CL-05 クライアントエラーハンドリング機能

■ 概要

クライアント AP で、以下のエラーを集約的に処理する機能を提供する。

- 入力値検証エラー
  - 必須入力チェックや形式チェックなど、画面や電文の入力値のみから発生するエラー。ユーザの再入力により復帰可能。
- 業務エラー
  - DB データとの突き合わせチェックや複雑なビジネスルールによるチェックなど、ビジネスロジック処理において発生するエラー。ユーザの再入力により復帰可能。
- システムエラー
  - DB やネットワークのエラーなど、システムや AP の不具合による復帰不可能なエラー。

なお、本機能では、クライアント AP 内部で発生したエラーだけでなく、サーバから受信したエラー電文により、サーバで発生したエラーも一元的に処理する。

- クライアント入力値検証エラー

  「CL-03 イベント処理実行機能」のイベント処理フローの「入力値検証」フェーズにおいて、入力値検証エラーが発生した場合にユーザに入力値検証エラーを通知するよう集約的に処理することができる。「入力値検証」フェーズを実装するインタフェースとして IEventProcCompViewDataValidator インタフェースが用意されている。TERASOLUNA フレームワークでは標準実装として、SimpleMessageViewDataValidator を提供し、イベント処理フローに組み込んでいる。

  SimpleMessageViewDataValidator では、「CL-06 メッセージ通知機能」を利用して、ユーザへ入力値検証エラーが発生した旨を表示する。

  また、入力値検証エラーの詳細なメッセージは、「CL-01 画面データ機能」を利用して、入力エラー対象の UI コントロールごとに ErrorProvider クラスでエラー原因を表示する。詳細は、「CL-01 画面データ機能」の機能説明書を参照のこと。

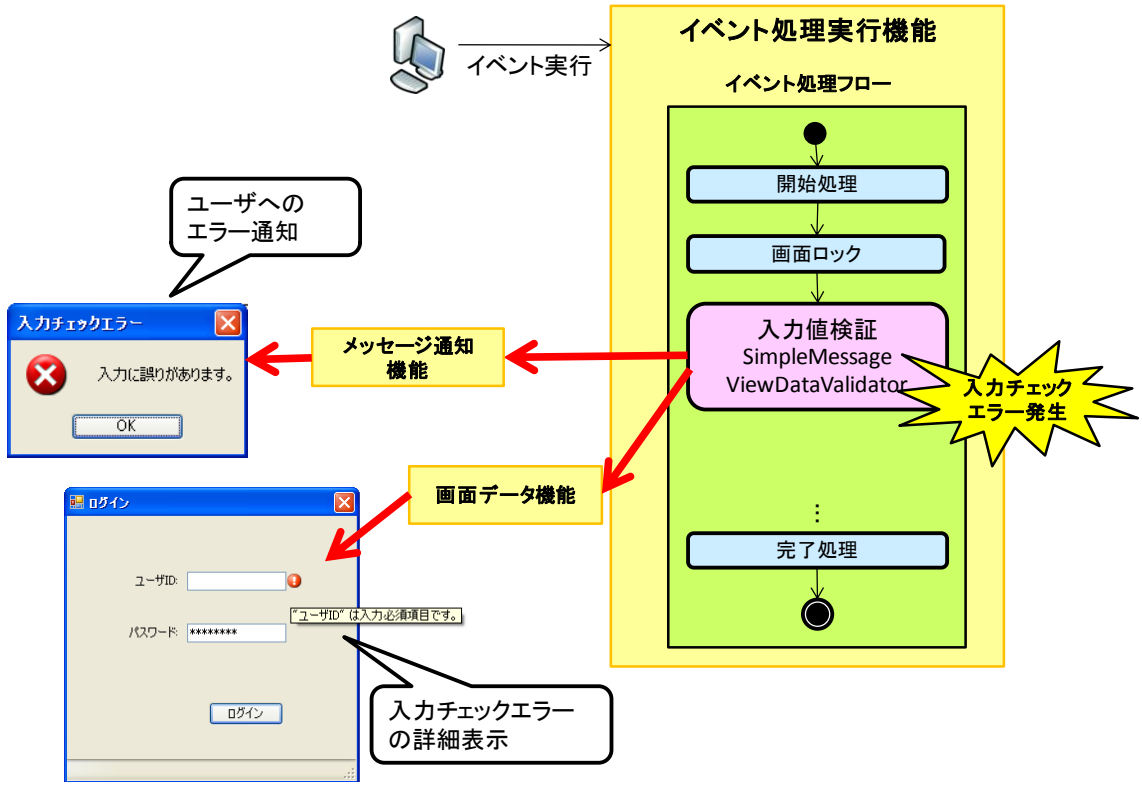


図 1 イベント実行中に発生した画面入力値検証エラーのハンドリング

- クライアント業務エラー

.NET には検査例外の機構がないため、業務エラーはメソッドの戻り値で扱うことが一般的である。

一方、TERASOLUNA フレームワークにおけるビジネスロジッククラスの実行は、「CL-03 イベント処理実行機能」により自動実行される。TERASOLUNA フレームワークでは、ビジネスロジッククラスを POCO(Plain Old CLR Object)で実装でき、開発者にとって柔軟な開発が可能なアーキテクチャを採用しているため、戻り値だけで業務エラーかどうかをフレームワークが一律判定することは困難である。

そこで、TERASOLUNA フレームワークでは、ビジネスロジッククラスの中で業務エラーを表す例外(BizLogicException)をスローすることで集約的に業務エラーを処理する機能を提供する。「CL-03 イベント処理実行機能」において、イベント処理フローの「エラー処理」フェーズを実装するインタフェースとして IEventProcCompErrorHandler を用意しており、標準実装として、DefaultErrorHandler クラスがイベント処理フローに組み込まれている。

DefaultErrorHandler クラスは、ビジネスロジッククラスで BizLogicException がスローされた場合に業務エラーとして判定し、BizLogicException 以外の例外であれば、例外をリスローし、システムエラーとして集約例外処理をする。(システムエラーについては後述)

業務エラーの場合は、リスローせず、イベント処理結果(EventProcessResult クラス)を「業務エラー」に設定し、エラー情報を結果に格納する。

業務エラー時の後処理は、通常、「CL-03 イベント処理実行機能」におけるイベント処理フローの「完了処理」フェーズで実施する。「完了処理」フェーズでは、業務エラーだけでなく、正

常終了、キャンセル終了など、イベント処理の実行結果を判定し、後処理を実行する。
「完了処理」を実装するインタフェースとして IEventProcCompCompletionHandler が用意されており、本機能の標準では、IEventProcCompCompletionHandler を実装した DefaultCompletionHandler クラスを提供し、イベント処理フローに組み込んでいる。
DefaultCompletionHanlder クラスは、業務エラー時には、「CL-06 メッセージ通知機能」により、ユーザにエラーメッセージを表示する。

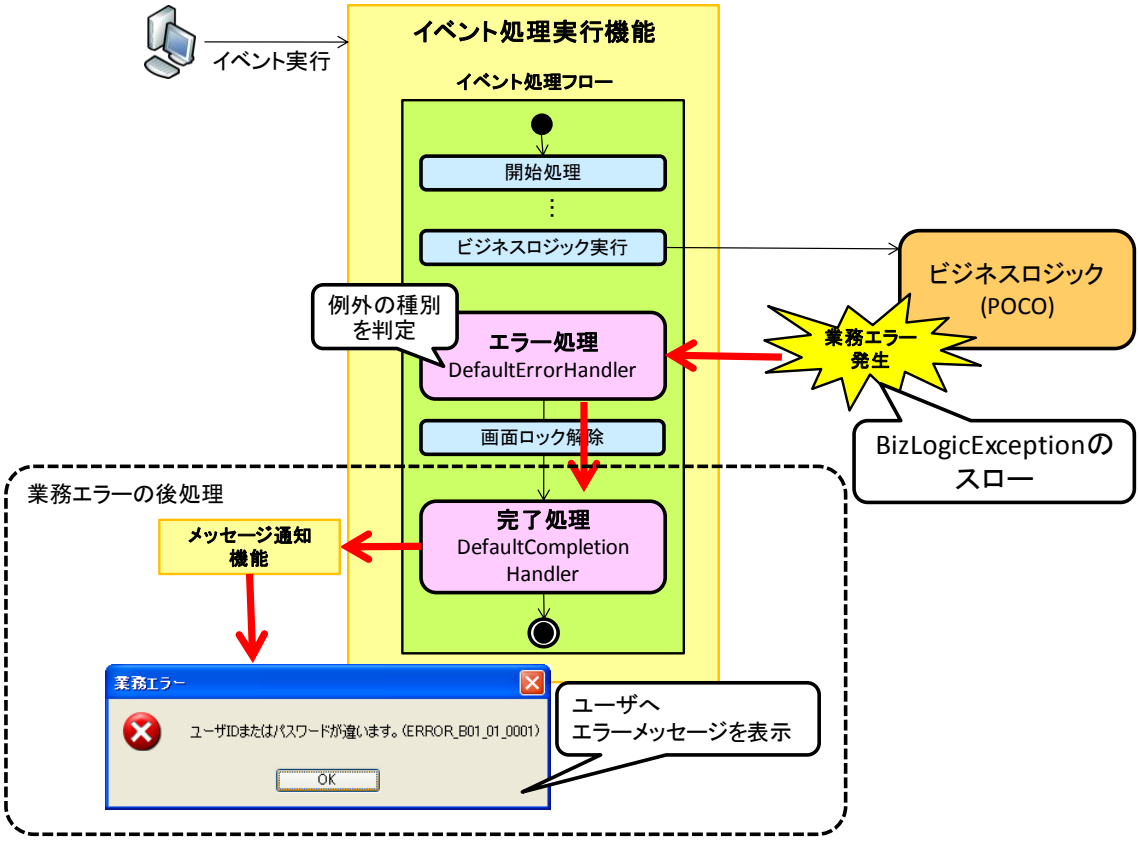


図 2 イベント処理実行中に発生した業務エラーのハンドリング

- クライアントシステムエラー
  前述の業務エラー用の例外(BizLogicException クラス)を除く例外は、全てシステムエラーとして扱う。
  TERASOLUNA Frameowork では、捕捉されなかった例外を集約的に処理する仕組みとして IExceptionHandler インタフェースを提供している。例外ハンドラは、FW 起動構成ファイル(TeasolunaBootstrap.config)に設定し、プロジェクトの要件により差し替え可能になっている。
  本機能は、クライアントで集約例外処理を実施する IExceptionHandler 実装クラスとして、標準で以下の3つのクラスを提供している。

- DefaultExceptionHandler クラス
  - フレームワーク起動後、AP 実行中に発生したエラーをハンドリングする。
    - 「CM-06 ログ出力機能」によりエラーログを出力する。
    - 「CL-06 メッセージ通知機能」によりユーザへのエラーメッセージを表示する。
- InitializationExceptionHandler クラス
  - フレームワークの起動時に発生したエラーをハンドリングする。
    - 「CM-06 ログ出力機能」によりエラーログを出力する。
    - MessageBox により、ユーザへのエラーメッセージを表示する。
- TerminationExceptionHandler クラス
  - フレームワークの終了時に発生したエラーをハンドリングする。
    - 「CM-06 ログ出力機能」によりエラーログを出力する。

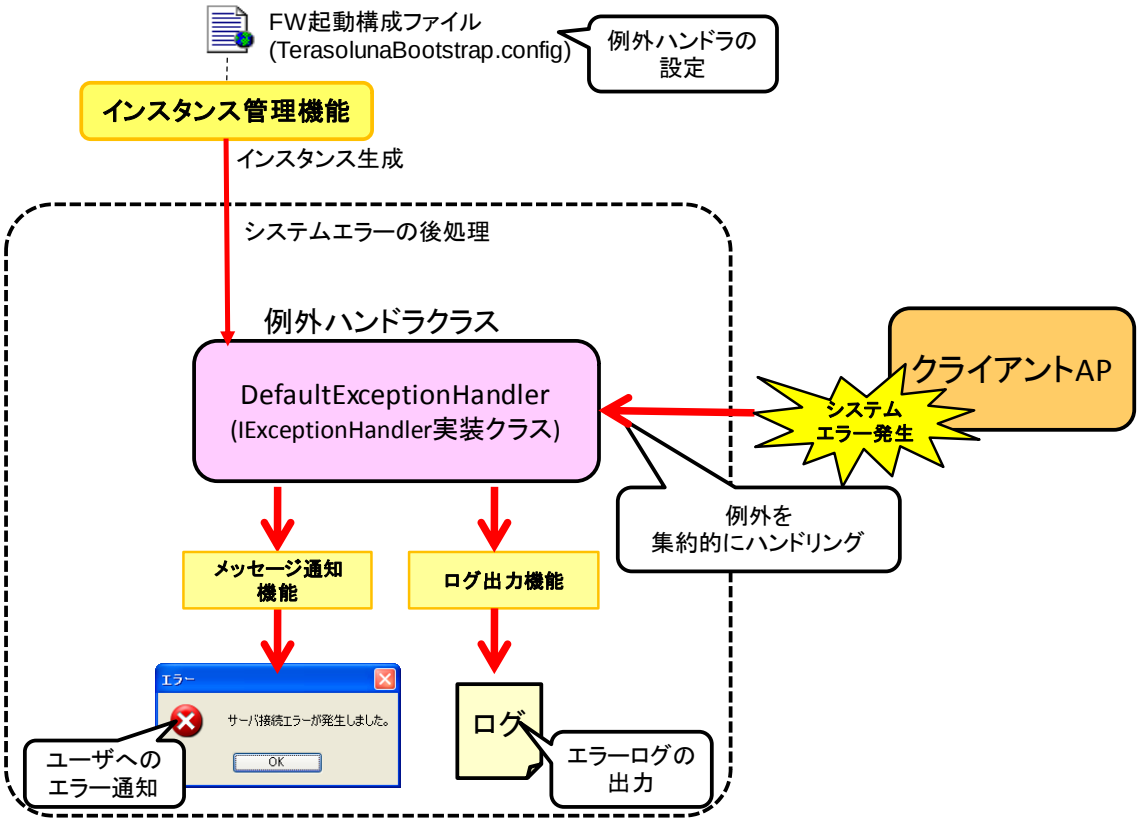


図 3 DefaultExceptionHandler による集約例外ハンドリング

イベント処理中に発生したシステムエラーは、前述のイベント処理フロー中の「エラー処理」フェーズ（DefaultErrorHandler クラス）で、BizLogicException 以外の例外であればシステムエラーとしてリスローすることで、イベント処理完了後に DefaultExceptionHandler クラスで集約例外処理を実施する。

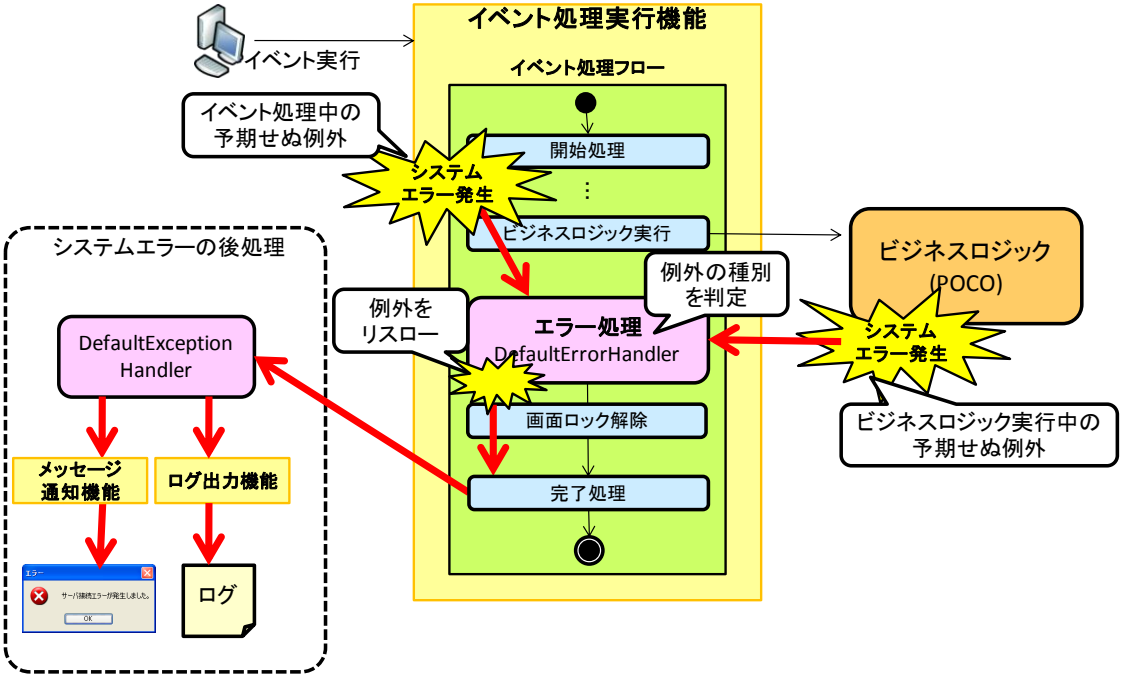


図 4 イベント処理実行中に発生したシステムエラーのハンドリング

- サーバ AP で発生したエラー
  TERASOLUNA フレームワークでは、サーバ AP で発生したエラーはエラー電文(SOAP Fault)で取り扱う。
  サーバ AP の開発言語(.NET や Java)を問わず、決まった形式の SOAP Fault を作成しクライアント側に返却することで、クライアント側のエラーと同様に集約的に処理することができる。
  TERASOLUNA フレームワークは通信基盤として、WCF(Windows Comunication Foundation)を採用している。WCF クライアントでは、サーバより SOAP Fault が返却されると、System.ServiceModel.FaultException 例外がスローされる。
  DefautlErrorHandler クラスは FaultException を捕捉し、FaultException に Detail プロパティが含まれている場合には、Detail プロパティに含まれている FaultContract の情報(SOAP Fault の<detail>要素)を取得してエラー種別を解析し、サーバ入力値検証エラー、サーバ業務エラー、サーバシステムエラーのどれであるかを判断する。サーバ入力値検証エラーおよびサーバ業務エラー時は、イベント処理の実行結果を格納後リスローせず、「完了処理」フェーズにおいてイベントの実行結果を判定し後処理を実施する。「完了処理」のフレームワークの標準提供クラスである DefaultCompletionHandler クラスは、ダイアログによりユーザへエラーメッセージを表示する。
  サーバシステムエラーの場合は例外をリスローし、イベント処理完了後に前述のシステムエラーの集約例外処理により例外を捕捉する。

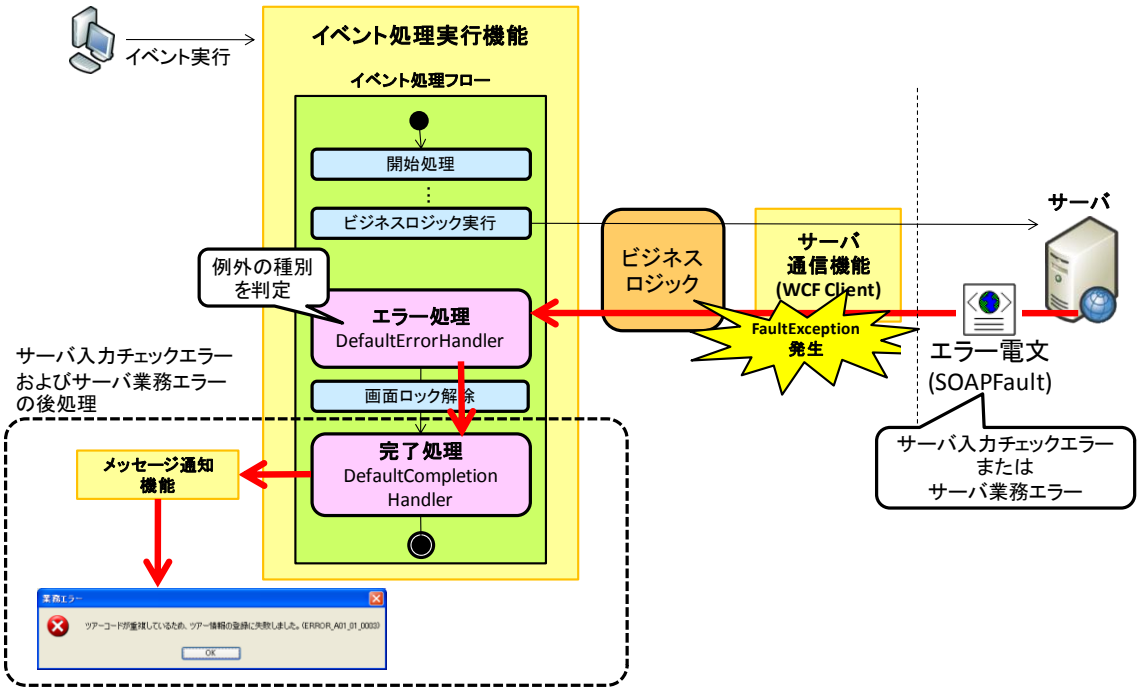


図 5 サーバ入力エラーおよびサーバ業務エラーのハンドリング

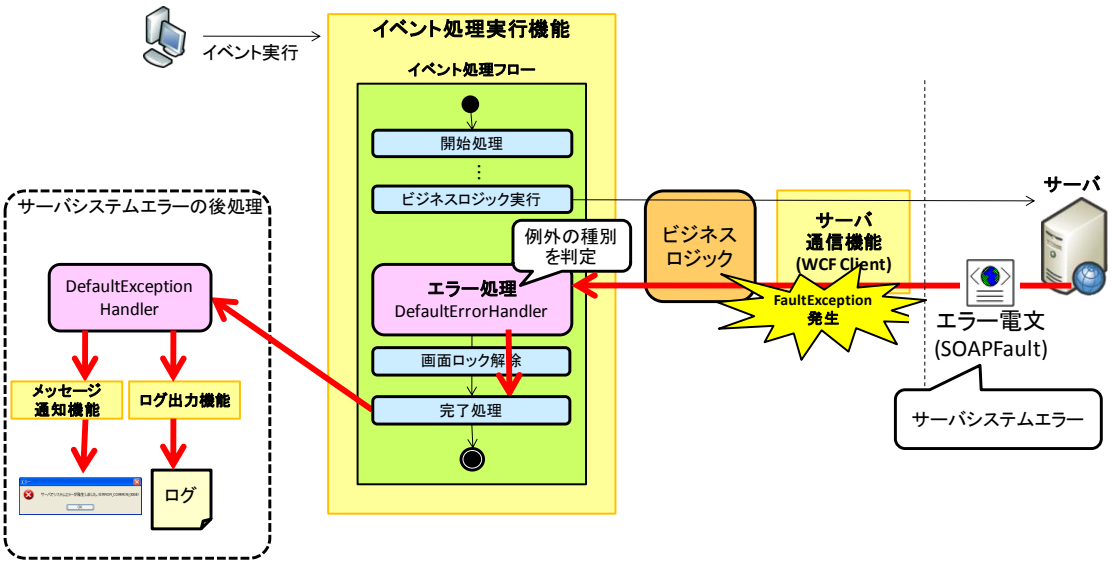


図 6 サーバシステムエラーのハンドリング

■ 使用方法

◆ 業務エラー用の例外(BizLogicException)の作成

ビジネスロジッククラス内において業務エラーを返却する場合、
BizLogicException をスローすることでフレームワークにより集約的に例外を処理することができる。以下に、通常使用する BizLogicException クラスのコンストラクタを示す。

表 1 BizLogicException クラスのコンストラクタ

項番	コンストラクタ	引数
1	public BizLogicException (string errorType, string bizLogicErrorType, IList<ErrorInfo> errors)	第１引数： エラー種別を表す任意の文字列。 ErrorType プロパティと対応。 第２引数： 業務エラーの分類を表す任意の文字列。 BizLogicErrorType プロパティと対応。 第３引数： エラー要因の詳細な情報(ErrorInfo)のリスト。 Errors プロパティと対応。 エラー要因が複数あれば ErrorInfo を複数指定する。

第1引数に対応する ErrorType プロパティは、「CL-03 イベント処理実行機能」における
イベント処理結果(EventProcessResult オブジェクト)の文字列として格納される。
BizLogicException.ErrorType の格納値とイベント処理結果の対応関係を以下に示す。

表 2 BizLogicException.ErrorType プロパティとイベント処理結果の対応関係

項番	BizLogicException.ErrorType プロパティの格納値	対応するイベント処理結果 （EventProcResultType の定数値）
1	BizLogicFailure	BizLogicFailure （クライアント AP 側ビジネスロジックで業務エラーが発生）
2	ValidationFailure	ValidationFailure （クライアント AP 側で入力値検証エラーが発生）

通常は、BizLogicException のコンストラクタでは、「BizLogicFailure」定数を指定する。
これによりイベント処理結果が「業務エラー」として処理される。
なお、「ValidationFailure」を指定することで、ビジネスロジック内で入力値検証エラーを扱うことも可能である。ただし、フレームワークが「CL-03 イベント処理実行機能」の「入力値検証」フェーズで画面データの検証を実施するため業務開発者が使用する必要は通常ない。
なお、業務エラーの種類によって処理を振り分けたい場合は、追加定義した任意の文字列を格納し「完了処理」フェーズでの処理を拡張することで柔軟に対応できる。

第2引数に対応する BizLogicErrorType プロパティは、業務エラーの分類を表す任意の文字列で、業務エラーをさらに分類したいときなどに、開発者が自由に使用する。
第3引数に対応する Errors プロパティは、Terasoluna.ErrorHandling.ErrorInfo クラスに格納する。以下に、ErrorInfo のコンストラクタを示す。

表 3 ErrorInfo のコンストラクタ

項番	コンストラクタ	引数
1	public ErrorInfo(string errorId, IList<string> arguments, string message, object rawData)	第１引数： エラーID。 エラーを特定するためのエラーコード。 第２引数： メッセージの置換文字列 第３引数： メッセージ 第４引数： エラー情報が格納されていた元データ

なお、BizLogicErrorType や Errors は、「CL-03 イベント処理実行機能」におけるイベント処理結果の文字列として格納される。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。

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 1 BizLogicException を使った業務エラーの記述例

◆ サーバエラー電文(SOAP Fault)の作成

TERASOLUNA フレームワークは、サーバで発生したエラー情報を、SOAP Fault の<detail>要素から取得する。
リスト 2 のような特定のフォーマットで<detail>要素を生成し返却することで、サーバ AP が、.NET、Java といった開発言語や、プラットフォームに依存することなく、クライアント AP はサーバエラーを集約的に処理することができる。
なお、<detail>要素が想定するフォーマットと異なる場合など、電文を解析できない場合は、システムエラーとして処理される。

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <s:Header xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"></s:Header>
    <S:Body>
        <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
            <faultcode xmlns="">S:Server</faultcode>
            <faultstring xmlns="">サーバで業務エラーが発生しました。</faultstring>
            <detail xmlns="">
                <ns2:SampleSoapFault xmlns:ns2="http://jp.terasoluna.toursample">
                    <ns2:ErrorType>BizLogicFailure</ns2:ErrorType>
                    <ns2:BizLogicErrorType>AppError</ns2:BizLogicErrorType>
                    <ns2:ErrorMessages>
                        <ns2:ErrorMessage>
                            <ns2:ErrorId>WARN_B01_01_0001</ns2:ErrorId>
                            <ns2:Message>ユーザIDまたはパスワードが違います。</ns2:Message>
                        </ns2:ErrorMessage>
                    </ns2:ErrorMessages>
                </ns2:SampleSoapFault>
            </detail>
        </S:Fault>
    </S:Body>
</S:Envelope>
```

リスト 2 SOAF Fault によるエラー電文のイメージ

以下に、サーバ AP が、.NET(WCF サービス)、.Java(JAX-WS Web サービス)の場合について、それぞれ SOAP Fault の作成方法を説明する。

(1) サーバ AP が.NET アプリケーション(WCF サービス)の場合

サーバ AP として、TERASOLUNA フレームワークを採用する場合、「SV-03 サーバエラーハンドリング機能」により、サーバ AP で発生したエラーは、フレームワークが提供する SOAP Fault (TerasolunaSoapFault クラス)として自動的にクライアントに返却されるため、開発者は FaultContract を意識することなく実装することができる。利用方法の詳細については、「SV-03 サーバエラーハンドリング機能」の機能説明書を参照のこと。
また、サーバ AP として、TERASOLUNA フレームワークを採用しない場合でも、「SV-03 サーバエラーハンドリング機能」が生成する FaultContract(TerasolunaSoapFault クラス)と同一のデータ構造をもつ FaultContract クラスを作成し FaultException<T>としてスローすることで、本機能による集約例外処理を利用できる。FaultCotract クラスには、以下のプロパティを定義する必要がある。

- 「ErrorType」プロパティ
  - 文字列型
  - クライアント側でフレームワークがエラーの種類によって処理を振り分けるための判定文字列
    - 「入力値検証エラー」の場合は、"ValidationFailure"を設定
    - 「業務エラー」の場合は、"BizLogicFailure"を設定
    - 「システムエラー」の場合は、値を設定しないか、上記の値以外を設定
  - ErrorType に設定した文字列に応じて、「CL-03 イベント処理実行機能」におけるイベント処理結果(EventProcessResult)に、対応する文字列が格納される。イベント処理結果の対応関係を以下に示す。

表 4 ErrorType プロパティとイベント処理結果の対応関係

項番	ErrorType の格納値	対応するイベント処理結果 (EventProcResultType の定数値)
1	"BizLogicFailure"	ServerBizLogicFailure (サーバ AP 側ビジネスロジックで業務エラーが発生。)
2	"ValidationFailure"	ServerValidationFailure (サーバ AP 側で入力値検証エラーが発生。)

- 「BizLogicErrorType」プロパティ
  - 文字列型
  - 開発者が利用する任意の文字列、業務エラーを細かく分類したい場合に使用
- 「ErrorMessages」プロパティ
  - 後述のエラーメッセージクラスに対する IEnumerable 型
  - エラーメッセージのリストを保持

なお、BizLogicErrorType、ErrorMessages プロパティは「CL-03 イベント処理実行機能」におけるイベント処理結果の文字列として格納される。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。

以下に、TERASOLUNA フレームワークを利用しない場合の FaultContract の実装例を示す。

```
[DataContract]
public class SampleSoapFault {
    [DataMember(Order = 0)]
    public string ErrorType { get; set; }

    [DataMember(Order = 1)]
    public string BizLogicErrorType { get; set; }

    [DataMember(Order = 2)]
    public IList<SampleSoapFaultErrorMessage> ErrorMessages { get; private set; }
```

リスト 3　FaultCotract クラスの実装例

この時、WCF サービスのメソッドに、この FaultContract クラスを引数として、FautlContract 属性を付与すると WSDL の<fault>要素の定義が生成される。なお、サーバ AP も TERASOLUNA フレームワークを使用する場合は、あらかじめ FautlContract 属性が適用される WCF サービスを自動的に生成するため、開発者により FautlContract 属性を付与する必要がない。

```
[ServiceContract]
public interface ICalcService
{
    [OperationContract]
    [FaultContract(typeof(SampleSoapFault))]
    CalcOutputDto Add(CalcInputDto inputDto);
}
```

リスト 4 FaultContract 属性の記述例

また、FaultContract クラスに定義する ErrorMessages プロパティは、エラーメッセージクラスをリストで保持する。エラーメッセージを表すクラスは、以下のプロパティを定義する必要がある。

- 「ErrorId」プロパティ
  - 文字列型のプロパティ
  - エラーコードなど対象エラーメッセージに対する ID を格納
- 「Arguments」プロパティ
  - 文字列型のプロパティ
  - クライアント側で、別途ユーザ向けに表示するメッセージを生成する等を考慮する必要がある場合に置換文字列を格納し利用可能
  - 複数置換文字列がある場合は、カンマ(,)区切りで格納
- 「Message」プロパティ
  - 文字列型のプロパティ
  - 対象エラーに対してクライアントへ通知するメッセージを格納

以下に、TERASOLUNA フレームワークを利用しない場合のエラーメッセージクラスの実装例を示す。

```
[DataContract(Name = "ErrorMessage")]
public class SampleSoapFaultErrorMessage
{
    [DataMember(Order = 0)]
    public string ErrorId { get; set; }
    [DataMember(Order = 1)]
    public string Arguments { get; set; }
    [DataMember(Order = 2)]
    public string Message { get; set; }

    public SampleSoapFaultErrorMessage(string errorId, string arguments, string message)
    {
        ErrorId = errorId;
        Arguments = arguments;
        Message = message;
    }
}
```

リスト 5　エラーメッセージクラスの実装例

クライアントにエラーを返却するには、定義した FaultContract を利用して、FaultException<T>をスローする。

- サーバ入力値検証エラーの場合
  FaultContract の ErrorType プロパティに”ValidationFailure”をセットする。
  また、サーバ側で発生した入力値検証エラー情報を、前述のエラーメッセージクラスに移し替え、FaultContract にセットする。

  以下に、例外の作成例を示す。(実際には、この後作成した例外をスローする。)
  この例では FaultException<ValidationFault> クラス(Validation AB の WCF Integration での入力値検証エラー)から FaultException<SampleSoapFault>へ移し替えている。

```
/// 入力値検証エラーを表すErrorType名
private const string VALIDATION_FAILURE_ERROR_TYPE = "ValidationFailure";
/// 入力値検証エラーを表す固定のエラーIDの例
private const string VALIDATION_ERROR_ID = "VALIDATION_ERROR";

/// 入力値検証エラー
protected virtual FaultException<SampleSoapFault>
                        CreateException(FaultException<ValidationFault> validationError)
{
        ValidationFault validationFault = validationError.Detail;
        List<SampleSoapFaultErrorMessage> messages =
                                    new List<SampleSoapFaultErrorMessage>();
        foreach (ValidationDetail detail in validationFault.Details)
        {
            ///エラーメッセージの追加
            messages.Add(new SampleSoapFaultErrorMessage(
                        VALIDATION_ERROR_ID, string.Empty, detail.Message));
        }
        return new FaultException<SampleSoapFault>(
            new SampleSoapFault()
            {
                ErrorType = VALIDATION_FAILURE_ERROR_TYPE,
                ErrorMessages = messages
            }
        );
 }
```

リスト 6 サーバ入力値検証エラーの場合の例外作成例

- サーバ業務エラーの場合
  FaultContract の ErrorType プロパティには”BizLogicFailure”をセットする。
  また、サーバ側で発生した業務エラー情報を、前述のエラーメッセージクラスに移し替え、FaultContract にセットする。

以下に、例外の作成例を示す。(実際には、この後作成した例外をスローする。)
この例では ApplicationException(業務エラーを表す例外の例)から、FaultException<SampleSoapFault>へ移し替えている。

```
/// 業務エラーを表すErrorType名
private const string BIZ_LOGIC_FAILURE_ERROR_TYPE = "BizLogicFailure";

/// 業務エラー
protected virtual FaultException<SampleSoapFault>
                        CreateException(BizLogicException bizLogicError)
{
        List<SampleSoapFaultErrorMessage> messages =
                                new List<SampleSoapFaultErrorMessage>();
        foreach (ErrorInfo errorInfo in bizLogicError.Errors)
        {
            ///エラーメッセージの追加
            messages.Add(
                new SampleSoapFaultErrorMessage(errorInfo.ErrorId,
                ///カンマ区切りの文字列に変換
                FormatToCommaSeparatedString(errorInfo.Arguments),
                        string.Format(errorInfo.Message, errorInfo.Arguments)));
        }
        return new FaultException<SampleSoapFault>(
            new SampleSoapFault()
            {
                ErrorType = BIZ_LOGIC_FAILURE_ERROR_TYPE,
                ErrorMessages = messages
            }
        );
}
```

リスト 7 業務エラーの場合の例外作成例

- サーバシステムエラーの場合

  FaultContract の ErrorType プロパティには何も設定する必要はない。
  もしくは、「ValidationFailure」、「BizLogicFailure」以外の値を設定することも可能である。
  また、サーバ側で発生したシステムエラーの情報を、前述のエラーメッセージクラスに移し替え、FaultContract にセットしている。
  以下に、例外の作成例を示す。(実際には、この後作成した例外をスローする。)
  この例では SystemException クラス(システムエラーを表す例外の例)から FaultException<SampleSoapFault>へ移し替えている。

```
/// システムエラーを表す固定的なエラーIDの例
private const string SYSTEM_ERROR_ID = "SYSTEM_ERROR";
/// システムエラー
protected virtual FaultException<SampleSoapFault> CreateException(Exception error)
{
    SampleSoapFault soapFault = new SampleSoapFault();
    soapFault.ErrorMessages = new List<SampleSoapFaultErrorMessage>();
    ///エラーメッセージの追加
    soapFault.ErrorMessages.Add(
        new SampleSoapFaultErrorMessage(SYSTEM_ERROR_ID, string.Empty, "予期せぬエラーが発生しました。"));
    return new FaultException<SampleSoapFault>(soapFault);
```

リスト 8 システムエラーの場合の例外作成例

(2) サーバ AP が Java アプリケーション(JAX-WS Web サービス)の場合

サーバ側を Java アプリケーション(JAX-WS 準拠の Web サービス)で実現する場合、
@WebFault アノテーションを付与した例外クラス(java.lang.Exception 継承クラス)を作成する。
この例外クラスをスローすることで SOAP Fault をクライアントへ返却することができる。
JAX-WS の仕様書[1]にもとづき、@WebFault を付与した例外クラスには、FaultBean を戻り値とする「getFaultInfo」メソッドを実装すること。
以下に、例外クラスの実装例を示す。

```
import javax.xml.ws.WebFault;

@WebFault(name = "SampleSoapFault", targetNamespace = "http://jp.terasoluna.toursample",
faultBean = "jp.terasoluna.toursample.server.common.exception.SampleFaultBean")
public class SampleSoapFaultException extends Exception {

        private static final long serialVersionUID = -9174594089171472166L;
        private SampleFaultBean faultBean;
        public SampleSoapFaultException(String message) {
                super(message);
                this.faultBean = new SampleFaultBean();
        }
        public SampleSoapFaultException(String message, SampleFaultBean faultInfo) {
                super(message);
                this.faultBean = faultInfo;
        }
        public SampleSoapFaultException(String message, SampleFaultBean faultInfo,
                        Throwable cause) {
                super(message, cause);
                this.faultBean = faultInfo;
        }
        public SampleFaultBean getFaultInfo() {
                return faultBean;
        }
}
```

リスト 9 例外クラスの実装例

この例外クラスを Web サービスクラスの Web メソッドの throws 句に宣言する。これにより、WSDL の<fault>要素の定義が生成される。
また、サーバビジネスロジックでのエラー発生時には、この例外クラスのインスタンスを生成し、エラー情報を格納後スローする。

[1] JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0
http://jcp.org/en/jsr/detail?id=224

```
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(serviceName = "A01Service", targetNamespace ="http://jp.terasoluna.toursample")
public class A01Service {
    @WebMethod(operationName = "ExecuteA01_01_01_S01")
    public void executeA01_01_01_S01(A01_01_01_S01InputDto input)
            throws SampleSoapFaultException {

        ///サーバビジネスロジック処理の実装
    }
}
```

リスト 10 JAX-WS による Web サービスクラスの実装例

FaultBean クラスには、Fault メッセージの<detail>要素に格納される情報を定義する。
FaultBean クラスには、以下の要素を定義する必要がある。

- 「ErrorType」要素
  - 文字列型の要素
  - クライアント側でフレームワークがエラーの種類によって処理を振り分けるための判定文字列
    - 「入力値検証エラー」の場合は、「ValidationFailure」を設定
    - 「業務エラー」の場合は、「BizLogicFailure」を設定
    - その他の値または値がない場合は、「システムエラー」として判定
  - 以下の実装例では、errorType フィールド、getErrorType メソッドに対応
  - 文字列は、「CL-03 イベント処理実行機能」におけるイベント処理結果(EventProcessResult)の文字列として格納される。イベント処理結果の対応関係を以下に示す。

表 5 ErrorType プロパティと EventProceResultType の定数値の対応関係

項番	ErrorType の格納値	対応するイベント処理結果 （EventProcResultType の定数値）
1	“BizLogicFailure”	ServerBizLogicFailure （サーバ AP 側ビジネスロジックで業務エラーが発生。）
2	“ValidationFailure”	ServerValidationFailure （サーバ AP 側で入力値検証エラーが発生。）

- 「BizLogicErrorType」要素
  - 文字列型の要素
  - 開発者が利用する任意の文字列、業務エラーを細かく分類したい場合に使用
  - 以下の実装例では、bizLogicErrorType フィールド、getErrorType メソッドに対応
- 「ErrorMessages」要素
  - 後述のエラーメッセージリストクラス
  - 以下の実装例では、errorMessages フィールド、getErrorMessages メソッドに対応

なお、BizLogicErrorType、ErrorMessages プロパティは「CL-03 イベント処理実行機能」におけるイベント処理結果の文字列として格納される。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。

以下に、FaultBean クラスの実装例を示す。

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "SampleSoapFault", namespace="http://jp.terasoluna.toursample")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SampleSoapFault", namespace="http://jp.terasoluna.toursample")
public class SampleFaultBean {

        @XmlElement(name = "ErrorType", namespace="http://jp.terasoluna.toursample")
        private String errorType;
        @XmlElement(name = "BizLogicErrorType",
                namespace="http://jp.terasoluna.toursample")
        private String bizLogicErrorType;
        @XmlElement(name = "ErrorMessages", namespace="http://jp.terasoluna.toursample")
        private ErrorMessages errorMessages;

        public String getErrorType() {
                return errorType;
        }
        public void setErrorType(String errorType) {
                this.errorType = errorType;
        }
        public String getBizLogicErrorType() {
                return bizLogicErrorType;
        }
        public void setBizLogicErrorType(String bizLogicErrorType) {
                this.bizLogicErrorType = bizLogicErrorType;
        }
        public ErrorMessages getErrorMessages() {
                return errorMessages;
        }
        public void setErrorMessages(ErrorMessages errorMessages) {
                this.errorMessages = errorMessages;
        }
}
```

リスト 11 FaultBean の実装例

また、FaultBean が持つエラーメッセージリストクラスでは、後述のユーザ定義のエラーメッセージクラスのリストデータを保持するように定義する。
以下に、エラーメッセージリストクラスの実装例を示す。

```
import java.util.List;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(namespace="http://jp.terasoluna.toursample")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace="http://jp.terasoluna.toursample")
public class ErrorMessages {

        @XmlElement(name = "ErrorMessage", namespace="http://jp.terasoluna.toursample",
                                                        nillable = true)
        private List<ErrorMessage> errorMessage;

        public List<ErrorMessage> getErrorMessage() {
                return errorMessage;
        }

        public void setErrorMessage(List<ErrorMessage> errorMessage) {
                this.errorMessage = errorMessage;
        }
}
```

リスト 12 FaultBean に含まれるエラーメッセージリスト(ErrorMessages)クラスの実装例

さらに、エラーメッセージリストクラスのエラーメッセージクラスの中には、以下のフィールド、プロパティを定義する必要がある。

- 「ErrorId」要素
  - 文字列型の要素
  - エラーコードなど、対象エラーメッセージに対する ID を格納
  - 以下サンプルコードで、errorId フィールド、getErrorId メソッドに対応
- 「Arguments」要素
  - 文字列型の要素
  - クライアント側で、別途ユーザ向けに表示するメッセージを生成する等を考慮する必要がある場合に置換文字列を格納し利用可能
  - 複数置換文字列がある場合は、カンマ(,)区切りで格納
  - 以下サンプルコードで、arguments フィールド、get Arguments メソッドに対応
- 「Message」要素
  - 文字列型の要素
  - 対象エラーに対してクライアントへ通知するメッセージを格納
  - 以下サンプルコードで、message フィールド、getMessage メソッドに対応

以下に、エラーメッセージリストクラスの実装例を示す。

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(namespace = "http://jp.terasoluna.toursample")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace = "http://jp.terasoluna.toursample")
public class ErrorMessage {

    @XmlElement(name = "ErrorId", namespace = "http://jp.terasoluna.toursample")
    private String errorId;
    @XmlElement(name = "Arguments", namespace = "http://jp.terasoluna.toursample")
    private String arguments;
    @XmlElement(name = "Message", namespace = "http://jp.terasoluna.toursample")
    private String message;

    public String getErrorId() {
        return errorId;
    }

    public void setErrorId(String errorId) {
        this.errorId = errorId;
    }

    public String getArguments() {
        return arguments;
    }

    public void setArguments(String arguments) {
        this.arguments = arguments;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

リスト 13 エラーメッセージ(Errormessage)クラスの実装例

SOAP Fault を返却するには、サーバ AP で発生した実際の例外やメソッドの戻り値の情報をもとに先ほど作成した例外クラスを作成し、スローする。

- サーバ入力値検証エラーの場合

FaultBean の ErrorType プロパティには”ValidationFailure”をセットする。
また、サーバ側で発生した入力値検証エラー情報を、前述のエラーメッセージクラスに移し替え、SOAP Fault 用の例外クラスにセットする。

以下に、例外の作成例を示す。(実際には、この後作成した例外をスローする。)
この例では、org.springframework.validation.BindException クラス(Spring Framework での入力値検証エラー)から、SampleSoapFaultException へ移し替えている。

```
/**
 * 入力値検証エラー用のSOAPFaultを作成
 *
 */
private SampleSoapFaultException createSoapFaultValidationException(
                    BindException e) {
        SampleFaultBean faultBean = new SampleFaultBean();
        ErrorMessages errorMessages = new ErrorMessages();
        List<ErrorMessage> messages = new ArrayList<ErrorMessage>();
        // エラーメッセージの作成
        List<ObjectError> errors = e.getAllErrors();
        for (ObjectError error : errors) {
                ErrorMessage errorMessage = new ErrorMessage();
                errorMessage.setErrorId(error.getCode());
                // エラーコードから取得した入力値検証エラーのメッセージをセット
                errorMessage.setMessage(messageAccessor.getMessage(error));
                // カンマ区切りの文字列に変換
                errorMessage.setArguments(formatToCommaSepratedStringForValidationError(
                        error.getArguments()));
                messages.add(errorMessage);
        }
        errorMessages.setErrorMessage(messages);
        faultBean.setErrorMessages(errorMessages);
        // Terasoluna Client FW for .NET 3.0では、ErrorTypeが
        // 「ValidationFailure」の場合、入力チェック例外であることを判定する
        faultBean.setErrorType("ValidationFailure");
        return new SampleSoapFaultException(messageAccessor
                        .getMessage("ERROR_COMMON_0002"), faultBean);
}
```

リスト 14 サーバ入力値検証エラー時の例外作成例

● サーバ業務エラーの場合

FaultBean の ErrorType プロパティには”BizLogicFailure”をセットする。
また、サーバ側で発生した業務エラー情報を、前述のエラーメッセージクラスに移し替え、SOAP Fault 用の例外クラスにセットする。

以下に、例外の作成例を示す。(実際には、この後作成した例外をスローする。)
この例では ApplicationException(業務エラーを表す例外の例)から、SampleSoapFaultException へ移し替えている。

```
/**
 * 業務エラー用のSOAPFaultを作成
 *
 */
private SampleSoapFaultException createSoapFaultAppException(
                        ApplicationException e) {
        SampleFaultBean faultBean = new SampleFaultBean();
        ErrorMessages errorMessages = new ErrorMessages();
        List<ErrorMessage> messages = new ArrayList<ErrorMessage>();
        // エラーメッセージの作成
        ErrorMessage errorMessage = new ErrorMessage();
        errorMessage.setErrorId(e.getErrorCode());
        errorMessage.setMessage(e.getMessage());
        messages.add(errorMessage);
        errorMessages.setErrorMessage(messages);
        // カンマ区切りの文字列に変換
        errorMessage.setArguments(formatToCommaSeparatedString(e.getOptions()));
        faultBean.setErrorMessages(errorMessages);
        // Terasoluna Client FW for .NET 3.0では、ErrorTypeが
        // 「BizLogicFailure」の場合、業務例外であることを判定する
        faultBean.setErrorType("BizLogicFailure";
        // 業務エラーを分類したければ分類可能
        faultBean.setBizLogicErrorType(messageAccessor
                                .getMessage("ERROR_COMMON_APP_ERROR"));
        return new SampleSoapFaultException(messageAccessor
                                .getMessage("ERROR_COMMON_0003"), faultBean);
}
```

リスト 15 サーバ業務エラー時の例外作成例

● サーバシステムエラーの場合

FaultBean の ErrorType プロパティには何も設定しない。
もしクライアントで、集約例外処理の拡張を検討するのであれば、「ValidationFailure」、「BizLogicFailure」以外の値を設定する。
また、サーバ側で発生したシステムエラーの情報を、前述のエラーメッセージクラスに移し替え、SOAP Fault 用の例外クラスにセットしている。

以下に、例外の作成例を示す。(実際には、この後作成した例外をスローする。)
この例では SystemException クラス(システムエラーを表す例外の例)から SampleSoapFaultException へ移し替えている。

```
/**
 * システムエラー用のSOAPFaultを作成
 *
 */
private SampleSoapFaultException createSoapFaultSysException(
                        SystemException e) {
        SampleFaultBean faultBean = new SampleFaultBean();
        ErrorMessages errorMessages = new ErrorMessages();
        List<ErrorMessage> messages = new ArrayList<ErrorMessage>();
        // エラーメッセージの作成
        ErrorMessage errorMessage = new ErrorMessage();
        errorMessage.setErrorId(e.getErrorCode());
        errorMessage.setMessage(e.getMessage());
        // カンマ区切りの文字列に変換
        errorMessage.setArguments(formatToCommaSeparatedString(e.getOptions()));
        messages.add(errorMessage);
        errorMessages.setErrorMessage(messages);
        faultBean.setErrorMessages(errorMessages);
        return new SampleSoapFaultException(messageAccessor
                                .getMessage("ERROR_COMMON_0004"), faultBean);
}
```

リスト 16 サーバシステムエラー時の例外作成例

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

システムエラーを集約例外処理できるように、アーキテクトは FW 起動構成ファイル(TerasolunaBootstrap.config)に、集約例外ハンドリングの設定をする。
集約例外処理を実施する IExceptionHandler 実装クラスは、「CM-02 インスタンス管理機能」を利用して実装クラスを FW 起動構成ファイルに指定することで、設定をすることができる。
集約例外処理するクラスは、システムで1つあればよいので、<lifetime>タグは「singleton」(ContainerControlledLifetimeManager)を指定する。
また、<type>タグの name 属性で指定した名前と、例外を捕捉するポイントは1対1に対応しており、表 6 の例外捕捉ポイントごとに、IExceptionHandler 実装クラスの設定を行う必要がある。

表 6 例外発生ポイントとの対応関係

項番	type の name 属性	例外捕捉ポイント	標準で設定する IExceptionHandler クラス
1	AppDomainUnhandledFailure	AppDomain.CurrentDomain.UnhandledException イベントが拾う例外	DefaultExceptionHandler
2	ApplicationThreadFailure	Application.ThreadException イベントが拾う例外	DefaultExceptionHandler
3	InitializationFailure	「CM-01 アプリケーション起動・終了機能」による、フレームワーク初期化処理で発生した例外	InitializationExceptionHandler
4	TerminationFailure	「CM-01 アプリケーション起動・終了機能」による、フレームワーク終了処理で発生した例外	TerminationExceptionHandler

以下に、FW 起動構成ファイルの記述例を示す。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>
    <typeAliases>
…
      <typeAlias alias="singleton"
                 type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager,
                     Microsoft.Practices.Unity,
                   Version=1.2.0.0,
                   Culture=neutral,
                   PublicKeyToken=31bf3856ad364e35" />
      <typeAlias alias="IExceptionHandler"
                  type="Terasoluna.ExceptionHandling.IExceptionHandler,
                      Terasoluna" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <!-- 集約例外処理の設定 -->
          <!-- AppDomain.CurrentDomain.UnhandledExceptionイベントのポリシー -->
          <type type="IExceptionHandler" name="AppDomainUnhandledFailure"
                mapTo="Terasoluna.Windows.Forms.ExceptionHandling.DefaultExceptionHandler,
                      Terasoluna.Windows.Forms" >
            <lifetime type="singleton" />
          </type>
          <!-- Application.ThreadExceptionイベントのポリシー -->
          <type type="IExceptionHandler" name="ApplicationThreadFailure"
                mapTo="Terasoluna.Windows.Forms.ExceptionHandling.DefaultExceptionHandler,
                      Terasoluna.Windows.Forms" >
            <lifetime type="singleton" />
          </type>
          <!-- 初期化処理失敗による例外発生時のポリシー -->
          <type type="IExceptionHandler" name="InitializationFailure"
                mapTo="Terasoluna.Windows.Forms.ExceptionHandling.InitializationExceptionHandler,
                      Terasoluna.Windows.Forms" >
            <lifetime type="singleton" />
          </type>
          <!-- 終了処理失敗による例外発生時のポリシー -->
          <type type="IExceptionHandler" name="TerminationFailure"
                mapTo="Terasoluna.Windows.Forms.ExceptionHandling.TerminationExceptionHandler,
                      Terasoluna.Windows.Forms" >
            <lifetime type="singleton" />
          </type>
        </types>
…
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 17 FW 起動構成ファイル(TerasolunaBootstrap.config)の記述例

◆ スタートアップオブジェクトの実装

アーキテクトは、システムエラーの集約例外処理ができるように、スタートアップオブジェクト（Program.cs など）のコードで、AppDomain.CurrentDomain.UnhandledException イベントと、Application.ThreadException イベントのイベントハンドラメソッドを実装し、以下の集約例外処理を埋め込む必要がある。
フレームワークが提供する ExceptionManager クラスの HandleException メソッドを呼び出し、パラメータとして発生した例外と、FW 起動構成ファイル(TerasolunaBootstrap.config)で指定した<type>タグの name 属性と同一の名前を指定する。なお、TERASOLUNA フレームワークが提供するプロジェクトテンプレートを利用して、「AP 起動用業務プロジェクト」を作成した場合、テンプレートに含まれるスタートアップオブジェクトには、以下の処理がすでに埋め込まれている。

```
static class Program
{
    /// <summary>
    /// アプリケーションのメイン エントリ ポイントです。
    /// </summary>
    [STAThread]
    static void Main()
    {
        ///AppDomain.CurrentDomain.UnhandledExceptionイベントの登録
        AppDomain.CurrentDomain.UnhandledException +=
                new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);
        ///Application.ThreadExceptionイベントの登録
        Application.ThreadException +=
                new System.Threading.ThreadExceptionEventHandler(Application_ThreadException);
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        ///アプリケーションの起動
        Application.Run(new TourSampleStartupForm());
    }
    /// <summary>
    /// システムエラーの集約例外ハンドリング処理
    /// </summary>
    private static void CurrentDomain_UnhandledException
                        (object sender, UnhandledExceptionEventArgs e)
    {
        Exception exception = e.ExceptionObject as Exception;
        if (exception != null)
        {
            ExceptionManager.HandleException(exception, "AppDomainUnhandledFailure");
        }
    }

    /// <summary>
    /// システムエラーの集約例外ハンドリング処理
    /// </summary>
    private static void Application_ThreadException
                        (object sender, System.Threading.ThreadExceptionEventArgs e)
    {
        ExceptionManager.HandleException(e.Exception, "ApplicationThreadFailure");
    }
```

リスト 18 Program.cs の実装例

◆ 利用するフレームワーク機能の動作設定

本機能は、以下のフレームワーク機能を利用している。フレームワーク機能を動作させるための設定は、各フレームワーク機能の機能説明書を参照のこと。

- 「CL-03 イベント処理実行機能」
- 「CL-04 サーバ通信機能」
- 「CL-01 画面データ機能」、
- 「CM-06 ログ出力機能」
- 「CL-06 メッセージ通知機能」

◆ エラーログの出力設定

TERASOLUNA フレームワークが標準提供するシステムエラーの集約例外処理では、「CM-06 ログ出力機能」を使用してエラー情報を Error レベルでログ出力する。
TraceSource を使った標準のログ出力機能を利用する場合、アーキテクトは、DefaultExceptionHandler クラスが存在するアセンブリのアセンブリ名である「Terasoluna.Windows.Forms」をカテゴリ名として設定する。
以下に、アプリケーション構成ファイル(App.config)の設定例を示す。
ログ出力の詳細な設定方法は、「CM-06 ログ出力機能」の機能説明書や MSDN のドキュメントを参照のこと。

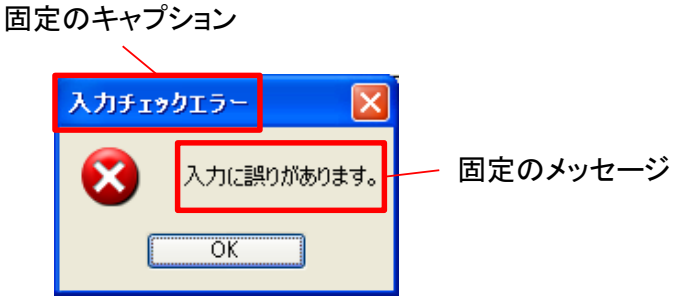
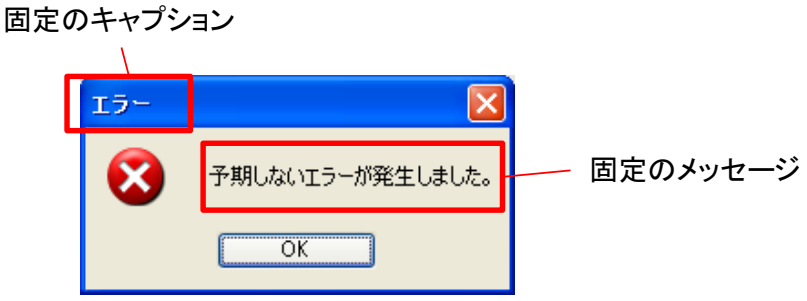
```
    ・・・
      <!--フレームワーク（Terasoluna.Windows.dll）のデフォルトログ -->
      <source name="Terasoluna.Windows.Forms" switchName="FrameworkLogSwitch"
                                              switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="XMLLogFile" />
          <add name="ConsoleLog" />
          <remove name="Default" />
        </listeners>
      </source>
    ・・・
    </sources>
      <!-- ログスイッチの設定 -->
    <switches>
      <add name="FrameworkLogSwitch" value="Error" />
      <add name="DefaultLogSwitch" value="All" />
    </switches>
    ・・・
  </system.diagnostics>
</configuration>
```

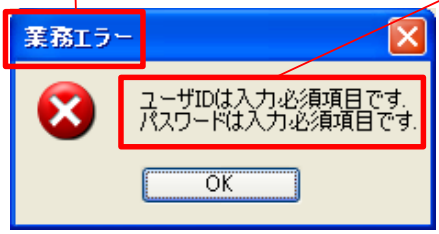
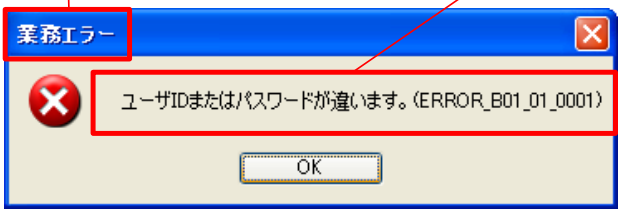
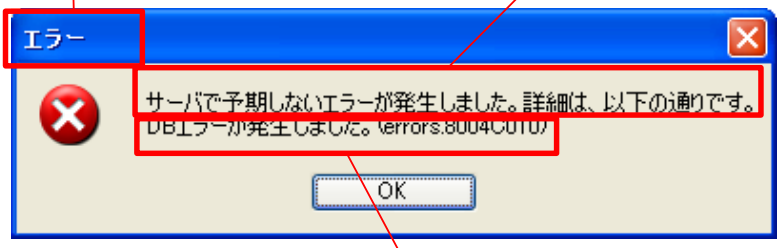
リスト 19　アプリケーション構成ファイル（App.config）のログの設定例

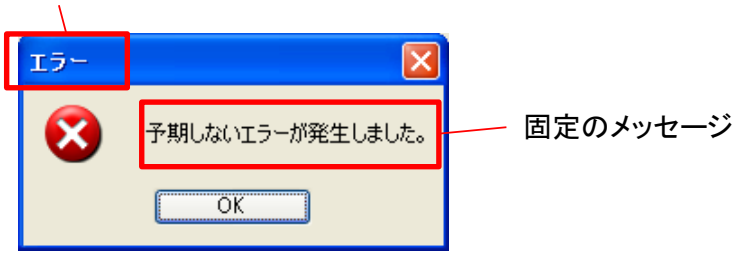
◆ 標準機能におけるエラーダイアログの表示仕様

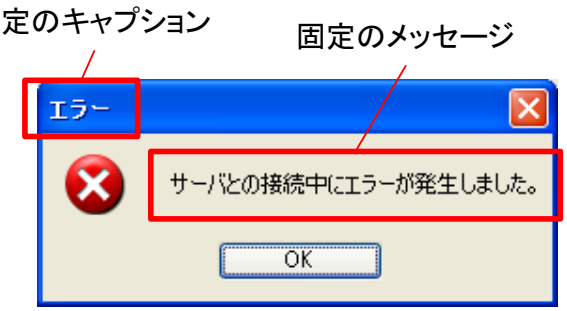
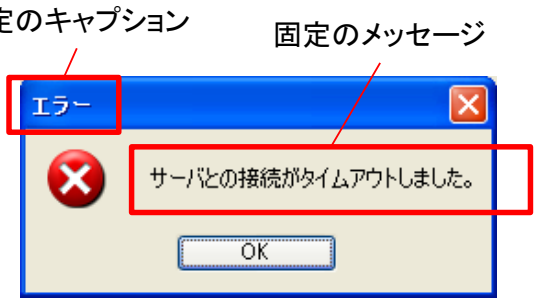
本機能の標準実装を利用した場合のエラーダイアログ表示イメージを以下に示す。

表 7 エラーダイアログの表示仕様

項番	エラー種別	ダイアログ
1	クライアント入力値検証エラー	
2	クライアント業務エラー	
3	クライアントシステムエラー	

<table>
<tr><th>項番</th><th>エラー種別</th><th>ダイアログ</th></tr>
<tr><td>4</td><td>サーバ入力値検証エラー</td><td>ErrorInfo.Messageを出力
エラーが複数ある場合は改行して列挙する
固定のキャプション
 
</td></tr>
<tr><td>5</td><td>サーバ業務エラー</td><td>ErrorInfo.Message + “(“ + ErrorInfo.ErrorId+“)”
の形式でメッセージ出力
エラーが複数ある場合は改行して列挙する
固定のキャプション
 
</td></tr>
<tr><td rowspan="2">6</td><td rowspan="2">サーバシステムエラー</td><td>【detail 要素が解析可能なフォーマットの SOAP Fault を受信した場合】
固定のキャプション
固定のメッセージ
 

ErrorInfo.Message + “(“ + ErrorInfo.ErrorId+“)”
の形式でメッセージ出力
エラーが複数ある場合は改行して列挙する</td></tr>
<tr><td>【detail 要素が解析不能なフォーマットの SOAP Fault を受信した場合】
固定のキャプション
 
</td></tr>
</table>

項番	エラー種別	ダイアログ
		【サーバ通信エラーの場合】(System.ServiceModel.CommunicationException) 
		【サーバ接続中にタイムアウトした場合】(System.TimeoutException) 

◆ 標準機能におけるログメッセージ出力仕様

本機能の標準実装を利用した場合のクライアントのログ出力仕様を示す。

表 8 ログ出力仕様

項番	エラー種別	ログレベル	ログ出力メソッドに渡す引数
1	クライアント入力値検証エラー	-	出力しない
2	クライアント業務エラー	-	出力しない
3	クライアントシステムエラー	ERROR	・発生した例外メッセージ ・例外
4	サーバ入力値検証エラー	-	出力しない
5	サーバ業務エラー	-	出力しない
6	サーバシステムエラー	ERROR	・発生した例外メッセージ ・例外

■ 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。

◆ JAX-WS RI 2.1.7(Metro1.5)を使用する場合の AP サーバの起動オプションの設定

JAX-WS 2.1.7 (Metro1.5)のデフォルト設定では、Web Fault の例外をスローすると、SOAP Fault の<detail>要素内に、Fault 自体の情報と発生した例外のスタックトレース情報の2つの要素が入ってしまう。一方、.NET クライアント側では、 <detail>要素内は1つの要素を期待しているため、SOAP 電文を正しく解析できず、FormatException になってしまう。

TERASOLUNA フレームワークでは、前述のとおり、Java サーバ AP で発生したエラーをクライアント AP に通知する際ユーザ定義の例外クラスを使用する。このため、Tomcat などの AP サーバの java VM 起動時のオプションとして、以下のシステムプロパティについて設定をオフ(=false)にすることで、クライアント AP へスタックトレースを送信しないようにする。

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

リスト 20　AP サーバの java 起動オプション

なお、JAX-WS2.2/Metor2.0 以降では、当該不具合が改修されており[2]、設定不要である。

[2] https://jax-ws.dev.java.net/issues/show_bug.cgi?id=761

■ 内部構成

◆ 構成クラス

本機能を構成するクラスを以下に示す。
本機能は、多くのフレームワーク機能を組み合わせて実現しているため、他のフレームワーク機能に属するクラスの記述は省略する。

表 9 構成クラス一覧

項番	クラス名	説明
Terasoluna.ExceptionHandling 名前空間		
1	ExceptionManager	本機能のエントリクラス
2	ExceptionUtility	例外に関するユーティリティクラス
3	IExceptionHandler	集約例外処理を実施するインタフェース
4	ReplaceExceptionHandler	例外の置き換えを行う IExceptionHandler 実装クラス
5	RethrowExceptionHandler	例外のリスローを行う IExceptionHandler 実装クラス
6	WrapExceptionHandler	例外を別の例外でラップする IExceptionHandler 実装クラス
7	ErrorInfo	エラー情報を保持するクラス
Terasoluna.ServiceModel 名前空間		
8	ITerasolunaSoapFault	TERASOLUNA フレームワークで利用する SOAP Fault の detail 要素の形式を定義するインタフェース。
9	ITerasolunaSoapFaultErrorMessage	TERASOLUNA フレームワークで利用する SOAP Fault の detail 要素に含まれるのエラーメッセージの形式を定義するインタフェース。
10	TerasolunaSoapFault	ITerasolunaSoapFault インタフェースを実装した DataContract クラス。
11	TerasolunaSoapFaultWrapper	SOAP Fault のメッセージ形式が一致すればハンドリング可能にするための ITerasolunaSoapFault 実装クラス
12	TerasolunaSoapFaultErrorMessage	ITerasolunaSoapFaultErrorMessage インタフェースを実装した DataContract クラス
13	TerasolunaSoapFaultErrorMessageWrapper	SOAP Fault のメッセージ形式が一致すればハンドリング可能にするための ITerasolunaSoapFaultErrorMessage 実装クラス。

項番	クラス名	説明
14	TerasolunaSoapFaultUtility	SOAP Fault を扱うためのユーティリティクラス。
Terasoluna.Windows.Forms.ExceptionHandling 名前空間		
15	DefaultExceptionHandler	クライアント AP 実行時に利用する IExceptionHandler のデフォルト実装クラス
16	InitializationExceptionHandler	クライアント AP の起動時に利用する IExceptionHandler のデフォルト実装クラス
17	TerminationExceptionHandler	クライアント AP の終了時に利用する IExceptionHandler のデフォルト実装クラス
18	DevelopmentTimeExceptionHandler	デバッグ情報を出力する開発時用 IExceptionHandler 実装クラス

■ 拡張ポイント

◆ 独自のスレッド処理の例外処理

TERASOLUNA フレームワークが管理しない独自スレッドを作成した場合、マニュアルスレッド(Thread クラスのインスタンスから生成)で発生した例外であれば、AppDomain.UnhandledException イベントで例外を捕捉することができるので標準の集約例外処理の設定で対応可能である。
一方、ThreadPool クラスや Delegete による非同期処理など、プールスレッド上で発生した例外は、AppDomain.UnhandledException イベントでは捕捉できない。
このため、タスクレスレッド中の処理を try-catch で囲み、catch 句で Form の BeginInvoke メソッドまたは、フレームワークの UIThreadInvoker クラスにより、タスクスレッドで発生した例外を UI スレッド(メインスレッド)へリスローする必要がある。
こうすることで、Application.ThreadException イベントで捕捉することができるので、同様に標準の集約例外処理の設定で対応可能である。
なお、「CL-03 イベント処理実行機能」の非同期実行時も内部的に同様の実装を行っており、タスクスレッドで発生した例外を集約処理できるようになっている。

```
public Form TargetForm   {get; ...}

private void DoAsync()
{
    try
    {
        //非同期で実施中の処理
    }
    catch(Exception e)
    {
        //タスクスレッドで発生した全ての例外UIスレッド上へリスロー
        UIThreadInvoker.BeginInvoke(TargetForm, new MethodInvoker(
            delegate()
            {
                throw e;
            }));
    }
}
```

リスト 21 非同期処理中で発生した例外処理の実装例

◆ クライアント入力値検証エラーの集約例外処理の拡張

「入力値検証」フェーズの実装を拡張し、イベント処理フローをカスタマイズすることで実現できる。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。
また、各 UI コントロールに対する詳細エラーの表示方法を変更するには、「CL-01 画面データ機能」の機能説明書を参照のこと。

◆ 業務エラーの集約例外処理の拡張

「エラー処理」フェーズおよび「完了処理」フェーズの実装を拡張し、イベント処理フローをカスタマイズすることで実現できる。イベント処理フローのカスタマイズ方法は、「CL-03 イベント処理実行機能」の機能説明書を参照のこと。

◆ システムエラーの集約例外処理の拡張

独自の集約例外処理を実施したい場合は、IExceptionHandler インタフェースを実装し、FW 起動構成ファイル(TerasolunaBootstrap.config)の設定を変更する。

```
public class SampleExceptionHandler : IExceptionHandler
{
    private static readonly ILogger _log = LogManager.GetLogger();
    // 例外に対する後処理を実装
    public bool HandleException(Exception exceptionToHandle, string name)
    {
        if (_log.IsErrorEnabled)
        {
            _log.Error(exceptionToHandle.Message, exceptionToHandle);
        }
        return true;
    }
}
```

リスト 22 IExceptionHandler インタフェースの実装例

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
…
  <unity>
…
    <containers>
      <container>
        <types>
          <!-- AppDomain.CurrentDomain.UnhandledExceptionイベントのポリシー -->
          <type type="IExceptionHandler" name="AppDomainUnhandledFailure"
                mapTo="Terasoluna.Sample.ExceptionHandling.SampleExceptionHandler,
                       Terasoluna.Sample" >
            <lifetime type="singleton" />
          </type>
…
        </types>
…
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 23 TerasolunaBootstrap.config の設定例

◆ 独自の SOAP Fault のハンドリング

TERASOLUNA が標準でサポートする SOAP Fault 以外のフォーマットで集約例外処理する場合は「エラー処理」フェーズの実装を拡張し、イベント処理フローをカスタマイズすることで実現できる。イベント処理フローのカスタマイズ方法は、「CL-03 イベント処理実行機能」の機能説明書を参照のこと。

◆ エラーメッセージ表示方法の変更

「CL-06 メッセージ通知機能」を拡張することによって、メッセージの表示方法を変更することができる。詳細な手順については「CL-06 メッセージ通知機能」の機能説明書を参照のこと。

◆ 標準提供のダイアログ上の固定メッセージの変更

本機能が標準で提供するエラーダイアログの「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

- CL-03 イベント処理実行機能
- CM-02 インスタンス管理機能
- CL-01 画面データ機能
- CL-04 サーバ通信機能
- CM-06 ログ出力機能
- CL-06 メッセージ通知機能