JFace Data Binding

1. JFace Data Binding とは 2
2. メリット・デメリット 4
3. 仕組みと構成要素 10
  3.1. JFace Data Binding の仕組み 10
  3.2. オブザーバブル 11
  3.3. バインディング 12
  3.4. コンバータ 13
  3.5. バリデータ 18
4. JFace Data Binding の応用 26
  4.1. Table と TableViewer でのデータバインディング 26
5. 今後の展望 28
6. 参考文献 28

1. JFace Data Binding とは

JFace Data Binding は、JFace および SWT の画面部品とデータを同期させる仕組みを提供しています。

JFace Data Binding を使用しない場合は、画面部品にデータを表示、またユーザによって入力された内容を受け取る時に JavaBean と画面部品のデータの受け渡しを手動で行なうことになります。JFace Data Binding を使用する場合では、JavaBean と画面部品のデータが自動的に受け渡され、データの同期が自動的に行われます。

下記に JFace Data Binding を使用する場合のデータの流れと、使用しない場合のデータの流れを図に表しました。この図の矢印はデータの流れを表しています。

図 1：JFace Data Binding を使用しない場合のデータの受け渡しの流れ

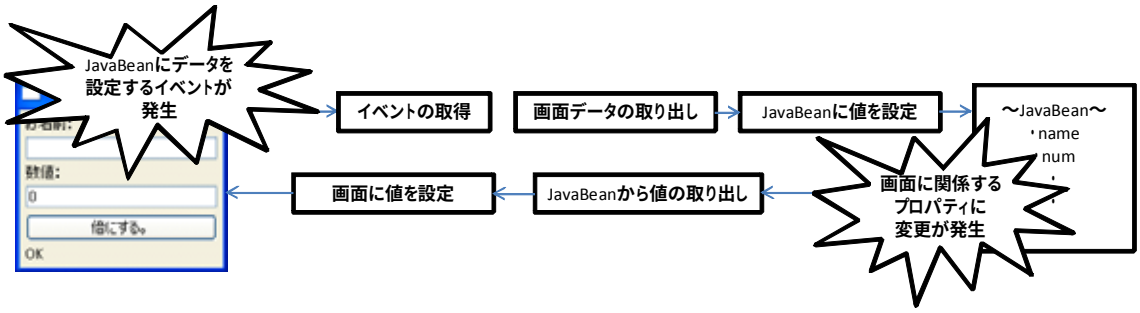


図 2：JFace Data Binding を使用する場合のデータの受け渡しの流れ

JFace Data Binding を使用しない場合はデータが変更になるイベントごとにデータの同期を行うよう作成する必要があることを示しています。それに対して JFace Data Binding を使用する場合は画面部品と JavaBean のプロパティを関連付けるようになっています。

これが何を意味するかと言いますと、JFace Data Binding を使用しない場合は常に画面と JavaBean の状態に齟齬が無いようにイベントの処理を記述する際に気をつける必要があります。また、複数のイベントが同じ画面部品やプロパティに値を更新する時には、同じデータの同期を複数の箇所で記述することになります。同じ意図で記述されているコードが複数存在することは保守性を低下させます。それに対して JFace Data Binding を使用する場合では、データの同期という目的で記述されるコードが一

箇所に纏められるため、JFace Data Binding を使用しない場合に懸念される問題は発生しません。それによって開発者は画面部品と JavaBean 間のデータ同期がもれなく用意されているか気をつける必要がなくなり、ビジネスロジックに集中することができます。

WEB アプリケーションの開発経験がある方には、JFace Data Binding は、Struts の ActionForm とよく似た機能の JFace および SWT 版と言えます。

そして、Struts の ActionForm と同様に入力チェックを行う機能も備えています。

JFace Data Binding の概要は以上です。

なお、この文書の作成のために確認作業を行った環境は以下の通りです。

OS	WindowsXP Professional SP3
Java	JDK 1.6.0_13
Eclipse	3.5.0 M6 (Galileo)
JFaceJFace Data Binding （org.eclipse.core.databinding）	1.3.0
SWT および JFaceJFace Data Binding （org.eclipse.jface.databinding）	1.2.0

なお、以降で使用するサンプルを以下のリンクからダウンロードできます。

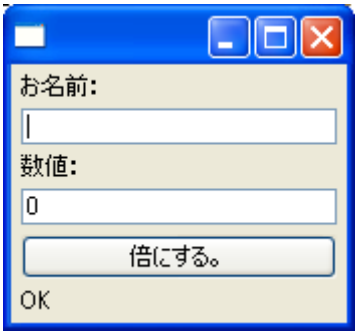
＜サンプルダウンロード＞

http://www.bbreak.co.jp/technique/doc/jface_data_binding/JFaceDataBinding.zip

2. メリット・デメリット

JFace Data Binding を使用する場合と、しない場合ではどのようなメリット・デメリットがあるのか、サンプルを使用して解説したいと思います。このサンプルでは、ボタン「倍にする。」を押すと、数値に入力した値を 2 倍します。また、算出した数値を名前の後ろに文字列結合するという仕様となっています。

図 3：サンプルの起動画面（画面は両方同じ）



【JFace Data Binding を使用しない場合】

リスト 1： SampleWindow.java

```
public class SampleWindow {
  private static DataBindSampleBean sampleBean = new DataBindSampleBean();
  private static void initialize(Shell shell) {
    // 部品宣言と配置は省略
    // ここからイベント処理の設定
    textName.addModifyListener(new ModifyListener() {
      public void modifyText(ModifyEvent e) {
        // bean に入力値を設定する
        sampleBean.setName(textName.getText());
      }
    });

    textNum.addModifyListener(new ModifyListener() {
      public void modifyText(ModifyEvent e) {
        try {
          String numStr = textNum.getText();
          if (numStr == null || numStr.length() == 0) {
```

テキストボックス「お名前」に変更があった場合のイベント処理

テキストボックス「数値」に変更があった場合のイベント処理

```
                return;
            }
            // bean に入力値を設定する
            sampleBean.setNum(Integer.parseInt(numStr));
          } catch (NumberFormatException ex) {
            textNum.setText("");
            labelError.setText("テキストボックス数値に半角数値以外を入力しないでください。");
            labelError.setSize(400, 50);
          }
        }
      });

      buttonDouble.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
          sampleBean.calcDoubleNum();
          // フォームに算出値を設定する
          textNum.setText(Integer.toString(sampleBean.getNum()));
          textName.setText(sampleBean.getName());
        }
```

ボタン「２倍にする。」が押された場合のイベント処理

リスト２： SampleBean.java

```
public class SampleBean implements Serializable{
  private static final long serialVersionUID = 5391067181749183131L;
  private String name = null;
  private int num = 0;
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public int getNum() { return num; }
  public void setNum(int num) { this.num = num; }
  public void calcDoubleNum() {
    setNum(this.num * 2);
    setName(this.name = this.name + "_" + this.num);
  }
}
```

【JFace Data Binding を使用する場合】

リスト 3：DataBindSampleWinodw.jav a

```
public class DataBindSampleWindow {
   private static DataBindSampleBean sampleBean = new DataBindSampleBean();
   private static void initialize(Shell shell) {
     // 部品宣言と配置は省略
     // ここからイベント処理の設定
     buttonDouble.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
             sampleBean.calcDoubleNum();
      }
     });
     // ここまでイベント処理の設定
     DataBindingContext dbc = new DataBindingContext();

     // JavaBean 側の監視設定
     IObservableValue nameObservable = BeansObservables.observeValue(sampleBean, "name");
     IObservableValue numObservable = BeansObservables.observeValue( sampleBean, "num");

     // 画面部品側の監視設定
     IObservableValue textNameObservable = SWTObservables.observeText(textName, SWT.Modify);
     IObservableValue textNumObservable = SWTObservables.observeText(textNum, SWT.Modify);

     // データバインディングコンテキストに Bean と画面の関連性を設定
     dbc.bindValue(textNameObservable, nameObservable, null, null);
     dbc.bindValue(textNumObservable, numObservable, null, null);

     // ここからバリデーション結果表示
     dbc.bindValue(SWTObservables.observeText(labelError),
          new AggregateValidationStatus(dbc.getBindings(),
               AggregateValidationStatus.MAX_SEVERITY), null, null);
     // ここまでバリデーション結果表示
}
```

ボタン「２倍にする。」が押された場合のイベント処理

Bean のプロパティ単位に監視設定を行う

画面部品の監視設定を行う

画面部品と Bean のプロパティをバインドする

リスト 4：DataBindSampleBean.jav a

```
public class DataBindSampleBean implements Serializable {
    private static final long serialVersionUID = 5391067181749183131L;
    private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(String propertyName, PropertyChangeListener listener) {
        changeSupport.addPropertyChangeListener(propertyName, listener);
    }
    public void removePropertyChangeListener(String propertyName, PropertyChangeListener listener) {
        changeSupport.removePropertyChangeListener(propertyName, listener);
    }

    private String name = null;
    private int num = 0;
    public String getName() { return name; }
    public void setName(String name) {
        String oldName = this.name;
        this.name = name;
        changeSupport.firePropertyChange("name", oldName, name);
    }
    public int getNum() { return num; }
    public void setNum(int num) {
        int oldNum = this.num;
        this.num = num;
        changeSupport.firePropertyChange("num", oldNum, num);
    }
    public void calcDoubleNum() {
        setNum(this.num * 2);
        setName(this.name + "_" + this.num);
    }
}
```

変更を通知するための PropertyChangeSupport を追加

ここでプロパティの変更を通知

ここでプロパティの変更を通知

～メリットについて～

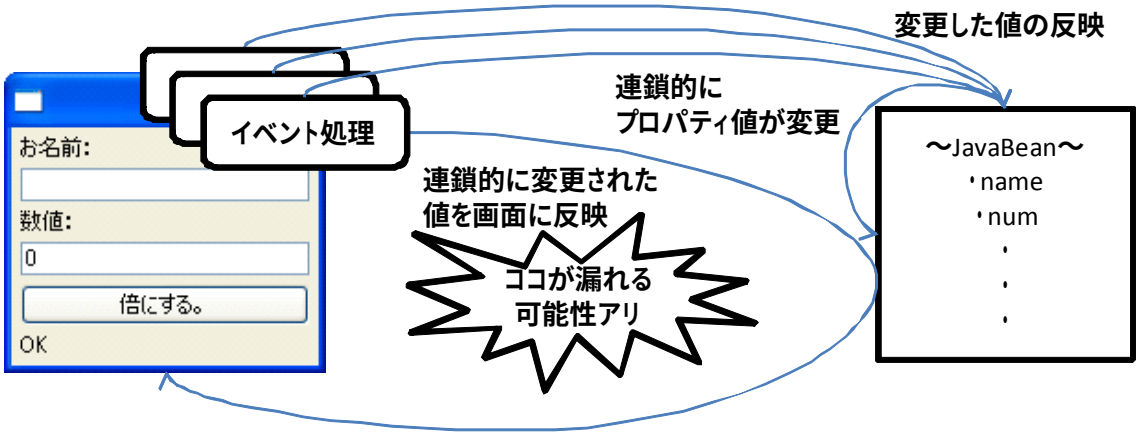
JFace Data Binding を使用しない場合、画面部品とフォーム用の JavaBean の間のデータは、イベント処理内で同期を取ります。そのため、あるイベントで変更された値が、JavaBean の他のプロパティに影響を及ぼす場合、影響範囲を把握し、影響のあるプロパティの値をフォームに設定し直す必要があります。具体的には、先のリスト 1 のボタン「倍にする。」を押下した際のイベント処理になります。

```
sampleBean.calcDoubleNum();
// フォームに算出値を設定する
textNum.setText(Integer.toString(sampleBean.getNum()));
textName.setText(sampleBean.getName());
```

ここでは、JavaBean のメソッドを呼び出し、数値を倍にし、さらに名前に倍にした数値を連結しています。このメソッドは JavaBean のプロパティ値だけを変更するので、変更した結果を画面部品に反映させるために JavaBean のプロパティ値を取得し、画面部品に設定しています。このメソッドはプロパティ”name”と”num”の両方の値に影響するため、両方に対応する画面部品に JavaBean のプロパティ値を設定し直しています。

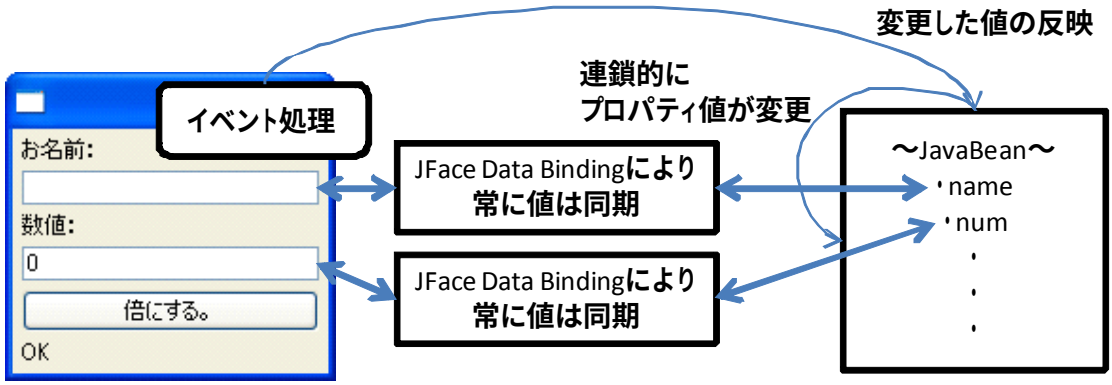
そのため、データバインドを使用しない場合は、変更された JavaBean のプロパティ”name”の値を画面部品に設定する必要があります。しかし、この仕様を担当者が把握していないと、画面と JavaBean とのデータに差異が発生します。

図 4：JFace Data Binding を使用しない場合のデータ反映漏れの可能性



JFace Data Binding を正しく設定すれば、画面部品と JavaBean のプロパティ値が常に同期されます。そのため、JavaBean 内部のプロパティが連鎖的に変わったとしても、それを意識しなくとも変更があり次第、双方の同期をとることができます。これが JFace Data Binding が作られた目的であり、導入により得られるメリットです。

図 5：JFace Data Binding を使用し連鎖的にプロパティが変更されている時の流れ



〜デメリットについて〜

JFace Data Binding ではバインディングするプロパティの名前を文字列で指定するため、リファクタリングを行うと手動で修正する必要が出てきます。

具体例をあげると、JavaBean のプロパティ名を Eclipse のリファクタリング機能でリスト 2 の JavaBean のプロパティ”name”を”fullName”に変更したとします。ここで、リスト 1 の JavaBean に対する監視設定を見てみましょう。

```
// JavaBean 側の監視設定
IObservableValue nameObservable = BeansObservables.observeValue(sampleBean, "name");
IObservableValue numObservable = BeansObservables.observeValue( sampleBean, "num");
```

ここでは、JavaBean のプロパティ値をバインドするための準備をしています。”BeansObservables”クラスの”observeValue”の第 1 引数は JavaBean、第 2 引数はプロパティ名を文字列で指定します。Eclipse のリファクタリングでは文字列で設定されている第 2 引数のプロパティ名を判断して修正する機能はありません。そのため、リファクタリングを後に第 2 引数のプロパティ名を変更し忘れると、起動時に異常終了します。

よって、リファクタリング時には監視設定は手動で修正する必要があり、リファクタリングの手間は増えることになります。

多少のデメリットがあるものの、JFace Data Binding のメリットは、それを補って余りあるものである事がお分かり頂けたのではないか、と思います。

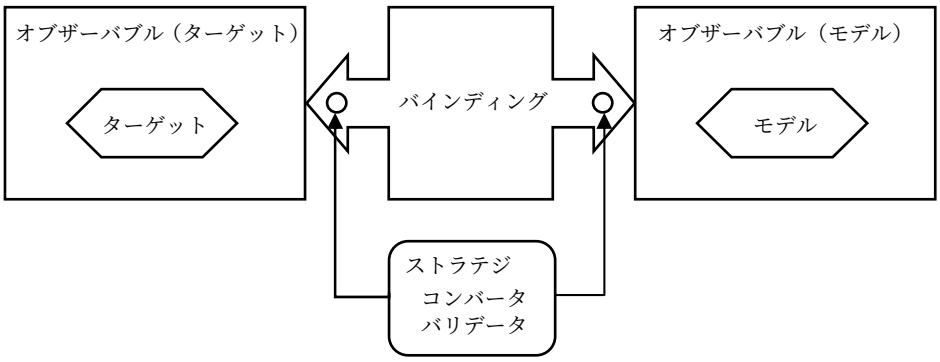
3. 仕組みと構成要素

この章では JFace Data Binding がどのように同期を行うのか理解するとともに、どのような要素で構成されているかを説明します。

3.1. JFace Data Binding の仕組み

JFace Data Binding はバインディングを仲介役として各オブジェクトのデータを同期させています。JFace Data Binding の各要素の関係を図にすると以下のようになります。

図 6：JFace Data Binding の全体イメージ



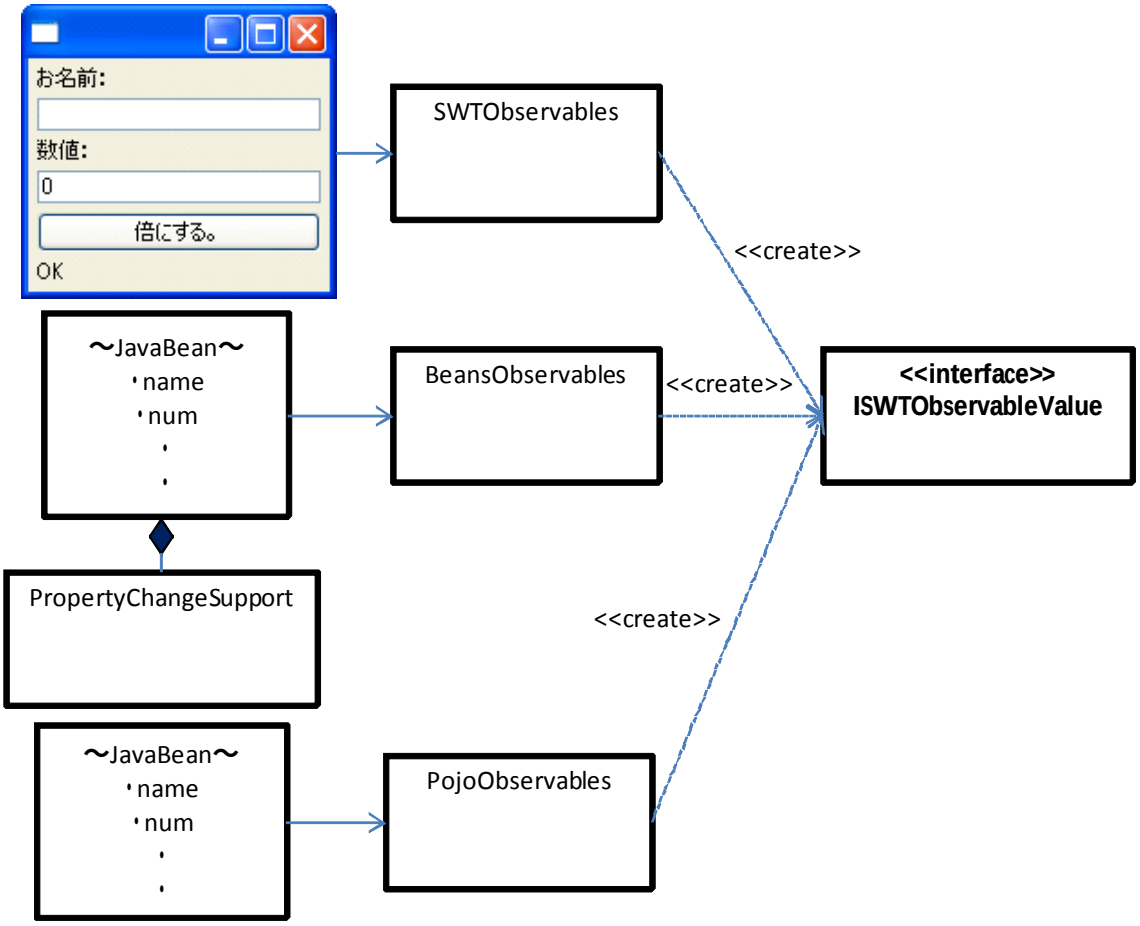
バインディングを仲介役として、左右両端のオブザーバブルに内包されているターゲットやモデルのデータを同期しています。ターゲットやモデルは画面部品や JavaBean を表します。また、ストラテジがバインディングの前後に入ります。このストラテジには必要に応じたバリデータやコンバータを設定し、入力チェックとデータ変換を行うことができます。

データを同期させる仕組みはとてもシンプルな形になっていることが分かると思います。

3.2. オブザーバブル

オブザーバブルは監視可能性を表すインターフェースです。オブザーバブルは画面部品、JavaBean の両方で作成します。

図 7：オブザーバブル関連のクラス図



画面部品ではファクトリクラス”SWTObservables”からオブザーバブルを作成します。この際に、引数として画面部品とともに、イベントタイプを指定しますが、”SWT”クラスの定数から感知したいイベントのタイプを適宜設定します。

JavaBean にはファクトリクラス” BeansObservables”からオブザーバブルを作成します。また、監視される JavaBean には必ず”PropertyChangeSupport”をメンバとして所有し、セッターメソッドが呼ばれる箇所など、プロパティ値の変更される箇所で”PropertyChangeSupport”の” firePropertyChange”メソッドを使い、プロパティ変更を通知できるようにします。この設定を忘れると、プロパティが変更されても、画面に変更が反映されません。

なお、同様のファクトリクラスに” PojoObservables”が存在します。これは”PropertyChangeSupport”を持たない JavaBean 用のサポートクラスです。

”PojoObservables”で作成したオブザーバブルは、画面部品から JavaBean への値の同期を行うことは可能ですが、JavaBean の変更が画面に通知されません。よって、画面の入力データを単純に受け取るだけの時にのみ使用できます。

表 1：オブサーバに関するクラス一覧

クラス	パッケージ	概要
SWTObservables	org.eclipse.jface.databinding.swt	画面部品用のオブザーバを作成するファクトリクラス
BeansObservables	org.eclipse.core.databinding.beans	JavaBean 用のオブザーバを作成するファクトリクラス
PojoObservables	org.eclipse.core.databinding.beans	POJO 用のオブザーバを作成するファクトリクラス
SWT	org.eclipse.swt	SWT の定数クラス、event type 指定に使用する
PropertyChangeSupport	java.beans	JavaBean に変更を通知機能を追加するために使用する

3.3. バインディング

バインディングは 2 つのオブザーバブルを関連付ける設定を指しています。バインディングでは” DataBindingContext”に画面部品のオブザーバブルと JavaBean のオブザーバブルが対であることを設定します。それに加えて、単に関連付けをするだけでなく、バインディング設定の際にはストラテジとして” UpdateValueStrategy”を設定し値の変換とバリデーションを同期時に行うよう設定できます。ストラテジは画面部品から JavaBean への同期時と、JavaBean から画面部品への同期時のそれぞれに設定できます。

” UpdateValueStrategy”で設定できるバリデーションと変換は以下の表の順序で処理が実行されます。

表 2：”UpdateValueStrategy”で設定できる内容と順序

順序	設定可能なインターフェース	ストラテジに設定するメソッド	内容
1	IValidator	setAfterGetValidator	値取得直後のバリデーション
2	IConverter	setConverter	変換処理
3	IValidator	setAfterConvertValidator	変換処理後のバリデーション
4	IValidator	setBeforeSetValidator	値設定前のバリデーション

リスト 3 と 4 では、ストラテジは設定しませんでした。該当するソースは以下の部

分になります。

```
// データバインディングコンテキストに Bean と画面の関連性を設定
dbc.bindValue(textNameObservable, nameObservable, null, null);
dbc.bindValue(textNumObservable, numObservable, null, null);
```

この場合、コンバータとバリデーションはデフォルトが使用され、データの同期を行っています。デフォルトの変換とバリデーションは変換前と変換後のデータ型で適切なデータ型のコンバータとバリデーション（バリデーションは値取得直後のみ、変換処理以降のバリデーションは行われない）が選択されています。

また、独自のコンバータをストラテジに設定した場合は、基本的なデータ型間の変換であってもデフォルトのバリデーションは選択されません。

詳しいストラテジ未設定時の動作は”UpdateValueStrategy”の”fillDefault”からたどることができます。

3.4. コンバータ

コンバータはその名前の通り、値の変換を行います。コンバータの具象クラスは”IConverter”を実装します。

自作のコンバータを作成する場合には、スレッドセーフではないクラスを使用する場合、synchronized でスレッドの同期をとることに注意が必要です。例えば、日時のデータを変換する場合には”SimpleDateFormat”を使用する場合がそれにあたります。これについては、後で記述するサンプルのリスト 7 の”convert”メソッドを参照して下さい。また、プリミティブ型への変換の場合は、変換する値が null の場合は何を返すか明確にする必要があります。

表 3：コンバータに関するクラス・インターフェース一覧

クラス	パッケージ	概要
IConverter	org.eclipse.core.databinding.conversion	コンバータのインターフェース
Convertor	org.eclipse.core.databinding.conversion	IConvertor を実装した抽象クラス
NumberToStringConverter	org.eclipse.core.databinding.conversion	Converter を継承しているコンバータクラス

今回のサンプルは、リスト 3 と 4 のサンプルに日付を入力するテキストボックスを追加し入力された文字列を日付データとして変換します。そして、ボタン「倍にする。」をクリックすると、入力された数値を 2 倍した数値だけ日を進めるように変更させます。

図 8：サンプルの起動画面

リスト 5：DataBindSampleConvertorWindow.java

```
public class DataBindSampleConvertorWindow {
  private static DataBindConvertSampleBean sampleBean = new DataBindConvertSampleBean();
  private static void initialize(Shell shell) {
    // 部品宣言と配置、イベント処理は省略
    DataBindingContext dbc = new DataBindingContext();
    // JavaBean側の監視設定
    IObservableValue nameObservable = BeansObservables.observeValue(sampleBean, "name");
    IObservableValue numObservable = BeansObservables.observeValue(sampleBean, "num");
    IObservableValue dateObservable = BeansObservables.observeValue(sampleBean, "date");
    // 日付用更新ストラテジの作成
    UpdateValueStrategy stringToDateStrategy = new UpdateValueStrategy();
    SimpleDateFormat sdf = new SimpleDateFormat();
    sdf.applyPattern("yyyy/MM/dd");
    StringToDateConvertor stringToDateConvertor = new StringToDateConvertor(sdf);
    stringToDateStrategy.setConverter(stringToDateConvertor);

    UpdateValueStrategy dateToStringStrategy = new UpdateValueStrategy();
    DateToStringConvertor dateToStringConvertor = new DateToStringConvertor(sdf);
    dateToStringStrategy.setConverter(dateToStringConvertor);

    // データバインディングコンテキストにBeanと画面の関連性を設定
    dbc.bindValue(SWTObservables.observeText(textName, SWT.Modify), nameObservable, null, null);
    dbc.bindValue(SWTObservables.observeText(textNum, SWT.Modify), numObservable, null, null);
```

```
        dbc.bindValue(
                SWTObservables.observeText(textDate, SWT.Modify),
                dateObservable,
                stringToDateStrategy,
                dateToStringStrategy);
    }
}
```

リスト 6：DataBindConvertSampleBean.java

```
public class DataBindConvertSampleBean implements Serializable {
    private static final long serialVersionUID = 5391067181749183131L;
    private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);
    public void addPropertyChangeListener(String propertyName, PropertyChangeListener listener) {
        changeSupport.addPropertyChangeListener(propertyName, listener);
    }
    public void removePropertyChangeListener(String propertyName, PropertyChangeListener listener) {
        changeSupport.removePropertyChangeListener(propertyName, listener);
    }
    private String name = null;
    private int num = 0;
    private Date date = new Date();
    public String getName() { return name; }
    public void setName(String name) {
        String oldName = this.name;
        this.name = name;
        changeSupport.firePropertyChange("name", oldName, name);
    }
    public int getNum() { return num; }
    public void setNum(int num) {
        int oldNum = this.num;
        this.num = num;
        changeSupport.firePropertyChange("num", oldNum, num);
    }
```

```
  public void calcDoubleNum() {
    setNum(this.num * 2);
    setName(this.name + "_" + this.num);
    Calendar cal = Calendar.getInstance();
    cal.setTime(this.date);
    cal.add(Calendar.DATE, this.num);
    setDate(cal.getTime());
  }
  public Date getDate() { return date; }
  public void setDate(Date date) {
    Date oldDate = this.date;
    this.date = date;
    changeSupport.firePropertyChange("date", oldDate, date);
```

リスト 7：StringToDateConvertor.java

```
/** 文字列を日付データに変換するコンバータ */
public class StringToDateConvertor implements IConverter {
  private SimpleDateFormat sdf = null;
  public StringToDateConvertor(SimpleDateFormat sdf) { this.sdf = sdf;
  }
  @Override
  public Object convert(Object fromObject) {
    if (fromObject == null || !(fromObject instanceof String)) {
      return null;
    }
    try {
      Date fromDate = null;
      synchronized (fromObject) {
        fromDate = sdf.parse((String) fromObject);
      }
      return fromDate;
    } catch (ParseException ex) {
      return null;
    }
```

```
    }
    @Override
    public Object getFromType() {
        // 受け取るデータの型を返却
        return String.class;
    }
    @Override
    public Object getToType() {
        // 返すデータの型を返却
        return Date.class;
    }
}
```

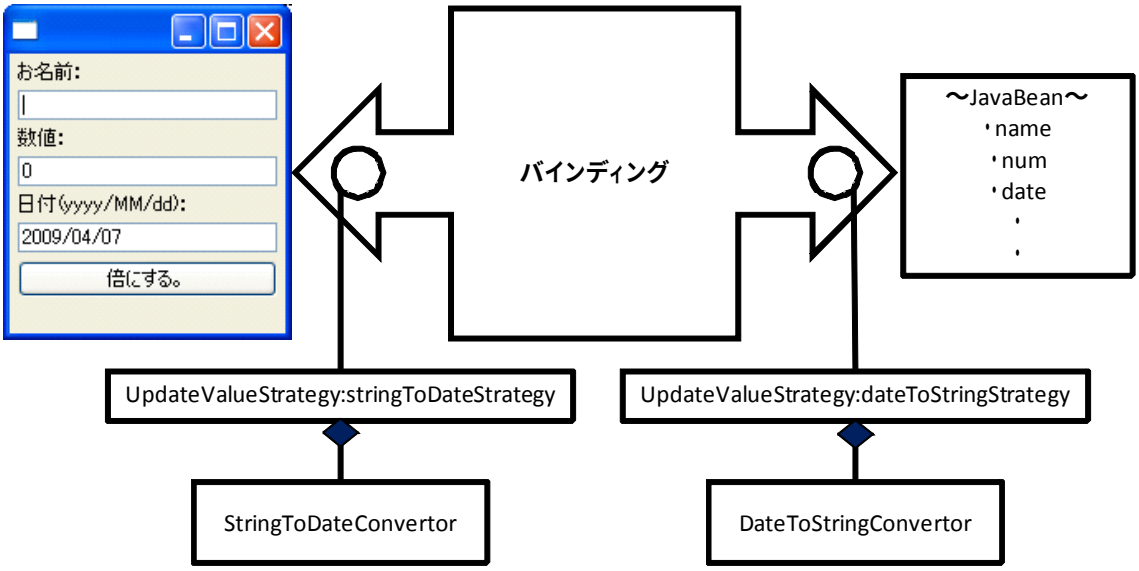
リスト 8：DateToStringConvertor.java

```
/** 日付データを文字列に変換するコンバータ */
public class DateToStringConvertor implements IConverter {
    private SimpleDateFormat sdf = null;
    public DateToStringConvertor(SimpleDateFormat sdf) { this.sdf = sdf; }
    @Override
    public Object convert(Object fromObject) {
        if (fromObject == null || !(fromObject instanceof Date)) {
            return null;
        }
        String fromString = null;
        synchronized (fromObject) {
            fromString = sdf.format((Date) fromObject);
        }
        return fromString;
    }
    @Override
    public Object getFromType() {
        // 受け取るデータの型を返却
        return Date.class;
    }
```

```
  @Override
  public Object getToType() {
    // 返すデータの型を返却
    return String.class;
  }
}
```

このサンプルの JavaBean には日付として”date”というプロパティが存在します。このパラメータと画面のテキストボックス「日付」がバインドされています。データの同期時の動きは 2 つ存在する。テキストボックス”textDate”からプロパティ”date”にデータを渡す場合は”StringToDateConvertor”の”convert”メソッドが実行され、入力値を Date 型に変換しています。また、逆の流れになると”DateToStringConvertor”の”convert”メソッドが実行され、プロパティ”date”の値が文字列に変換されます。

図 9：サンプルのストラテジとコンバータの関連図



ただし、ここで使用したコンバータは入力値が異常な場合、null を返却しています。そのため、誤った入力がある場合には、このアプリケーションは異常終了してしまいます。よって、コンバータを使用する時には入力値の確認を、後述するバリデータを使用し、入力エラーの制御を行うことになります。

3.5. バリデータ

バリデータは入力された値が期待された値かを確認します。バリデータの具象クラス

は”IValidator”を実装します。バリデーションを作成するときは、基本的なバリデーションを自作ライブラリとして作成することが推奨されています。

バリデーションの結果は”ValidationStatus”クラスを使用して”Istatus”クラスで返却します。返却されたステータスは JFace Data Binding コンテキストに記録され、バリデーション結果の表示領域に出力することができます。返却するステータスの種類と挙動については以下の表に記載します。

表 4：返却するステータスの一覧

ステータス	挙動
ok	バリデーションの成功を表す。結果表示領域を作成している場合は”OK”と表示される。
info	メッセージ付きのバリデーションの成功を表す。結果表示領域を作成している場合は設定したメッセージが表示される。
warning	警告を表す。後続の処理を行うが、注意を喚起する場合に使用。結果表示領域を作成している場合は設定したメッセージが表示される。
error	異常を表す。後続の処理を行わない。結果表示領域を作成している場合は設定したメッセージが表示される。エラーが返却されると、バインドされている値が同期されていない。（画面にエラーとなる値を入力した場合は、画面上はエラーの入力内容がそのまま残るが、JavaBean の方は入力エラーになる前の値が設定されている）
cancel	キャンセルを表す。後続の処理は行わない。Error と同様の挙動となる。

表 5：バリデーションに関するクラス・インターフェース一覧

クラス	パッケージ	概要
IValidator	org.eclipse.core.databinding.validation	バリデータのインターフェース
IStatus	org.eclipse.core.runtime	バリデーションを行った後の戻り値のインターフェース
ValidationStatus	org.eclipse.core.databinding.validation	バリデーションの結果の返却値を作成するクラス
MultiValidator	org.eclipse.core.databinding.validation	複数フィールドにまたがるバリデーションを行うためのヘルパクラス

では、先ほどのリスト 5～8 のサンプルにバリデーションを追加し、異常終了しないように入力チェックを行うように改良してみましょう。

以下のリスト 9 で文字列を変換する前に入力した文字列を確認し、入力した文字列が日付に変換可能かを判定しています。また、日付に変換できる文字列であっても、あまりにも大きな値が指定された場合は警告メッセージを表示するようになっていま

す。なお、警告の場合は、JavaBean の値は更新されます。

リスト 9：StringToDateAfterGetValidator.java

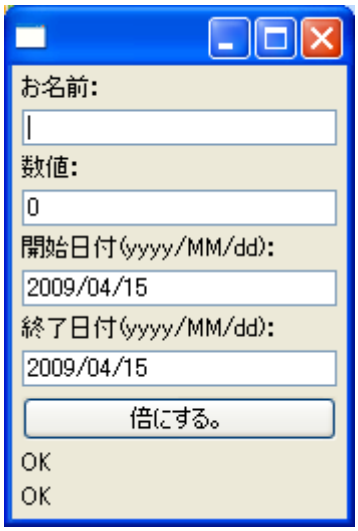
```
public class StringToDateAfterGetValidator implements IValidator {
  @Override
  public IStatus validate(Object value) {
    if (value == null || !(value instanceof String)) {
      return ValidationStatus.error("入力値が異常です");
    }
    String valueString = (String) value;
    if (valueString.length() == 0) {
      return ValidationStatus.error("入力値が空です。");
    }
    String[] splitedValue = valueString.split("/");
    if (splitedValue.length != 3) {
      return ValidationStatus.error("日付は yyyy/MM/dd 形式で入力して下さい");
    }
    Iterator<String> iter = Arrays.asList(splitedValue).iterator();
    boolean tooLong = false;
    while (iter.hasNext()) {
      String checkTarget = iter.next();
      try {
        if (checkTarget.length() > 4) {
          tooLong = true;
        }
        Integer.parseInt(checkTarget);
      } catch (NumberFormatException ex) {
        return ValidationStatus.error("日付は yyyy/MM/dd 形式で入力して下さい", ex);
      }
    }
    if (tooLong) {
      return ValidationStatus.warning("少し大きすぎる値が設定されていませんか？");
    }
    return ValidationStatus.ok();
  }
```

リスト 10：DataBindSampleConvertorAndValidationWindow.java 追加部分
（DataBindSampleConvertorWindow を改造）

```
～ストラテジの設定を作成後の箇所に以下を追加～
StringToDateAfterGetValidator stringToDateAfterGetValidator = new StringToDateAfterGetValidator();
stringToDateStrategy.setAfterGetValidator(stringToDateAfterGetValidator);
```

また、複数のフィールドにまたがるバリデーションを行う場合は、”MultiValidator”を使うことで実現できます。”MultiValidator”は抽象クラスなので、継承して複数フィールドの相関チェックを実現するクラスを作成します。以下に”MultiValidator”のサンプルとして、開始日付と終了日付が前後して入力されないようにチェックするクラスを作成します。

図 10：サンプルの起動画面



リスト 10：PeriodMultiValidator.java

```
public class PeriodMultiValidator extends MultiValidator {
  private IObservableValue fromValue = null;
  private IObservableValue toValue = null;
  public PeriodMultiValidator(IObservableValue fromValue, IObservableValue toValue) {
    this.fromValue = fromValue;
    this.toValue = toValue;
  }
```

```
  @Override
  protected IStatus validate() {
    if (fromValue == null || toValue == null) {
      return ValidationStatus.error("両方の日付が必要です。");
    }
    Date fromDate = (Date) fromValue.getValue();
    Date toDate = (Date) toValue.getValue();
    if (toDate.before(fromDate)) {
      return ValidationStatus.error("日付の前後が逆です。");
    }
    return ValidationStatus.ok();
  }
}
```

リスト 11：DataBindMultiValidateSampleBean.java（DataBindConvertSampleBean を改造）

```
/**
 * 属性 date を削除し、startDate、endDate を追加する。
 * セッター、ゲッターは date 同様の物を作成する
 */
/** メソッド calcDoubleNum を置き換える 開始 */
  public void calcDoubleNum() {
    setNum(this.num * 2);
    setName(this.name + "_" + this.num);

    Calendar cal = Calendar.getInstance();
    cal.setTime(this.startDate);
    cal.add(Calendar.DATE, -this.num);
    setStartDate(cal.getTime());

    cal.setTime(this.endDate);
    cal.add(Calendar.DATE, this.num);
    setEndDate(cal.getTime());
  }
/** メソッド calcDoubleNum を置き換える 終了 */
```

リスト 12：DataBindSampleMultiValidationWindow.java
（DataBindSampleConvertorAndValidationWindow を改造）

```
/** JavaBean 側の監視設定の日付の監視設定を置き換える 開始 */
IObservableValue startDateObservable = BeansObservables.observeValue(sampleBean, "startDate");
IObservableValue endDateObservable = BeansObservables.observeValue( sampleBean, "endDate");
/** JavaBean 側の監視設定の日付の監視設定を置き換える 終了 */
/** データバインディングコンテキストに Bean と画面の関連性を設定の「日付」の箇所を置き換える 開始*/
// 中間オブザーバの作成
IObservableValue startDateMiddleObservable = new WritableValue(null, Date.class);
IObservableValue endDateMiddleObservable = new WritableValue(null, Date.class);

// 画面部品～中間オブザーバ間でデータを同期させる
dbc.bindValue(SWTObservables.observeText(textStartDate, SWT.Modify),
    startDateMiddleObservable, stringToDateStrategy,
    dateToStringStrategy);
dbc.bindValue(SWTObservables.observeText(textEndDate, SWT.Modify),
    endDateMiddleObservable, stringToDateStrategy,
    dateToStringStrategy);

// 相関チェックの準備
PeriodMultiValidator periodMultiValidator = new PeriodMultiValidator(
    startDateMiddleObservable, endDateMiddleObservable);

// 中間オブザーバから相関チェックを行い、チェック後オブザーバを作成する
IObservableValue startDateValidatedObservable = periodMultiValidator
    .observeValidatedValue(startDateMiddleObservable);
IObservableValue endDateValidatedObservable = periodMultiValidator
    .observeValidatedValue(endDateMiddleObservable);

// チェック後オブザーバ～JavaBean間でデータを同期させる
dbc.bindValue(startDateValidatedObservable, startDateObservable, null, null);
dbc.bindValue(endDateValidatedObservable, endDateObservable, null, null);
/** データバインディングコンテキストに Bean と画面の関連性を設定の「日付」の箇所を置き換える 終了*/
```

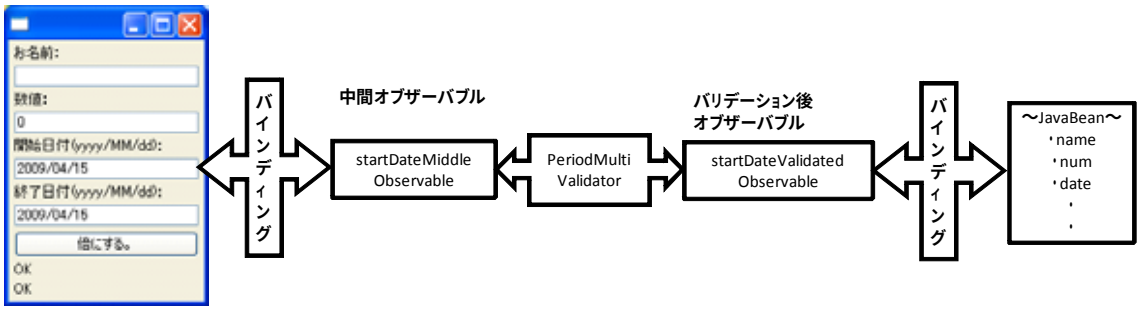
相関チェックの実行

```
/** 相関チェック出力用のバインドをエラーラベルのバインドの後に追加 開始 */
dbc.bindValue(SWTObservables.observeText(labelMultiError),
      periodMultiValidator.getValidationStatus(), null, null);
/** 相関チェック出力用のバインドをエラーラベルのバインドの後に追加 終了 */

/** 相関チェックのステータスを画面に出力するバインド設定 開始 */
dbc.bindValue(SWTObservables.observeText(labelMultiError),
      periodMultiValidator.getValidationStatus(), null, null);
/** 相関チェックのステータスを画面に出力するバインド設定 終了 */
```

オブザーバブルは今まで画面部品と JavaBean の関連付けを直接バインドしていたが、今回、相関チェックを行うため、間に中間オブザーバを作成し、相関チェックを行った後に、JavaBean との同期を行うよう修正した。

図 11：相関チェックの流れ



相関チェックを行う”PeriodMultiValidator”クラスは”IValidator”インターフェースのバリデータではないので、使い方が異なります。今回はスーパークラスの”MultiValidator”から”observeValidatedValue”メソッドを呼び出し、バリデーションを実行しています。（リスト 13 の吹き出しの箇所）ただし、相関チェック出力用のバインドで使用している”getValidationStatus”メソッド内でもバリデーションが再実行されますので、2 回同じチェックを行っています。これは、チェック後オブザーバを作成する個所（リスト 13 の吹き出しの箇所）を省略すると、相関チェックが通らなかった場合にも JavaBean に入力値が反映されてしまうため、あえて 2 回同じチェックをすることが分かってこのような作りをしています。

上記の通り、”MultiValidator”を使用すれば、画面部品間の相関チェックは可能なことが分かりました。しかしながら、今回のサンプルではバリデーションの結果を今までの通常のバリデーション結果と別に出力せざるを得ませんでした。API ドキュメン

トを見る限り、” addValidationStatusProvider”に相関チェッククラスを引数に指定すればJFace Data Bindingコンテキストのバリデーション結果に入れる事が出来るように”MultiValidator”クラスの JavaDoc に記述されていましたが、正しくバリデーションは行われませんでした。今後、何らかの形で修正されると思われますが、現状はサンプルのように作るのが妥当と思われます。

4. JFace Data Binding の応用

4.1. Table と TableViewer でのデータバインディング

Table は Excel のスプレッドシートの様な表の画面部品です。JFace Data Binding の応用として、編集可能な Table を配置し、その部品と JavaBean のデータをバインドしてみたいと思います。使用する JavaBean はリスト 4 の”DataBindSampleBean”になります。ここでは、複数の JavaBean を使用し、一行が 1 つの JavaBean に対応します。すべてのセルは編集可能で、編集が行われると、対応する JavaBean の値が変更されます。そして、ボタン「倍にする。」を押下すると、選択状態にある行の数値を倍にし、倍にした数値が名前の後ろに付くようになっています。

図 12：サンプルの起動画面

リスト 13：DataBindSampleTableViewerWindow.java

```
public class DataBindSampleTableViewerWindow {
   private static List<DataBindSampleBean> sampleBeanList = createBeans();
   private static void initialize(Shell shell) {
      // 画面部品の配置は省略
      // 行定義の設定
      ObservableListContentProvider listContentProvider = new ObservableListContentProvider();
      tableViewer.setContentProvider(listContentProvider);

      // テーブルに表示するデータの設定
      WritableList writableList = new WritableList(sampleBeanList, DataBindSampleBean.class);
      tableViewer.setInput(writableList);

      DataBindingContext dbc = new DataBindingContext();
```

```
    // セルを編集するタイプの設定を作成
    IValueProperty cellEditorControlText = CellEditorProperties.control()
        .value(WidgetProperties.text());

    // カラム毎に対応する Bean のプロパティ名で IValueProperty を作成する
    IValueProperty nameProperty = BeanProperties.value("name");
    IValueProperty numProperty = BeanProperties.value("num");

    // 列の編集サポートクラスとして、監視可能な編集サポートクラスを作成する
    tableViewerNameColumn.setEditingSupport(ObservableValueEditingSupport
        .create(tableViewer, dbc, new TextCellEditor(table),
            cellEditorControlText, nameProperty));
    tableViewerNumColumn.setEditingSupport(ObservableValueEditingSupport
        .create(tableViewer, dbc, new TextCellEditor(table),
            cellEditorControlText, numProperty));

    // テーブルと JavaBean のバインディング
    ViewerSupport.bind(tableViewer, (IObservableList) writableList,
        new IValueProperty[] { nameProperty, numProperty });
  }
}
```

Table にデータをバインドする場合は TableViewer の”ContentProvider”にとして”ObservableListContetnProvider”を設定します。これにより JavaBean に変更があった場合に同期が行えるようになります。また、列を変更可能にするために TableViewer の”EditingSupport”として”ObservableValueEditingSupport”を設定します。これによりセルの編集があった場合に同期が行えるようになります。

最後に、バインドですが、今までと異なり”ViewerSupport”クラスを使用します。このクラスは名前の通り、テーブルのバインドを簡単に行うためのクラスです。ここで、TableViewer と対応する Bean のリスト、列毎に対応する JavaBean のプロパティを設定し、バインドします。

5. 今後の展望

JFace Data Binding は Eclipse3.3 から追加された新機能であるため、まだ API が固まりきっていない感があります。今回開発途中である 3.5 を使用したのは、3.3 では存在した TableViewer に値をバインドするためのサポートクラスが現行の 3.4 でなくなってしまったためです。また、TableViewer だけでなくサポートクラス全般が一時的に削除されています。複雑なウィジェットに JavaBean のデータをバインドするためにはサポートクラスは重要なため、3.5 以上のバージョンで作成することが望ましいと思われます。今後も突然 API が変わることもあり得るため、今後も動向には十分注意する必要があります。

しかしながら、JFace Data Binding は画面と JavaBean のデータ同期をロジックから分離し、イベント毎にデータの状態を考慮しなければならない状況からエンジニアを解放してくれます。よって、JFace Data Binding を活用することで、多少の API の変更があったとしても、余りあるメリットを享受できるのではないかと思われます。

6. 参考文献

・JFace Data Binding – Eclipsepedia
JFace Data Binding 本家（英語）
http://wiki.eclipse.org/JFace_Data_Binding

・Eclipse Databinding with Eclipse RCP applications – Tutorial
コミッタによるチュートリアル（英語）
http://www.vogella.de/articles/EclipseDataBinding/article.html

・fireChangeEvent()
開発メンバによるブログ（英語）
http://fire-change-event.blogspot.com/

・Tip: Validation with a MultiValidator << EclipseSource Blog
相関チェックのサンプル（英語）
http://eclipsesource.com/blogs/2009/02/27/databinding-crossvalidation-with-a-multivalidator/

開発部
多田　丈晃