Rapid Application Development With Data Binding and Object Persistence

Rakesh Vidyadharan
President and CEO
Sans Pareil Technologies, Inc.

2006-04-15

Abstract

This article discusses techniques for Rapid Application Development through the use of Data Binding and Object Persistence. A few frameworks such as JGoodies[1], Spring[2] exist that facilitate automatic data binding between Model and View components. In this article, I will demonstrate a simple approach for Rapid Application Development through automatic code generation of Model objects and persisting them using Object Persistence frameworks. This technique is by no means meant as a replacement for pre-existing frameworks, it is intended purely to illustrate the principles involved in Data Binding and Object Persistence. The techniques described in this article may be used to develop custom Rapid Application Development frameworks if no pre-existing framework meets your organisations requirements.

1 Introduction

Client application development in general revolve around applying the MVC design pattern or variations of it. Traditional application development revolved around developing the Model, Controller and View components, often by different members of a development team. Development tools have made it possible to generate Model components from an existing database (usually a Relational Database Management System), developing the View components using visual editors, and subsequently linking up the Model and View components through implementing stub classes generated by the development tools for handling modifications made to either the View or the Model.

Synchronising modifications applied between the View and Model is often the most repetitive and error prone task in the application development lifecycle. In this article, I present a simple framework for automatically generating the Model component from the View component and persisting the Model to a Data Store. The frame work also includes generic Controller components that facilitate binding the View and Model components together.

1.1 Rapid Application Development

Rapid Application Development has traditionally been associated with frameworks and technologies such as NextStep/OpenStep/Cocoa, Visual Basic, TCL/TK etc. Java for a long time lagged behind these technologies due to a lack of tools and frameworks. The situation has changed with the arrival of IDE's that offer easy to use user interface builders and object

[1] http://www.jgoodies.com/
[2] http://www.springframework.org/

and persistence modelling tools. Binding frameworks have also been developed that makes the task of creating applications even simpler. It is now fair to say that Java may be used for Rapid Application Development just as readily as the more popular technologies.

1.2 Data Binding

Data Binding frameworks work by configuring the bindings between the View and Model manually. The bindings are usually expressed using one or a combination of both techniques below:

1. Configuration files (usually XML).
2. Invocations to binding methods defined by the framework.

Some frameworks may also require you to implement the View and/or Controller components as derived classes from base classes provided by the framework, or implement interfaces defined by the framework. Once the bindings have been established, modifications affected to either the View or Model components would automatically be propagated to the other by the framework.

I have developed a very simple framework for the purpose of illustrating the concepts of Data Binding, using which I have developed the sample application (see section 2). In the simple framework I have chosen to make the process even simpler by delegating the binding actions to generic binding classes - sub-classes of `Updater` (see Figure 2(a)) and `PropertyChangeListener` (see Figure 2(b)). This approach reduces the amount of code that needs to be written (Java or XML).

The downside to this approach is that the binding framework is not as flexible as the more involved ones. There is a definite requirement that field names in the View and Model be identical.[3] This shortcoming can be addressed through more complex code in the Controller components, however that would just be moving code from one component into another.

2 Sample Application

I have developed a simple web-based form for collecting user information and storing to a database as a means to illustrate the techniques discussed in this article.[4] Figure 1(a) shows the simple View components as displayed in a web-browser. I have used the Echo2[5] Ajax based *event-driven* framework for developing the View.[6]

For Object Persistence I have chosen to use JDO[7] since it enables transparent storage to Object, Relational or XML Database Management Systems. For the purpose of this example I chose to use JPOX[8] as the JDO implementation to persist the data to a PostgreSQL database.

I will use a code generator to generate the Model object that represents the data necessary to render the View, as well as a **D**ata **A**ccess **O**bject design pattern for saving/fetching instances of the Model from the Data Store. Figure 1(b) shows the inheritance hierarchy for the code generator, as well as the `JDOMetaDataGenerator` support class that generates the JDO metadata file. I will use generic Controller objects to bind the View and Model components. The source code for the application may be downloaded from SPT[9].

[3] This can be a problem if you wish to hold all the UI Components for the View in a `Collection` of some kind for ease of UI Component generation.

[4] You can access the sample application at http://apps.sptci.com/echodemo/

[5] http://www.nextapp.com/platform/echo2/echo/

[6] I chose Echo2, since it would be very simple to rewrite the View and the framework to work with AWT/Swing as the programming interface is identical. It would even be possible to write a code generator that would convert the Echo2 View components into an AWT/Swing component. Sans Pareil Technologies, Inc. uses Echo2 for rapid prototyping of web applications and creating rich web based management interfaces.

[7] http://java.sun.com/products/jdo/

[8] http://www.jpox.org/

[9] http://www.sptci.com/products/articles/rad.jar

3 Development Process

The steps involved in developing an application using the Rapid Application Development process outlined in this article are as follows:

1. Develop the View class with all the UI Components declared in it (see section 3.1).
2. Compile the View class.
3. Generate the Model, Data Access Object design pattern and JDO metadata files using the code generator (see section 3.2).
4. Modify the Model and Data Access Object design pattern classes generated to meet any application specific requirements (see sections 3.3 and 3.4).
5. Modify the JDO metadata file to comply with JDO vendor requirements.
6. Add the initialisation code for the UI Components (see Listing 2).
   - 6.1. Add `JDOModelPropertyChangeListener` instances to the UI Components if so desired to trigger updates of the Model.
   - 6.2. Register `ViewPropertyChangeListener` instances with the Model if so desired to trigger updates of the View.
7. Add event handling code necessary for the View.
   - 7.1. Apply the appropriate `Updater` to event handling code.
   - 7.2. Apply `PropertyChangeListener` to the UI Components if so desired.
8. Assemble the application, and test the Data Binding and Object Persistence logic.

3.1 Developing the View

The View component was written by hand and not developed using any interface builder.[10] Listing 1 displays the various components used to build the View. This is sufficient to generate the Model using the code generator (see section 3.2).

Listing 1: Basic View Components

```
public class InputForm extends ContentPane
{
  private TextField textField;
  private TextArea textArea;
  private HashMap<String, CheckBox>
    checkBoxes;
  private ListBox listBox;
  private SelectField selectField;

  private Button first;
  private Button previous;
  private Button last;
  private Button next;
  private Button save;
  private Button delete;
  private Label index;

  public InputForm() {}
}
```

After generating the Model (see section 3.3) the View component can be enhanced to include the Model that was generated as well as the code necessary to properly initialise and layout the various components. Listing 2 displays the modifications to the source file[11]. Most of those methods do not strictly speaking belong to a View component, however I have chosen to combine Controller features (by implementing the `ActionListener` interface) into the View for the sake of simplicity.

Listing 2: Basic View Implementation

```
public class InputForm extends ContentPane
  implements ActionListener
{
  private InputFormModel model;
  private InputFormModelFactory dao =
    InputFormModelFactory.getInstance();

  public InputForm()
  {
    this( new InputFormModel() );
  }

  public InputForm( InputFormModel model )
```

[10] NextApp offers EchoStudio an Eclipse based development environment for easily developing the View.
[11] Listing omits the UI Component initialisation code.

```
  {
    this.model = model;
    initForm();
  }

  final void setModel(
      InputFormModel model )
  {
    this.model = model;
    ( new UIUpdater(
        this, model ) ).update();
  }

  public void actionPerformed(
      ActionEvent event )
  {
    if ( "save".equals(
          event.getActionCommand() ) )
    {
      doSave();
    }
    else if ( "delete".equals(
          event.getActionCommand() ) )
    {
      doDelete();
    }
    else if ( "previous".equals(
        event.getActionCommand() ) )
    {
      doNavigate( Integer.parseInt(
            index.getText() ) - 1 );
    }
  }

  private void doSave()
  {
    try
    {
      ( new JDOBeanUpdater( this, model )
        ).update();
      dao.save( model );
      setModel( new InputFormModel() );
      setIndexValue( dao.lastIndex() + 1 );
    }
    catch ( Throwable t )
    {
      ErrorPane pane = new ErrorPane(
          t.getMessage(),
          StringUtilities.stackTrace( t ) );
      add( pane );
    }
  }

  private void doDelete()
  {

    try
    {
      dao.delete( model );
      setModel( new InputFormModel() );
      setIndexValue( dao.lastIndex() + 1 );
    }
    catch ( Throwable t )
    {
      ErrorPane pane = new ErrorPane(
          t.getMessage(),
          StringUtilities.stackTrace( t ) );
      add( pane );
    }
  }

  private void doNavigate( int index )
  {
    InputFormModel data =
      dao.fetchByIndex( index );
    if ( data == null )
    {
      data = new InputFormModel();
    }

    setModel( data );
    setIndexValue( index );
  }
}
```

3.2 Code Generation

The repetitive work involved in developing the Model is handled through a code generator. The source code for the Model and **Data Access Object design pattern** can be generated using the `JDOModelGenerator` (see Figure 1(b)). The generator *reflects* upon the class file for the View and develops the following three files:

1. Source file for the Model.
   1.1. The class will have the same name as the View class, with a "Model" appended to it.
   1.2. The class will belong to the same `package` as the View.
   1.3. The fields in the Model will match the names assigned to the UI Components in the View.
   1.4. The fields will be assigned types based upon the type of UI Component it represents. Complex UI Components such as `ListBox`, `SelectField` are modelled as `List<ListItem>` fields.[12] Other UI Components such as `CheckBox`, `RadioButton`

[12] `ListItem` if a general purpose JavaBean for representing entries in a UI Component that models a list.

are modelled as `Map<String, Boolean>` fields.

1.5. All the `mutator methods` in the Model fire a `PropertyChangeEvent` to any registered `PrpoertyChangeListeners`.

2. Source file for the Data Access Object design pattern.

2.1. The class will have the same name as the Model class with a "Factory" appended to it.

2.2. Standard methods for saving an object to the Data Store and deleting an object from the Data Store are generated for the Data Access Object design pattern.

2.3. The Data Access Object design pattern will be responsible for binding the Model with the Data Store. The data store may be any ODBMS, RDBMS, or XDBMS for which a JDO implementation is available.[13]

3. JDO metadata file.

3.1. The generated file only lists the fields in the Model and support classes.

3.2. This will usually need additional configuration depending upon your JDO vendor and Data Store.

A more robust and production ready code generator could split the Model into a pure data Model and a decorator for that Model object. This would enable direct mapping between the View and the decorator while leaving flexibility in designing and implementing the Model. This is an approach I am currently involved in implementing to enhance the Rapid Application Development framework we use at Sans Pareil Technologies, Inc.. With that additional level of indirection the Rapid Application Development framework can be used to generate most of the source code necessary to implement production grade applications.

3.3 Developing the Model

You may need to modify the Model generated by the code generator (see section 3.2) to meet your application requirements (see Listing 3). Listing 3 displays the modified default constructor in the generated Model. No other modifications were necessary for the sample application.

You will probably need to decompose the Model into different Model sub-components depending upon the complexity of the data displayed in the View. This can easily be handled through the Data Access Object design pattern controller class that was generated.

Listing 3: Model Modifications

```
public class InputFormModel
  implements InstanceCallbacks, Cloneable,
    Comparable<InputFormModel>, Serializable
{
  public InputFormModel()
  {
    checkBoxes =
      new TreeMap<String, Boolean>();
    listBox = new ArrayList<ListItem>();
    selectField = new ArrayList<ListItem>();
    for ( int i = 0; i < 3; ++i )
    {
      checkBoxes.put(
          "Check " + i, false );
      listBox.add( new ListItem(
            "List Item " + i ) );
      selectField.add( new ListItem(
            "Select " + i ) );
    }
  }
}
```

3.4 Developing the Controller

The following Controller components take care of most of the work required to successfully bind the View to the Model and the Model to the data store:

1. ActionListener. The Listener for *actions* triggered by UI Components in the View. 2).

[13] For true Rapid Application Development and performance an ODBMS is recommended.
[14] The full source code for the application includes commented out code for enabling `PropertyChangeListeners`.

2. PropertyChangeListener. Handlers for updating View or Model components through PropertyChangeEvents. These are not currently implemented in the sample application, although the controllers exist for handling PropertyChangeEvents.[14]

3. Data Access Object design pattern for binding the Model to the Data Store. A standard Data Access Object design pattern is generated by the code generator.

4. JDOModelUpdater. A Controller used to update fields in JDO Model components with the data in corresponding fields in the View (see the doSave method in Listing 2).

5. ViewUpdater. A Controller used to update UI Component fields in the View with data in the corresponding fields in the Model (see the setModel method in Listing 2).

For the sample application, the InputForm class also acts as a Controller. The doSave, doDelete, doNavigate and setModel methods (see Listing 2) illustrate the binding code necessary to keep the View, Model and Data Store synchronised with each other.

The binding of the View and Model is achieved through concrete sub-classes of Updater. The update method will update the View or Model as appropriate from its associated component. Invoking the update method from the Controller is the only action that needs to be performed to synchronise the View and Model with each other.

The two lines shown below replaces the usual lines of code that would need to be written to update the Model with the data from the View and persist the Model to the Data Store. The JDOModelUpdater handles the task of updating the fields of the Model with the data in the corresponding fields in the View.

```
(new JDOModelUpdater(this, model)).update();
dao.save( model );
```

Similarly the following two lines of code replaces the usual lines of code that would be necessary to fetch data from the Data Store to the Model and update the View with fresh data[15]. The ViewUpdater handles the task of updating the UI Components of the View with data from their corresponding fields in the Model.

```
model = dao.fetchByIndex( index );
( new ViewUpdater( this, model ) ).update();
```

The Data Access Object design pattern should be updated with additional methods necessary to support application requirements. Listing 4 shows the additional methods that were added to the Data Access Object design pattern to meet the requirements of the sample application.

Listing 4: DAO Modifications

```
public class InputFormModelFactory
  implements Serializable
{
  public final int lastIndex()
  {
    return ( fetchAll().size() - 1 );
  }

  public final InputFormModel fetchByIndex(
      int index )
  {
    ArrayList<InputFormModel> list =
      (ArrayList<InputFormModel>)fetchAll();
    InputFormModel model = null;
    if ( index >= 0 && list.size() > index )
    {
      model = list.get( index );
    }

    return model;
  }
}
```

3.5 Binding Framework

The binding framework revolves around instances of the Updater and PropertyChangeListener classes. The instances handle the task of updating correspondingly named fields in the View or Model with the data in their associated component.

[15] This step is usually not necessary in traditional web-application development, since the binding is automatically applied at page generation time. Echo2 however requires this additional step similar to desktop application development frameworks. This additional capability is also useful if you wish to implement server-side push to update web pages.

Figure 2(a) shows the inheritance hierarchy for the `Updater` classes used to bind the View and Model together. Appropriate instances of `Updater` may be added to the appropriate Controller components to update the View or Model from the other as required.

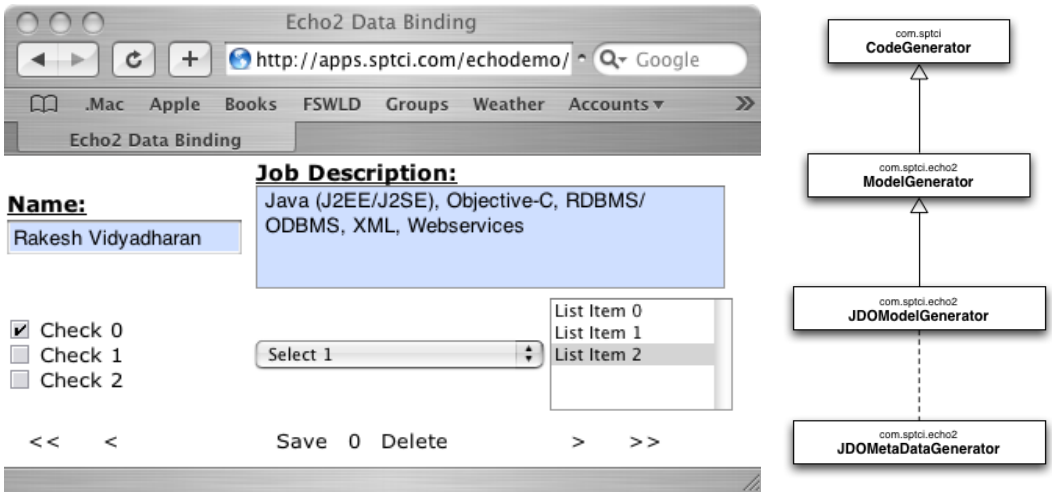
Figure 2(b) shows the inheritance hierarchy for the `PropertyChangeListener`. Appropriate instances of `PropertyChangeListener` may be added to the View components to trigger updates of the Model. Similarly the appropriate `PropertyChangeListener` may be registered with the Model to trigger updates of the View.

4 Conclusion

In this article I have outlined a general purpose Data Binding framework that in combination with the choice of an appropriate Persistence Framework can facilitate Rapid Application Development. As illustrated in the article, a framework that handles the time consuming repetitive tasks of application development can help a developer concentrate on the important task at hand - developing the application.

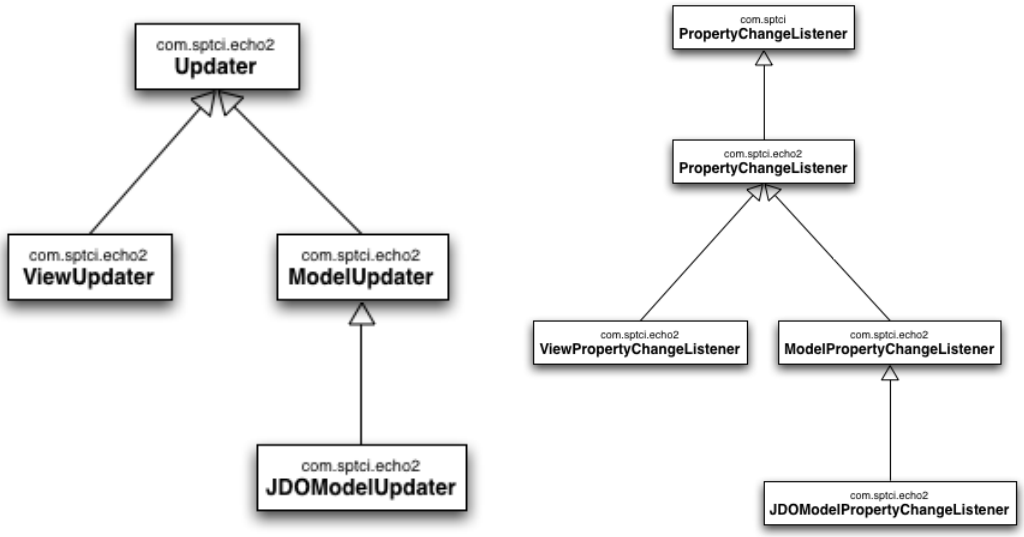
There are a number of choices when it comes to selecting binding frameworks for application development. Choosing the appropriate framework can help drastically reduce application development time and lead to higher quality products. Popular frameworks are actively developed and heavily tested not just by the vendors, but by all the other organisation using those frameworks.

It is also possible to create a custom framework as illustrated in the sample application with fairly little effort if your organisation has specific requirements that are not handled by the popular frameworks. Sans Pareil Technologies, Inc. is involved in developing a more robust and production grade framework along the principles outlined in this article to facilitate its application development requirements.



(a) User Input Form (b) CodeGenerator

Figure 1: User Input Form and Code Generator



(a) Updater (b) PropertyChangeListener

Figure 2: Binding Class Inheritance Hierarchy