

# Presentation Model Architectures with JGoodies Binding

Felix Leipold

January 2009

## Abstract

This tutorial explains how to implement an application using the presentation model architecture, as a way to structure non-trivial applications. Thereby it also tries to shed light on what the presentation model architecture is and how data binding facilitates its implementation. The examples are using Java Swing as the UI-layer and Karsten Lentzsch's JGoodies binding library.

## 1 Overview

### 1.1 Presentation Model Architecture

The term presentation model pattern or architecture is unfortunately neither very well defined nor understood [6]. So I will give a short description of my understanding. The presentation model approach aims at making applications with user interfaces more testable and also more understandable.

The core idea is to have a very thin, effectively stateless, view layer, that merely mirrors a model. The model is not allowed to call explicitly into the view layer. The fact that powerful user interfaces often visualise state, that is not strictly part of the domain, calls for the introduction of a new layer of decorators around the domain that keeps presentational state (e.g. the currently selected filter criteria in a mailbox view).

While the idea, that a clear separation between model and user interface helps understanding and testing applications, is widely recognised, there is still a lot of ad-hoc coding in the UI parts of "real-life" applications. The difficulty in implementing this pattern is the necessity to synchronize the presentation model and the view layer. This requires copying state back and forth. Data binding frameworks help solving this problem in a declarative way, so the programmer doesn't have to worry about when to copy state around.

### 1.2 Example Code

This tutorial introduces Karsten Lentzsch's JGoodies binding framework, which aims at solving the data binding problem for Java Swing applications. It is in turn a port of the application framework in Visual Works Smalltalk. While we

use Java and Swing as examples, the underlying patterns are applicable to all rich user interface technologies.

Throughout this tutorial we are going to use a simple contact manager application as an example. I will first describe the outlines of this application. Then I'll introduce the concepts that JGoodies binding[2] provides to help us build the whole application.

In section 4 we are going to put it all together. The sourcecode for the examples including all required libraries is available for download [4]. Most code snippets that are presented in this document are located in the `CodeSnippedTest` class. Please also note that most of the quoted code is actually tweaked and shortened for readability, so when in doubt browse the code from the download archive.

While this tutorial does not focus on actually building UIs, the examples are relying heavily on the JGoodies Forms library, which is well described in [3].

## 2 Introducing the Example Application



Figure 1: Example Application

The user interface (UI) of the contact manager is shown in figure 1. The listbox in the top part of the UI allows to choose a contact, while the text fields at the bottom allow to edit the selected contacts details. The two buttons add a new contact or remove the currently selected contact respectively. Even though the functionality seems rather trivial, there is a separation between the actual domain model and purely presentational state. While the list of contacts are part of the domain, the currently selected contact is purely presentational state.

The actual domain consists of only two classes: `Contact` representing a single contact and `ContactManager` representing a collection of contacts, that

has methods to create or remove contacts. Listing 1 shows excerpts from `Contact`. It extends the class `Model`, which is part of JGoodies to support the implementation of observability according to the JavaBeans Specification [1]. The property `firstName` is such a so-called bound property. The setter has to call the `firePropertyChange` method on `Model` to notify listeners.

```
public class Contact extends Model {

    private String firstName = "";
    private String lastName = "";
    private boolean fictional;

    public Contact() {}

    public Contact(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        String oldValue = this.firstName;
        this.firstName = firstName;
        firePropertyChange("firstName", oldValue, firstName);
    }

    ...
}
```

Listing 1: `Contact.java`

The `ContactManager` class is even lighter, as it is just using an existing `List` implementation with observation support. It is shown in Listing 2.

```
public class ContactManager {
    private ObservableList<Contact> contacts;

    public ContactManager() {
        contacts = new BeanListModel<Contact>();
    }

    public Contact createNewContact() {
        Contact contact = new Contact();
        contacts.add(contact);
        return contact;
    }

    public ListModel getContacts() {
        return contacts;
    }

    public void removeContact(Contact contact) {
        contacts.remove(contact);
    }
}
```

```
}  
}
```

Listing 2: ContactManager.java

## 3 Data-Binding

### 3.1 ValueModels

The basic problem in a layered approach is to get the data from the domain (or presentation) to the UI-elements and back again. A very basic albeit common solution to the problem looks like that:

```
TextField firstName = new TextField();  
firstName.setText(contact.getFirstName());  
  
//and at some point:  
contact.setFirstName(firstName.getText());
```

One of the issues here is the fact that you never know when to copy data back. The obvious approach here is to register an event listener with the UI and update the model whenever the UI changes.

```
TextField firstName = new TextField();  
firstName.setText(contact.getFirstName());  
firstName.addTextListener(new TextListener() {  
    public void textValueChanged(TextEvent e) {  
        contact.setFirstName(firstName.getText());  
    }  
});
```

Listing 3: Event driven Ssynchronisation

That is more elegant, but still a lot of code to say a simple thing: connect this model property with that UI-element. Data binding, as supported by JGoodies, does exactly this. It allows you to *bind* UI-elements straight to properties of your domain or presentation model. Therefore it uses the observer pattern. A central concept that JGoodies introduces to facilitate observation is the `ValueModel` interface shown in Listing 4. It allows to access a value and observe its changes. You might think of it as a kind of pointer.

```
public interface ValueModel {  
    Object getValue();  
    void setValue(Object o);  
    void addValueChangeListener(PropertyChangeListener l);  
    void removeValueChangeListener(PropertyChangeListener l);  
}
```

Listing 4: ValueModel interface

Given a `ValueModel` instance representing a property of the domain model the code in 3 can be generalised. A naïve implementation of such a generic bind function might look like this:

```

void bind(final TextField textField, final ValueModel value) {
    textField.setText((String)value.getValue());
    textField.addTextListener(new TextListener() {
        public void textValueChanged(TextEvent e) {
            value.setValue(textField.getText());
        }
    });
}

```

To keep the implementation simple this binding just synchronizes from the UI to the model. JGoodies provides a lot of two way bindings for standard UI elements in its Bindings class. This is however rarely used directly. Instead JGoodies provides the class BasicComponentFactory a factory that can create various UI elements and binds them to a given ValueModel, e.g. JTextField createTextField(ValueModel model). The value model pattern has actually been around and described for some time [5].

The simplest implementation of the ValueModel is the ValueHolder. It is best described using a simple test case, which uses Mockito to verify the event notification:

```

ValueHolder holder = new ValueHolder();
holder.setValue("Old");

PropertyChangeListener listener = mock(PCL.class);
holder.addPropertyChangeListener(listener);

holder.setValue("New");

verify(listener)
    .propertyChange(
        eq(new PropertyChangeEvent(holder, "value", "Old", "New"));

```

## 3.2 Adapting Properties

In the last section no explanation was given, where the ValueModel instance comes from. In most cases we want to bind properties of our domain. For this purpose JGoodies provides the PropertyAdapter class. The workings are demonstrated in the following test:

```

Contact contact = new Contact();
ValueModel firstNameModel =
    new PropertyAdapter(contact, "firstName");

contact.setFirstName("Felix");
assertEquals("Felix", firstNameModel.getValue());

firstNameModel.setValue("Holden");
assertEquals("Holden", contact.getFirstNameModel());

```

A property adapter projects a single an object to a single property. This is illustrated in figure 2. If the property has listener support, the property adapter will also fire a change event whenever the underlying property changes.

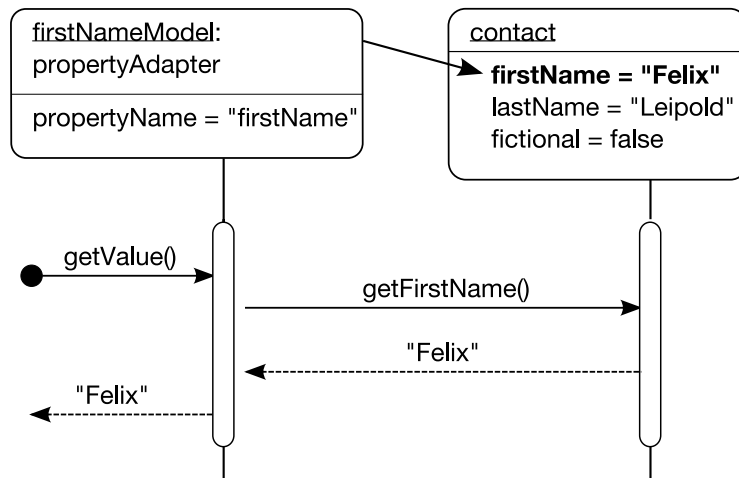


Figure 2: PropertyAdapter Example

ValueModels are not statically typed. PropertyAdapters are however bound to the type of the underlying property. Hence trying to set the value of the first-NameModel in the example above to an integer will yield an exception.

### 3.3 Converting ValueModels

Domain models are not always using strings (well, might be wishful thinking on my side). The ValueModel approach actually allows for an easy conversion between the domain and the UI layer, by decorating ValueModels with converters. The following example shows how to adapt an integer property to a string value, which in turn can be bound to a text field:

```

LineItem lineItem = new LineItem();

ValueModel quantityModel =
    new PropertyAdapter(lineItem, "quantity");

ValueModel quantityStringValue =
    new IntegerAsStringConverter(quantityModel);

lineItem.setQuantity(14);

assertEquals(14, quantityModel.getValue());
assertEquals("14", quantityStringValue.getValue());

quantityStringValue.setValue("17");

assertEquals(17, lineItem.getQuantity());

class IntegerAsStringConverter extends AbstractConverter {
    public IntegerAsStringConverter(ValueModel valueModel) {
        super(valueModel);
    }
}
  
```

```

    }

    @Override
    public Object convertFromSubject(Object subjectValue) {
        return subjectValue.toString();
    }

    public void setValue(Object newValue) {
        subject.setValue(Integer.parseInt((String) newValue));
    }
}

```

Listing 5: Adapting and converting a property

This decoration process is illustrated in figure 3. The implementation of the converter given here doesn't contain any error handling. The class `ConverterFactory` provides a number of robust converter implementations that cover most use cases.

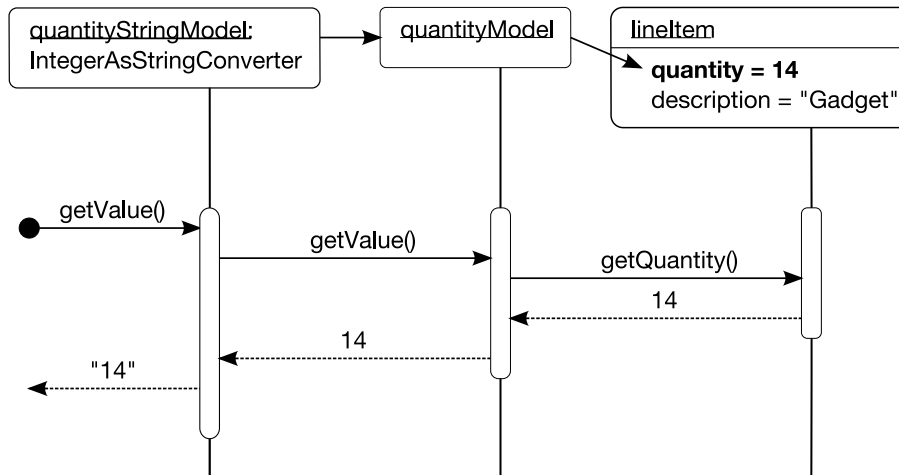


Figure 3: Decorating a ValueModel with a Converter

### 3.4 Binding Properties and ValueModels

The class `PropertyConnector` supports binding properties and value models or two properties. This helps implementing new custom bindings. A short example looks like this:

```

Contact felix = new Contact("Felix", "Leipold");
Contact holden = new Contact("Holden", "Caulfield");

PropertyConnector.connect(felix, "firstName", holden, "lastName")
    .updateProperty2();

assertEquals("Felix", holden.getLastName());

```

```

felix.setFirstName("Holden");

assertEquals("Holden", holden.getLastName());

holden.setLastName("Felix");

assertEquals("Felix", felix.getFirstName());

```

The static factory method `connect` creates a new connector that connects the first name of `felix` with the last name of `holden`. As these two have different values we'll have to decide which one wins. This is what the `updateProperty2()` call is about. The connector will subsequently listen to all changes on either side and update the other side accordingly. There is also a factory method to bind a value model to a property called `connectAndUpdate`.

### 3.5 More Indirection

In most cases there is more than one property per domain object. The class `PresentationModel` is among other things providing a nice way to get to `ValueModels` on multiple domain properties. This helps keeping the code more compact:

```

Contact contact = new Contact("Felix", "Leipold");
PresentationModel contactModel = new PresentationModel(contact);

ValueModel firstName = contactModel.getModel("firstName");
ValueModel lastName = contactModel.getModel("lastName");

assertEquals("Felix", firstName.getValue());
assertEquals("Leipold", lastName.getValue());

```

So far we did only access properties of a single domain object. However presentation models also allow for swapping the underlying domain object. This comes very handy, when an existing UI is to be retargeted to a new model object. This is for example the case with the detail panel of our example app.

Instead of passing in our domain object to the presentation model, we pass in a value model. This value model is acting as a bean channel. The following test case illustrates how to use a bean channel:

```

Contact felix = new Contact("Felix", "Leipold");
Contact holden = new Contact("Holden", "Caulfield");

ValueModel contactChannel = new ValueHolder();
((ValueHolder)contactChannel).setIdentityCheckEnabled(true);

contactChannel.setValue(felix);

PresentationModel contactModel =
    new PresentationModel(contactChannel);

ValueModel firstName = contactModel.getModel("firstName");
ValueModel lastName = contactModel.getModel("lastName");

assertEquals("Felix", firstName.getValue());

```

```

assertEquals("Leipold", lastName.getValue());

contactChannel.setValue(holden);

assertEquals("Holden", firstName.getValue());
assertEquals("Caulfield", lastName.getValue());

```

### 3.6 Modelling Selections in Lists

While Swing does provide models to represent the state of a listbox these are difficult to use. The class `SelectionInList` is filling this gap. It implements both `ListModel`, the Swing interface for observable lists, as well as `ValueModel`. The `ListModel` aspect represents the elements from which the user is allowed to choose, while the `ValueModel` represents the selected object (and not the selection index as Swing would). The example below illustrates how to use a `SelectionInList` with a `PropertyAdapter`.

```

import static jgoodies...BasicComponentFactory.createList;
import static jgoodies...BasicComponentFactory.createTextField;

//Domain
Contact holden = new Contact("Holden");
Contact felix = new Contact("Felix");
ListModel contacts = new BeanListModel(felix, holden);

//Presentation Model
SelectionInList contactSelection =
    new SelectionInList(contacts);

ValueModel firstNameModel =
    new PropertyAdapter(contactSelection, "firstName");

contactSelection.setValue(holden);

//UI
JList listbox = createList(contactSelection);
JTextField nameField = createTextField(firstNameModel, false);

DefaultFormBuilder builder = UIUtils.singleColumnFormBuilder();

builder.append(new JScrollPane(listbox), 3);
builder.append("First name", nameField);

builder.setDefaultDialogBorder();

UIUtils.showInFrame(builder.getPanel());

```

Listing 6: `SelectionInListExample.java`

The example uses a `FormBuilder` from the `jgoodies forms` library [3]. Though all of the code lives in a single method, the objects fall clearly into the domain, presentation model, and view layers respectively. Another thing to notice is that all that this code does is creating and configuring objects. Apart from `Contact` no custom classes are being used. Moreover the initial selection in

the listbox gets set, before the UI-element even exists. A screenshot is provided in figure 4.

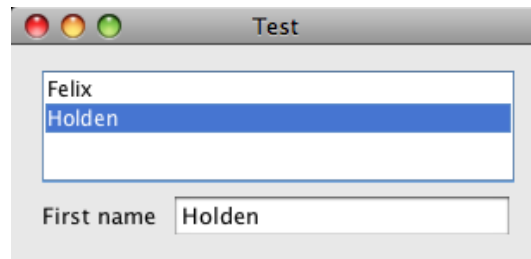


Figure 4: SelectionInListExample screenshot

### 3.7 Actions and Models for Components

Value models are very useful for binding state of UI-elements to corresponding (presentation) model objects. However more often than not the UI also has to trigger actions in the domain or the presentation model, e.g. when the user is pushing buttons or choosing menu entries. The model associated with these UI elements is an *Action*. Fortunately the Swing API provides this interface and supports the binding to buttons and menu entries. Actions have an *actionPerformed* method, that acts as a callback. It also has several properties that influence the appearance of the action in the UI, such as its title and enablement state. In order to deactivate the remove button, when no element is selected, we do not need to fiddle with the view directly. Just disabling the action will also disable any bound UI-element.

```
Action add = new AbstractAction("Add") {
    public void actionPerformed(ActionEvent e) {
        add();
    }
};

// Still without view involvement:
add.setEnabled(false);

// Building the view:
JButton addButton = new JButton(add);
```

Similar to buttons other UI-elements do also have state/ configuration that goes beyond the immediate value, such as enabled or editable state or the color. Normally these properties are left unchanged. However if they become important for the application (make it into acceptance criteria), they should be bound to the presentation layer.

For the most common properties like enable and editable JGoodies provides a support class called *ComponentValueModel*, that extends *ValueModel*, but moreover provides properties that are automatically bound to the UI-elements, created by the *BasicComponentFactory*.

```

Contact contact = new Contact("Felix", "Leipold");
PresentationModel contactModel =
    new PresentationModel(contact);
ComponentValueModel firstName =
contactModel.getComponentModel("firstName");

JTextField nameField =
BasicComponentFactory.createTextField(firstName);

assertEquals("Felix", nameField.getText());

firstName.setEditable(false);
assertFalse(nameField.isEditable());

```

As the example above shows this allows to move the enabling and disabling of UI-elements into the presentation layer.

## 4 Back to the Example

### 4.1 Presentation Layer

Using the building blocks presented in the last section allows us to describe the (implied) interface of the presentation model for our contact manager as follows:

```

public class ContactManagerPresentation {
    public ContactManagerPresentation(ContactManager domain) {
    public SelectionInList getContactSelection();
    public Action getAdd();
    public Action getRemove();
    public ComponentValueModel getLastName();
    public ComponentValueModel getFirstName();
    public ComponentValueModel getFictional();
}

```

The components of the presentation model correspond to the elements of the UI. However they are a simplification in that they represent only these aspects that are relevant from an application point of view, while it abstracts from detail like colours, positions, or fonts.

The important point is that it allows us to test drive the application regardless of whether we have got a UI or not. To ensure that the add action creates and selects a user, we can now write the following test:

```

@Test
public void shouldCreateAndSelectUserOnAdd() {
    assertNull(presentation.getContactSelection().getValue());

    presentation.getAdd().actionPerformed(EVENT);

    assertEquals(1, model.getContacts().getSize());
    assertEquals(model.getContacts().getElementAt(0),
        presentation.getContactSelection().getValue());
}

```

The actual implementation of the presentation model is written in a very declarative style. The building blocks provided are just being wired up. By using value models we end up with very little moving parts. Most work is done in the constructor. After construction only the action handlers are being called. The source of the constructor and the action handlers is given in listing ??:

```

public ContactManagerPresentation(ContactManager domain) {
    this.domain = domain;

    contactSelection =
        new SelectionInList(this.domain.getContacts());

    hasSelection = new IsNotNullConverter(contactSelection);

    PresentationModel contactAdapter =
        new PresentationModel(contactSelection);

    firstName = contactAdapter.getComponentModel("firstName");
    lastName = contactAdapter.getComponentModel("lastName");
    fictional = contactAdapter.getComponentModel("fictional");

    add = new AbstractAction("Add") {
        public void actionPerformed(ActionEvent e) {
            add();
        }
    };

    remove = new AbstractAction("Remove") {
        public void actionPerformed(ActionEvent e) {
            remove();
        }
    };

    PropertyConnector.connectAndUpdate(hasSelection, remove,
        "enabled");
}

private void remove() {
    if (contactSelection.getValue() == null) {
        return;
    }
    domain.removeContact((Contact) contactSelection.getValue());
}

private void add() {
    Contact newContact = domain.createNewContact();
    contactSelection.setValue(newContact);
}

```

Listing 7: ContactManagerPresentation.java

Figure 5 shows the objects involved in the presentation and domain layers. The objects with a thick green box are actually mapping nicely to UI-elements. The only moving part is the selection of the selection in list. It should also become obvious, that the presentation layer is a projection of the domain to the user interface.

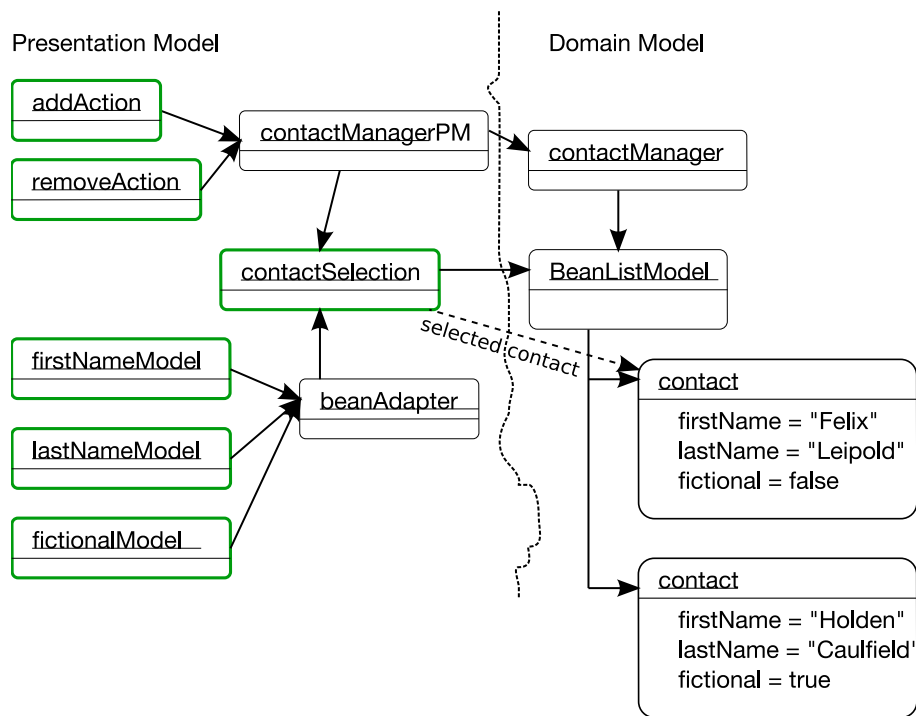


Figure 5: Objects in presentation and domain layer

## 4.2 View Layer

The actual view consists of a single class that acts as a builder. Note how we don't need to extend JFrame:

```

import static jgoodies...BasicComponentFactory.createTextField;
import static jgoodies...BasicComponentFactory.createList;
import static jgoodies...BasicComponentFactory.createCheckBox;

public class ContactManagerUI {

    ...

    public ContactManagerUI(ContactMgrPresentation presenter) {
        contactList = createList(presenter.getContactSelection());
        firstName = createTextField(presenter.getFirstName());
        lastName = createTextField(presenter.getLastName());
        fictional = createCheckBox(presenter.getFictional());
    }
  
```

```

        add = new JButton(presenter.getAdd());
        remove = new JButton(presenter.getRemove());

        buildPanel();
    }

    private void buildPanel() {
        DefaultFormBuilder b = UIUtils.singleColumnFormBuilder();

        b.appendSeparator("Contacts");
        b.append(new JScrollPane(contactList), 3);
        b.append(ButtonBarFactory.buildAddRemoveBar(add, remove)
            , 3);

        b.appendSeparator("Detail");
        b.append("First name", firstName);
        b.append("Last name", lastName);
        b.append("Type", fictional);

        b.setDefaultDialogBorder();

        panel = b.getPanel();
    }
    ...
}

```

Listing 8: ContactManagerUI.java

The constructor instantiates and binds the UI-elements. The build panel method defines the layout by adding them to a panel. It is making use of a programtic form builder. Using programatic builders helps ensuring a uniform appearance and is a very good option for non-trivial applications.

However there is also a number of visual UI-Builders supporting JGoodies binding.

### 4.3 All together now...

To kick off the whole application we need to instantiate all three layers (as the main method of ContactManagerUI does):

```

ContactManagerUI ui =
    new ContactManagerUI(
        new ContactManagerPresentation(
            new ContactManager()));

UIUtils.showInFrame(ui.getPanel(), "Contact Manager");

```

The nested constructor calls once again illustrate the dependencies.

## 5 Philosophy

- No relevant state kept in UI components.
- As little explicit event handlers as possible.
  - Use binding to keep properties in sync.
  - Use value models to manage changing target objects.
- Never call into the view, even if it is behind an interface.
- The behaviour of the presentation model has to be independent from the number of registered views.

## 6 Advanced topics

### 6.1 Using the Action Reflector

### 6.2 Validation

### 6.3 Opening new Windows

### 6.4 Test Visualisation

## 7 TODO

- Contrast with MVC approach.
- Show how to use with SWT to demonstrate flexibility.
- How does this relate to the jface way of adapters.
- structuring ui into reusable components

## References

- [1] JavaBeans Specification 1.01, Sun Microsystems, 1997 <http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html>
- [2] JGoodies Binding <https://binding.dev.java.net/>
- [3] The JGoodies Forms Framework, Karsten Lentzsch, 2004 <http://www.jgoodies.com/articles/forms.pdf>
- [4] Sourcecode for the examples: <http://wuetender-junger-mann.de/downloads/jgoodies-tutorial.zip>
- [5] Understanding and Using ValueModels, Bobby Woolf, 1994, <http://c2.com/ppr/vmodels.html>
- [6] Presentation Model, Martin Fowler, 2004, <http://martinfowler.com/eaDev/PresentationModel.html>