Understanding JFace data binding in Eclipse, Part 1: The pros and cons of data binding

Stop writing data synchronization code and start solving problems

Scott Delap

Large amounts of the development time for Java™ graphical user interface (GUI) applications involve simply moving data out of domain objects into GUI components and back again. In recent years, several data binding frameworks have stepped to the forefront to automate this data synchronization process. This article explains what a data binding framework does, introduces several popular Java GUI data binding frameworks, and covers the pros and cons of using data binding.

Most modern Web applications feature view layers that are smart enough to synchronize request and response variables with HTML input tags. This process is easy to accomplish because input from the user is routed through the structured layers of Web applications and HTTP. Java GUI applications, on the other hand, do not often feature this order. Regardless of whether they are written in Standard Widget Toolkit (SWT) or Swing, these applications usually do not have a defined path between the domain objects of the application and its GUI controls (also commonly referred to as components).

Boilerplate data synchronization

What results instead of order is, at worst, chaos and, at best, large blocks of boilerplate synchronization code prone to bugs. Consider the code excerpt in Listing 1. You see the definition of a simple domain object called a FormBean. When a dialog wishes to make use of this data, it has to extract it from the domain object and insert it into components for display, as shown in the method call at the end of the constructor. Correspondingly, after the user changes the information, it must be extracted from the GUI components and set back in the domain model. This back-and-forth process is performed by the syncBeanToComponents() and syncComponentsToBean() methods. Finally, notice that references to the GUI components need to be kept available in the object scope so they can be accessed in the synchronization methods.

Listing 1. A Swing dialog without data binding

```java
package com.nfjs.examples;
```
import com.jgoodies.forms.layout.FormLayout;
import com.jgoodies.forms.layout.CellConstraints;
import com.jgoodies.forms.builder.DefaultFormBuilder;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionEvent;

public class NoBindingExample {
    private JFrame frame;
    private JTextField firstField;
    private JTextField lastField;
    private JTextArea descriptionArea;
    private FormBean bean;

    public NoBindingExample() {
        frame = new JFrame();
        firstField = new JTextField();
        lastField = new JTextField();
        descriptionArea = new JTextArea(6, 6);
        DefaultFormBuilder builder =
            new DefaultFormBuilder(new FormLayout("r:p, 2dlu, f:p:g"));
        builder.setDefaultDialogBorder();
        builder.append("First:", firstField);
        builder.append("Last:", lastField);
        builder.appendRelatedComponentsGapRow();
        builder.appendRow("p");
        builder.add(new JLabel("Description:"),
            new CellConstraints(1,
                5, CellConstraints.RIGHT,
                CellConstraints.TOP),
            new JScrollPane(descriptionArea),
            new CellConstraints(3,
                5, CellConstraints.FILL,
                CellConstraints.FILL));
        builder.nextRow(2);
        builder.append(new JButton(new MessageAction()));
        frame.add(builder.getPanel());
        frame.setSize(300, 300);
        bean = new FormBean();
        syncBeanToComponents();
    }

    private void syncBeanToComponents() {
        firstField.setText(bean.getFirst());
        lastField.setText(bean.getLast());
        descriptionArea.setText(bean.getDescription());
    }

    private void syncComponentsToBean() {
        bean.setFirst(firstField.getText());
        bean.setLast(lastField.getText());
        bean.setDescription(descriptionArea.getText());
    }

    public JFrame getFrame() {
        return frame;
    }
}

private class FormBean {
    private String first;
    private String last;
}
private String description;

public FormBean() {
    this.first = "Scott";
    this.last = "Delap";
    this.description = "Description";
}

public String getFirst() {
    return first;
}

public void setFirst(String first) {
    this.first = first;
}

public String getLast() {
    return last;
}

public void setLast(String last) {
    this.last = last;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}
}

private class MessageAction extends AbstractAction {
    public MessageAction() {
        super("Message");
    }

    public void actionPerformed(ActionEvent e) {
        syncComponentsToBean();
        JOptionPane.showMessageDialog(null, "First name is " + bean.getFirst());
    }
}

public static void main(String[] args) {
    NoBindingExample example = new NoBindingExample();
    example.getFrame().show();
}

Data binding to the rescue

While this example is a simple one, there are 10 lines of extra code if you count the component reference assignments in the constructor. If a new field is added to the bean, there are three more lines to add for initialization and synchronization in each direction between the GUI components and domain model. Writing this code over and over is boring, often resulting in the introduction of bugs into an application. Fortunately, there is a better solution.

Data binding frameworks allow developers to easily "glue" JavaBean properties and GUI components together. The JavaBean property is normally referred by a string that tells the data binding framework to look for corresponding getters and setters on the JavaBean. For example, "first" signifies that there are getFirst() and setFirst() methods on a given JavaBean.
Components are initialized with data automatically. When the value in the component changes, the associated JavaBean property is changed. Likewise, the JavaBean supports property change listeners, so the GUI component can be updated when its corresponding JavaBean property is changed.

You can also configure when modern Java data binding frameworks synchronize changes (usually when a key is pressed, a mouse is clicked, or focus is lost). They also support a wide range of GUI components, such as text fields, check boxes, lists, and tables.

Listing 2 shows the code excerpt from Listing 1 rewritten to use the JGoodies data binding framework.

**Listing 2. The same Swing dialog using JGoodies data binding**

```java
package com.nfjs.examples;
import com.jgoodies.forms.layout.FormLayout;
import com.jgoodies.forms.layout.CellConstraints;
import com.jgoodies.forms.builder.DefaultFormBuilder;
import com.jgoodies.binding.beans.BeanAdapter;
import com.jgoodies.binding.adapter.BasicComponentFactory;
import javax.swing.*;
import java.awt.event.ActionEvent;
public class BindingExample {
    private JFrame frame;
    private FormBean bean;
    public BindingExample() {
        frame = new JFrame();
        bean = new FormBean();
        BeanAdapter adapter = new BeanAdapter(bean);

        JTextField firstField = BasicComponentFactory.createTextField(
                adapter.getValueModel("first"));
        JTextField lastField = BasicComponentFactory.createTextField(
                adapter.getValueModel("last"));
        JTextArea descriptionArea = BasicComponentFactory.createTextArea(
                adapter.getValueModel("description"));
        DefaultFormBuilder builder =
                new DefaultFormBuilder(new FormLayout("r:p, 2dlu, f:p:g"));
        builder.append("First:", firstField);
        builder.append("Last:", lastField);
        builder.appendRelatedComponentsGapRow();
        builder.addRow(2);
        builder.add(new JButton(new MessageAction()));
        frame.add(builder.getPanel());
```

Understanding JFace data binding in Eclipse, Part 1: The pros and cons of data binding
I'll cover the implementation details in a moment. For now, notice the differences in terms of what is not present in comparison to the first example:

- References to the components don't have to be exposed simply for synchronization purposes. They aren't propagated outside of the constructor scope.
- Neither synchronization method is present.
- There is no initial synchronization in the constructor to populate the components.
- There is no synchronization in the action before displaying the dialog.

Despite the fact that all these items are now missing, this example performs exactly the same as the first one.

**JGoodies data binding implementation specifics**

Covering the entire JGoodies data binding framework is outside the scope of this article. However, it is useful to look at the implementation specifics of the example shown in Listing 2. Two lines explain all of the magic:

```java
BeanAdapter adapter = new BeanAdapter(bean);
JTextField firstField =
    BasicComponentFactory.createTextField(adapter.getValueModel("first"));
```

The first line creates a JGoodies object, called `BeanAdapter`, which is used to create value model objects. Value models define a generic way to access a JavaBean property without requiring the specifics of its name. Listing 3 shows the `ValueModel` interface definition.
Listing 3. The ValueModel interface

```java
public interface ValueModel {
    java.lang.Object getValue();
    void setValue(java.lang.Object object);
    void addValueChangeListener(PropertyChangeListener propertyChangeListener);
    void removeValueChangeListener(PropertyChangeListener propertyChangeListener);
}
```

The `BasicComponentFactory` class contains methods that create Swing components that are glued to the `ValueModel` supplied. The second line uses `BasicComponentFactory` to create a `JTextField`. In this case, a `JTextField` is glued to the "first" property of the `FormBean`. The JGoodies data binding API handles the rest in terms of performing initialization of the text field with the data from the `FormBean`. It also synchronizes any changes made in the text field back to the `FormBean`.

### Getting the value from the domain object into the model

It may still seem like all of this synchronization is performed with smoke and mirrors. However, that is not the case. Almost all modern GUI components have a model behind them. The task of a data binding framework is to get the value stored in the domain object into that model. Frameworks take two approaches to performing this task.

#### Adapting the bean field

One approach is to adapt the bean field being bound into becoming the model of the component itself. With this approach, when the view part of the component attempts to retrieve or modify the value, it goes directly to the value in the bean. In many cases this approach is used by the JGoodies Data Binding Framework.

Figure 1 shows how JGoodies uses the `DocumentAdapter` and `PropertyAdapter` classes to decorate a bean into being a model for a `JTextComponent`.

![Adapting a field into a JTextComponent Model](image-url)

**Figure 1. JGoodies adapting of a field into a JTextComponent Model**

...
Automating the calling of getters and setters

Another approach to synchronizing model and GUI values is to automate the calling of getters and setters between each side of the relationship when a value changes in the other. This technique is leveraged by the JFace Data Binding framework for use with SWT.

Listing 4 shows the same example as before rewritten using SWT and JFace data binding. This framework features a context object that it uses to glue fields together. Notice the three `context.bind()` method calls used to associate the text controls to the `FormBean` fields.

Listing 4. The same Swing dialog using JFace data binding

```java
import org.eclipse.jface.examples.databinding.nestedselection.BindingFactory;
import org.eclipse.jface.internal.databinding.provisional.DataBindingContext;
import org.eclipse.jface.internal.databinding.provisional.description.Property;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.MessageBox;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class JFaceBindingExample {
    private Shell shell;
    private FormBean bean;

    public void run() {
        Display display = new Display();
        shell = new Shell(display);

        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 2;
        shell.setLayout(gridLayout);

        bean = new FormBean();

        DataBindingContext context = BindingFactory.createContext(shell);

        Label label = new Label(shell, SWT.SHELL_TRIM);
        label.setText("First:");
        GridData gridData = new GridData(GridData.FILL_HORIZONTAL);
        Text text = new Text(shell, SWT.BORDER);
        text.setLayoutData(gridData);
        context.bind(text, new Property(bean, "first"), null);

        label = new Label(shell, SWT.NONE);
        label.setText("Last:");
        gridData = new GridData();
        gridData.horizontalAlignment = SWT.CENTER;
        gridData.horizontalSpan = 2;
        Button button = new Button(shell, SWT.PUSH);
```
button.setLayoutData(gridData);
button.setText("Message");
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        MessageBox messageBox = new MessageBox(shell);
        messageBox.setMessage("First name is " + bean.getFirst());
        messageBox.open();
    }
});
shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
display.dispose();
}

public static void main(String[] args) {
    JFaceBindingExample example = new JFaceBindingExample();
    example.run();
}

Data binding with Swing

A number of promising data binding frameworks have appeared on the scene for use with Java GUI APIs. The following are the main open source API initiatives for use with Swing applications.

JGoodies data binding API

The popular JGoodies data binding API has been available as an open source project on Java.net for the past few years. It was written by Karsten Lentzsch, who is also the author of the popular JGoodies FormLayout. This framework has been around the longest, and has been through many revisions and bug fixes to make it stable for production use.

Spring RCP

The Spring Rich Client Platform (RCP) project -- a subproject of the popular Spring Application Framework -- also contains a Swing data binding framework. Both Spring RCP and JGoodies are influenced by data binding designs from VisualWorks Smalltalk. Spring RCP has not yet featured a V1.0 release. That said, the data binding portion of the code base is stable and is used by many developers.

SwingLabs

The SwingLabs project on Java.net has included development on a Swing data binding framework for the past few years. However, effort on this project has recently been transferred to a new Sun Microsystems-sponsored data binding Java Specification Request (JSR).

JSR 295: Beans binding

This recently created JSR, lead by Sun’s Scott Violet with expert group members, including Ben Galbraith and Karsten Lentzsch, intends to provide a standard API for data binding for use in
desktop and server environments. JSR 295 will not be ready for some time, so it is not an option for those who need a solution today.

**Data binding with SWT**

There are two main open source data binding APIs for use with SWT.

**SWTBinding**

Jaysoft ported the popular JGoodies data binding API for use with SWT. The core classes are virtually the same as JGoodies. The Swing-specific models were replaced with ones suitable for SWT controls.

**JFace data binding**

Another recent newcomer to the Java data binding scene is the JFace data binding framework. A provisional version of the API is included in the Eclipse V3.2 release. Unlike the SWTBinding/JGoodies framework, JFace data binding was built from the ground up to work specifically with SWT and JFace.

**Advantages of data binding**

In addition to addressing the synchronization problem, there are some other advantages to using a data binding framework in your application. Because you are using the same synchronization code over and over, instead of creating your own, you have fewer bugs. Another major gain is application testability.

The popular Presentation Model (see Resources) advocates separating out the state and business logic of an application into a model layer that is separate from the GUI controls in the view. The state of the model is synchronized frequently with the view, as shown in Figure 2.

**Figure 2. Relationships using Presentation Model**
This type of design allows you to test all the business logic of your application without having to instantiate the view. An example of this is the enabling of certain controls on a form when a total is greater than 100. There is an enablement condition of "if total > 100". There is also an associated state based on the evaluation of this condition.

With a Presentation Model pattern, this state is set in a variable in the Presentation Model and synchronized with the view to modify control enablement. As a result, you can test the logic without accessing the GUI components in the view.

Accessing GUI components and mocking them is often difficult with SWT and Swing. You run all tests against the Presentation Model because it contains the conditional logic and a hold place for the state that changes as a result of executing it. The one catch to this whole pattern is when or how you synchronize data changes back and forth between the Presentation Model and the view. Before data binding, this was difficult. Now it's as easy as binding controls to fields in the Presentation Model or associated domain objects.

Disadvantages of data binding

There are some disadvantages to using a data binding framework in your application. At first, applications are more difficult to debug because the extra binding layer makes it harder to trace the flow of data between controls and domain objects. However, this becomes easier as you become familiar with the implementation specifics of the framework you're using.

Applications also may be more brittle during refactoring due to the use of strings to signify properties. Consider the code excerpt from Listing 5. The string "first" is used to tell the JFace Data Binding framework to bind to the getFirst() / setFirst() property. Refactoring getFirst() and setFirst() to getFirstName() and setFirstName() requires changing the string to "firstName". Current IDE refactoring tools wouldn't catch this change.

Listing 5. Areas refactoring won't catch

```java
class FormBean {
  private String first;

  public FormBean() {
    this.first = "Scott";
    this.last = "Delap";
    this.description = "Description";
  }

  public String getFirst() {
    return first;
  }

  public void setFirst(String first) {
    this.first = first;
  }
}
```
Conclusion

Regardless of whether you're developing in SWT or Swing, there is much to be gained from incorporating a data binding framework into your project. No one likes writing or maintaining boilerplate GUI to domain model synchronization code. I hope this introduction has demonstrated how Java data binding frameworks can provide a welcome relief from those activities. As an added benefit, they can also increase testability when combined with proper GUI design patterns.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)