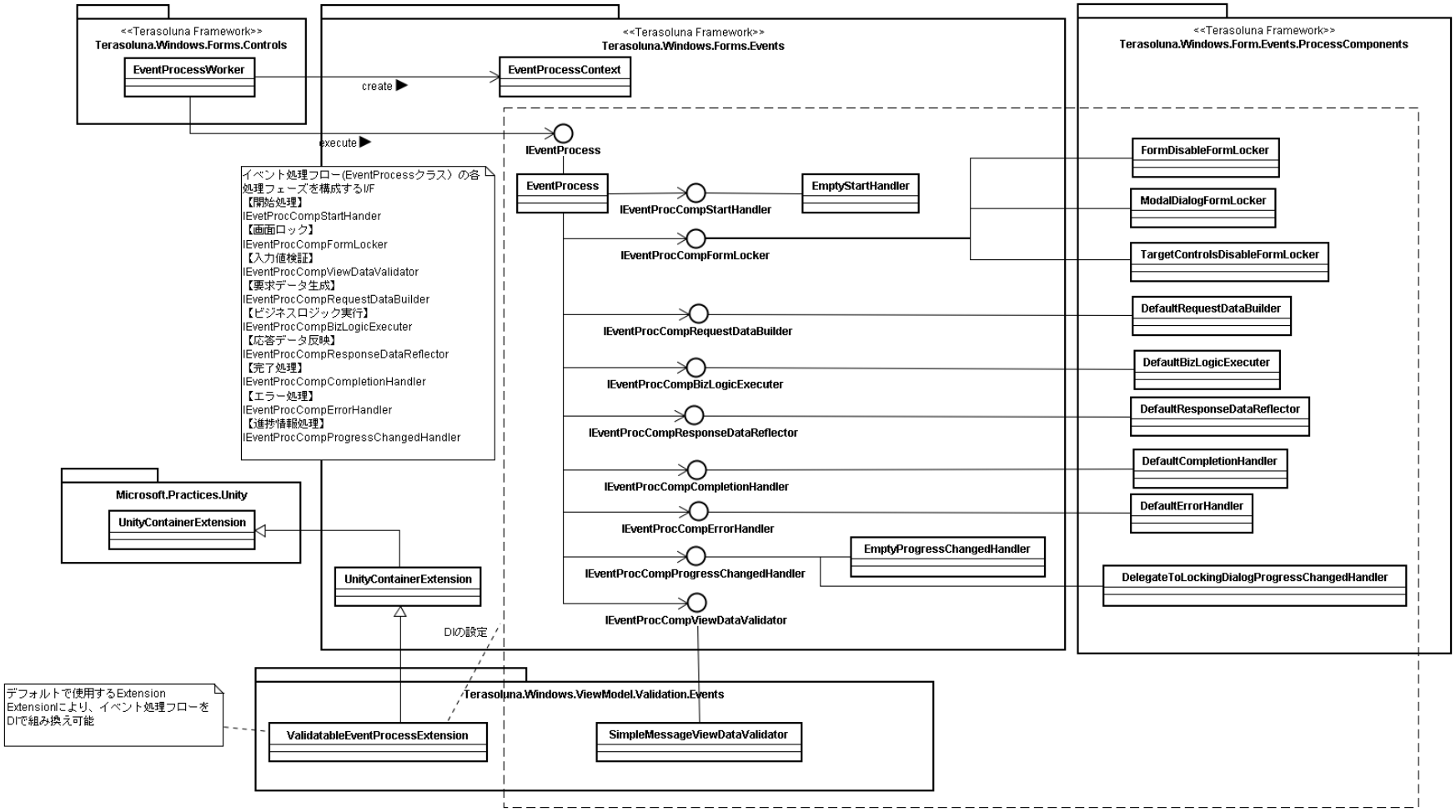


図 38 クラス図

◆ シーケンス図

イベント処理の同期実行の場合のシーケンス図を図 39、非同期実行の場合のシーケンス図を図 40 に示す。

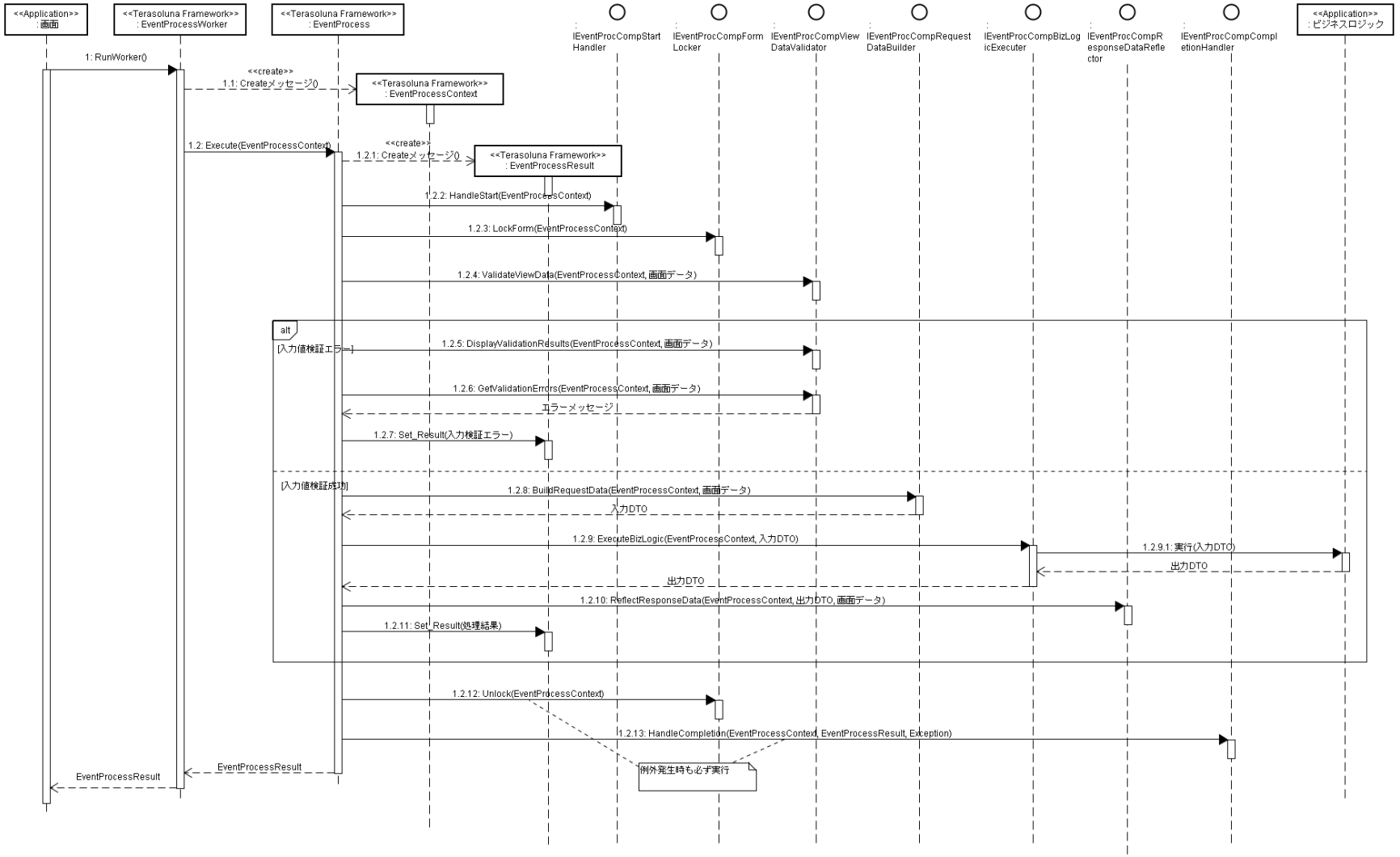


図 39 シーケンス図(イベント処理の同期実行の場合)

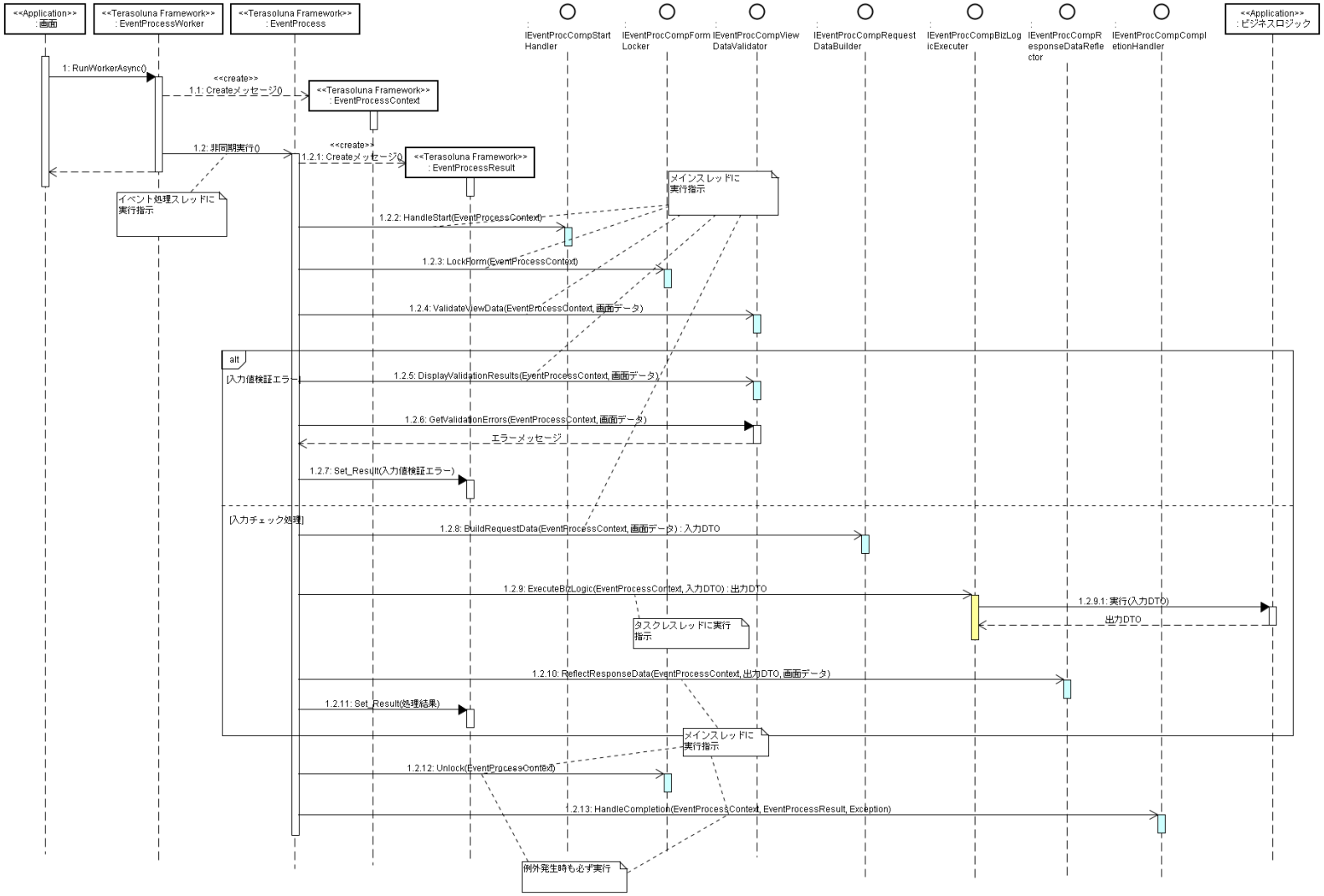


図 40 シーケンス図(イベント処理の非同期実行の場合)

◆ 構成クラス

本機能を構成するクラスを以下に示す。

表 27 構成クラス一覧

項番	クラス名	説明
Terasoluna.BizLogic 名前空間		
1	BizLogicInfo	ビジネスロジック情報を保持するクラス。
2	BizLogicInfoTypeConverter	BizLogicInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
3	BizLogicException	業務エラーを表す例外クラス。
4	BizLogicExceptionErrorType	発生した業務エラーの種別を定義するクラス。
5	IBizLogicExecutor	BizLogicInfo オブジェクトをもとにビジネスロジックを実行するインタフェース
6	IBizLogicInfoManager	BizLogicInfo オブジェクトをもとにビジネスロジック情報を取得するインタフェース
7	IBizLogicTypeFilter	サポートするビジネスロジッククラスであるか判定するインタフェース
8	BizLogicManager	ビジネスロジック実行のためのエントリクラス。
9	BizLogicInfoManagerBase	IBizLogicInfoManager 実装の基底クラス。
10	EmptyBizLogicTypeFilter	判定ロジックが空実装の IBizLogicTypeFilter 実装クラス。
11	PocoBizLogicExecutor	ユーザ定義クラスを実行する IBizLogciExecutor 実装クラス。
12	PocoBizLogicInfoManager	ユーザ定義クラスを実行する IBizLogicInfoManager 実装クラス。
13	PocoBizLogicTypeFilter	ユーザ定義クラスを実行する際の IBizLogicTypeFilter 実装クラス。
14	WcfBizLogicInfoManagerBase	WCF クライアントを実行する IBizLogicManager の基底クラス。
15	WcfChannelFactoryBizLogicExecutor	ChannelFactory クラスを使って WCF クライアントを実行する IBizLogicExecutor 実装クラス。
16	WcfChannelFactoryBizLogicInfoManager	ChannelFactory クラスを使って WCF クライアントを実行する IBizLogicManager 実装クラス。

17	WcfProxyBizLogicExecutor	「サービス参照の追加」によって生成されたプロキシ（ClientBase 派生）クラスを使って WCF クライアントを実行する IBizLogicExecutor 実装クラス。
18	WcfProxyBizLogicInfoManager	「サービス参照の追加」によって生成されたプロキシ（ClientBase 派生）クラスを使って WCF クライアントを実行する IBizLogicInfoManager 実装クラス。
19	BizLogicExtension	ビジネスロジック実行のための機能を有効するため UnityContainerExtension 継承クラス。
Terasoluna.Windows.Forms.Controls		
20	EventProcessWorker	本機能を実行するためのコンポーネントクラス。
Terasoluna.Windows.Forms.Events 名前空間		
21	IEventProcess	イベント処理フローを定義するインタフェース。
22	EventProcess	IEventProcess インタフェースのデフォルト実装クラス。
23	EventProcessContext	イベント処理に関する情報を保持するクラス。
24	EventProcResultTypes	イベント処理結果の種別を定義するクラス。
25	EventProcessResult	イベント処理の実行結果を保持するクラス。
26	EventProcState	イベント処理の実行状態を定義するクラス。
27	EventProcRunState	イベント処理が同期実行か非同期実行かの状態を定義するクラス。
28	EventProcEventArgs	EventProcessWorker で発生するイベント共通の EventArgs クラス。 EventProcessWorker.Started イベントの EventArgs クラス。
29	ViewDataValidatingEventArgs	EventProcessWorker.ViewDataValidating イベント時の EventArgs クラス。
30	ViewDataValidatedEventArgs	EventProcessWorker.ViewDataValidated イベント時の EventArgs クラス。
31	ViewDataValidationEventArgs	EventProcessWorker.DisplayedViewDataValidation イベント時の EventArgs クラス。
32	RequestDataBuildingEventArgs	EventProcessWorker. RequestDataBuilding 時の EventArgs クラス。
33	RequestDataBuiltEventArgs	EventProcessWorker. RequestDataBuilt イベント時の EventArgs クラス。

34	BizLogicExecutingEventArgs	EventProcessWorker.BizLogicExecuting イベント時の EventArgs クラス。
35	BizLogicExecutedEventArgs	EventProcessWorker.BizLogicExecuted イベント時の EventArgs クラス。
36	ResponseDataReflectingEventArgs	EventProcessWorker. ResponseDataReflecting イベント時の EventArgs クラス。
37	ResponseDataReflectedEventArgs	EventProcessWorker.ResponseDataReflected イベント時の EventArgs クラス。
38	EventProcCancelEventArgs	中断可能なイベントにおける基底の EvnetArgs クラス。
39	EventProcCompletedEventArgs	EventProcessWorker.Completed イベント時の EventArgs クラス。
40	EventProcCancelException	EventProcessWorker.CancelAsync メソッドにより、キャンセルが発生した場合の例外クラス。
41	EventProcInterruptedException	「～ing」イベントによりイベント処理を中断した場合の例外クラス。
42	EventProcException	イベント処理内で、予期しない例外が発生したことを通知するための例外クラス
43	EventProcUnexpectedServerException	イベント処理内で、予期せぬサーバエラーが発生したことを通知するための例外クラス。
44	IEventProcCompStartHandler	イベント処理フローの「開始処理」を実現するためのインタフェース。
45	IEventProcCompFormLocker	イベント処理フローの「画面ロック」を実現するためのインタフェース。
46	IEventProcCompViewDataValidator	イベント処理フローの「入力値検証」を実現するためのインタフェース。
47	IEventProcCompRequestDataBuilder	イベント処理フローの「要求データ生成」を実現するためのインタフェース。
48	IEventProcCompBizLogicExecuter	イベント処理フローの「ビジネスロジック実行」を実現するためのインタフェース。
49	IEventProcCompResponseDataReflector	イベント処理フローの「応答データ生成」を実現するためのインタフェース。
50	IEventProcCompCompletionHandler	イベント処理フローの「完了処理」を実現するためのインタフェース。
51	IEventProcCompErrorHandler	イベント処理フローの「エラー処理」を実現するためのインタフェース。
52	IEventProcCompProgressChangedHandler	イベント処理フローの「進捗情報処理」を実現するためのインタフェース。
53	EmptyStartHandler	IEventProcCompStartHandler の空実装クラス。

54	EmptyFormLocker	IEventProcCompFormLocker の空実装クラス。
55	EmptyViewDataValidator	IEventProcCompViewDataValidator の空実装クラス。
56	EmptyRequestDataBuilder	IEventProcCompRequestDataBuilder の空実装クラス。
57	EmptyBizLogicExecuter	IEventProcCompBizLogicExecuter の空実装クラス。
58	EmptyResponseDataReflector	IEventProcCompResponseDataReflector の空実装クラス。
59	EmptyCompletionHandler	IEventProcCompCompletionHandler の空実装クラス。
60	EmptyErrorHandler	IEventProcCompErrorHandler の空実装クラス。
61	EmptyProgressChangedHarandler	IEventProcCompProgressChangedHandler の空実装クラス。
62	EventProcEventReporter	IEventProcCompErrorHandler の空実装クラス。
63	EventProcessInfo	イベント処理に関する設定情報を保持するクラス
64	IEventProcessInfoManager	イベント処理に関する情報を管理するためのインタフェース。
65	EventProcessInfoManager	IEventProcessInfoManager のデフォルト実装クラス。
66	EventProcessManager	イベント処理に関するインタフェースに対する実装オブジェクトを管理するクラス。
67	EventProcProgressInfo	イベント処理の進捗情報を保持するクラス。
68	IEventProcEventReporter	イベント処理で発生するイベントの通知を制御するためのインタフェース。
69	IEventProcProgressInfoManager	イベント処理の進捗情報を管理するためのインタフェース。
70	IEventProcProgressManager	.NET 標準の進捗通知イベント(ProgressChangedEventHandler)に、一定間隔で通知する機能を提供するインタフェース。
71	EventProcProgressInfoManager	IEventProcProgressInfoManager インタフェースのデフォルト実装クラス。
72	EventProcProgressManager	IEventProcProgressManager インタフェースのデフォルト実装クラス。
73	LockControlInfo	ロック対象となるコントロールについての情報を保持するクラス。

74	LockControlInfoConverter	LockControlInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
75	ViewDataValidationInfo	画面データを入力値検証するのに必要な情報を保持するクラス。
76	ViewDataValidationInfoConverter	ViewDataValidationInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
77	RequestDataBuildInfo	要求データを生成するのに必要な情報を保持するクラス。
78	RequestDataBuildInfoConverter	RequestDataBuildInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
79	BizLogicExecutionInfo	ビジネスロジック実行に必要な情報を保持するクラス
80	BizLogicExecutionInfoConverter	BizLogicExecutionInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
81	ResponseDataReflectionInfo	応答データを画面に反映するのに必要な情報を保持するクラス。
82	ResponseDataReflectionInfoConverter	ResponseDataReflectionInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
83	ProgressInfo	進捗情報を保持するクラス
84	ProgressInfoConverter	ProgressInfo を Visual Studio のデザイナでシリアライズ可能にするための TypeConverter。
85	PropertyInstanceMapping	EventProcessExtension で登録する プロパティのインスタンス名を保持するクラス。
86	PropertyValueMapping	EventProcessExtension で登録する プロパティに設定する値を保持するクラス。
87	EventProcessExtension	イベント定義のための UnityContainerExtension 継承クラス。
Terasoluna.Windows.Forms.Events.ProcessComponents 名前空間		
88	DelegateStartHandler	EventProcessContext の情報に基づき取得した IEventProcCompStartHandler 実装クラスに処理を委譲する IEventProcCompStartHandler 実装クラス。
89	DelegateFormLocker	EventProcessContext の情報に基づき取得した IEventProcCompFormLocker 実装クラスに処理を委譲する IEventProcCompFormLocker 実装クラス。

90	DelegateViewDataValidator	EventProcessContext の情報に基づき取得した IEventProcCompViewDataValidator 実装クラスに処理を委譲する IEventProcCompViewDataValidator 実装クラス。
91	DelegateRequestDataBuilder	EventProcessContext の情報に基づき取得した IEventProcCompRequestDataBuilder 実装クラスに処理を委譲する IEventProcCompRequestDataBuilder 実装クラス。
92	DelegateBizLogicExecuter	EventProcessContext の情報に基づき取得した IEventProcCompBizLogicExecuter 実装クラスに処理を委譲する IEventProcCompBizLogicExecuter 実装クラス。
93	DelegateResponseDataReflector	EventProcessContext の情報に基づき取得した IEventProcCompResponseDataReflector 実装クラスに処理を委譲する IEventProcCompResponseDataReflector 実装クラス。
94	DelegateCompletionHandler	EventProcessContext の情報に基づき取得した IEventProcCompCompletionHandler 実装クラスに処理を委譲する IEventProcCompCompletionHandler 実装クラス。
95	DelegateErrorHandler	EventProcessContext の情報に基づき取得した IEventProcCompErrorHandler 実装クラスに処理を委譲する IEventProcCompErrorHandler 実装クラス。
96	DelegateProgressChangedHandler	EventProcessContext の情報に基づき取得した IEventProcCompProgressChangedHandler 実装クラスに処理を委譲する IEventProcCompProgressChangedHandler 実装クラス。
97	DelegateToFormStartHandler	画面クラスに処理を委譲する DelegateStartHandler クラス。
98	DelegateToFormFormLocker	画面クラスに処理を委譲する DelegateFormLocker クラス。
99	DelegateToFormViewDataValidator	画面クラスに処理を委譲する DelegateViewDataValidator クラス。
100	DelegateToFormRequestDataBuilder	画面クラスに処理を委譲する DelegateRequestDataBuilder クラス。
101	DelegateToFormBizLogicExecuter	画面クラスに処理を委譲する DelegateBizLogicExecuter クラス。
102	DelegateToFormResponseDataReflector	画面クラスに処理を委譲する DelegateResponseDataReflector クラス。
103	DelegateToFormCompletionHandler	画面クラスに処理を委譲する DelegateCompletionHandler クラス。

104	DelegateToFormErrorHandler	画面クラスに処理を委譲する DelegateErrorHandler クラス。
105	DelegateToFormProgressChangedHandler	画面クラスに処理を委譲する DelegateProgressChangedHandler クラス。
106	DevelopementTimeCompletionHandler	開発時用に、デバッグ情報をダイアログ表示する IEventProcCompCompletionHandler クラス。
107	FormDisableFormLocker	フォームを無効化することで画面ロックする IEventProcCompFormLocker 実装クラス。
108	ModalDialogFormLocker	ダイアログを表示することで画面ロックする IEventProcCompFormLocker 実装クラス。
109	DelegateToLockingDialogProgressChangedHandler	進捗ダイアログに処理を委譲する DelegateProgressChangedHandler クラス。
110	ICancelableForm	キャンセル可能な画面にイベント処理の実行時情報を渡すためのインタフェース。
111	RunningForm	ダイアログによる画面ロックをする場合に利用するダイアログクラス。 IEventProcCompProgressChangedHandler と ICancelableForm を実装する。
112	TargetControlsDisableFormLocker	指定したコントロールを無効化することで画面ロックする IEventProcCompFormLocker 実装クラス。
113	DefaultRequestDataBuilder	IEventProcCompRequestDataBuilder インタフェースのデフォルト実装クラス。 「CM-04 データコピー機能」を利用し要求データを生成する。
114	DefaultBizLogicExecuter	IEventProcCompBizLogicExecuter インタフェースのデフォルト実装クラス。 Terasoluna.BizLogic 名前空間の実装クラスを利用してビジネスロジックを実行する。
115	DefaultResponseDataReflector	IEventProcCompResponseDataReflector インタフェースのデフォルト実装クラス。 「CM-04 データコピー機能」を利用し応答データを反映する。
116	DefaultCompletionHandler	IEventProcCompCompletionHandler インタフェースのデフォルト実装クラス。
117	DefaultErrorHandler	IEventProcCompErrorHandler インタフェースのデフォルト実装クラス。
Terasoluna.Windows.Forms.Events.Extensions 名前空間		
118	DelegateToFormEventProcessExtension	イベント処理の各フェーズを画面クラスに委譲するイベント処理を定義した EventProcessExtension クラス。

119	DevelopementTimeEventProcessExtension	開発時用に、完了フェーズでデバッグ情報をダイアログ表示するイベント処理を定義した EventProcessExtension クラス。
120	DefaultEventProcessExtension	標準的なイベント処理を定義した EventProcessExtension クラス。
Terasoluna.Windows.ViewModel.Validation.Events 名前空間		
121	SimpleMessageViewDataValidator	本機能のデフォルトの IEventProcCompViewDataValidator。「CL-01 画面データ」機能を実装した「画面データ」クラスを「CM-05 入力値検証機能」で入力値検証する。
122	ValidatableDelegateToFormEventProcessExtension	入力値検証機能を可能なイベント処理を定義した DelegateToFormEventProcessExtension クラス。
123	ValidatableEventProcessExtension	入力値検証機能を可能なイベント処理を定義した DefaultEventProcessExtension。本機能のデフォルトの EventProcessExntesion クラス。

■ 拡張ポイント

◆ イベント処理フローの変更・追加

EventProcessWorker.EventProcessName プロパティで選択可能なイベント処理フローの定義を変更するには、イベント処理定義のための Extension クラス（EventProcessExtension 継承クラス）で、イベント処理フローの構成要素となる各インタフェースを実装し、標準実装クラスの代わりに DI で差し替える。

以下に、イベント処理フローを変更した実装例を示す。
これは、イベント処理フローの「入力値検証処理」フェーズを実装するインタフェースである IEventProcCompViewDataValidator インタフェースのデフォルトの実装クラスを SampleMessageViewDataValidator クラスに差し替えた例である。
デフォルトの Extension クラスである ValidatableEventProcessExtension を継承し、SetupEventProcessComponents メソッドをオーバーライドし、RegisterType メソッドでデフォルトの DI 設定を上書きする。

```
public class SampleEventProcessExtension：ValidatableEventProcessExtension
{
        protected override void SetupEventProcessComponents()
        {
            base.SetupEventProcessComponents();
            ///イベント処理フロー中の入力チェックエラー機能のDI設定を上書き
            Container.RegisterType<IEventProcCompViewDataValidator,
SampleMessageViewDataValidator>(
                    new ContainerControlledLifetimeManager());
        }
    }
}
```

リスト 24　イベント処理フローを変更した Extension の実装例

「入力値検証処理」フェーズの実装は、IEventProcCompViewDataValidator インタフェースを実装するが、デフォルト実装クラスである SimpleMessageViewDataValidator を継承するとよい。

```
public class SampleMessageViewDataValidator：SimpleMessageViewDataValidator
{
    public override void DisplayValidationResults(
                            EventProcessContext context, object viewData)
    {
        //入力チェックエラー時のダイアログ表示を要件に合わせて変更
    }
}
```

リスト 25　IEventProcCompViewDataValidator の実装イメージ

なお、他のフェーズの拡張についても、同様に各種インタフェースを実装したクラスを作成すればよい。この際、デフォルトの実装クラスは、拡張ポイントが細かく分割されているため、変更したいポ

イントのみを上書き(オーバーライド)するよう実装することで、拡張のコストを最小化できる。
また、イベント処理フローを追加する場合には、SetupEventProcessComponents メソッドをオーバーライドし、各処理フェーズを実装したコンポーネントを登録後、SetupEventProcesses メソッドをオーバーライドして、RegisterEventProcess メソッドで、イベント処理フローを実現するコンポーネントの組み合わせを定義すればよい。
以下に、イベント処理フローを追加した例を示す。
RegisterEventProcess メソッドの第2引数以降の可変パラメータは Terasoluna.Windows.Forms.Events.PropertyInstanceMapping オブジェクトを列挙し、各処理フェーズを定義している。この例では全ての処理フェーズを定義しているが、差し替えたい処理フェーズのみを定義するだけでもよい。最低限の定義として「画面ロック」のコンポーネントについての PropertyInstanceMapping オブジェクトが指定されていればよい。
PropertyInstanceMapping のコンストラクタの第1引数には、イベント名を表す文字列を渡す。文字列は、表 28 のように処理フェーズに対応する EventProcessExtension クラスの定数として定義されている。
第2引数には、追加したコンポーネントの登録名(UnityContainer.RegisterType メソッドで指定した名前)を必要な分指定する。

```
public class SampleEventProcessExtension : ValidatableEventProcessExtension
{
    protected override void SetupEventProcesses()
    {
        base.SetupEventProcesses();
        ・・・

        ///イベント処理フローの追加定義
        ///イベント処理フローを実現するコンポーネントをDIで組み合わせる
        RegisterEventProcess<EventProcess>("NewEventProcess",
            ///開始処理
            new PropertyInstanceMapping(EventProcPropStartHandler, "NewStartHandler"),
            ///画面ロック
            new PropertyInstanceMapping(EventProcPropFormLocker, "NewFormLock"),
            ///入力値検証
            new PropertyInstanceMapping(EventProcPropFormLocker, "NewViewDataValidator"),
            ///要求データ生成
            new PropertyInstanceMapping(EventProcPropRequestDataBuilder,
                "NewRequestDataBuilder"),
            ///ビジネスロジック実行
            new PropertyInstanceMapping(EventProcPropBizLogicExecuter, "NewBizLogicExecuter"),
            ///応答データ反映
            new PropertyInstanceMapping(EventProcPropResponseDataReflector,
                "NewResponseDataReflector"),
            ///完了処理
            new PropertyInstanceMapping(EventProcPropCompletionHandler,
                "NewCompletionHandler"),
            ///エラー処理
            new PropertyInstanceMapping(EventProcPropErrorHandler, "NewErrorHandler"),
            ///進捗情報処理
            new PropertyInstanceMapping(EventProcPropProgressChangedHandler,
                "NewProgressChangedHandler")
            );
    }
・・・
```

```
・・・
    /// イベント処理フローを各フェーズを構成するコンポーネントを登録
    protected override void SetupEventProcessComponents()
    {
        base.SetupEventProcessComponents();

        ///開始処理のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompStartHandler, SampleStartHandler>(
            "NewStartHandler",
            new ContainerControlledLifetimeManager());
        ///画面ロックのコンポーネントを追加定義
        Container.RegisterType<IEventProcCompFormLocker, SampleFormLocker>(
            "NewFormLock",
            new ContainerControlledLifetimeManager());
        ///入力値検証のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompViewDataValidator,
            SampleMessageViewDataValidator>(
            "NewViewDataValidator",
            new ContainerControlledLifetimeManager());
        ///要求データ生成のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompRequestDataBuilder, SampleRequestDataBuilder>(
            "NewRequestDataBuilder",
            new ContainerControlledLifetimeManager());
        ///ビジネスロジック実行のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompBizLogicExecuter, SampleBizLogicExecuter>(
            "NewBizLogicExecuter",
            new ContainerControlledLifetimeManager());
        ///応答データ反映のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompResponseDataReflector,
            SampleResponseDataReflector>(
            "NewResponseDataReflector",
            new ContainerControlledLifetimeManager());
        ///完了処理のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompCompletionHandler, SampleCompletionHandler>(
            "NewCompletionHandler",
            new ContainerControlledLifetimeManager());
        ///エラー処理のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompErrorHandler, SampleErrorHandler>(
            "NewErrorHandler",
            new ContainerControlledLifetimeManager());
        ///進捗情報処理のコンポーネントを追加定義
        Container.RegisterType<IEventProcCompProgressChangedHandler,
            SampleProgressChangedHandler>(
            "NewProgressChangedHandler",
            new ContainerControlledLifetimeManager());
    }
```

リスト 26 イベント処理フローの追加定義した Extension の実装例

表 28 EventProcessExtension クラスの定数値

項番	変数名	対応する処理フェーズ
1	EventProcPropStartHandler	開始処理
2	EventProcPropFormLocker	画面ロック処理
3	EventProcPropViewDataValidator	入力値検証処理
4	EventProcPropRequestDataBuilder	要求データ生成処理
5	EventProcPropBizLogicExecuter	ビジネスロジック実行処理
6	EventProcPropResponseDataReflector	応答データ反映処理
7	EventProcPropCompletionHandler	完了処理
8	EventProcPropErrorHandler	エラー処理
9	EventProcPropProgressChangedHandler	進捗通知処理

Extension クラス作成後、AP 共通構成ファイル(TerasolunaApplicaiton.config)でデフォルトの Extension クラスの代わりに設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
・・・
  <unity>
    <containers>
      <container>
・・・
        <extensions>
          <!-- イベント処理フロー定義のプロジェクト拡張設定 -->
          <add type="TourSampleAppCommon.Event.SampleEventProcessExtension,
                    TourSampleAppCommon" />
・・・
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 27 TerasolunaApplication.config の記述例

ビルド後、EventProcessWorker の EventProcessName プロパティで、追加したイベント処理フローを選択することができる。
プルダウンメニューには、RegisterEventProcess メソッドの第1引数に指定した文字列が表示される。
なお、デフォルトのイベント処理フロー(EventProcessName プロパティを未指定時)を変更したい場合は、RegisterEventProcess メソッドの第1引数を null で指定する。

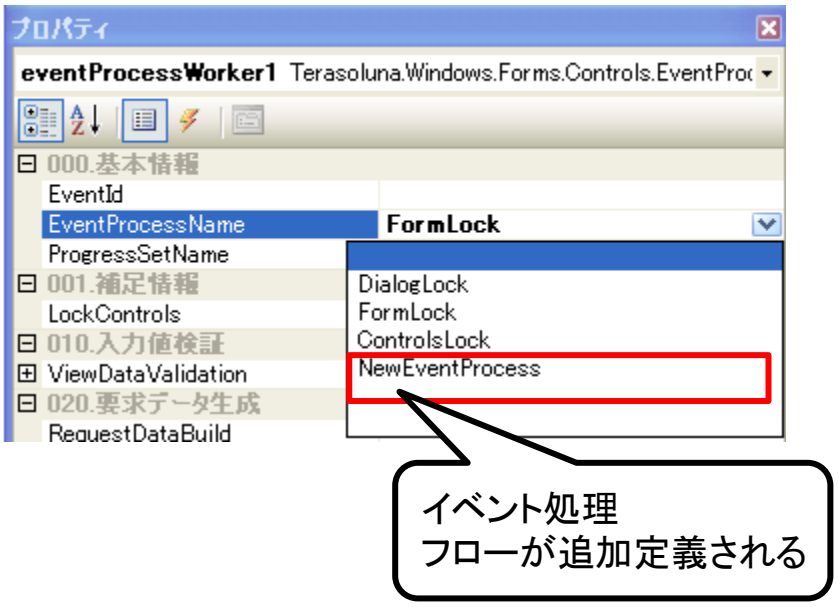


図 41 EventProcessWorker.EventProcessName プロパティの表示

◆ 進捗率計算方法の定義の変更・追加

EventProcessWorker.ProgressSetName プロパティで選択可能な進捗率の計算方法を追加・変更するには、イベント処理定義のための Extension クラス(EventProcessExtension 継承クラス)で、定義を追加する。

以下に、実装例を示す。
まず、SetupProgressSets メソッドをオーバーライドし、RegisterEventProgressSets メソッドにより進捗率の計算方法を追加する。
RegisterEventProgressSets メソッドの第2引数以降の可変パラメータは Terasoluna.Windows.Forms.Events.PropertyInstanceMapping オブジェクトを列挙し、各処理フェーズが取りうる進捗率を変更している。この例では全ての処理フェーズで定義しているが、進捗率の計算が不要な処理フェーズは記述不要である。
PropertyInstanceMapping のコンストラクタの第1引数には、各フェーズを表す文字列を渡す。文字列は、表 29 のように処理フェーズに対応する EventProcessExtension クラスの定数として定義されている。
第2引数には、Terasoluna.Windows.Forms.Events.ProgressInfo オブジェクトを指定して、開始時の進捗率(%)、終了時の進捗率(%)、進捗率の増分(%)、進捗率をカウントアップする時間間隔(ms)を指定する。

```
public class SampleEventProcessExtension : ValidatableEventProcessExtension
{
    /// 進捗率計算方法の追加定義
    protected override void SetupEventProgressSets()
    {
        base.SetupEventProgressSets();

        RegisterEventProgressSet( "NewProgressSet",
            ///イベント処理開始(0%)
            new PropertyValueMapping<ProgressInfo>(
                ProgressInfoPropStart, new ProgressInfo(0)),
            ///入力値検証(10%～20%)
            new PropertyValueMapping<ProgressInfo>(
                ProgressInfoPropViewDataValidation, new ProgressInfo(10, 20)),
            ///要求データ生成(20%～30%)
            new PropertyValueMapping<ProgressInfo>(
                ProgressInfoPropRequestDataBuild, new ProgressInfo(20, 30)),
            ///ビジネスロジック実行(30%～90%、100msごとに1%ずつカウントアップ)
            new PropertyValueMapping<ProgressInfo>(
                ProgressInfoPropBizLogicExecution, new ProgressInfo(30, 90, 1, 100)),
            ///応答データ反映（90%～100%）
            new PropertyValueMapping<ProgressInfo>(
                ProgressInfoPropResponseDataReflection, new ProgressInfo(90, 100)),
            ///イベント処理終了(100%)
            new PropertyValueMapping<ProgressInfo>(
                ProgressInfoPropCompletion, new ProgressInfo(100)));
        }
```

リスト 28 進捗率の計算方法を追加定義した Extension の実装例

表 29 EventProcessExtension クラスの定数値

項番	変数名	対応する処理フェーズ
1	ProgressInfoPropStart	イベント開始時
2	ProgressInfoPropViewDataValidation	入力値検証処理
3	ProgressInfoPropRequestDataBuild	要求データ生成処理
4	ProgressInfoPropBizLogicExecution	ビジネスロジック実行処理
5	ProgressInfoPropResponseDataReflection	応答データ反映処理
6	ProgressInfoPropCompletion	応答データ反映処理完了後

表 30 ProgressInfo の主要なコンストラクタ

項番	コンストラクタ	第１引数	第２引数	第３引数	第４引数
1	public ProgressInfo(int startPercentage)	開始時の進捗率	-	-	-
2	public ProgressInfo(int startPercentage, int endPercentage)	開始時の進捗率	終了時の進捗率	-	-
3	public ProgressInfo(int startPercentage, int endPercentage, int incrementStep, int incrementInterval)	開始時の進捗率	終了時の進捗率	進捗率の増分	カウントアップする時間間隔(ms)

Extension クラス作成後、AP 共通構成ファイル(TerasolunaApplicaiton.config)でデフォルトの Extension クラスの代わりに設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
・・・
  <unity>
    <containers>
      <container>
・・・
        <extensions>
          <!-- イベント処理フロー定義のプロジェクト拡張設定 -->
          <add type="TourSampleAppCommon.Event.SampleEventProcessExtension,
                    TourSampleAppCommon" />
・・・
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 29 TerasolunaApplication.config の記述例

ビルド後、EventProcessWorker の EventProgressSetName プロパティで、追加した進捗率の計算方法を選択することができる。
プルダウンメニューには、RegisterEventProgressSet メソッドの第1引数に指定した文字列が表示される。
なお、デフォルトの進捗率の計算方法(EventProgressSetName プロパティを未指定時)を変更したい場合は、RegisterEventProgressSet メソッドの第1引数を null で指定する。

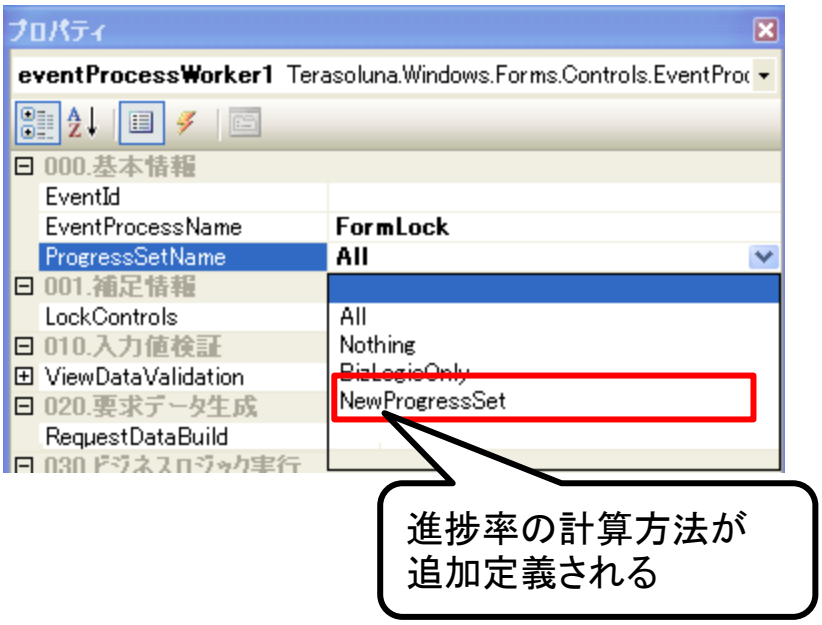


図 42 EventProcessWorker.ProgressSetName プロパティの表示

◆ 標準提供のダイアログ上の固定メッセージの変更

エラーダイアログや進捗ダイアログやキャンセル時のダイアログなど、本機能が標準で提供するダイアログの「メッセージ」内容は、フレームワーク内で定義されたリソースファイル(TerasolunaWindows.Forms.dll および Terasoluna.ViewModel.Validation.dll の DisplayResources.resx)でデフォルトメッセージを管理している。「メッセージ」内容をデフォルト定義のものから変更する場合は、「CM-07 メッセージ管理機能」により、デフォルトメッセージの置き換えを実施する。
詳細な手順については、「CM-07 メッセージ管理機能」の機能説明書を参照のこと。

■ 関連機能

- CM-02 インスタンス管理機能
- CM-03 スレッド制御機能
- CL-01 画面データ機能
- CM-05 入力値検証機能
- CM-04 データコピー機能
- CL-04 サーバ通信機能
- CL-05 クライアントエラーハンドリング機能
- CM-07 メッセージ管理機能