

Understanding JFace data binding in Eclipse, Part 3: Exploiting advanced features

Skill Level: Intermediate

[Scott Delap \(scott@clientjava.com\)](mailto:scott@clientjava.com)
Desktop/Enterprise Java Consultant

17 Oct 2006

Almost all applications require synchronization of data between various objects and services. This tutorial, Part 3 of a series titled "[Understanding JFace data binding in Eclipse](#)," shows you how to use advanced features to accomplish this task while writing testable UIs.

Section 1. Before you start

About this series

This "[Understanding JFace data binding in Eclipse](#)" series introduces advanced features of the new JFace data binding application program interface (API) included in Eclipse V3.2.

Data binding APIs relieve you from having to write boilerplate synchronization code. The JFace data binding API provides this functionality for user interfaces (UIs) written in the Standard Widget Toolkit (SWT) and JFace. Part 1 in the series introduces the basic components in the API. Part 2 moves on to advanced topics such as testing, tables, converters, and validation.

About this tutorial

This tutorial addresses using advanced features of JFace data binding, such as converters, validation, and table. It also covers how to structure UIs in a more

testable manner. You will learn how to leverage the JFace data binding API to write Java™ UI applications that are well structured and testable.

Prerequisites

This is written for developers with some experience with the Java programming language and Eclipse. You should have a basic understand of SWT and JFace and have read [Part 1](#).

System requirements

To run the examples in this tutorial, you need a copy of the Eclipse V3.2 software development kit (SDK) and a machine capable of running it. The examples in this tutorial use Java V5 autoboxing. As a result, a Java V1.5 Java Runtime Environment (JRE) is preferred.

Section 2. Writing testable code

What does synchronization have to do with UI testing? It's a powerful tool that a Java UI developer can use to write testable UIs.

Most desktop application developers don't test their UIs. Although server-side code is often rigorously tested, a large portion of desktop business logic never gets near a JUnit test. Tools are available for this task, such as Abbot, Mercury Interactive Corp. products, and Eggplant from Redstone Software Inc. (see [Resources](#)). However, many organizations don't use these tools.

Why aren't UIs tested? There are three common culprits:

1. Poorly organized code -- Server-side applications have well-separated layers, such as persistence and business logic, but desktop applications often feature a tangled mess of concerns.
2. UIs change -- They're moving targets, with their functionality seeming to change constantly due to user demands. Even the best UI-testing tool may have trouble keeping up with such fluid UIs.
3. Commonly marketed UI testing solutions aren't the right level at which to be testing much of the UI logic.

Would you test your entire Web application at the HTTP level? Excising all the application logic would be difficult with this as your only exposed access point. Similarly, using a UI to test business logic, client/sever concerns is a trying process.

Section 3. Mangled code is bad code

As a software developer, you've long been taught about the benefits of separation of concerns. Tight coupling leads to code that isn't reusable, is hard to test, and is hard to maintain. Interestingly, all these lessons often are thrown out the window when you're developing UIs. This can be best illustrated by looking at an example.

Download the project from the [Downloads](#) section. Import it into your workspace by selecting **File > Import** from the menu. In the dialog, select **Existing Projects Into Workspace**. Select the archive file option on the following screen, then browse to select the zipped file you just downloaded. After clicking **Finish** to import it, you should now have a databinding-tutorial2 project in your workspace.

Right-click MangledConcernsExample, and select **Run As > SWT Application** from the pop-up menu. You'll see a window similar to the one shown in Figure 1. It features a simple enablement rule tying together the Name, Spouse, and Years Married fields. If you fill in a value for the Name and Spouse fields, the Years Married field becomes enabled. Removing a value from either the Name or the Spouse field causes the Years Married field to clear and become disabled. The code enabling this functionality is shown in Listing 1.

Figure 1. Example UI



Listing 1. The mangled enablement code

```
private void createControls(Composite c) {  
    ...  
    YearsMarriedEnablementListener listener = new YearsMarriedEnablementListener();  
    this.nameTxt.addModifyListener(listener);  
    this.spouseTxt.addModifyListener(listener);  
}  
  
private class YearsMarriedEnablementListener implements ModifyListener {  
    public void modifyText(ModifyEvent e) {  
        boolean enable = false;  
        if ((nameTxt.getText().trim().length() > 0)  
            && (spouseTxt.getText().trim().length() > 0)) {  
            enable = true;  
        } else {  
            yearsMarriedTxt.setText("");  
        }  
        yearsMarriedTxt.setEnabled(enable);  
    }  
}
```

There are number of issues with this example. First, note that the `YearsMarriedEnablementListener` is more of an afterthought than a strategic portion of the application. A second issue is that the code in this listener references UI controls directly. To test this code, you would have to instantiate the entire form, including UI controls. This code could be structured much better using the UI design pattern of the Presentation Model.

Section 4. Introducing the Presentation Model

One of the core patterns in desktop application development is the Model-View-Controller (MVC) pattern. This pattern doesn't quite fit modern UI development. Each widget is its own mini MVC triad, leaving little for the application to do at the widget level. There are larger application-level concerns in terms of enablement, validation, and data synchronization that need to be handled, however.

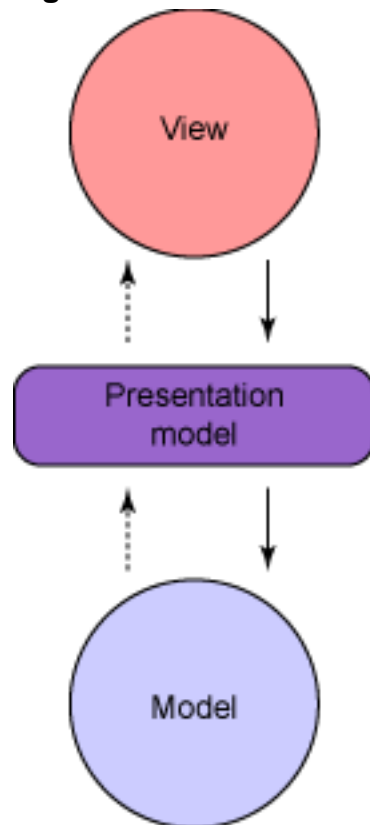
One pattern that tries to address these concerns is the Model-View-Presenter pattern. With this, UI controls delegate to a controller object for business tasks such as "save" when a button is clicked. Moving business logic to a controller is a step in the right direction in terms of testability. However, it overlooks one key factor: The controller logic often needs to access data and change state in the UI. If this state is held in widgets, such as the enabled property of a Text widget, testing the controller requires either the full UI or a stub mocking its state.

This situation can be remedied by extracting the state and the business logic out of the UI, as advocated by another variation of MVC called the Presentation Model.

You can test the business logic and state changes in the Presentation Model without any need for UI code. This separation also causes interaction between the UI and the Presentation Model to be limited to synchronization of data and state.

Figure 2 shows a diagram of the Presentation Model pattern. Now that you know some background about the Presentation Model, you can convert the mangled example using this better-structured pattern.

Figure 2. Presentation Model pattern

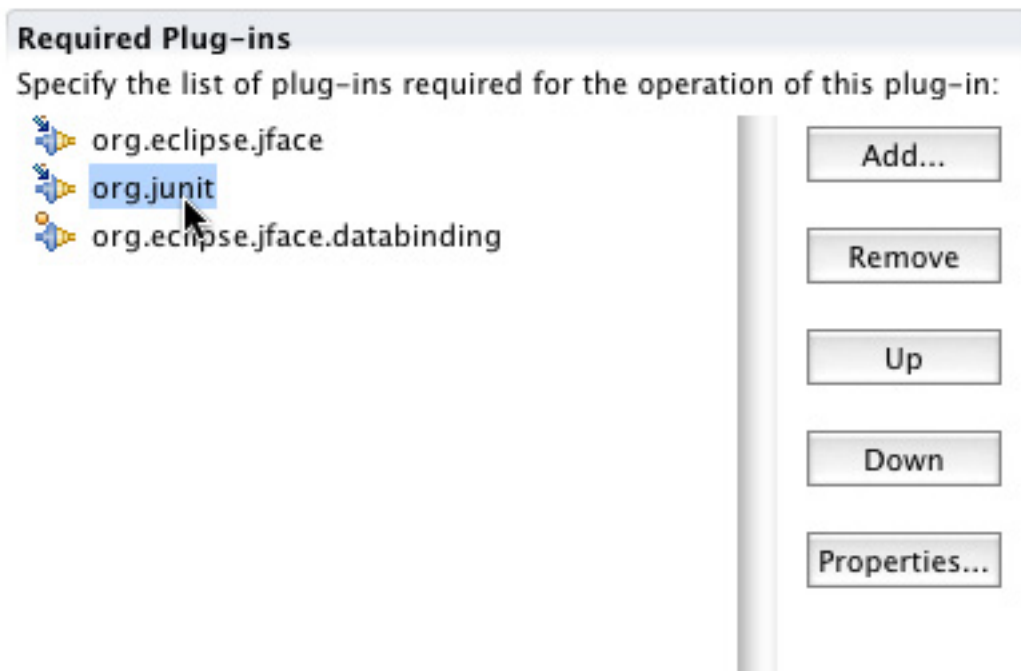


Section 5. Writing a testable Presentation Model: The test

The first step in refactoring the previous example into one using the Presentation Model is to write a test. To do so, you need to add JUnit support to your project. Open the Eclipse MANIFEST.MF editor by clicking the META-INF/MANIFEST.MF file in the package explorer. Then, click the **Dependencies** tab and the **Add** button in the Required Plug-ins section. Select the org.junit plug-in, and click **OK**. Your editor now appears similar to Figure 3.

Figure 3. Manifest.MF editor after adding dependencies

Dependencies



Next, create a new package and then a new class named `ContactPresentationModelTest` that extends `TestCase`. Insert the test method shown in Listing 2.

Listing 2. Test for Presentation Model

```
public void testYearsMarriedEnablement() {
    Contact contact = new Contact();
    ContactPresentationModel presentationModel = new
    ContactPresentationModel(
        contact);

    assertFalse(presentationModel.getEnableYearsMarried());

    presentationModel.getContact().setName("Name");
    assertFalse(presentationModel.getEnableYearsMarried());

    presentationModel.getContact().setSpouse("Spouse");
    assertTrue(presentationModel.getEnableYearsMarried());

    presentationModel.getContact().setYearsMarried("5");
    presentationModel.getContact().setSpouse("");
    assertFalse(presentationModel.getEnableYearsMarried());
    assertNull(presentationModel.getContact().getYearsMarried());
}
```

This code won't compile because you don't have the referenced `PresentationModel` yet. Create a new class called `ContactPresentationModel` in the same

package. Paste in the code from Listing 3.

Listing 3. Stubbed-in Presentation Model code

```
private Contact contact;
private boolean enableYearsMarried;

public ContactPresentationModel(Contact contact) {
    this.contact = contact;
}

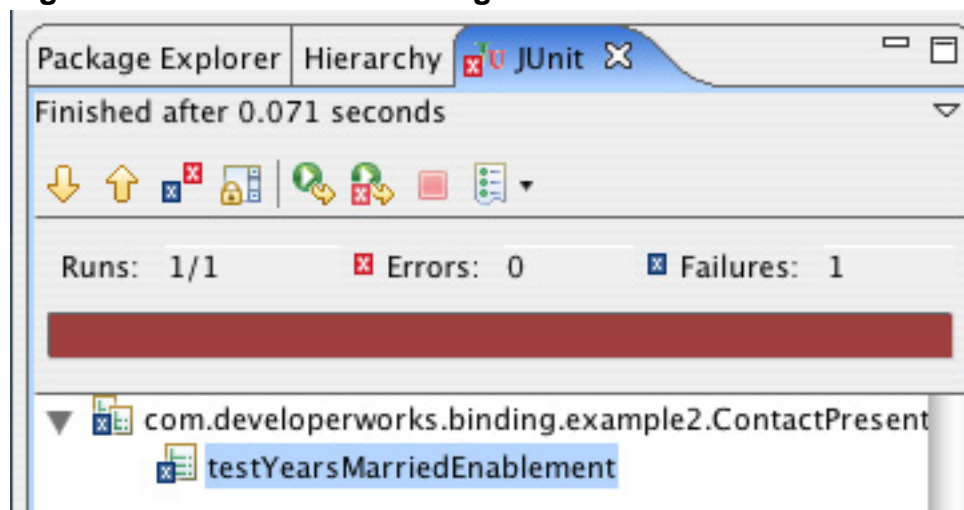
public Contact getContact() {
    return contact;
}

public void setContact(Contact contact) {
    this.contact = contact;
}

public boolean getEnableYearsMarried() {
    return this.enableYearsMarried;
}
```

At this point, the test you previously created will compile. Right-click the class in the package explorer, and then select **Run As > JUnit Test** from the pop-up menu. Your JUnit view should show the test failing, as in Figure 4.

Figure 4. JUnit view after failing tests



Let's look at exactly what the method in Listing 2 is testing. The first few lines set up a new `ContactPresentationModel` and populate it with a new `Contact` object. Because this object doesn't have a value for name or spouse, the state it holds in its `enableYearsMarried` variable (which is accessed through the `getEnableYearMarried()` getter method) should be false at initialization. The test then sets the name property and asserts that the enabled state is still false. After populating the spouse property, which should change the enablement state, an

assert is included to test for this condition. The `yearsMarried` property is then set, and the `spouse` property is cleared. The clearing of the `spouse` property should cause the enablement state to revert to false and clear the `yearsMarried` property. As a result, the two final asserts check that this has occurred.

Section 6. Writing a testable Presentation Model: The business logic

Now that you have a complete test for the desired business logic for the `ContactPresentationModel`, it's time to implement the functionality.

First, it's worth examining the stubbed-in class. It contains two properties: `contact` and `enableYearsMarried`. The `Contact` object reference is exposed so other classes can access it through the Presentation Model if needed. The `enableYearsMarried` property is added to the Presentation Model, instead of the `Contact` object, because the state and business logic that change it are tied to the editing of the `Contact`, not the object itself.

You can now modify the `ContactPresentationModel` to implement the contract specified in the test. Open the class in the Eclipse Java editor. The first change needed is to implement property change support for the `enableYearsMarried` property, for later use with JFace data binding. This can be done by changing its setter to match the code shown in Listing 4.

Listing 4. Enablement state setter with support for property changes

```
public
void
setEnableYearsMarried(boolean
enableYearsMarried)
{
boolean
oldVal
=
this.enableYearsMarried;
this.enableYearsMarried
=
enableYearsMarried;
firePropertyChange("enableYearsMarried",
\
oldVal,
this.enableYearsMarried);
}
```

Next, you need to rewrite the logic in the `ModifyListener` from the mangled example to something that operates on the `Contact` object, instead of directly on

the widgets of the UI. This new property-change listener is shown in Listing 5.

Listing 5. New property-change listener implementing the business logic in the Presentation Model

```
private class EnablementPropertyChangeListener implements
    PropertyChangeListener {

    public void propertyChange(PropertyChangeEvent evt) {
        boolean enable = false;
        if ((getContact().getName() != null &&
            getContact().getName().trim().length() > 0) &&
            (getContact().getSpouse() != null &&
            getContact().getSpouse().trim().length() > 0)) {
            enable = true;
        } else {
            getContact().setYearsMarried(null);
        }
        setEnableYearsMarried(enable);
    }
}
```

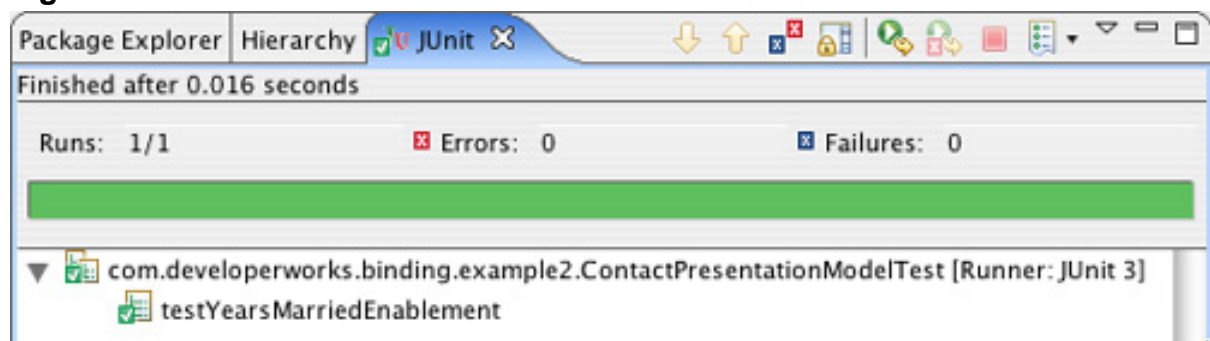
Finally, the constructor of the `ContactPresentationModel` needs to wire this new listener to the name and spouse properties of the `Contact`, as shown in Listing 6.

Listing 6. Adding the listener to the contact's fields

```
EnablementPropertyChangeListener enablementPropertyChangeListener
    = new EnablementPropertyChangeListener();
this.contact.addPropertyChangeListener("name",
    enablementPropertyChangeListener);
this.contact.addPropertyChangeListener("spouse",
    enablementPropertyChangeListener);
```

If you right-click the `ContactPresentationModel` test and run it as a JUnit test, you see the famous JUnit green bar of success, as shown in Figure 5.

Figure 5. JUnit view after successful tests



You now have the same business logic as in the mangled coded example, rewritten in a fully testable UI-independent manner.

Section 7. Synchronizing the Presentation Model with the UI

You may wonder what all this has to do JFace data binding. The Presentation Model makes code easier to test, as you just saw. However, the data and state in the Presentation Model still aren't reflected in a UI. Writing all this synchronization code yourself would be a chore. Luckily, you have JFace data binding at your disposal. The `ContactForm` from the mangled example can easily be refactored by changing its constructor and `bindGUI()` methods, as shown in Listing 7.

Listing 7. Refactored ContactForm

```
public
ContactForm(Composite
c,
ContactPresentationModel
presentationModel)
{
this.contact
= new
Contact();
createControls(c);
createButtons(c);
bindGUI(presentationModel);
}

private
void
bindGUI(ContactPresentationModel
presentationModel)
{
DataBindingContext
ctx =
createContext();
ctx.bind(nameTxt,
new
Property(presentation\
Model.getContact(),
"name"),
new
BindSpec());
ctx.bind(spouseTxt,
new
Property(presentation\
Model.getContact(),
"spouse"),
new
BindSpec());
ctx.bind(yearsMarriedTxt,
new
Property(presentation\
Model.getContact(),
"yearsMarried"),
new
```

```
BindSpec();
ctx.bind(new
Property(yearsMarriedTxt,
"enabled"),
new
Property(presentation\
Model,
"enableYearsMarried"),
new
BindSpec());
}
```

Next, change the `run()` method in the example runner, as shown in Listing 8.

Listing 8. The refactored example runner

```
public void run() {
    ...
    ContactPresentationModel presentationModel = \
    new ContactPresentationModel(contact);
    ContactForm contactForm = new ContactForm(shell, presentationModel);

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
    display.dispose();
}
```

Section 8. Changing the Presentation Model

Now that you've separated your code into UI, Presentation Model, and domain model layers, it can be easily modified to meet changing requirements. Suppose your client wants to be more explicit and have the UI appear as shown in Figure 6, with a Married checkbox. Checking/unchecking this box enables/disables both the Spouse and Years Married fields.

Figure 6. Example UI after adding an explicit checkbox



Because the functionality of the Presentation Model will change, you need to modify its test to validate what to implement. Listing 9 shows the trimmed-down test method. The checkbox handles changing the enablement state, so the only other logic required in the listener is to clear the two marriage-related fields as needed.

Listing 9. The refactored enablement test

```
public void testYearsMarriedEnablement() {
    Contact contact = new Contact();
    ContactPresentationModel presentationModel = new ContactPresentationModel(
        contact);

    assertFalse(presentationModel.getEnableYearsMarried());

    presentationModel.setEnableYearsMarried(true);
    presentationModel.getContact().setSpouse("spouse");
    presentationModel.getContact().setYearsMarried("5");
    presentationModel.setEnableYearsMarried(false);

    assertNull(presentationModel.getContact().getSpouse());
    assertNull(presentationModel.getContact().getYearsMarried());
}
```

To make the test pass, refactor the `ContactPresentationModel` constructor and the `EnablementPropertyChangeListener`, as shown in Listing 10.

Listing 10. Modifying the Presentation Model with explicit enablement

```
public ContactPresentationModel(Contact contact) {
    this.contact = contact;

    EnablementPropertyChangeListener enablementPropertyChangeListener =
        new EnablementPropertyChangeListener();
    addPropertyChangeListener("enableYearsMarried",
        enablementPropertyChangeListener);
}
```

```
private class EnablementPropertyChangeListener implements
    PropertyChangeListener {

    public void propertyChange(PropertyChangeEvent evt) {
        if (!getEnableYearsMarried()) {
            getContact().setYearsMarried(null);
            getContact().setSpouse(null);
        }
    }
}
```

The only thing left is to modify the UI. You need to add a label and the checkbox to the `createControls()` method. This new checkbox must then be bound to the Presentation Model. Finally, to enable/disable the Spouse field as you do the Years Married field, bind its enabled property to the same place on the Presentation Model. These changes are shown in Listing 11.

Listing 11. Adding a checkbox to the UI

```
private void bindGUI(ContactPresentationModel
    presentationModel) {
    . . .
    ctx.bind(chkIsMarried,
        new Property(presentationModel, "enableYearsMarried"),
        new BindSpec());
    ctx.bind(new Property(spouseTxt, "enabled"),
        new Property(presentationModel, "enableYearsMarried"),
        new BindSpec());
}

private void createControls(Composite c) {
    . . .

    Label labelMarried = new Label(c, SWT.SHELL_TRIM);
    labelMarried.setText("Married");

    gridData = new GridData(GridData.FILL_HORIZONTAL);
    this.chkIsMarried = new Button(c, SWT.CHECK);
    this.chkIsMarried.setLayoutData(gridData);
    . . .
}
```

As you can see, compact changes can be made to each of the example's building blocks. It's easy to modify enablement logic to clear a second field on the model. It also required only one more binding line to apply enablement to the additional Spouse field in the UI and keep it synchronized.

Section 9. Introducing the BindSpec

Up to this point in this tutorial, as well as Part 2, you've seen instances of `BindSpec` created with no other follow-up information provided. Sometimes, when you're binding the properties of two objects, more configuration is needed to achieve the desired data flow back and forth. This is where the `BindSpec` class comes in. It serves as a way to specify more binding configuration, as well as provide validation and conversion functionality during data synchronization.

Looking at the `BindSpec` class, you'll see that it contains setter methods for model-to-target and target-to-model converters. Each method requires a class implementing the `IConverter` interface, shown in Listing 12.

Listing 12. The `IConverter` interface

```
public
interface
IConverter
{

public
Object
getFromType();

public
Object
getToType();

public
Object
convert(Object
fromObject);
}
```

To allow JFace data binding to check that you've specified a valid converter for the target and model being bound, the interface requires types for each side of the conversion. Normally, this is something like `String.class`. The `DataBindingContext.bind` method then compares these types against the types of the model and target at bind time for consistency. The only other method required does the actual conversion.

The other main option on the `BindSpec` is that of a validator. Validators can be set on the target and model sides. Calling `setValidator` defaults to the target side. This causes data from a widget, for example, to be validated before being synchronized with a model. Validators must implement the `IValidator` interface shown in Listing 13.

Listing 13. The `IValidator` interface

```
public interface IValidator {

    public ValidationError isPartiallyValid(Object value);
}
```

```
public ValidationError isValid(Object value);  
}
```

The `isPartiallyValid()` method allows you to perform validation while a value is being changed -- for example, while a text field has focus and the user is typing. The `isValid()` method, in comparison, is invoked after all changes have been made, but before synchronization with the model -- for example, tabbing off a text field.

Section 10. Implementing a custom converter

Moving back to the example, imagine that the client has requested that the plain-looking Married checkbox be changed to a combo box containing the words "Yes" and "No," as shown in Figure 7.

Figure 7. Example UI with a combo instead of a checkbox



However, remember that the checkbox is bound to the `enableYearsMarried` property, which is of type `boolean`. There is a type mismatch, with a `String` on one side and a `boolean` on the other. This is a perfect place for a converter.

Although you could write tests for the functionality presented in the rest of this tutorial, the example focuses on implementations. Only changes needed to keep the `ContactPresentationModelTest` are made in regard to testing. In a development environment, however, it's always a good idea to write tests.

Create a new class named `BooleanToStringConverter`. For its `fromType`,

return `Boolean.TYPE`. Return `String.class` for its `toType`. In the `convert()` method, cast the object to a `Boolean` and return `Yes` if true and `No` if false. Now, create the corresponding converter by creating the class `StringToBooleanConverter`. Swap the from and to types, and invert the `convert()` method to return true if the value is `Yes` and false if the value is `No`. Java 5 autoboxing takes care of the object conversion.

Next, you need to change the UI. Remove the code associated with the checkbox and replace it with the code in Listing 14. This listing also contains the replacement binding method calls for the `bindGUI()` method. Recall from Part 2 that the combo widget has a selection property to which it can be bound. The binding line also uses a different constructor for the `BindSpec` class, which allows you to specify usage of the `BooleanToStringConverter` and `StringToBooleanConverter`.

Listing 14. Replacing the checkbox with a combo

```
gridData = new GridData(GridData.FILL_HORIZONTAL);
this.comboIsMarried = new Combo(c, SWT.BORDER);
this.comboIsMarried.setLayoutData(gridData);
this.comboIsMarried.add("Yes");
this.comboIsMarried.add("No");

. . .

ctx.bind(new Property(comboIsMarried, SWTProperties.SELECTION),
         new Property(presentationModel,
                     "enableYearsMarried"),
         new BindSpec(new BooleanToStringConverter(),
                     new StringToBooleanConverter(), null, null));
```

Section 11. Implementing a custom validator

The fields in the example all take strings at the moment. However, the `Years Married` field should be restricted to being a number. One way to accomplish this is with a custom validator. Create a class called `YearsMarriedValidator` and paste in the code from Listing 15.

Listing 15. A custom validator

```
public
class
YearsMarriedValidator
implements
IValidator
{
public
```

```

ValidationError
isPartiallyValid(Object
value)
{
try {
Integer.valueOf((String)
value);
return
null;
} catch
(NumberFormatException
nfe) {
return
new
ValidationError(ValidationError.ERROR,
"Not A
Number");
}
}

public
ValidationError
isValid(Object
value)
{
if
("5".equals(value))
{
return
ValidationError.error("5
Is Not
Allowed");
} else
{
return
null;
}
}
}

```

This code implements both the `isPartiallyValid()` and `isValid()` methods. For the `isPartiallyValid()` method, an attempt is made to convert the string entered to an `Integer`. If a `NumberFormatException` is thrown, then you know this attempt failed. As a result, a `ValidationError` is returned. For the sake of example, the `isValid()` method returns a `ValidationError` if the number 5 is entered. The only thing left is to include the validator in the `BindSpec` of the `yearsMarriedTxt` field, as shown in Listing 16.

Listing 16. Binding the validator

```

ctx.bind(validationErrorLabel, binding.getValidationError(),
        new BindSpec(new ValidationErrorToStringConverter(),
                    new ReadOnlyConverter(String.class,
                    ValidationError.class), null, null));

```

At this point, open your modified example and enable the **Years Married** field by selecting **Yes** from the combo box. Try to enter the letters `abc` in the **Years Married** field. Nothing happens because the validator's partial validation check prevents numbers from being entered. Now enter the number 1 and try to backspace over the

number. Interestingly, you'll find that you can't. You didn't code the validator to allow nulls or empty strings, so removing the number isn't allowed because it doing so would result in an invalid value. Modify the partial validation method to handle these cases, and rerun the application.

In order to more easily explore the `isValid()` method on the `YearsMarriedValidator`, create and bind a label widget to the property with the code in Listing 17.

Listing 17. Adding a label showing the Years Married value

```
Label yvLabel = new Label(c, SWT.NONE);
yvLabel.setText("YM Value:");
this.ymValLabel = new Label(c, SWT.BORDER);
gridData = new GridData(GridData.FILL_HORIZONTAL);
this.ymValLabel.setLayoutData(gridData);

...

ctx.bind(ymValLabel, new Property(presentationModel.getContact(),
"yearsMarried"), new BindSpec());
```

Now you can visually see changes to the property in the contact object as they're triggered by JFace data binding. Enable the Years Married field, and type `abc` again. You'll notice that nothing appears in the label because the invalidated changes aren't synchronized. Next, enter the number `1`. It's synchronized with the `Contact` object. Because the label is also bound to the `Contact` object, it also changes to `1`. Enter the number `5` and tab out of the field. Notice that although the value stays in the Text widget, it doesn't appear in the label. This is due to the validator preventing synchronization.

Section 12. Observing validation errors

The changes you've made are useful, but it would also be nice to be able to notify the user when a validation error occurs. This functionality can be implemented by binding to special observables.

If you look at the method signature for the `bind()` method in the `DataBindingContext` class, you'll notice that it returns a `Binding` object, which you've ignored up to this point. This `Binding` object is the mediator that is responsible for keeping the data between the model and the target in harmony. It also invokes converters and validators at appropriate times. Each `Binding` object also has an observable for partial and full `ValidatorErrors`. You can observe these to determine when errors have occurred. Modify the `ContactForm` class by

adding two labels in which to view the results, and then bind them as shown in Listing 18. Fix any imports as needed. This code relies on a few classes in the extras package included in the project.

Listing 18. Displaying errors in labels

```

ctx.bind(partialValidationErrorLabel,
binding
.getPartialValidationError(),
new
BindSpec(
new
ValidationErrorToStringConverter(),
\
new
ReadOnlyConverter(
String.class,
ValidationError.class),
null,
null));
ctx.bind(validationErrorLabel,
binding.getValidationError(),
new
BindSpec(new
ValidationErrorToStringConverter(),
new
ReadOnlyConverter(String.class,
ValidationError.class),
null,
null));
. . .

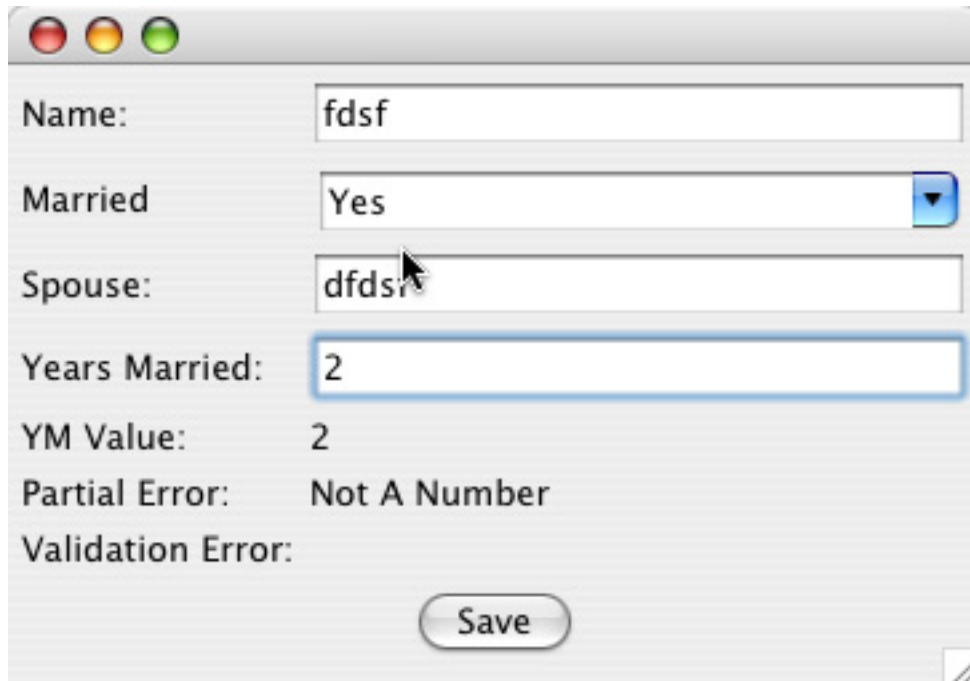
Label
partialLabel
= new
Label(c,
SWT.NONE);
partialLabel.setText("Partial
Error:");
this.partialValidationErrorLabel
= new
Label(c,
SWT.BORDER);
gridData
= new
GridData(GridData.FILL_HORIZONTAL);
this.partialValidationErrorLabel.setLayoutData(gridData);

Label
fullLabel
= new
Label(c,
SWT.NONE);
fullLabel.setText("Validation
Error:");
this.validationErrorLabel
= new
Label(c,
SWT.BORDER);
gridData
= new
GridData(GridData.FILL_HORIZONTAL);
this.validationErrorLabel.setLayoutData(gridData);

```

Right-click your example runner, and run the application again as an SWT application. You should see a dialog similar to the one shown in Figure 8. Enable the Years Married field and enter a nonnumeric character. Notice the error message that appears. Next, try entering the number 5 and tabbing out of the field to test the other validation label. Finally, change the label to the number 4. Both error labels should be blank because no validation errors have occurred.

Figure 8. Example UI with labels presenting validation errors



Section 13. Master-detail relationships with tables

Applications often present summary lists of objects. When you select one, the detail appears in a form. Such functionality can be coded in with JFace data binding by using a collection widget, such as a List or Table. The selected value can then be bound lazily to the target form for displaying the detail records.

The first step in implementing such functionality is to create another Presentation Model to hold the list for the table to present. Code for this is shown in Listing 19. This Presentation Model also holds on to a `WritableValue` to preserve the selection state of the table. Again, notice that all state has been extracted out of the UI table widget and represented in the Presentation Model.

Listing 19. Presentation Model for the TableForm

```
public
class
TablePresentationModel
extends
PropertyChangeAware
{

private
List
contacts;

private
WritableValue
selectedContact;

public
TablePresentationModel(List
contacts)
{
this.contacts
=
contacts;
this.selectedContact
= new
WritableValue(Contact.class);
this.selectedContact.setValue(contacts.get(0));
}

public
List
getContacts()
{
return
contacts;
}

public
void
setContacts(List
contacts)
{
this.contacts
=
contacts;
}

public
WritableValue
getSelectedContactObservable()
{
return
selectedContact;
}

public
void
setSelectedContactObservable\
(WritableValue
selectedContact)
{
this.selectedContact
=
selectedContact;
}
}
```

With a Presentation Model created, you now need a UI. Listing 20 shows the code for a `TableForm` class.

Listing 20. Implementation of the `TableForm`

```
public class TableForm {

    private TableView contactsTableView;

    public TableForm(Composite c, TablePresentationModel presentationModel) {
        createControls(c);
        bindGUI(presentationModel);
    }

    private void bindGUI(TablePresentationModel presentationModel){
        DataBindingContext ctx = createContext();

        ctx.bind(new Property(this.contactsTableView,
            ViewersProperties.CONTENT), new TableModelDescription(new
            Property(presentationModel, "contacts", Contact.class, true),
            new String[] {"name", "spouse"}), null);
        ctx.bind(new Property(this.contactsTableView,
            ViewersProperties.SINGLE_SELECTION),
            presentationModel.getSelectedContactObservable(), null);
    }

    private void createControls(Composite c) {
        GridData gridData = new
        GridData(GridData.FILL_HORIZONTAL);
        gridData.horizontalSpan = 2;
        this.contactsTableView = new TableView(c,
        SWT.BORDER);

        this.contactsTableView.getTable().setLayoutData(gridData);
    }

    public static DataBindingContext createContext() {
        DataBindingContext context = new DataBindingContext();
        context.addObservableFactory(new
            BeanObservableFactory(context, null,
            new Class[] { Widget.class }));
        context.addObservableFactory(new SWTObservableFactory());
        context.addObservableFactory(new ViewersObservableFactory());
        context.addBindSupportFactory(
            new DefaultBindSupportFactory());
        context.addBindingFactory(new DefaultBindingFactory());
        context.addBindingFactory(new ViewersBindingFactory());
        return context;
    }
}
```

Similar to the `ContactForm`, the `TableForm` takes a Presentation Model and binds its contents to a UI. In this case, the widget is a Table. The first line in the `bindGUI()` method connects the `Contact` objects in the list from the Presentation Model to the table. Instead of a simple `Property` object supplied during binding, a `TableModelDescription` object is used. This object allows you to pass along an array of strings to indicate which properties of the `Contact` object to bind to columns in the table. The second bind line in the method binds the selected value in the table to the `WritableValue` selection holder you created in the Presentation

Model. Finally, notice that the `createContext()` method at the end of the class definition adds `ViewerObservableFactory` and `ViewersBindingFactory` to the context. Without these factories, the context won't know how to bind data to a table.

This is a good stopping point to test your code thus far. Modify the example runner with the code in Listing 21. This code creates a few example contacts, passes them into the Presentation Model, then builds the `TableForm`. Right-clicking the runner and running it as a SWT application opens a window similar to the one shown in Figure 9.

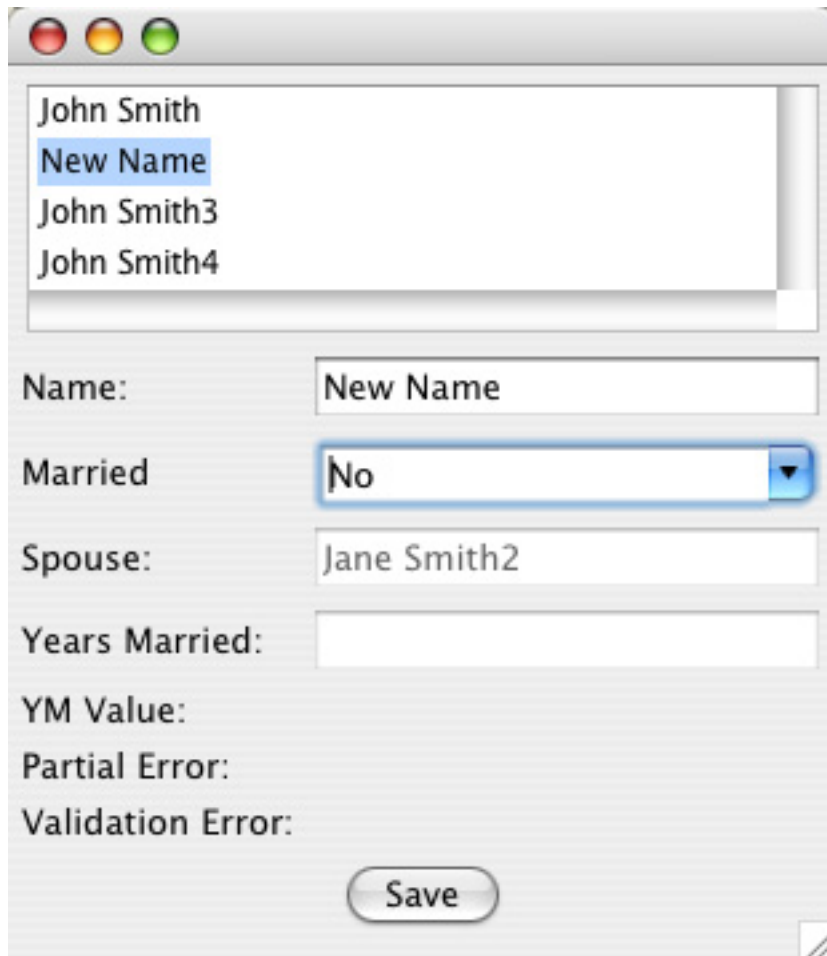
Listing 21. Modifying the example runner to try the `TableForm`

```
Contact contact = new Contact();
List contacts = new ArrayList();
contacts.add(new Contact("John Smith", "Jane Smith"));
contacts.add(new Contact("John Smith2", "Jane Smith2"));
contacts.add(new Contact("John Smith3", "Jane Smith3"));
contacts.add(new Contact("John Smith4", "Jane Smith4"));

TablePresentationModel tablePresentationModel = new
TablePresentationModel(
    contacts);
TableForm tableForm = new TableForm(shell,
tablePresentationModel);

ContactPresentationModel presentationModel = new
ContactPresentationModel(
    contact);
ContactForm contactForm = new ContactForm(shell,
presentationModel);
```

Figure 9. Example UI implementing a master-detail relationship



The screenshot shows a graphical user interface for a contact form. At the top, there is a table with four rows of names: "John Smith", "New Name", "John Smith3", and "John Smith4". The "New Name" row is highlighted in blue. Below the table, there are several form fields: "Name:" with the value "New Name", "Married" with a dropdown menu set to "No", "Spouse:" with the value "Jane Smith2", "Years Married:" (empty), "YM Value:" (empty), "Partial Error:" (empty), and "Validation Error:" (empty). A "Save" button is located at the bottom center of the form.

Section 14. Introducing indirection

All that's left is to hook the selection from the table into `ContactPresentationModel` for viewing. Wikipedia defines indirection in computer programming as "the ability to reference something using a name, reference, or container instead of the value itself." You use this approach with `ContactForm` and `ContactPresentationModel` by binding to a placeholder that will be filled in later. Refactor it so that the `Contact` variable is now instead an `IObservable` value. Change the variable and method names accordingly to be `contactObservable`. Doing so causes a few compilation errors. Fix the `enablementChangeListener` and the test with the code in Listing 22.

Listing 22. Connecting the `TablePresentation` model and `ContactPresentationModel`

```
if
(!getEnableYearsMarried())
{
Contact
contact
=
(Contact)
\
getContactObservable().getValue();
if
(contact
!=
null) {
contact.setYearsMarried(null);
contact.setSpouse(null);
}
}
. . .

ContactPresentationModel
presentationModel
= new
ContactPresentationModel(
new
WritableValue(Contact.class));
presentationModel.getContactObservable().setValue(contact);

assertFalse(presentationModel.getEnableYearsMarried());

presentationModel.setEnableYearsMarried(true);
contact.setSpouse("spouse");
contact.setYearsMarried("5");
presentationModel.setEnableYearsMarried(false);

assertNull(contact.getSpouse());
assertNull(contact.getYearsMarried());
```

Now you need to fix the `ContactForm`. Make sure the previous `getContact()` methods have been refactored to be `getContactObservable()`. Because you're now binding to an `IObservableValue` instead of a `Contact` object directly, you must be more explicit at binding time. Modify the `Property` object constructors for `name`, `spouse`, and `yearsMarried` to have a third argument of `String.class` and a fourth argument of `false`. Doing so specifies the type of the property you're binding to and the fact that it isn't a collection. Finally, fix the errors in the example runner by changing the `ContactForm` constructor to take the `WritableValue` instance from the `TablePresentationModel`.

Run the example again. Notice that the first value from the table is selected and also shown in the form below. If you change the value of the **Name** field, it changes in the table. Changing the selection in the table changes the object displaying in the form.

Section 15. Conclusion

This tutorial has provided a more advanced introduction to core features in the JFace data binding API. It has also shown how data binding can help you write more testable code that implements the Presentation Model pattern. Along the way, you've seen how data binding relieves you from writing the tedious boilerplate synchronization code often necessary in desktop applications. In its place, the JFace data binding API provides a set of interfaces and implementations to generically reference JavaBean properties and properties of SWT/JFace widgets. JFace data binding also includes powerful API features to handle conversions, validation, and indirect binding.

Downloads

Description	Name	Size	Download method
Part 3 source code	os-ecl-jfacedb3.source.zip	115KB	HTTP

[Information about download methods](#)

Resources

Learn

- Learn more about the [Presentation Model](#) pattern.
- Read "[How Many Data Binding Frameworks = A Bad Thing](#)" at ClientJava.com.
- Find out more about [JFace data binding](#).
- Learn more about the [Eclipse Foundation](#) and its many projects.
- Expand your Eclipse skills by visiting IBM developerWorks' [Eclipse project resources](#).
- Browse all of the [Eclipse content](#) on developerWorks.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Stay current with developerWorks [technical events and webcasts](#).
- To listen to interesting interviews and discussions for developers, be sure to check out [developerWorks podcasts](#).

Get products and technologies

- Check out [Mercury WinRunner](#) from Mercury Interactive Corp.
- Check out [EggPlant](#) from Redstone Software Inc.
- Download the SWT binding alternative [SWTBinding](#).
- Check out the latest [Eclipse technology downloads](#) at [alphaWorks](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- The [Eclipse newsgroups](#) has many resources for people interested in using and extending Eclipse.
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Scott Delap



Scott Delap is an independent consultant specializing in Java EE and rich Java clients. He has presented papers at JavaOne and is actively involved in the desktop Java community. He is also the administrator of ClientJava.com, a portal focused on desktop Java development. ClientJava.com is frequently featured all over the Web, from JavaBlogs to Sun Microsystems' Web site.