エラーチェック（バリデーション）の体系的な考え方と実装パターンについて

マイクロソフト株式会社
コンサルティングサービス統括本部
プリンシパルコンサルタント
赤間　信幸　（http://blogs.msdn.com/nakama/）

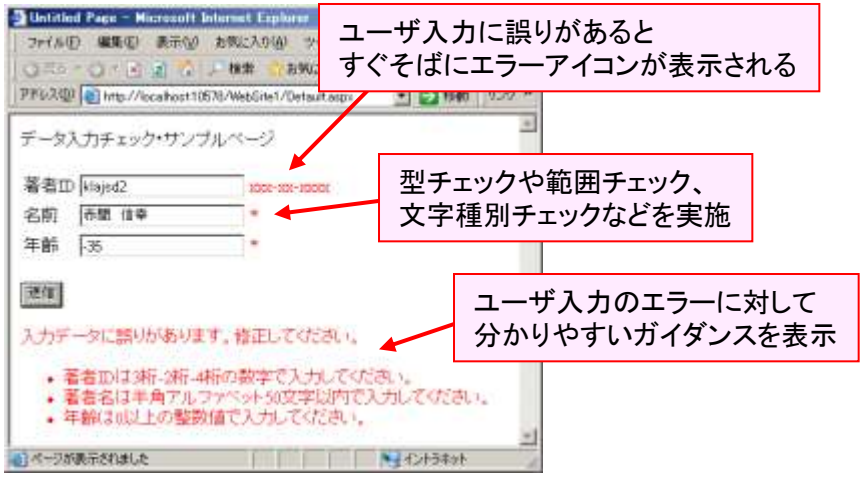
業務アプリケーションの分類

- 業務アプリケーションは、参照系／更新系に大別される
  - 参照系／更新系では、求められるアプリケーションの機能が異なる
  - 更新系では、適切なエラーチェックが実装上の重要なポイントになる

	参照系	更新系
種類	・DB からデータを取得して一覧表示 ・分かりやすく・見やすくデータを表示	・入力フィールドからデータをエントリ ・必要に応じてエラーチェックを実施
操作	・マウス操作が主体 ・ドラッグ &ドロップなども実施 ・直観的な操作（マニュアル不要）	・キーボード操作が主体 ・ファンクションキー、IME 制御、キーボードによるフォーカス制御 ・各種の視覚的なガイダンス（ウィザードやポップアップなど）
支援機能	・各種のデータのビジュアル化 －表（グリッド） －グラフ －etc. ・データの印刷（レポート）	・各種のデータ入力支援 －フリガナ変換 －郵便番号／住所変換 －エラー表示やガイダンス表示 －etc.

エラーチェック（ユーザ入力検証）の意味

- エラーチェック（ユーザ入力検証）には、2 つの目的がある
  - ① アプリケーションの保護
    - ユーザから入力された値をそのまま利用すると、エラーやセキュリティ脆弱性の原因になってしまう（SQL 挿入、Cross-Site Scripting など）
  - ② ユーザビリティの向上
    - エンドユーザに親切なエラーメッセージを表示するように作成すると、使いやすいアプリケーションを実現することができる
- しかし、場当たり的にエラーチェック機能を実装すると、生産性が大幅に損なわれる
  - このため、ランタイムが持つ機能をうまく活用して実装していくことが望ましい



エラーチェック（ユーザ入力検証）の意味 ーランタイムが持つ機能を活用するために

- .NET ランタイム（.NET Framework）が持つエラーチェック機能を活用するためには、以下の知識が欠かせない
  - A. アプリケーションの終了パターンの分類
    - 正常終了／業務エラー／システムエラーを正しく分類すること
    - 業務エラーが、さらに単体入力エラーと突合せエラーに分類されること
  - B. アーキテクチャ的な観点から見た、エラーチェックの実装方法
    - 論理 3 階層型アプリケーションにおいて、どこで何をチェックすべきか
  - C. ランタイムが持つバリデーション機能（エラーチェック機能）の狙い
    - ランタイムが持つバリデーション機能は、それぞれにコンセプトが違う
    - どの部分をカバーする目的で作られているのかの対応関係と、その限界点を理解することが重要
- これらについて、以下に順番に解説していく

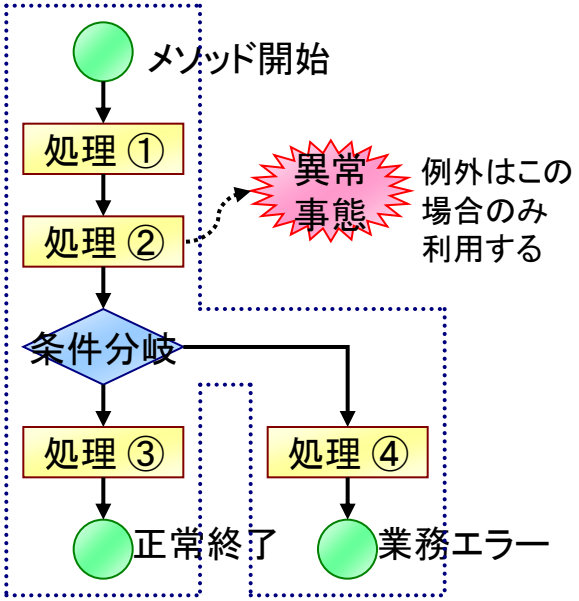
※ 詳細な実装コード・手順は、ここでは解説しません
※ 本コースでは、データ検証に対する考え方そのものを学習してください

本セミナーの目的

- 世の中に存在する、エラーチェック（バリデーション）の体系的な分類と、実装パターンの分類を理解する
  - 1. エラーチェックの体系的な分類方法
    - 正常終了／業務エラー／システムエラーの分類
    - 業務エラーの分類
  - 2. アーキテクチャから見たエラーチェックの実装場所
    - A. Web アプリケーションの場合
    - B. スマートクライアントアプリケーションの場合
  - 3. 単体入力エラーチェックの実装パターン
    - ① ASP.NET Web フォームの場合
    - ② Silverlight 3, WPF 3 の場合
    - ③ Windows フォーム 2.0, WPF 3.5 の場合

1. エラーチェックの体系的な分類方法

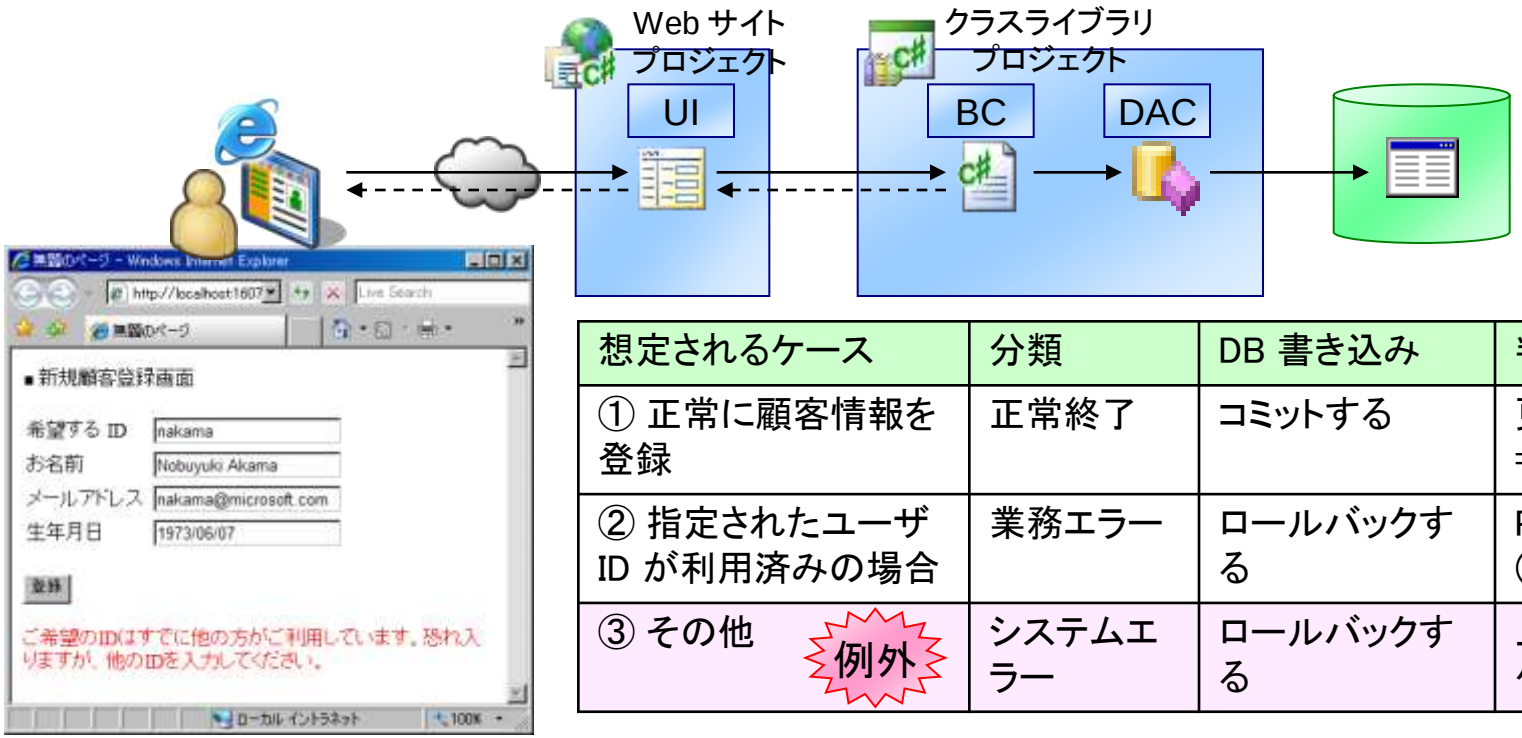
- 業務アプリケーションの終了パターンは、大別して以下の 3 通りに分類される
  - A. 正常終了：特に問題なく、期待通りに業務処理が終了できた
  - B. 業務エラー：ユーザ入力値の問題で、処理が完遂できなかった
  - C. システムエラー：システムトラブルで、処理が完遂できなかった



分類	対応するケース	.NETでの表現方法
正常終了	業務で期待された主たる処理が問題なく終了した場合	戻り値の一部として表現
業務エラー	業務設計の中で想定されている範囲内で、処理が分岐し、正常終了できなかった場合	戻り値の一部として表現
システムエラー	業務設計の想定範囲外の異常事態が発生し、アプリケーション処理を正しく遂行できなくなった場合	例外を用いて表現

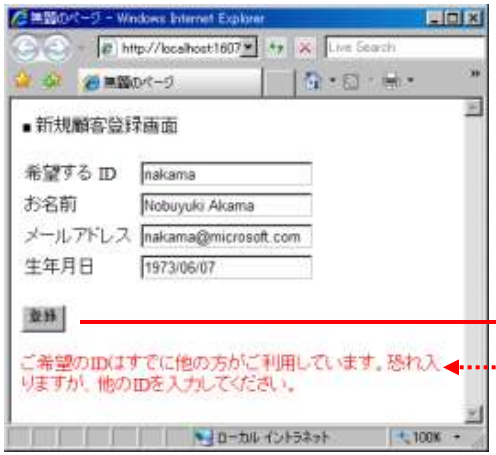
1. エラーチェックの体系的な分類方法 ー正常終了／業務エラー／システムエラーの分類

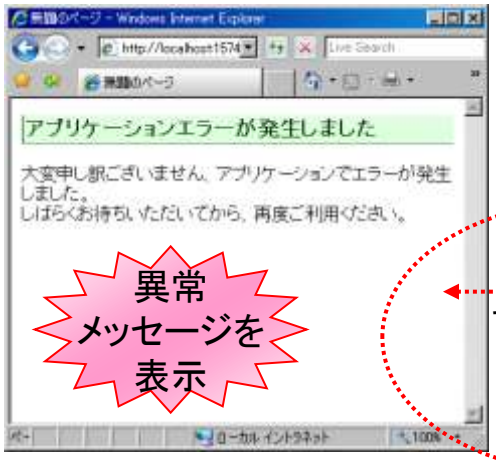
- この分類はエンドユーザへの通知方法を考えるとすぐわかる
  - 具体例）新規顧客登録業務（データエントリページ）の場合
    - 指定されたユーザ ID がすでに使われていた → 業務エラー
    - DB サーバが停止していた → システムエラー →「ごめんなさい」画面

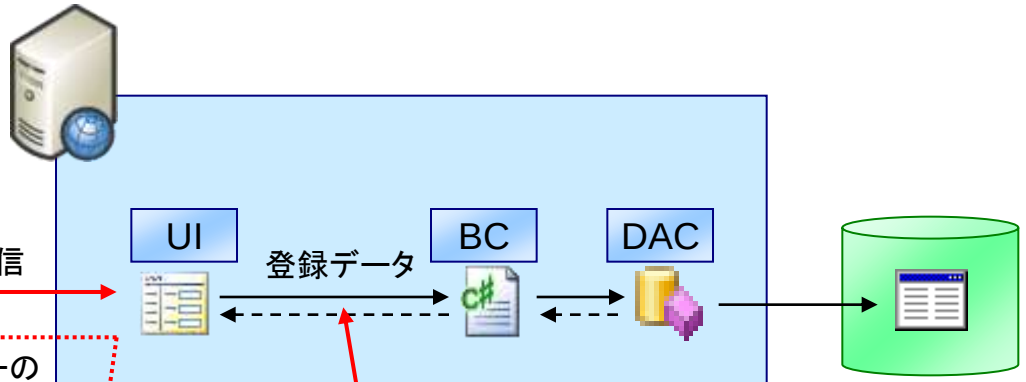


想定されるケース	分類	DB 書き込み	判定方法
① 正常に顧客情報を登録	正常終了	コミットする	更新結果行数 = 1
② 指定されたユーザ ID が利用済みの場合	業務エラー	ロールバックする	PK 制約違反 (#547 エラー)
③ その他 例外	システムエラー	ロールバックする	上記以外のケース

1. エラーチェックの体系的な分類方法
－正常終了／業務エラー／システムエラーの分類







データ送信

業務エラーの
メッセージ表示

① 正常終了：正しく登録できたケース
② 業務エラー：希望 ID が重複したケース
③ アプリケーション・システムエラー：その他
（DB サーバが動作していなかった等）

異常
事態

異常
メッセージを
表示

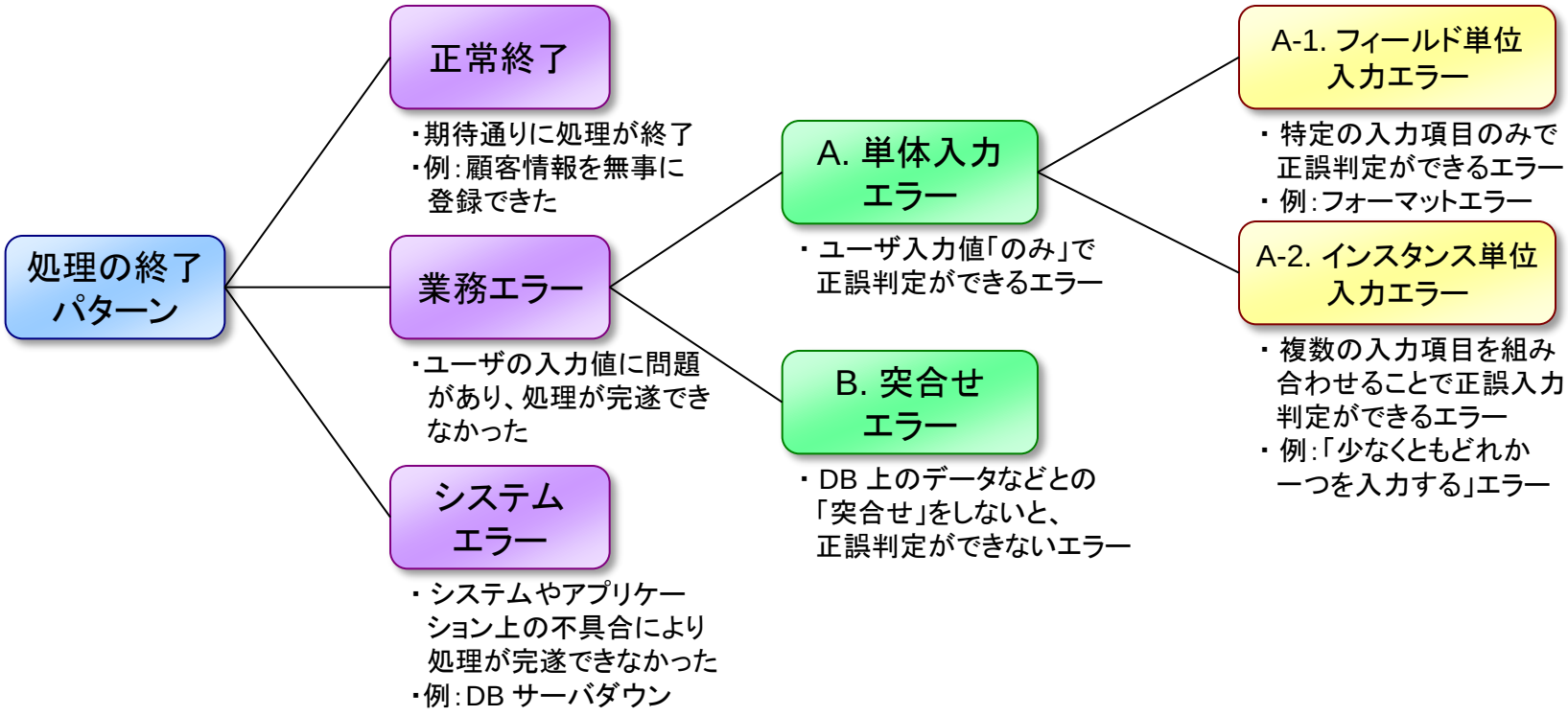
アプリケーション・
システムエラー
発生時

例外で
表現

※ .NET では、実装上のルールとして、
システムエラーの場合に限って例外を
使うのが望ましい
※ 詳細 → http://blogs.msdn.com/nakama
/archive/2008/12/29/net-part-1.aspx

1. エラーチェックの体系的な分類方法 ー業務エラーの分類

- 業務エラーは、さらに以下のように細分化される
  - この業務エラーの分類方法は、Web / Win に拠らない（極めて重要）
  - 単票形式のデータ入力フォームを取り上げて解説する

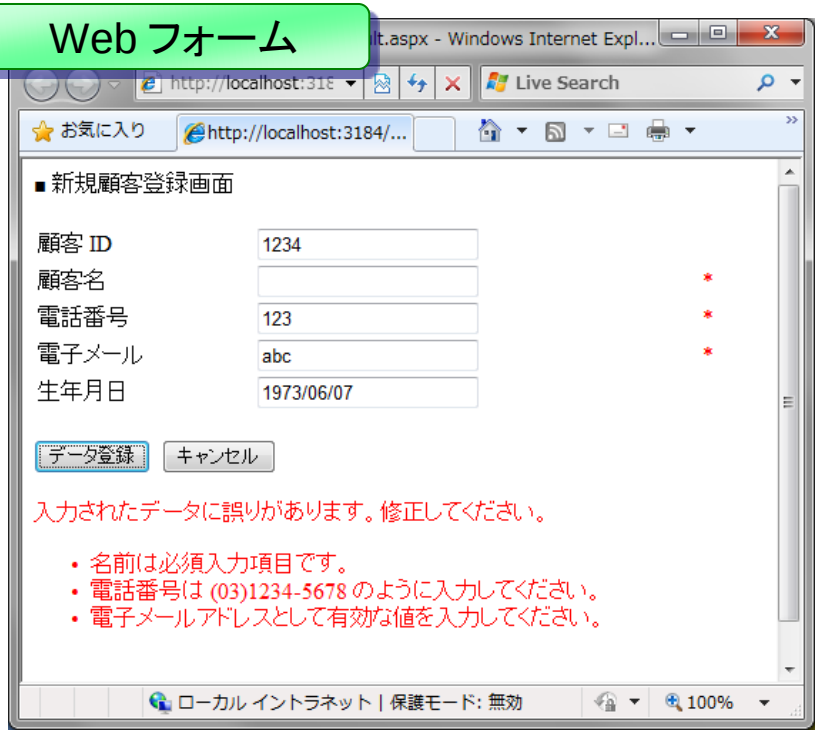


1. エラーチェックの体系的な分類方法 ー業務エラーの分類

- 具体例）新規顧客登録画面の場合
  - 以下のような新規顧客登録画面を考えてみる
  - この場合、Windows フォーム、Web フォームを問わず、データ入力に関連するエラーは次のページのように分類できる

Windows フォーム

Web フォーム



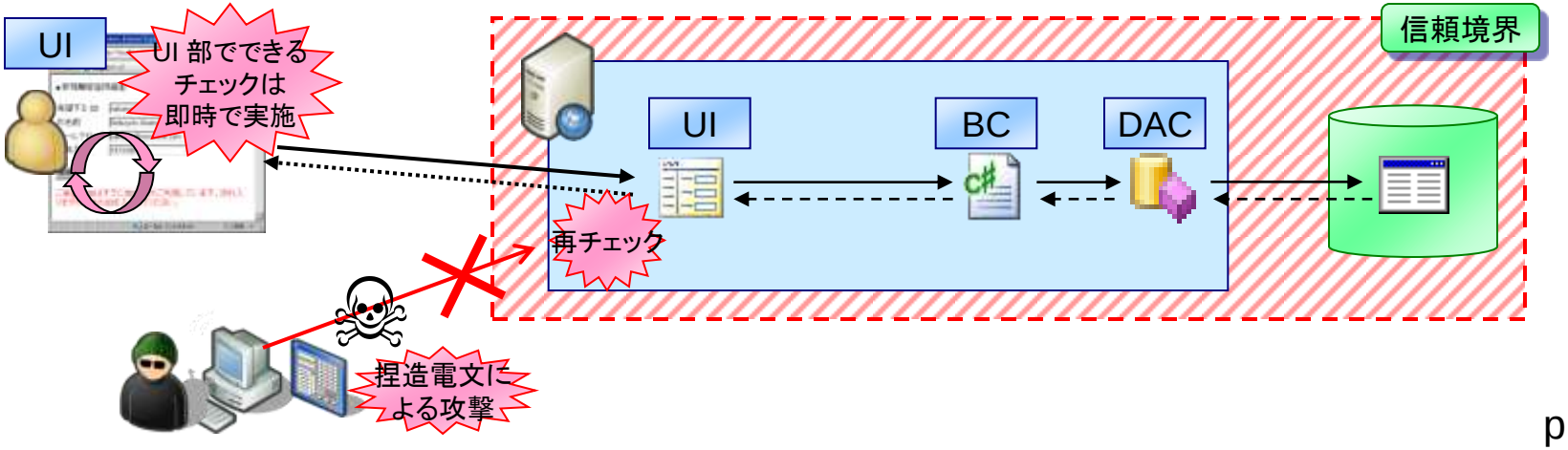
1. エラーチェックの体系的な分類方法 ー業務エラーの分類

- 具体例）新規顧客登録画面の場合（続き）

大分類	中分類	小分類	具体的なケース
正常終了			入力項目が適切であり、データベースに適切にデータが登録できた
業務エラー	A. 単体入力エラー	A-1. フィールド単位の入力エラー	顧客 ID が入力されていない
			顧客 ID が半角英数大文字 4 文字ではない
			顧客名が入力されていない
			顧客名が半角英数文字 40 文字以内ではない
			電子メールアドレスのフォーマットが正しくない
			電話番号のフォーマットが正しくない
			生年月日が日付になっていない
		A-2. インスタンス単位の入力エラー	電子メールアドレス、電話番号が両方とも入力されていない
	B. 突き合わせエラー		指定された顧客 ID がすでに DB 上に存在していた（使われていた）
システムエラー			DB サーバが停止していた
			ネットワークが切断しており、DB サーバへの接続が開けなかった
			メモリ不足が発生し、アプリケーションがクラッシュした
			（その他いろいろ．．．）（※ システムエラーは無限にパターンがあるため、洗い出しきれない）

2. アーキテクチャから見たエラーチェックの実装場所

- 前述したエラーチェックを実装する場所には、以下の 2 つの基本セオリーがある
  - エラーチェックは、可能な限り、ユーザに近い場所で行う
    - エンドユーザにとって、「UI が即時反応すること」はユーザビリティ上重要
    - このため、UI 部でできるチェックは必ず UI 部で行い、エラーを表示する
  - 信頼境界の端点では、必ずデータの再チェックを行う
    - 信頼境界（Trust Boundary）= 不正な攻撃を受ける危険性のある境界
    - 典型的には、ネットワークアクセスを受け付ける場所では必ず再チェック

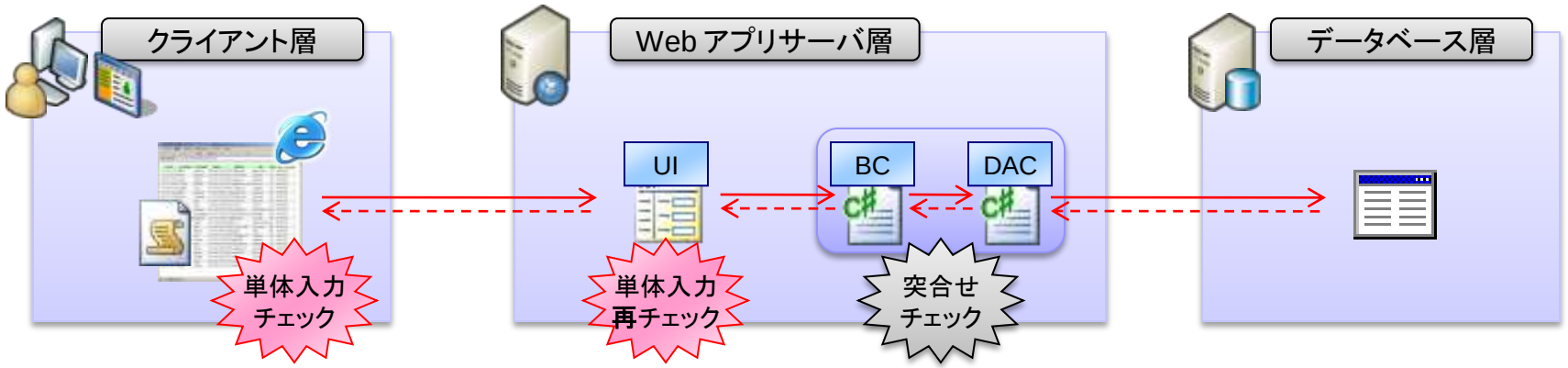


2. アーキテクチャから見たエラーチェックの実装場所 ー実装に関する基本セオリーの適用方法

- 前述の基本セオリーを、各アーキテクチャパターンに適用する方法を以下に示す
  - ① Web アプリケーションの場合
  - ② スマートクライアントアプリケーションの場合

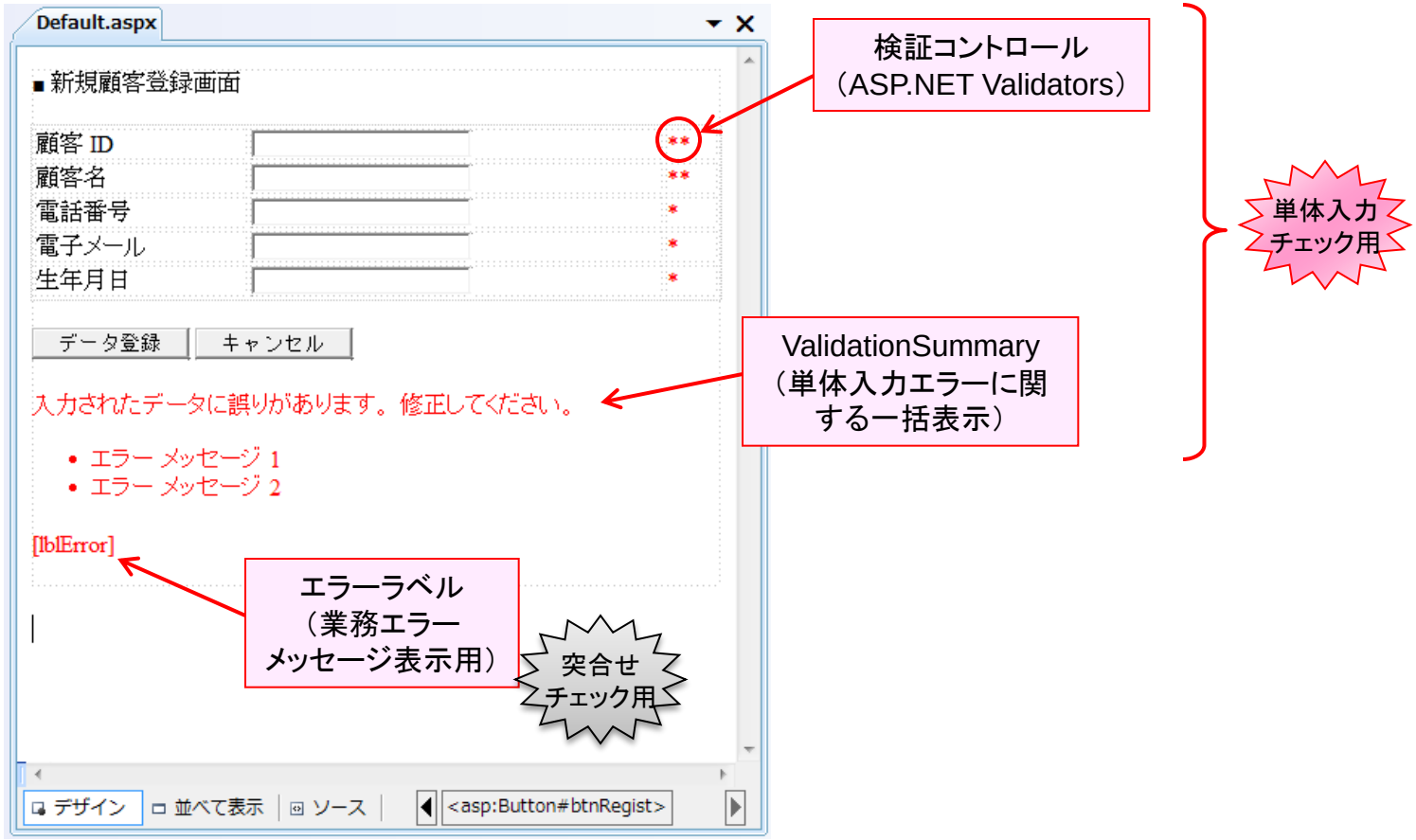
2. アーキテクチャから見たエラーチェックの実装場所 ー① Web アプリケーションの場合

- ASP.NET Web アプリケーションの基本実装パターン
  - A. 単体入力チェックについて
    - UI 部(*.aspx 上)に、検証コントロール(バリデータ)を使って実装
    - 検証コントロールが JavaScript を出力するため、クライアントでもチェックがかかる
  - B. 突合せ入力チェックについて
    - BC, DAC 部で、業務処理の一部として実装する
    - BC から UI 部に対して、業務エラー情報として返し、UI 部ではエラーラベルに表示を行う



2. アーキテクチャから見たエラーチェックの実装場所 ー① Web アプリケーションの場合

- 画面設計と実装例



2. アーキテクチャから見たエラーチェックの実装場所 ー① Web アプリケーションの場合

- 画面設計と実装例(続き)
  - UI 部 → BC 部呼び出しの部分の処理コード

C#

```
protected void btnRegist_Click(object sender, EventArgs e) {
    // ASP.NET 検証コントロールを使って、単体入力チェックを実施
    if (Page.IsValid == false) return;

    // BC 呼び出し
    CustomerBizLogic biz = new CustomerBizLogic();
    CustomerBizLogic.RegistCustomerResult result = biz.ResistCustomer(tbxId.Text,
            tbxName.Text, tbxPhone.Text, tbxMail.Text, DateTime.Parse(tbxBirthday.Text));

    // 正常終了と業務エラー（突き合わせエラー）を切り分けてメッセージ表示
    switch (result) {
        case CustomerBizLogic.RegistCustomerResult.Success:
            lblResult.Text = "正しく顧客登録を行いました。";
            break;
        case CustomerBizLogic.RegistCustomerResult.DuplicateCustomerIDError:
            lblResult.Text = "指定された ID はすでに利用されています。";
            break;
    }
}
```

UI

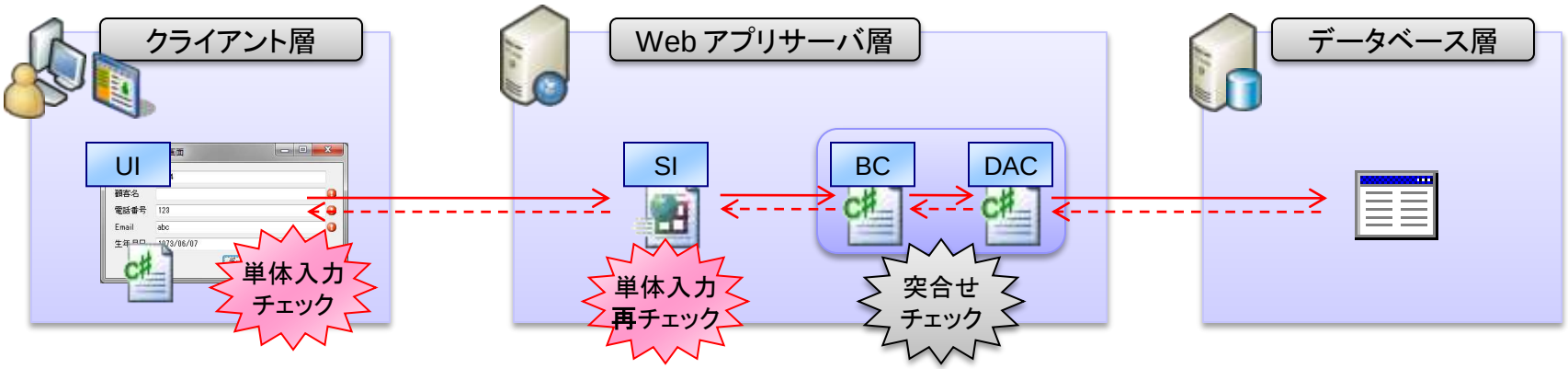
単体入力チェックを実施する

戻り値を switch 文などにより分岐させて後処理を行う

※ RIA（Silverlight など）も同様のアーキテクチャパターンとなる

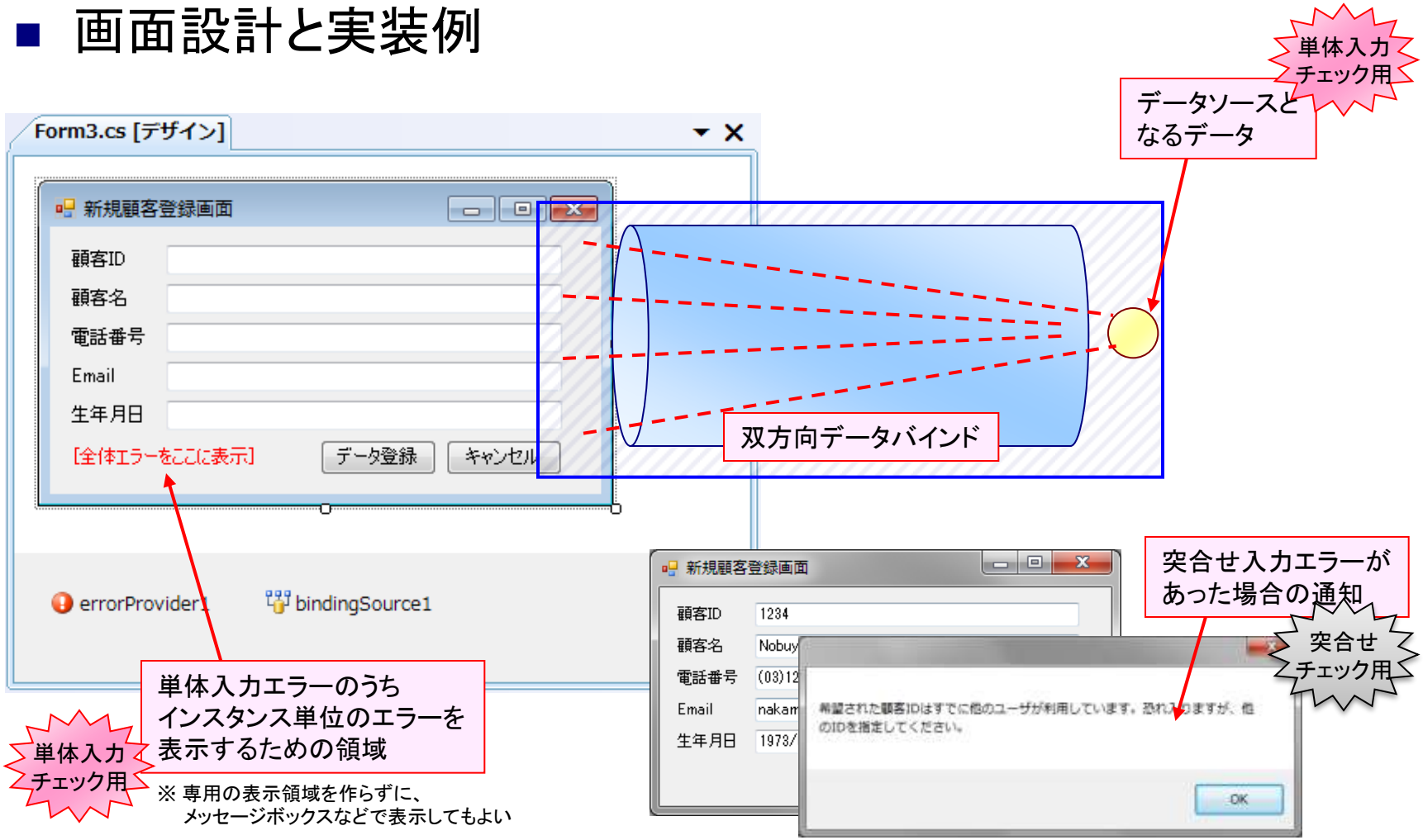
2. アーキテクチャから見たエラーチェックの実装場所 ―② スマートクライアントアプリケーションの場合

- スマートクライアントの場合の基本実装パターン
  - A. 単体入力チェックについて
    - UI 部に、双方向データバインドを使って実装
    - SI 部が信頼境界端点になるため、SI 部にも単体入力チェックを重複実装する必要がある
  - B. 突合せ入力チェックについて
    - BC, DAC 部で、業務処理の一部として実装する
    - SI から UI 部に対して、業務エラー情報として返し、UI 部ではエラーラベルに表示を行う



2. アーキテクチャから見たエラーチェックの実装場所 ー② スマートクライアントアプリケーションの場合

- 画面設計と実装例



2. アーキテクチャから見たエラーチェックの実装場所 ー② スマートクライアントアプリケーションの場合

C#

UI

```
private void btnRegist_Click(object sender, RoutedEventArgs e)
{
    // フィールド単位の単体入力再チェック
    string[] errorMessages = ValidationUtility.GetErrorMessages(this);
    if (errorMessages.Length != 0)
    {
        MessageBox.Show("入力エラーがあります。修正してください。");
        return;
    }

    // インスタンス単位の単体入力チェック
    CustomerInput ci = this.Resources["objCustomer"] as CustomerInput;
    if (ci.Email == null && ci.Phone == null)
    {
        MessageBox.Show("電話番号または電子メールアドレスの少なくとも片方は入力してください。");
        return;
    }

    // 単体入力チェックが OK なら、ビジネスロジックを呼び出す
    ServiceReference1.CustomerServiceSoapClient proxy = new ServiceReference1.CustomerServiceSoapClient();
    var result = proxy.ResistCustomer(ci.ID, ci.Name, ci.Phone, ci.Email, ci.Birthday);

    // 正常終了と業務エラー（突き合わせエラー）を切り分けてメッセージ表示
    switch (result)
    {
        case ServiceReference1.RegistCustomerResult.Success:
            MessageBox.Show("正しく顧客登録を行いました。");
            break;
        case ServiceReference1.RegistCustomerResult.DuplicateCustomerIDError:
            MessageBox.Show("指定された ID はすでに利用されています。");
            break;
    }
}
```

単体入力チェックを実施する

戻り値を switch 文などにより分岐させて後処理を行う

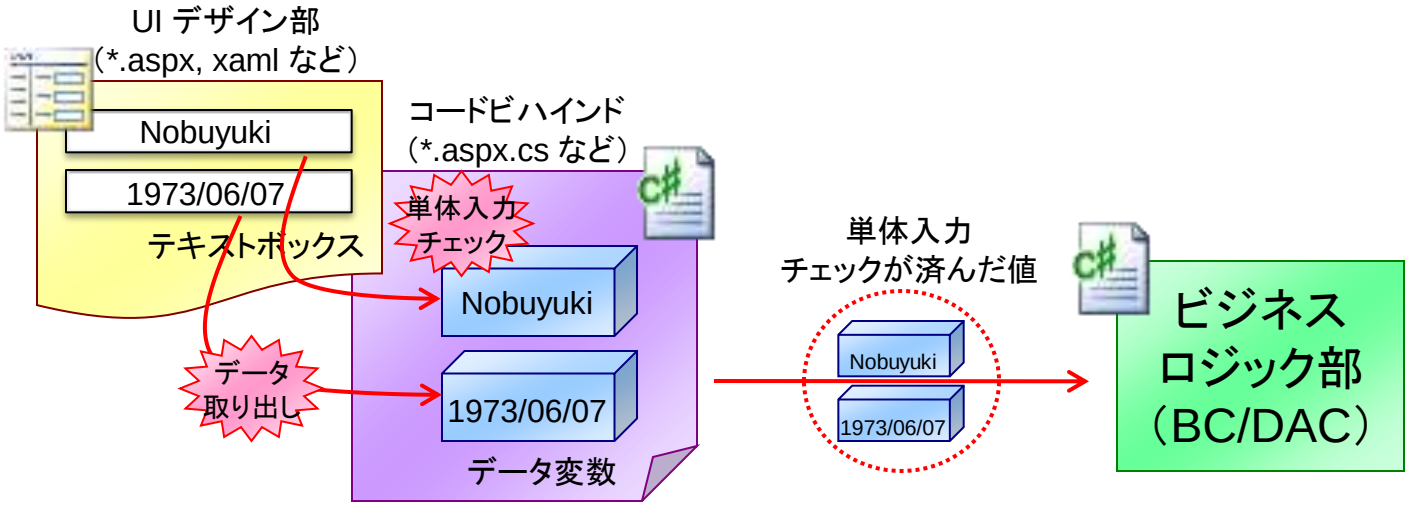
※ この実装コードは、Windows アプリ部の双方向データバインドの方式によって変化する → 後述

3. 単体入力エラーチェックの実装パターン

- UI 部の単体入力エラーチェックの実装パターンは、利用するテクノロジによって全くといっていいほど異なる
  - 「単体入力チェックを行う」ことや、「フィールド単位のチェックとインスタンス単位のチェックがある」ことは同じだが、実装方法が全く違う
  - この実装方法の特性の違いを理解しておかないと、単体入力チェックロジックを適切に実装できない
- 大別すると、実装パターンは以下の 3 種類に分類される
  - ① ASP.NET Web フォームの場合：ASP.NET 検証コントロール
    - 検証コントロールを使って、「正しい文字列」を作成する方式
  - ② Silverlight 3, WPF 3 の場合：例外ベース双方向データバインド
    - 双方向データバインドを使うが、反映に失敗するケースがある方式
  - ③ Windows フォーム 2.0, WPF 3.5 の場合：IDataErrorInfo
    - 双方向データバインドを使うが、反映に失敗するケースがない方式

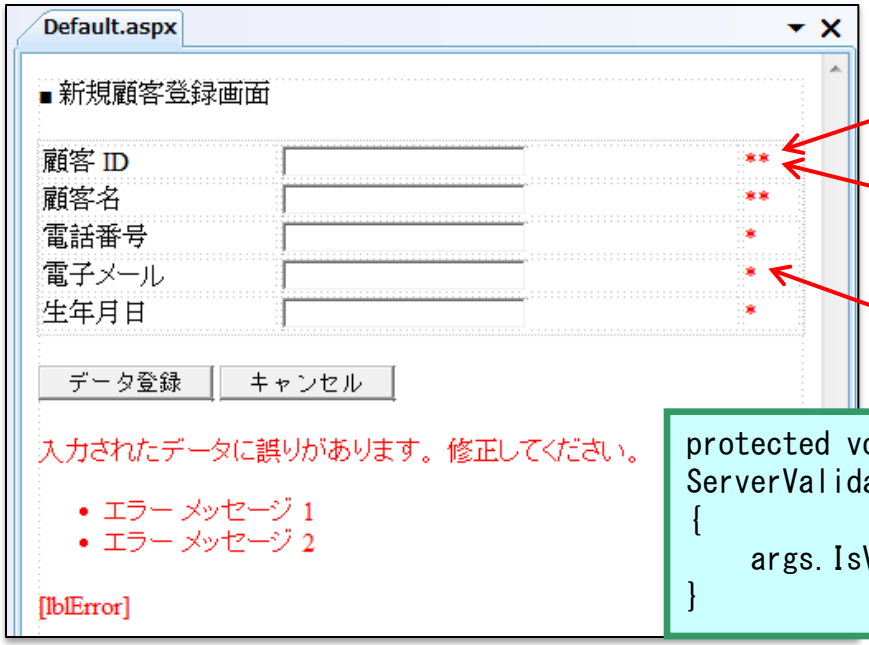
3. 単体入力エラーチェックの実装パターン ー以降の解説を読むにあたって

- 基本的に、どのテクノロジであっても、UI 部でやるべきことは以下の通り
  - UI 上のテキストボックスなどから値を入力してもらう
  - 入力された値を、コードビハインドのデータ変数に取り出す
  - 単体入力チェックが済んだ値を、BC/DAC に送出する
- これらのうち下線部のやり方が、テクノロジにより大きく違う



3. 単体入力エラーチェックの実装パターン ー① ASP.NET Web フォームの場合

- 検証コントロールを使って、単体入力チェックを実施する
  - 4 種類の標準チェックロジックが用意されている
    - 必須入力チェック、フォーマットチェック、比較チェック、範囲チェック
  - これでカバーできないときは、CustomValidator を利用して自力実装
    - インスタンス単位の単体入力チェックなどは CustomValidator で実装



必須入力チェック
RequiredFieldValidator

フォーマットチェック
RegularExpressionValidator

特殊なチェック（片方必須入力チェック）
CustomValidator

C#

```
protected void CustomValidator1_ServerValidate(object source,
ServerValidateEventArgs args)
{
    args.IsValid = !(tbxEmail.Text == "" && tbxPhone.Text == "");
}
```

3. 単体入力エラーチェックの実装パターン
ー① ASP.NET Web フォームの場合

C#

```
protected void btnRegist_Click(object sender, EventArgs e)
{
    // サーバでの単体入力チェックの再チェック
    if (IsValid == false) return;

    // UI からのデータの取り出し
    string customerID = tbxCustomerID.Text;
    string customerName = tbxCustomerName.Text;
    string phone = tbxPhone.Text;
    string email = tbxEmail.Text;
    DateTime? birthday = (tbxBirthday.Text == "" ? null :
        (DateTime?)DateTime.Parse(tbxBirthday.Text));

    // BC の呼び出し
    CustomerBizLogic biz = new CustomerBizLogic();
    CustomerBizLogic.RegistCustomerResult result = biz.ResistCustomer(
        customerID, customerName, phone, email, birthday);
    switch (result)
    {
        case CustomerBizLogic.RegistCustomerResult.Success:
            lblResult.Text = "正しく顧客登録を行いました。";
            break;
        case CustomerBizLogic.RegistCustomerResult.DuplicateCustomerIDError:
            lblResult.Text = "指定された ID はすでに利用されています。";
            break;
    }
}
```

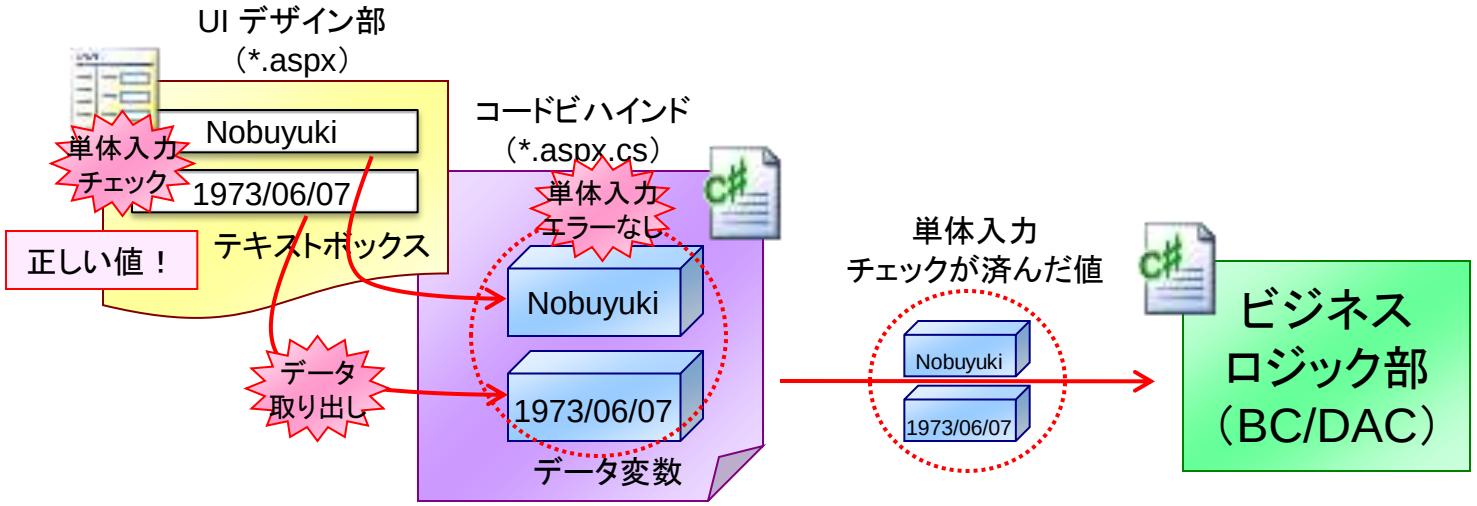
単体入力チェック

単体入力チェックが済んだテキストボックスから値を取り出すので、型変換などで失敗することが絶対にない！

ビジネスロジック部（BC/DAC）

3. 単体入力エラーチェックの実装パターン ー① ASP.NET Web フォームの場合

- ASP.NET Web フォームの検証コントロールの特徴
  - 「テキストボックスに、適切な値を作る」ように動作する
    - 検証コントロールによるチェックが通過していれば（IsValid = true なら）、データ変数への取り出しの際に失敗したりすることは絶対にない
  - すなわち、コードビハインドで値をテキストボックスから取り出す際には、すでに単体入力チェックが終わっている状態になっている！
    - ただし、データ取り出し作業自体は自力で記述する必要がある

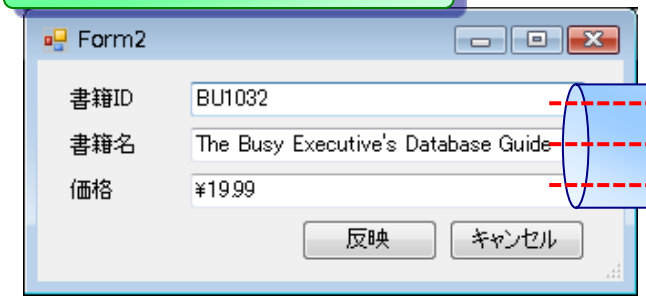


3. 単体入力エラーチェックの実装パターン ー② Silverlight 3, WPF 3 の場合

- これに対して、Silverlight などでは、双方向データバインドと呼ばれるテクニックで、データ検証とデータ取り出しを行う
  - 双方向データバインドとは、UI コントロールの表示と、データソースオブジェクト間の値をリアルタイムに同期させるための技術である

※ 技術的には以下の 2 種類があるが、ここでは単一値のみ扱う

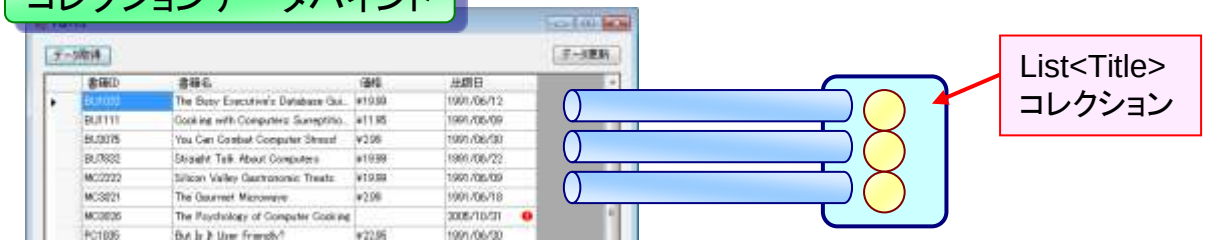
単一値データバインド



C#

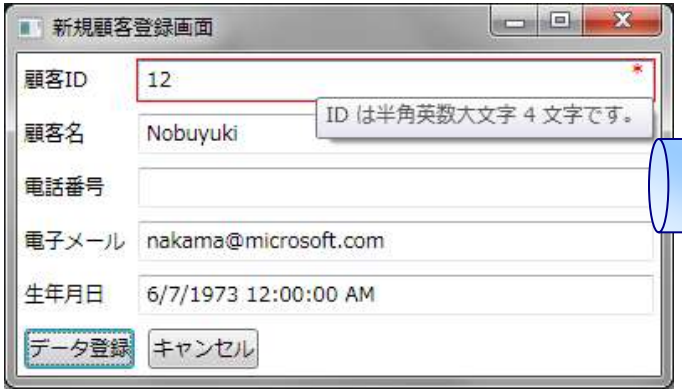
```
public class Title
{
    public string title_id { get; set; }
    public string title { get; set; }
    public decimal? price { get; set; }
    public DateTime? pubdate { get; set; }
}
```

コレクションデータバインド



3. 単体入力エラーチェックの実装パターン ー② Silverlight 3, WPF 3 の場合

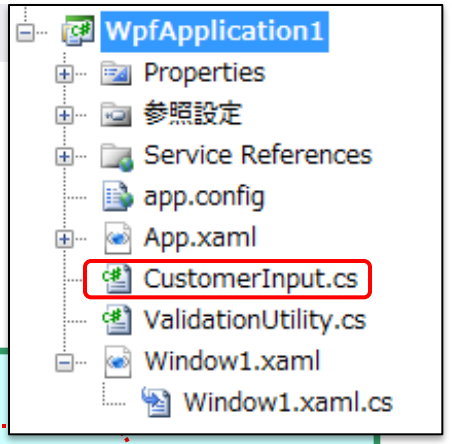
- Silverlight 3 や WPF 3 の場合には、バインドするオブジェクト側に、フィールド単位のデータチェックロジックを持たせる
  - 双方向データバインドの "ValidatesOnException" 機能を使う
    - データ反映に失敗した場合に、例外メッセージをエラーとして表示できる
  - これにより、単体入力データチェックのうち、フィールド単位の入力チェックができる



C#

```
public class CustomerInput
{
  private string _id;
  public string ID
  {
    get { return _id; }
    set
    {
      if (value == null)
        throw new ArgumentException("ID は必須入力項目です。");
      if (Regex.IsMatch(value, @"^[0-9A-Z]{4}$") == false)
        throw new ArgumentException("ID は半角英数大文字 4 文字です。");
      _id = value;
    }
  }
...
```

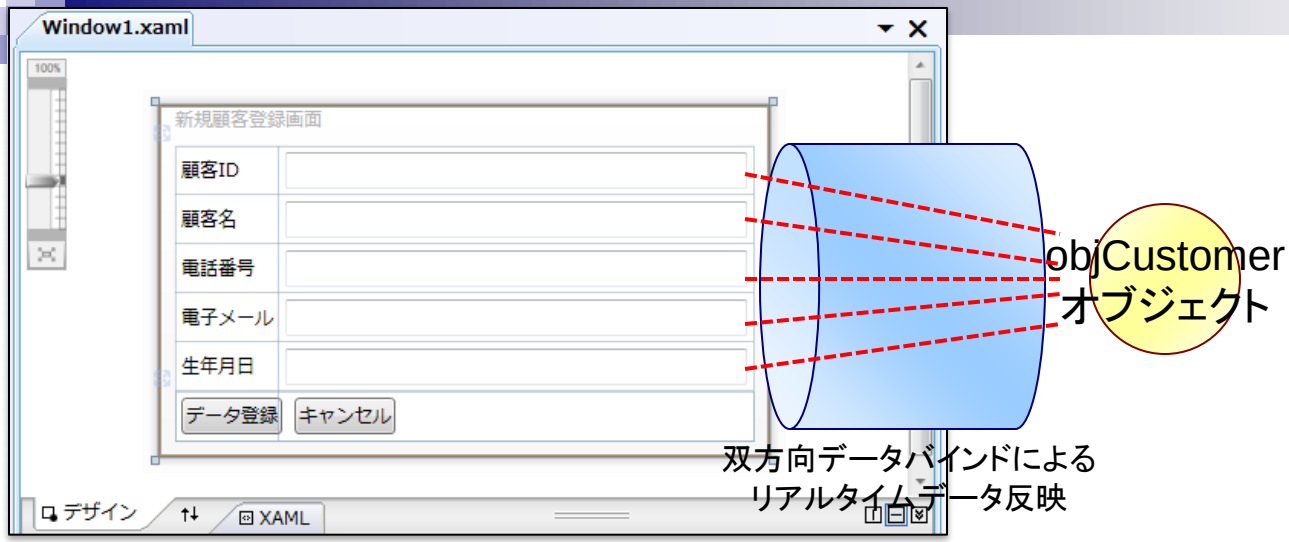
3. 単体入力エラーチェックの実装パターン ー② Silverlight 3, WPF 3 の場合

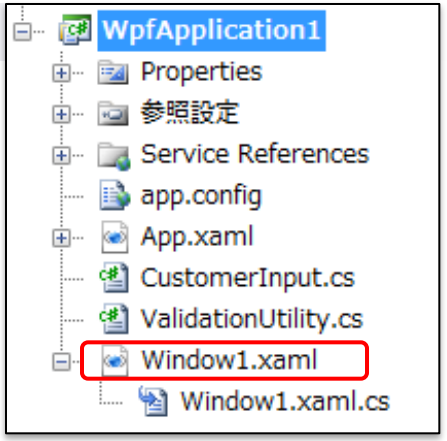



```
public class CustomerInput
{
    private string _id;
    public string ID
    {
        get { return _id; }
        set
        {
            if (value == null) throw new ArgumentException("ID は必須入力項目です。");
            if (Regex.IsMatch(value, @"^[0-9A-Z]{4}$") == false)
              throw new ArgumentException("ID は半角英数大文字 4 文字です。");
            _id = value;
        }
    }

    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            if (value == null || value == "") throw new ArgumentException("名前は必須入力項目です。");
            if (Regex.IsMatch(value, @"^[¥u0020-¥u007e]{1,40}$") == false)
              throw new ArgumentException("名前は半角英数文字 40 字以内で入力してください。");
            _name = value;
        }
    }

    ....
```

- フィールド単位の入力チェック機能を実装
- フィールドに不適切な値が入力されそうになったら例外を発生させる

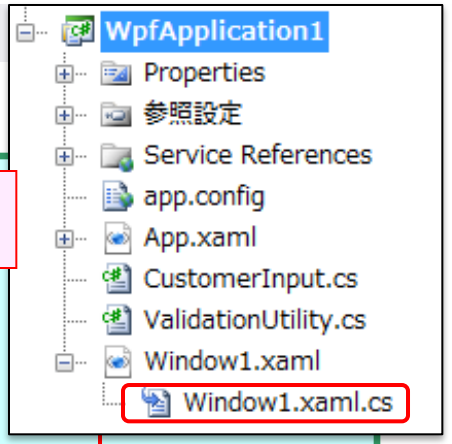
例外ベースの双方向バインドオブジェクト





XAML

```
<Window x:Class="WpfApplication1.Window1" ...>
    <Window.Resources>
        <src:CustomerInput x:Key="objCustomer" />
    </Window.Resources>
    <Grid>
        ....
        <TextBox Grid.Row="0" Grid.Column="1" Margin="4" Text="{Binding Source={StaticResource objCustomer}, Path=ID,
Mode=TwoWay, ValidatesOnExceptions=True}" />
        <TextBox Grid.Row="1" Grid.Column="1" Margin="4" Text="{Binding Source={StaticResource objCustomer}, Path=Name,
Mode=TwoWay, ValidatesOnExceptions=True}" />
        <TextBox Grid.Row="2" Grid.Column="1" Margin="4" Text="{Binding Source={StaticResource objCustomer}, Path=Phone,
Mode=TwoWay, ValidatesOnExceptions=True}" />
        <TextBox Grid.Row="3" Grid.Column="1" Margin="4" Text="{Binding Source={StaticResource objCustomer}, Path=Email,
Mode=TwoWay, ValidatesOnExceptions=True}" />
        <TextBox Grid.Row="4" Grid.Column="1" Margin="4" Text="{Binding Source={StaticResource objCustomer},
Path=Birthday, Mode=TwoWay, ValidatesOnExceptions=True}" />
        <StackPanel Grid.Row="5" Grid.Column="0" Grid.ColumnSpan="2" Orientation="Horizontal">
            <Button x:Name="btnRegist" Click="btnRegist_Click" Content="データ登録" Margin="4" />
            <Button x:Name="btnCancel" Click="btnCancel_Click" Content="キャンセル" Margin="4" />
        </StackPanel>
    </Grid>
</Window>
```



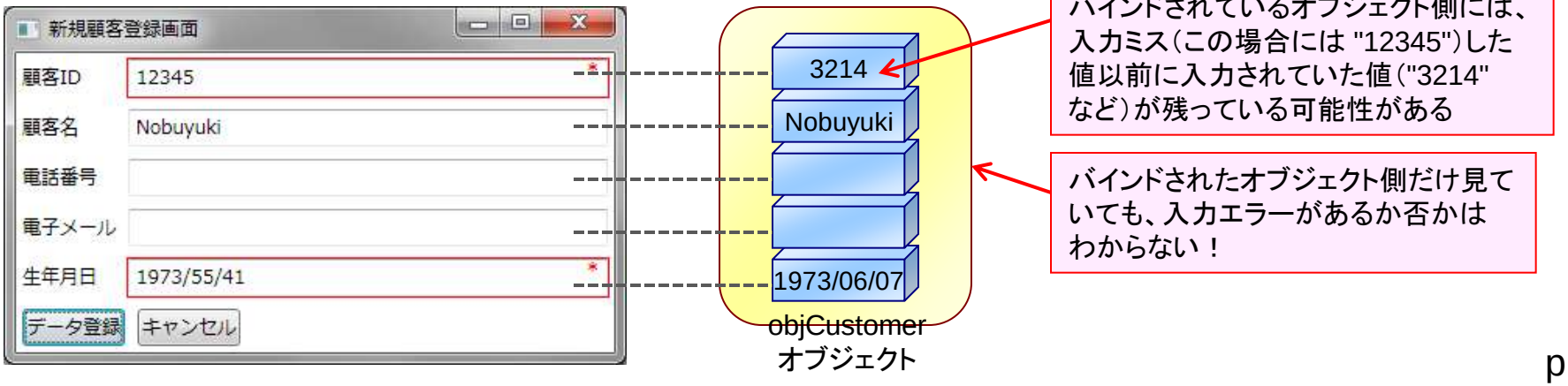
【フィールド単位の単体入力チェック】
画面上のデータバインドを再度行わせることで実施

【インスタンス単位の単体入力チェック】
バインドされているオブジェクトの中のデータを再チェック

```
private void btnRegist_Click(object sender
{
    // フィールド単位の単体入力再チェック
    string[] errorMessages = ValidationUtility.GetErrorMessages(this);
    if (errorMessages.Length != 0)
    {
        MessageBox.Show("入力エラーがあります。修正してください。");
        return;
    }

    // インスタンス単位の単体入力チェック
    CustomerInput ci = this.Resources["objCustomer"] as CustomerInput;
    if (ci.Email == null && ci.Phone == null)
    {
        MessageBox.Show("電話番号または電子メールアドレスの少なくとも片方は入力してください。");
        return;
    }

    // 単体入力チェックが OK なら、ビジネスロジックを呼び出す
    ServiceReference1.CustomerServiceSoapClient proxy = new ServiceReference1.CustomerServiceSoapClient();
    var result = proxy.ResistCustomer(ci.ID, ci.Name, ci.Phone, ci.Email, ci.Birthday);

    // 正常終了と業務エラー（突き合わせエラー）を切り分けてメッセージ表示
    switch (result)
    {
        case ServiceReference1.RegistCustomerResult.Success:
            MessageBox.Show("正しく顧客登録を行いました。");
            break;
        case ServiceReference1.RegistCustomerResult.DuplicateCustomerIDError:
            MessageBox.Show("指定された ID はすでに利用されています。");
            break;
    }
}
```

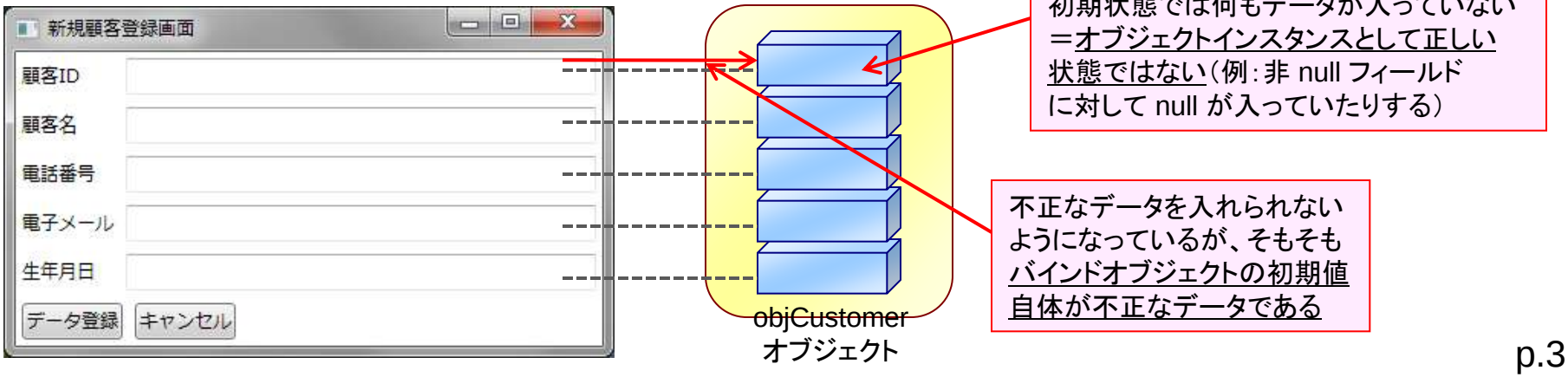
3. 単体入力エラーチェックの実装パターン ー② Silverlight 3, WPF 3 の場合

- 例外ベースの双方向データバインドには、以下のような注意点がある
  - 注意点 1. 双方向データバインドであるにもかかわらず、UI コントロールとバインドオブジェクトの間に、ずれが発生する危険性がある
    - 本来、データバインド＝二点間のデータの同期を保つための技術
    - しかし、誤ったデータが UI から入力された時は、反映が行われないため、UI 表示とバインドオブジェクトのデータがずれることがある
    - このため、入力エラーがあるか否かは、バインドオブジェクトだけを見ていても分からない



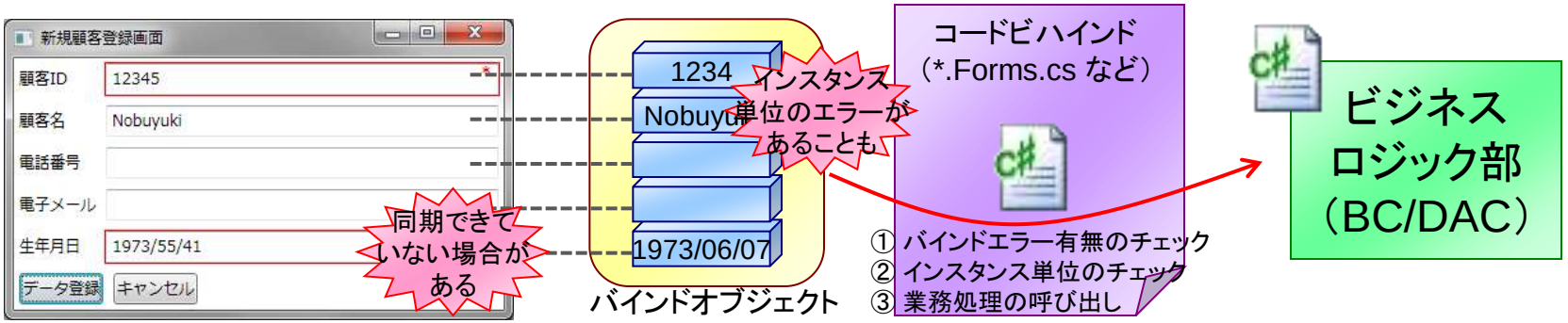
3. 単体入力エラーチェックの実装パターン ー② Silverlight 3, WPF 3 の場合

- 注意点 2. バインドオブジェクトが、必然的に不整合状態に陥っていることがありうる
  - 例外ベースの検証＝不正なデータをバインドオブジェクトが受け付けないようになっている、ということ
  - しかし、そもそも入力フォームの表示直後のバインドオブジェクトは、何も入力されていない＝オブジェクトとして正しい状態ではない
  - あるいは、入荷予定日と出荷予定日を入力するような場合、フィールド間の大小比較関係は、片方ずつデータ入力されてもうまくチェックできない
  - 結果的に、インスタンス全体チェックは、イベントハンドラでの実装が必要



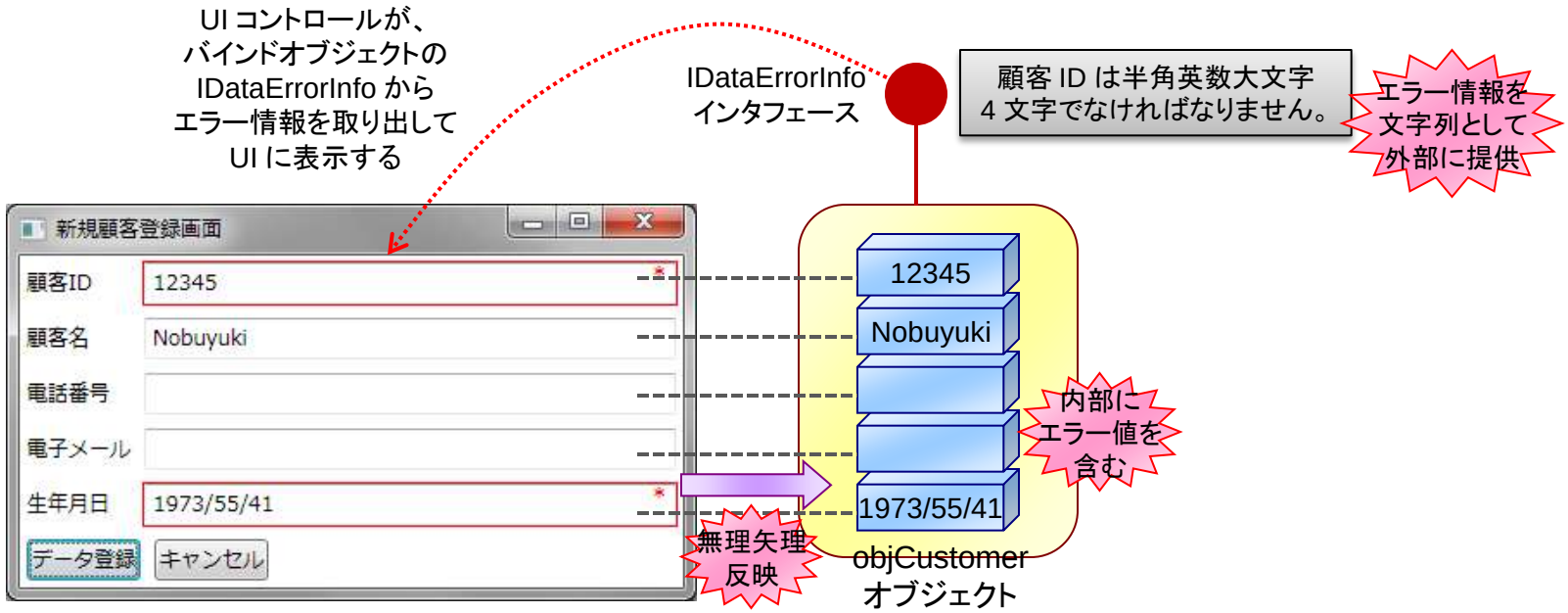
3. 単体入力エラーチェックの実装パターン ー② Silverlight 3, WPF 3 の場合

- つまり、ここまでの解説をまとめると、例外ベースの双方向データバインドの動作イメージは以下の通りになる
  - バインドエラーがない場合に限り、UI からの入力がすべてバインドオブジェクトに反映されている
    - このためイベントハンドラ内では、まずバインドエラーのチェックが必要
  - 仮にバインドエラーがなかったとしても、インスタンス単位のチェックをイベントハンドラ内で行う必要がある
- このように、例外ベースの双方向データバインドは、実装がすっきりしないところがある



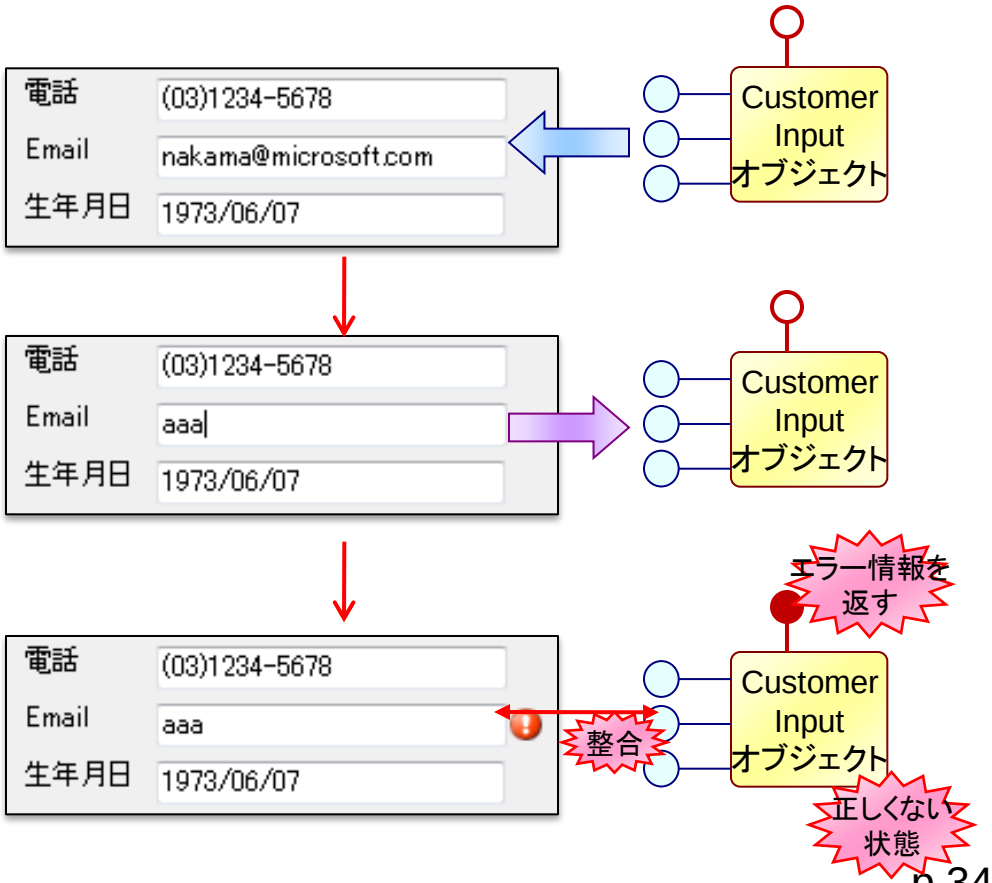
3. 単体入力エラーチェックの実装パターン ー③ Windows フォーム 2.0, WPF 3.5 の場合

- こうした問題を解決するため、Windows フォーム 2.0 や WPF 3.5 では、IDataErrorInfo 入力検証がサポートされた
  - IDataErrorInfo インタフェースは、オブジェクトインスタンス内部にエラーが含まれていることを、文字列情報として返すためのもの
  - これを使うことにより、前述の問題をきれいに解決することができる



3. 単体入力エラーチェックの実装パターン ー③ Windows フォーム 2.0, WPF 3.5 の場合

- IDataErrorInfo オブジェクトを使った双方向データバインドは、下図のように動作する
  - 入力値が正しかろうと間違っていようと、とにかくオブジェクトに反映してしまう
  - オブジェクトインスタンスが不正な状態にある場合にはこれを IDataErrorInfo インタフェースから公開する
  - これにより、常に UI とオブジェクト内の値とが同期される
  - 具体的な実装 → 次ページ



C#

```
public class CustomerInput : IDataErrorInfo
{
    private Dictionary<string, string> _errors = new Dictionary<string, string>();

    private string _id;
    public string ID
    {
        get { return _id; }
        set
        {
            _id = value;
            if (_id == null)
            {
                _errors["ID"] = "ID は必須入力項目です。";
            }
            else if (Regex.IsMatch(value, @"^[0-9A-Z]{4}$") == false)
            {
                _errors["ID"] = "ID は半角英数大文字 4 文字です。";
            }
            else
            {
                _errors["ID"] = null;
            }
        }
    }

    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            if (_name == null || _name == "")
            {
                _errors["Name"] = "名前は必須入力項目です。";
            }
```

単体入力エラーがある値であっても、とりあえず受け付けてデータの同期を図る

当該入力値が不適切な場合には、エラー情報をため込んでおく

IDataErrorInfo ベースの双方向バインドオブジェクト

```
            else
            {
                _errors["Name"] = null;
            }
        }
    }

    private string _email;
    public string Email
    {
        get { return _email; }
        set
        {
            _email = value;
            if (value == null || Regex.IsMatch(value, @"¥w+([-+.']¥w+)*@¥w+([-.]¥w+)*¥.¥w+([-.]¥w+)*"))
            {
                _errors["Email"] = null;
            }
            else
            {
                _errors["Email"] = "電子メールアドレスとして有効な値を入力してください。";
            }
        }
    }

    private string _phone;
    public string Phone
    {
        get { return _phone; }
        set
        {
            _phone = value;
            if (value == null || Regex.IsMatch(value, @"(0¥d{1,4}-|¥(0¥d{1,4}¥) ?)?¥d{1,4}-¥d{4}"))
            {
                _errors["Phone"] = null;
            }
            else
```

IDataErrorInfo ベースの双方向バインドオブジェクト

```
            {
                _errors["Phone"] = "電話番号は (03)1234-5678 のように入力してください。";
            }
        }
    }

    public DateTime? Birthday { get; set; }

    // 全体整合チェック
    public string Error
    {
        get
        {
            if (_email == null && _phone == null)
            {
                return "電子メールアドレスか電話番号かのいずれか一方は必須入力です。";
            }
            else
            {
                return null;
            }
        }
    }

    public bool HasErrors
    {
        get { return (_errors.Count != 0 || Error != null); }
    }

    public string this[string columnName]
    {
        get
        {
            return (_errors.ContainsKey(columnName) ? _errors[columnName] : null);
        }
    }
}
```

IDataErrorInfo インタフェース経由でエラー情報を返すための処理

オブジェクト全体に対するデータ検証内容は、ここに記述する
（エラーがない場合には null を返す）

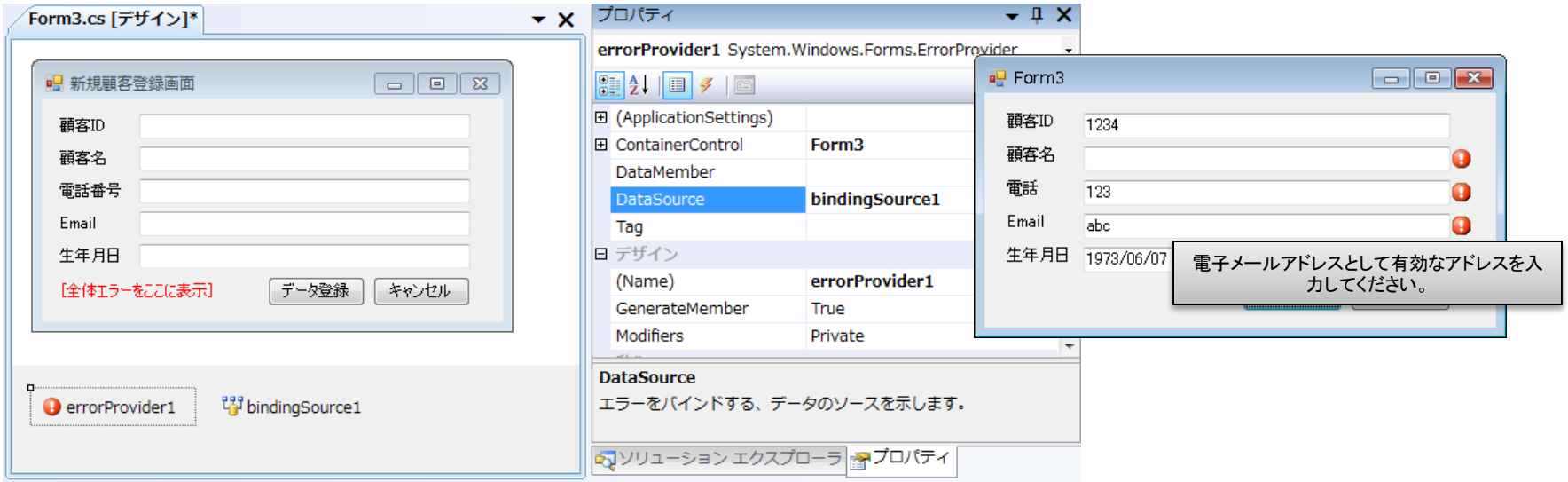
※ このメソッドは必須ではないが、実装しておくと UI 実装がラクになる

特定カラム（プロパティ）にエラーがある場合には、そのエラーの情報をメッセージで返す
※ ここから入手されるエラー情報は、ErrorProvider と BindingSource により自動的にアイコンで表示される

IDataErrorInfo ベースの双方向バインドオブジェクト

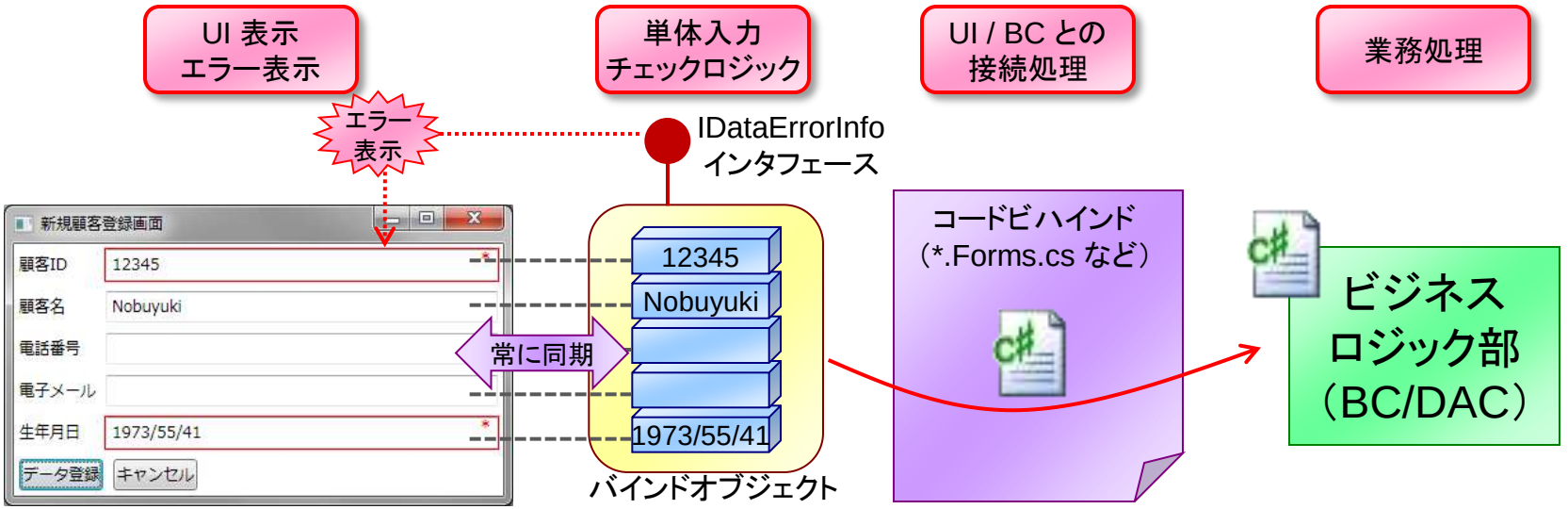
3. 単体入力エラーチェックの実装パターン ー③ Windows フォーム 2.0, WPF 3.5 の場合

- 具体例）Windows フォーム 2.0 の場合の IDataErrorInfo 双方向データバインドの実装方法
  - 前述のように作成した IDataErrorInfo オブジェクトを BindingSource コントロールにより UI コントロール群と接続する
  - さらに画面上に ErrorProvider コントロールを貼り付けておくと、これが自動的にエラー情報をチェックし、アイコンなどの表示をしてくれる



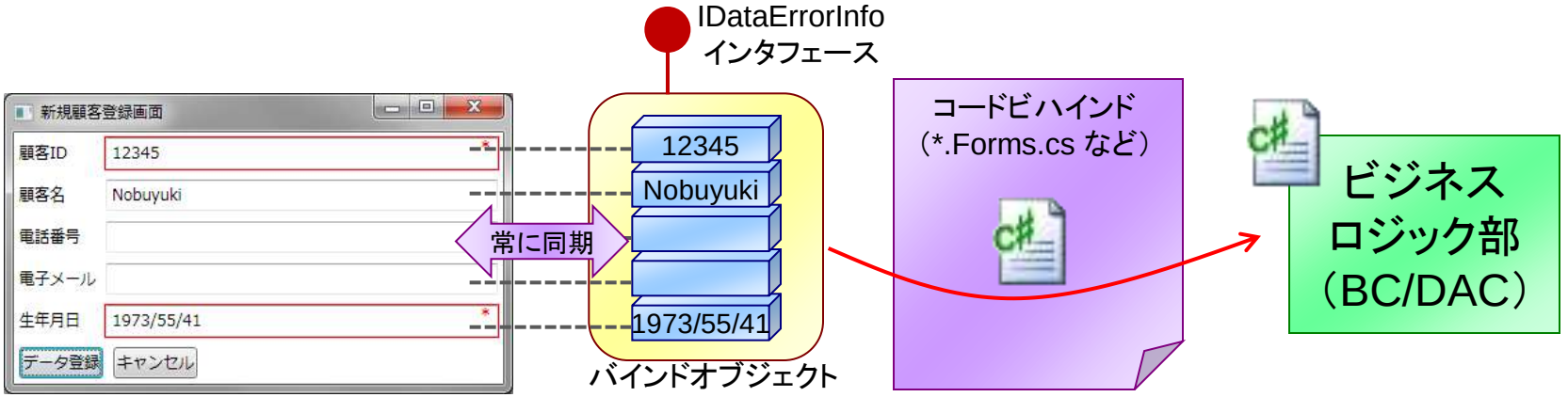
3. 単体入力エラーチェックの実装パターン ー③ Windows フォーム 2.0, WPF 3.5 の場合

- IDataErrorInfo ベースの双方向データバインドには、以下のようなメリットがある
  - 単体入力チェック処理を、バインドオブジェクトに固めることができる
    - このため、モジュールの役割分担が明確になる（MVC 的なモデル）
    - しかも、単体入力チェックロジック部分だけを重点的に単体機能テストすることもできる



3. 単体入力エラーチェックの実装パターン ー③ Windows フォーム 2.0, WPF 3.5 の場合

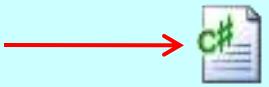
- 入力仕掛り状態の維持が簡単にできる
  - バインドオブジェクトをそのままシリアル化して保存すれば、入力しかけのデータをそのまま保存しておくこともできる
- コードビハインドの記述が簡単になる
  - コードビハインドのイベントハンドラでは、バインドオブジェクトだけを操作すればよく、UI コントロールを触る必要がない
  - このため、コードビハインドのコードの見通しも非常によくなる
  - → 次ページ参照



UI

```
public partial class Form1 : Form // ※ 一部コードを省略
{
    private CustomerInput ci;

    private void Form1_Load(object sender, EventArgs e)
    {
        ci = new CustomerInput();
        bindingSource1.DataSource = ci;
    }

    private void bindingSource1_BindingComplete(object sender, BindingCompleteEventArgs e)
    {
        lblError.Text = ci.Error;
    }

    private void button1_Click(object sender, EventArgs e)
    {
        // 単体入力チェックの結果を確認
        if (ci.HasErrors)
        {
            MessageBox.Show("入力データに誤りがあります。修正してください。");
            return;
        }

        // 単体入力チェックが OK なら、ビジネスロジックを呼び出す
        localhost.CustomerService proxy = new localhost.CustomerService();
        var result = proxy.ResistCustomer(ci.ID, ci.Name, ci.Phone, ci.Email, ci.Birthday);

        // 正常終了と業務エラー（突き合わせエラー）を切り分けてメッセージ表示
        switch (result)
        {
            case localhost.RegistCustomerResult.Success:
                MessageBox.Show("正しく顧客登録を行いました。");
                break;
            case localhost.RegistCustomerResult.DuplicateCustomerIDError:
                MessageBox.Show("指定された ID はすでに利用されています。");
                break;
        }
    }
}
```

単体入力チェックを実施する



戻り値を switch 文などにより分岐させて後処理を行う

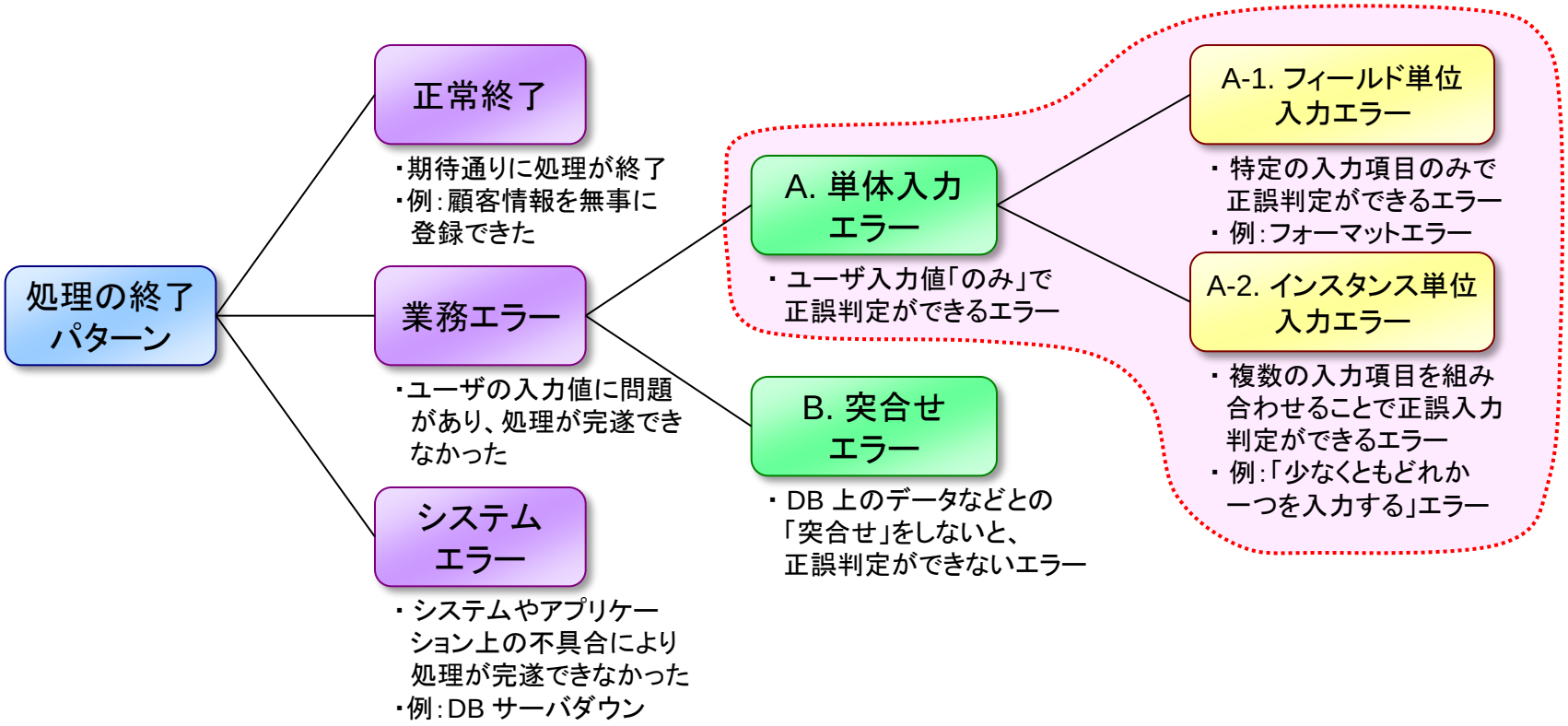
3. 単体入力エラーチェックの実装パターン ー3 つの実装方式の比較

- これらの 3 つの実装方式は、単体入力チェックに対する考え方やアプローチが異なるため、違いを理解することが大切

	① ASP.NET 検証コントロール	② 例外ベースの双方向データバインド	③ IDataErrorInfo ベースの双方向データバインド
コンセプト	正しい入力値を持ったテキストを作る	正しい値しか設定できないバインドオブジェクトを使う	正しくない値も設定できるバインドオブジェクトを使う
双方向バインド	×	○	○
値の同期	なし	△ 部分的	○ 完全
フィールド単位の検証ロジック	○ 検証コントロールで実装（必要に応じて CustomValidatorを利用）	○ 例外でチェック	○ IDataErrorInfo でバインドオブジェクトに実装
インスタンス単位の検証ロジック	○ 検証コントロールで実装（必要に応じて CustomValidatorを利用）	× イベントハンドラ側で記述する	○ IDataErrorInfo でバインドオブジェクトに実装
イベントハンドラ実装	・IsValid で単体入力チェック ・テキストボックスなどからデータ値を取り出す（必要に応じて型変換も実施） ・BC を呼び出す	・フィールド単位のデータバインドをまとめて再チェック ・バインドオブジェクトの、インスタンス単位としての有効性をチェック ・BC を呼び出す	・バインドしているオブジェクトの HasErrors プロパティをチェック ・BC を呼び出す

まとめ

- 業務アプリの終了パターンは、以下のように分類される
  - .NET Framework が持つエラーチェック（入力検証機能）は、このうち単体入力エラーの制御の部分に特化している



まとめ

- 単体入力チェックに対するアプローチは、ランタイムによってかなり異なる
  - ① ASP.NET Web フォームの場合：ASP.NET 検証コントロール
    - 検証コントロールを使って、「正しい文字列」を作成する方式
  - ② Silverlight 3, WPF 3 の場合：例外ベース双方向データバインド
    - 双方向データバインドを使うが、反映に失敗するケースがある方式
  - ③ Windows フォーム 2.0, WPF 3.5 の場合：IDataErrorInfo
    - 双方向データバインドを使うが、反映に失敗するケースがない方式
- それぞれに特徴があるので、データ検証に対する考え方をよく理解した上で活用することが重要

参考情報

- ① ASP.NET 検証コントロール
  - Visual Studio 2005 による Web アプリケーション構築技法
    - 第 5 章「入力検証コントロール」
- ② 例外ベースの双方向データバインド
  - MSDN : データバインディング（Silverlight 2）
    - http://msdn.microsoft.com/ja-jp/library/cc278072(VS.95).aspx
  - データ バインドと WPF でデータの表示をカスタマイズする
    - http://msdn.microsoft.com/ja-jp/magazine/cc700358.aspx
- ③ IDataErrorInfo ベースの双方向のデータバインド
  - スマートクライアントにおける単体入力データ検証
    - http://blogs.msdn.com/nakama/archive/2009/02/26/part-2.aspx