How to build a bean for Servoy – A step by step tutorial brought to you by Servoy Stuff

PART 6

AA. Creating an interface for our SliderBean

In the last part of this tutorial, we saw what it takes to make a bean Web client compatible, using a simple Swing JTextField for the Smart client and a Wicket Panel and TextField for the Web client.
Now it's time to go back to our ServoySlider bean and prepare it to be Web client compatible, so in this tutorial we will adapt what we've learned and do the same thing for our Slider, so that the Smart client will use our Swing bean while the Web client will use a simple input TextField (inside a Panel) for now.
Once we will have this done, we will look in the next part at how we can use JavaScript in the browser to emulate our slider with a DHTML component that we will use.

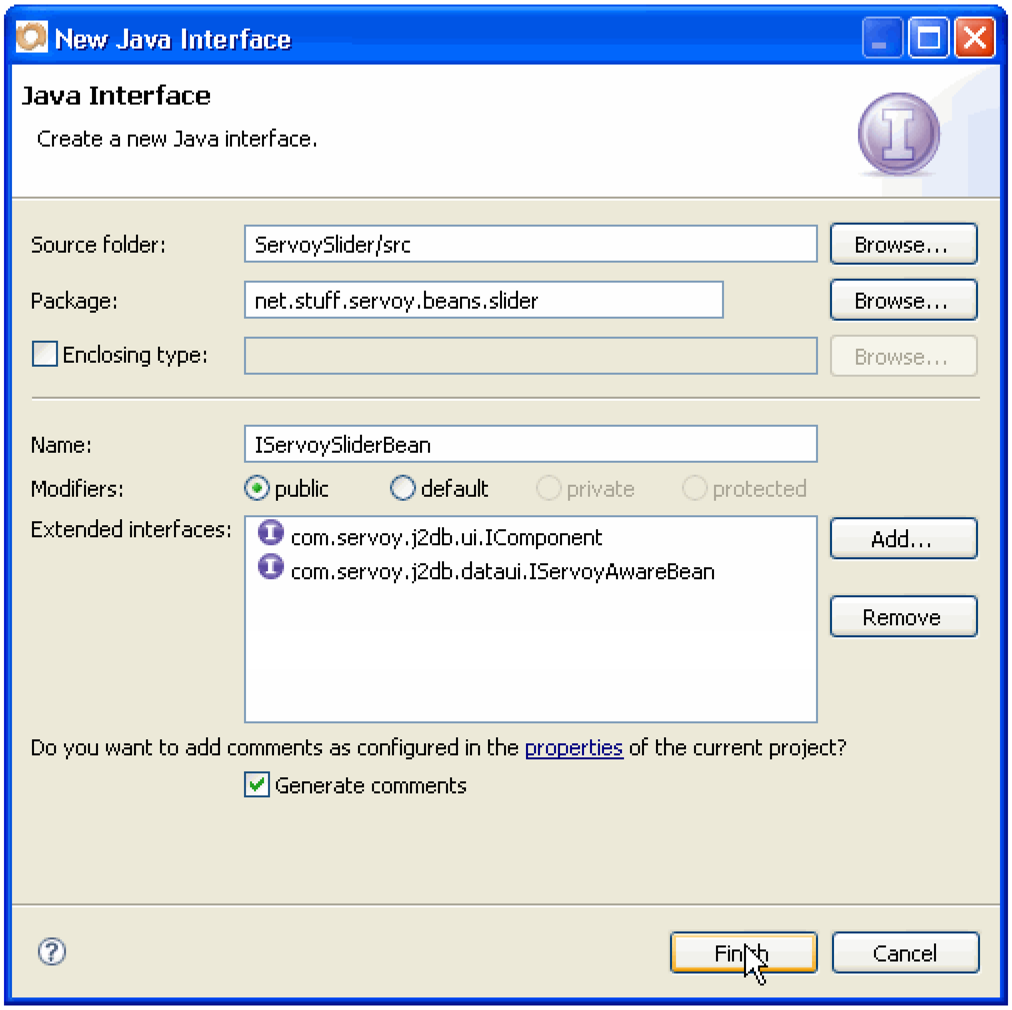
So if you remember from the last tutorial, a Web compatible Java bean has in fact 2 implementations, and even if there are some similarities in these implementations, they are based on 2 very different objects, coming from 2 very different libraries: Swing and Wicket. The thing that ties them together is the IServoyBeanFactory that is responsible to return to Servoy the type of object it is expecting, depending on the type of client (Smart or Web).

But to be able to manipulate these 2 different types of objects easily (passing them parameters coming from the Properties View as set by the user) you remember that we created an interface where we put all the necessary methods.

You remember also that this interface was extending the IComponent interface (because that's the kind of objects the IServoyBeanFactory getBeanInstance() method is returning to Servoy), and that it was also extending the IServoyAwareBean interface (because we want our bean to be an input bean).

So let's create this interface now in our project, and name it IServoySliderBean, so that its final signature will be:

```
public interface IServoySliderBean extends IComponent, IServoyAwareBean {
```



Source folder: ServoySlider/src
Package: net.stuff.servoy.beans.slider
Enclosing type:
Name: IServoySliderBean
Modifiers: [✓ public] default private protected
Extended interfaces:
com.servoy.j2db.ui.IComponent
com.servoy.j2db.dataui.IServoyAwareBean
Do you want to add comments as configured in the properties of the current project?
[✓ Generate comments]

By extending these 2 interfaces, we know that we will need to implement 29 methods already. And in fact we are going to add a few methods of our own too, because if we look at the properties that are used in our bean (look at the code of the ServoySliderBeanInfo, that's easier), we can see that some of them are not covered by any of these interfaces – some of them are, like background, foreground, font, etc., and the others are:

- focusable,
- orientation,
- paintLabels,
- paintTicks,
- paintTrack,
- snapToTicks,
- dataProviderID,
- precision,
- usingDecimals,
- inverted,
- majorTicks,
- minorTicks,
- maximumValue,
- minimumValue,
- toolTipMessage,
- currentValue.

So that's 16 properties for which we will need getter and setter methods, meaning 32 more methods, so we will have a whooping total of 61 methods to implement in our new Wicket bean (the Swing bean already implements all of that so there will be no need to change it!).
But relax! Most of them are accessors, and we will look at the others more closely...
Anyway, the methods related to our properties are:

```
public void setFocusable(boolean focusable);
public boolean isFocusable();

public void setOrientation(int orientation);
public int getOrientation();

public void setPaintLabels(boolean paintLabels);
public boolean getPaintLabels();

public void setPaintTicks(boolean paintTicks);
public boolean getPaintTicks();

public void setPaintTrack(boolean paintTrack);
public boolean getPaintTrack();

public void setSnapToTicks(boolean snapToTicks);
public boolean getSnapToTicks();

public void setDataProviderID(String dataProviderID);
public String getDataProviderID();

public void setPrecision(int precision);
public int getPrecision();

public void setUsingDecimals(boolean usingDecimals);
public boolean isUsingDecimals();

public void setInverted(boolean inverted);
public boolean getInverted();

public void setMajorTicks(int majorTicks);
public int getMajorTicks();

public void setMinorTicks(int minorTicks);
public int getMinorTicks();

public void setMaximumValue(int maximumValue);
public int getMaximumValue();

public void setMinimumValue(int minimumValue);
public int getMinimumValue();

public void setToolTipMessage(String toolTipMessage);
public String getToolTipMessage();

public void setCurrentValue(int currentValue);
public int getCurrentValue();
```

So that's our IServoySliderBean interface all done, now let see to implement it.

BB. First steps in the Swing bean

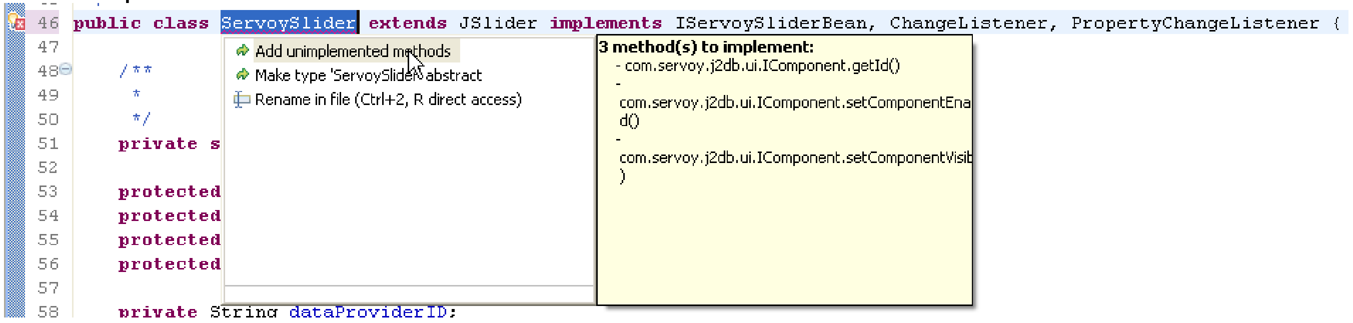
The first thing we can do now is to make our Swing bean implement this interface, and this is easy! Right now our class has this signature:

```
public class ServoySlider extends JSlider implements IServoyAwareBean,
ChangeListener, IScriptObject, PropertyChangeListener {
```

Let's get rid of the IServoyAwareBean interface in the signature and replace it with the new IServoySliderBean interface, so that our signature will be:

```
public class ServoySlider extends JSlider implements IServoySliderBean,
ChangeListener, IScriptObject, PropertyChangeListener {
```

But of course Eclipse is complaining that this interface is not entirely covered by this implementation, so let's "quick fix" it:



```
public class ServoySlider extends JSlider implements IServoySliderBean, ChangeListener, PropertyChangeListener {

    /**
     *
     */
    private s

    protected
    protected
    protected
    protected

    private String dataProviderID;
```

Add unimplemented methods
Make type 'ServoySlider' abstract
Rename in file (Ctrl+2, R direct access)

3 method(s) to implement:
- com.servoy.j2db.ui.IComponent.getId()
- com.servoy.j2db.ui.IComponent.setComponentEnabled()
- com.servoy.j2db.ui.IComponent.setComponentVisible()

There's only 3 methods left to implement and they are really easy:

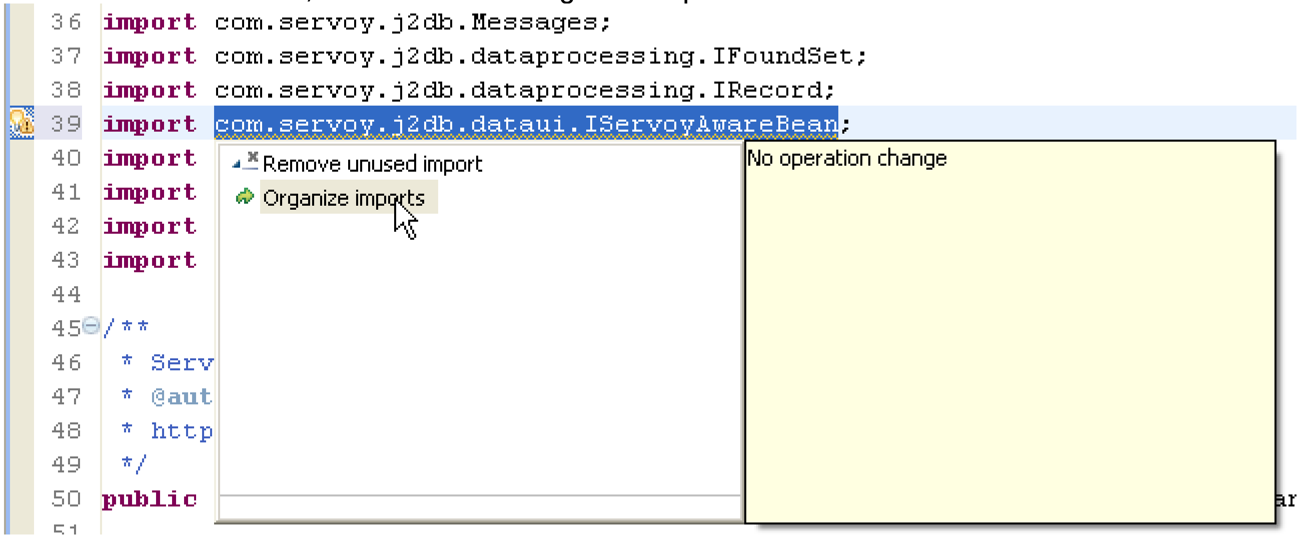
```
        /* (non-Javadoc)
         * @see com.servoy.j2db.ui.IComponent#getId()
         */
        public String getId() {
              return null;
        }

        /* (non-Javadoc)
         * @see com.servoy.j2db.ui.IComponent#setComponentEnabled(boolean)
         */
        public void setComponentEnabled(boolean enabled) {
              setEnabled(enabled);
        }

        /* (non-Javadoc)
         * @see com.servoy.j2db.ui.IComponent#setComponentVisible(boolean)
         */
        public void setComponentVisible(boolean visible) {
              setVisible(visible);
        }
```

You know that getId() is only used by Wicket components, so we can safely return null, and the 2 others are just alias methods for the setters of enabled and visible properties of our bean.

So Eclipse seems a lot happier now, except that there is a warning sign at the top of the class in the “import” section. Basically, the interface that was referenced before is not needed anymore, so you can do like me, and get rid of the imports, using another “quick fix”, “Remove unused import” (which will do that one at the time, or even better “Organize imports”:



```
import com.servoy.j2db.Messages;
import com.servoy.j2db.dataprocessing.IFoundSet;
import com.servoy.j2db.dataprocessing.IRecord;
import com.servoy.j2db.dataui.IServoyAwareBean;
import
import
import
import

/**
 * Serv
 * @aut
 * http
 */
public
```

I always prefer to have clean imports, and more importantly clean classes with no warnings, because if these (related to unused imports) are not problematic at all, they could potentially hide more important warnings that you will prefer to know about!

Now that we have a clean class, what do we need to do?
Remember from the last tutorial that we extracted what could be used as utilities methods inside a utility class with static methods? Let’s do that too!

Create a “ServoyUtils” class (you can copy the one from the TestBean project if you want.
I noticed from last time that one of the 2 methods we did could have been written better (yes, there’s always room for improvement ;-), by breaking its logic into 2 methods, and in fact we are going to use the two separately, so our ServoyUtils implementation will be:

```
/**
 * Test a String
 * @return true if the passed String is not empty
 */
public static boolean isNotEmpty(String s) {
      return (s != null && s.trim().length() > 0);
}

/**
 * Test the existence of inited foundset and record
 */
public static boolean isExisting(IFoundSet foundset, IRecord record) {
      return (foundset != null && foundset.getSize() > 0 && record != null);
}
```

```
      /**
       * Test method for the presence of the dataprovider in the record,
       * could be enhanced with search for related foundset
       * @param record the currentRecord
       * @param dataProviderID the dataProviderID
       * @return true id the dataProvider exists in the current FoundSet
       */
      public static boolean isContainedInFoundset(IRecord record,
            String dataProviderID) {
            if (record != null && isNotEmpty(dataProviderID)) {
                  IFoundSet foundset = record.getParentFoundSet();
                  if (isExisting(foundset, record)) {
                        boolean isContained = false;
                        String[] providers =
                  foundset.getDataProviderNames(IFoundSet.COLUMNS);
                        if (providers != null) {
                              for (int i = 0; i < providers.length; i++) {
                                    if (dataProviderID.equals(providers[i])) {
                                          isContained = true;
                                          break;
                                    }
                              }
                        }
                        if (isContained && foundset.getSelectedIndex() > -1) {
                              return true;
                        }
                  }
            }
            return false;
      }
```

So let’s use these methods in our ServoySlider (remember that the whole idea is to centralize as much as possible our logic so that there is as little as possible duplicated code: the DRY principle!).

The first place that we can use our utility methods is in the updateSlider() method, let’s replace the 2 first tests:

```
if (currentFoundset != null
      && currentFoundset.getSize() > 0
            && currentRecord != null) {
      if (hasDataProvider()&& isContainedInFoundset(getDataProviderID())) {
```

by calls to our ServoyUtils static methods:

```
if (ServoyUtils.isExisting(currentFoundset, currentRecord)) {
      if (ServoyUtils.isContainedInFoundset(currentRecord,getDataProviderID())) {
```

Then in our stateChanged() method, we can do the same, replacing the 2 first tests:

```
if (!ignoreUpdate &&
      (currentFoundset != null
            && currentFoundset.getSize() > 0
            && currentRecord != null)) {

      if (hasDataProvider() && isContainedInFoundset(getDataProviderID())) {
```

by calls to our ServoyUtils static methods:

```
if (!ignoreUpdate && ServoyUtils.isExisting(currentFoundset, currentRecord)) {
      if (ServoyUtils.isContainedInFoundset(currentRecord,getDataProviderID())) {
```

Then we can also change our hasDataProvider() method to call our ServoyUtils class:

```
	private boolean hasDataProvider() {
		return ServoyUtils.isNotEmpty(dataProviderID);
	}
```

And we can get rid of the private boolean isContainedInFoundset() method of the ServoySlider class, that is not used anymore (and generates a warning BTW).

We are done with our Swing class, now let’s look into the Wicket implementation...

CC. The ServoyWicketSlider

We are going to do the simplest implementation in Wicket for our Slider: a text input field where the value will be editable. In the next part we will implement a DHTML component to emulate our Swing slider, and it will be based on an input text field anyway, but for now there is already enough to do so we will concentrate on the Java implementation aspects...

You remember that to implement an input field, you need to put it inside a Panel (otherwise Servoy’s Wicket will complain that the tag is not right!). You also remember that a Panel needs some html markup to work, so let’s add an html file, name it “ServoyWicketSlider.html” and put it inside our “net.stuff.servoy.beans.slider” package.

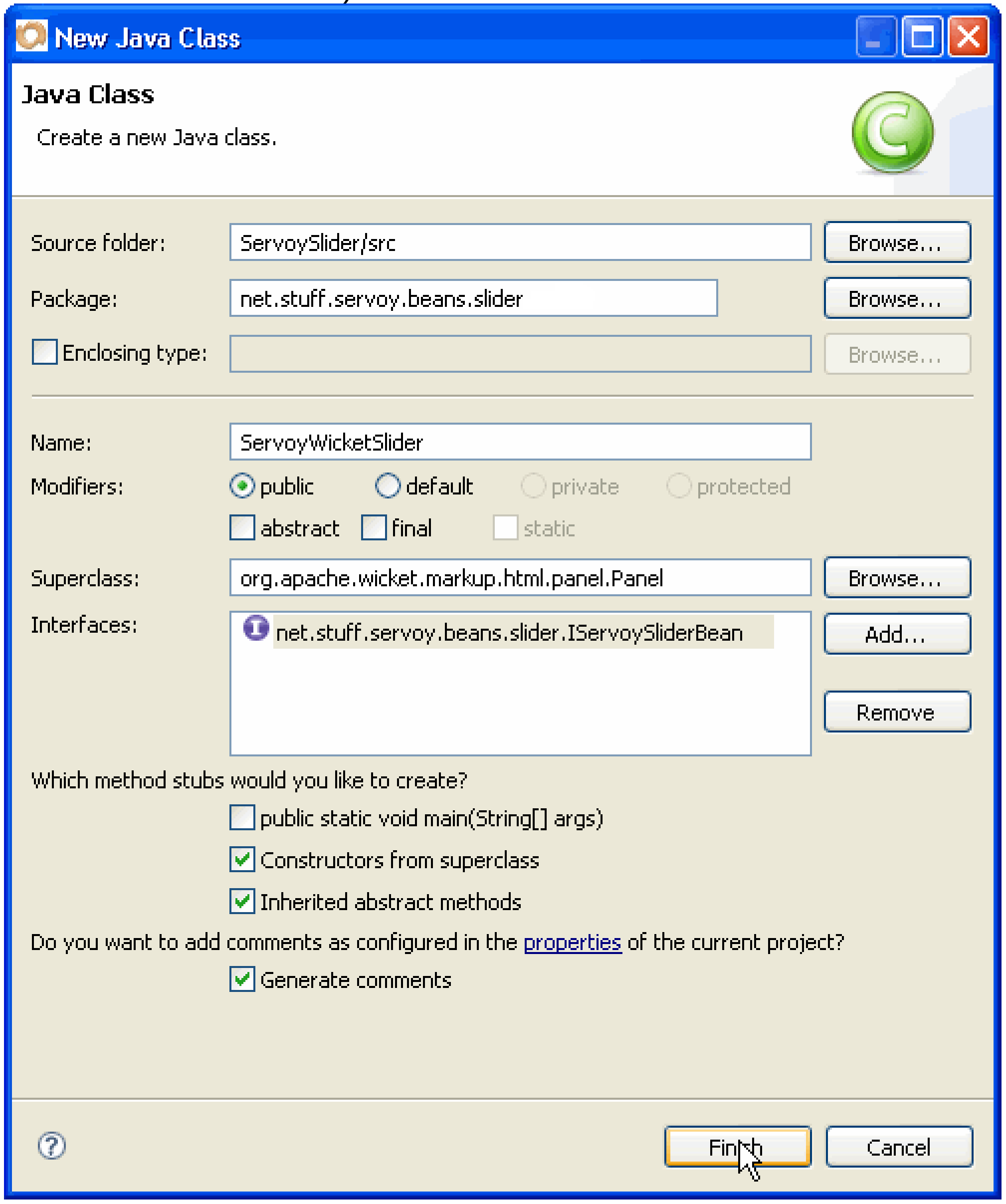
Now open it (Open with > Text editor), and put that html markup in it:

```
<wicket:panel>
	<table style='height: 100%;width: 100%;margin: 0px;padding: 0px;border-collapse: collapse;table-layout: fixed;overflow: hidden;' >
		<tr style='height: 100%;width: 100%;margin: 0px;padding: 0px' >
			<td style='height: 100%;width: 100%;margin: 0px;padding: 0px' >
				<input wicket:id="slider-input" type="text" style="border:none; background-color: transparent; height: 100%; width: 100%; margin: 0px; padding: 0px;"/>
			</td>
		</tr>
	</table>
</wicket:panel>
```

That’s basically the same markup as our last time WicketTestBean, a **wicket:panel** tag defining the component’s markup, and inside it a table for easy layout, and then an input type text with a **wicket:id** that we will use later in our class to bind our TextField component to this markup.
We used “**slider-input**” for the ID. Note it down! We will use it in a minute.

Of course the spaces here are only used for readability, but the less there is the better, so you can get rid of them in your final html file.

Right, it's time to create our class, let's call it ServoyWicketSlider, with a superclass of Panel (org.apache.wicket.markup.html.panel.Panel), and let it implement the IServoySliderBean interface we created earlier (also don't forget to check the "Constructors from superclass" because Panel doesn't have a default constructor) :



Java Class

Create a new Java class.

Source folder: ServoySlider/src

Package: net.stuff.servoy.beans.slider

[ ] Enclosing type:

Name: ServoyWicketSlider

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: org.apache.wicket.markup.html.panel.Panel

Interfaces: net.stuff.servoy.beans.slider.IServoySliderBean

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[✓] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

[✓] Generate comments

Wow, that's our 61 method stubs created in one go + 2 constructors! 63 TODO tasks in our tasks view. It's not that bad, really, because we know what to do, right? We did it already for the WicketTestBean!

And there are also some methods and properties that don't make sense at all in an input of type text, those directly related to the Slider like the paintTicks, paintLabels, minimum and maximum, etc... These will not have any possible translation in this version of the bean, so we will leave them blank for now (we will do something useful in our next part where we will use a real DHTML slider) and concentrate on the only useful property: the currentValue.

But the currentValue property of our slider is an int and it should really be a double, while a TextField is really only dealing with Text values, so we will use a double numberValue and we'll use a special constructor for our property model...

So let's add a property for that:

```
      private double numberValue;
```

And generate the getter/setter for it:

```
      public double getNumberValue() {
            return this.numberValue;
      }
      public void setNumberValue(double numberValue) {
            this.numberValue = numberValue;
      }
```

There are a few values that make sense for our current text implementation, so we will set properties for them too, and they are:

```
      private Dimension size;
      private Border border;
      private Point location;
      private Color background;
      private Color foreground;
      private Font font;
      private String toolTipText;
      private String name;
      private boolean opaque;
      protected Cursor cursor;
```

And there are also a few that we know we will need, to deal with the IServoyAwareBean implementation, and these are:

```
      protected IFoundSet currentFoundset;
      protected IRecord currentRecord;
      protected boolean validationEnabled = true;
      private String dataProviderID;
      protected double previousValue = -1;
```

Now all we have to do is to hook these properties to the stub methods which are already here. Exactly how we did it for our TestBean. But we know that we will have to use an essential class to get our changes updating the markup, and that's a ChangesRecorder, so we add (and construct) it too:

```
public ChangesRecorder jsChangeRecorder = new ChangesRecorder(new Insets(2,2,2,2),
new Insets(0,0,0,0));
```

Let's modify the constructor, like we did for our TestBean:

```
public ServoyWicketSlider(String id) {
      super(id);

      setNumberValue(0);

      add(StyleAttributeModifierModel.INSTANCE);

      add(TooltipAttributeModifier.INSTANCE);

      add(new SimpleAttributeModifier("class", getId()+" field"));

      setVersioned(false);
      setOutputMarkupId(true);
```

```
		TextField field = new TextField("slider-input",
			new PropertyModel(this, "numberValue"), Double.class);

		add(field);

	}
```

We call our superclass constructor's first, then we set our default numberValue then we add the two StyleAttributeModifierModel and ToolTipAttributeModifierModel to deal with styles and tooltip changes, we setup the class attribute of our markup div to be the id + "field" (to be styled as a field by Servoy). We set the versioning of this class to false (no dealing with the back button involved), but we want our markup to contain the id, and then finally we add our TextField.

Note that we use a special constructor for our TextField, we give it's ID (coming from the markup in out html file), a PropertyModel, based on our numberValue (a double value!) and because this property is not a String, we precise its base class (Double.class). This will ask Wicket to perfom String conversions on our Double and Number conversion back on the text value typed by the user.

The next constructor of our ServoyWicketSlider class we don't need (the one with an ID and a Model), so we can get rid of it.

Last time also, you must remember that we used 2 interfaces on our TestBean to tell Servoy to track for style changes in our bean, and to help him place our bean on a form, and they were the IStylePropertyChanges and the ISupportWebBounds respectively.

So let's add these 2 interfaces to our class signature which will now be:

```
public class ServoyWicketSlider extends Panel implements IServoySliderBean,
IStylePropertyChanges, ISupportWebBounds {
```

And of course, a quick fix will add 6 more methods to our very empty class.
We can deal with them straight away:

```
	public Properties getChanges() {
		return jsChangeRecorder.getChanges();
	}
	public boolean isChanged() {
		return jsChangeRecorder.isChanged();
	}
	public void setChanged() {
		jsChangeRecorder.setChanged();
	}
	public void setChanges(Properties paramProperties) {
		jsChangeRecorder.setChanges(paramProperties);
	}
	public void setRendered() {
		jsChangeRecorder.setRendered();
	}
	public Rectangle getWebBounds() {
		Dimension localDimension = jsChangeRecorder.calculateWebSize(
			size.width, size.height, border, null, 0, null);
		return new Rectangle(location, localDimension);
	}
```

Now, let's go back to the rest of the methods, still empty, and see which one we can implement now with the help of our properties, and the jsChangesRecorder class.

```
	public String getDataProviderID() {
		return dataProviderID;
	}
	public String getToolTipMessage() {
		return toolTipText;
	}
	public void setDataProviderID(String dataProviderID) {
		this.dataProviderID = dataProviderID;
	}
	public void setToolTipMessage(String toolTipMessage) {
		setToolTipText(Messages.getStringIfPrefix(toolTipMessage));
	}

	public Color getBackground() {
		return background;
	}
	public Border getBorder() {
		return border;
	}
	public Font getFont() {
		return font;
	}
	public Color getForeground() {
		return foreground;
	}
	public Point getLocation() {
		return location;
	}
	public String getName() {
		return name;
	}
	public Dimension getSize() {
		return size;
	}
	public String getToolTipText() {
		return toolTipText;
	}
	public boolean isOpaque() {
		return opaque;
	}
```

And then you would remember the use of the PersistHelper class, and the ComponentFactoryHelper class along with the ChangesRecorder instance to update our styles:

```
	public void setBackground(Color c) {
		this.background = c;
		if (c != null) {
			jsChangeRecorder.setBgcolor(PersistHelper.createColorString(c));
		}
	}
```

```
public void setBorder(Border border) {
      this.border = border;
      if (this.border != null) {
            ComponentFactoryHelper.createBorderCSSProperties(
                  ComponentFactoryHelper.createBorderString(this.border),
                        jsChangeRecorder.getChanges());
      }
}

public void setComponentEnabled(boolean enabled) {
      setEnabled(enabled);
      setChanged();
}

public void setComponentVisible(boolean visible) {
      setVisible(visible);
      jsChangeRecorder.setVisible(visible);
      setChanged();
}

public void setCursor(Cursor cursor) {
      this.cursor = cursor;
}

public void setFont(Font font) {
      this.font = font;
      if (font != null) {
            jsChangeRecorder.setFont(PersistHelper.createFontString(font));
      }
}

public void setForeground(Color c) {
      this.foreground = c;
      if (c != null) {
            jsChangeRecorder.setFgcolor(PersistHelper.createColorString(c));
      }
}

public void setLocation(Point location) {
      this.location = location;
      if (this.location != null) {
            jsChangeRecorder.setLocation(location.x, location.y);
      }
}

public void setName(String name) {
      this.name = name;
}
public void setOpaque(boolean b) {
      this.opaque = b;
      jsChangeRecorder.setTransparent(!b);
}

public void setSize(Dimension size) {
      this.size = size;
      if (this.size != null) {
            jsChangeRecorder.setSize(size.width, size.height,
                  this.border, null, 0);
      }
}
```

```
	public void setToolTipText(String toolTipText) {
		this.toolTipText = toolTipText;
	}
```

All of this is classic and we saw it in the previous part of the tutorial.

The initialize method we can ignore:
```
	public void initialize(IClientPluginAccess paramIClientPluginAccess) {
		// ignore
	}
```
The isReadonly will get a basic implementation:
```
	public boolean isReadOnly() {
		return !isEnabled();
	}
```
We will use the setValidationEnabled to update our property :
```
	public void setValidationEnabled(boolean validationEnabled) {
		this.validationEnabled = validationEnabled;
	}
```
And of course we need to return true to the stopUIEditing() method:
```
	public boolean stopUIEditing(boolean paramBoolean) {
		return true;
	}
```

So we are left with the 2 really important methods of our bean, the setSelectedRecord() (called by Servoy) and the setNumberValue() (called by Wicket on update of the TextField). For our setSelectedRecord() method, let's do this:

```
	public void setSelectedRecord(IRecord record) {
		if (record != null) {
			this.currentFoundset = record.getParentFoundSet();
			this.currentRecord = record;
			updateSlider();
		}
	}
```

Which leads us to add an updateSlider() method, that will be like so:

```
private void updateSlider() {
	if (ServoyUtils.isContainedInFoundset(currentRecord,getDataProviderID())) {
		Object o = currentRecord.getValue(getDataProviderID());
		this.numberValue = (Double)o;
		this.previousValue = getNumberValue();
		setChanged();
	}
}
```

Note that we use the same construct (using the ServoyUtils static method) to test our dataProvider.

And the setNumberValue() method will be:

```
public void setNumberValue(double numberValue) {
      this.numberValue = numberValue;
      if (previousValue != numberValue) {
            if (ServoyUtils.isContainedInFoundset(currentRecord,
                  getDataProviderID())) {
                  if (currentRecord.startEditing()) {
                        currentRecord.setValue(getDataProviderID(), numberValue);
                        setChanged();
                  }
            }
      }
}
```

That's it for the methods we can implement, we will leave the rest of them as TODO tasks for the next tutorial when the properties they deal with will actually make sense (on a real Slider!).

From the last time, you must also remember that we subclassed 2 methods from our Wicket Panel ancestors, and we are going to do it here too, these are:

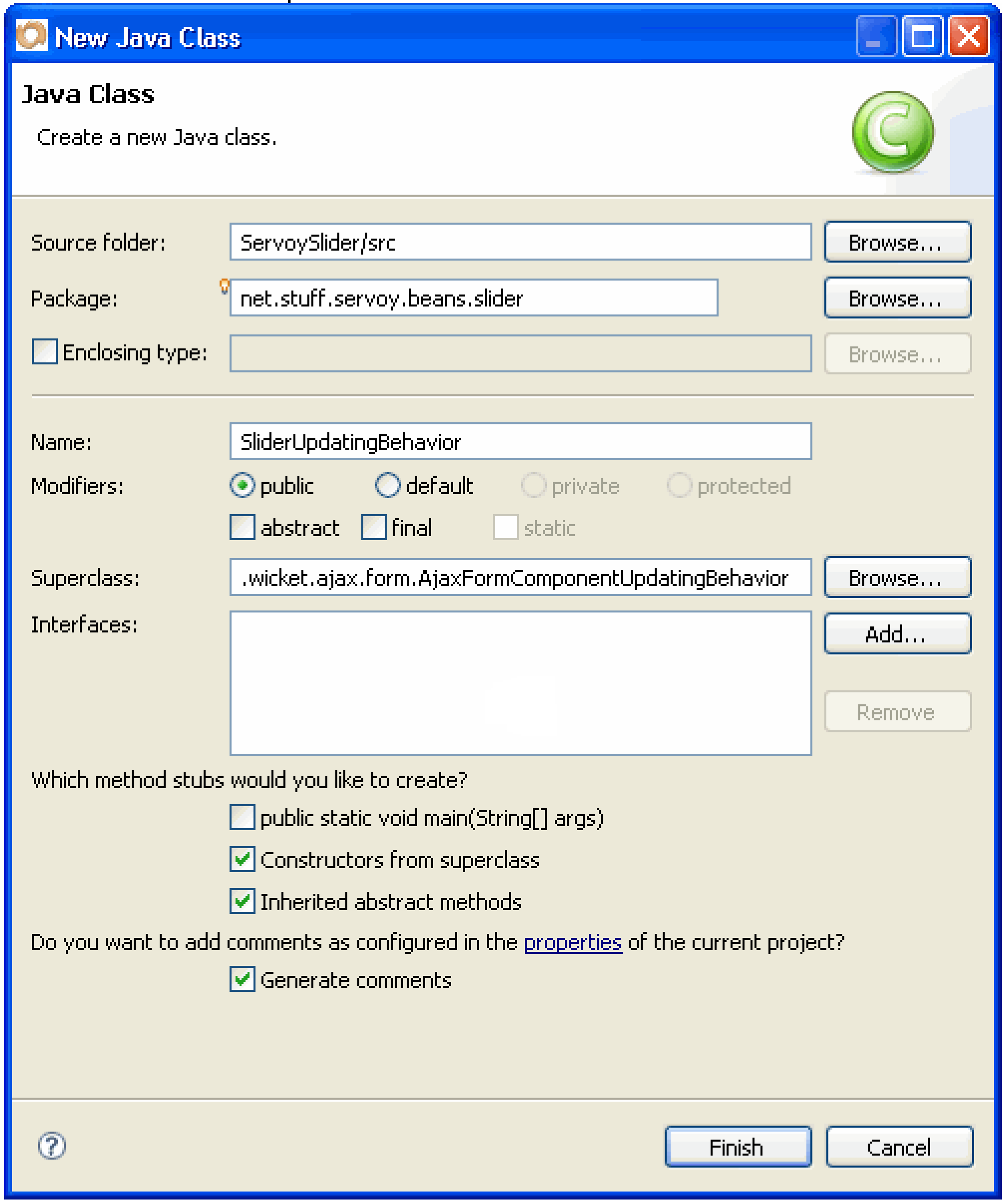
```
      @Override
      public String getMarkupId() {
            return getId();
      }

      @Override
      protected void onRender(MarkupStream markupStream) {
            super.onRender(markupStream);
            jsChangeRecorder.setRendered();
      }
```

The first one to make sure that the ID is used as the markup ID of our class (it will be used by Servoy's Ajax method calls), and the second to make sure we updated the state of the ChangesRecorder class when the class has finished rendering.

And finally, finally, you also remember that we added a TextChangesUpdatingBehavior (a subclass of AjaxFormComponentUpdatingBehavior) to our class for 2 browser's events: "onchange" and "onblur"... This was to trigger update to Servoy whenever the value changed in our field.

We'll do the same here, first create a class, let's call it SliderUpdatingBehavior this time, with a superclass of "org.apache.wicket.ajax.form.AjaxFormComponentUpdatingBehavior" and create it with "Constructors from superclass" checked:



Source folder: ServoySlider/src
Package: net.stuff.servoy.beans.slider
[ ] Enclosing type:
Name: SliderUpdatingBehavior
Modifiers: [✓ public] default private protected
[ ] abstract [ ] final [ ] static
Superclass: .wicket.ajax.form.AjaxFormComponentUpdatingBehavior
Interfaces:
Which method stubs would you like to create?
[ ] public static void main(String[] args)
[✓] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments as configured in the properties of the current project?
[✓] Generate comments

Our class signature is:

```
public class SliderUpdatingBehavior extends AjaxFormComponentUpdatingBehavior {
```

Let's quick fix to add the required long required by the Serialize interface:

```
	private static final long serialVersionUID = 1L;
```

Then let's add 2 constant to define the browser's events we want to deal with:

```
	public final static String DATA_CHANGE = "onchange";
	public final static String LOST_FOCUS = "onblur";
```

And finally a variable to hold a pointer to our parent:

```
	protected ServoyWicketSlider parent;
```

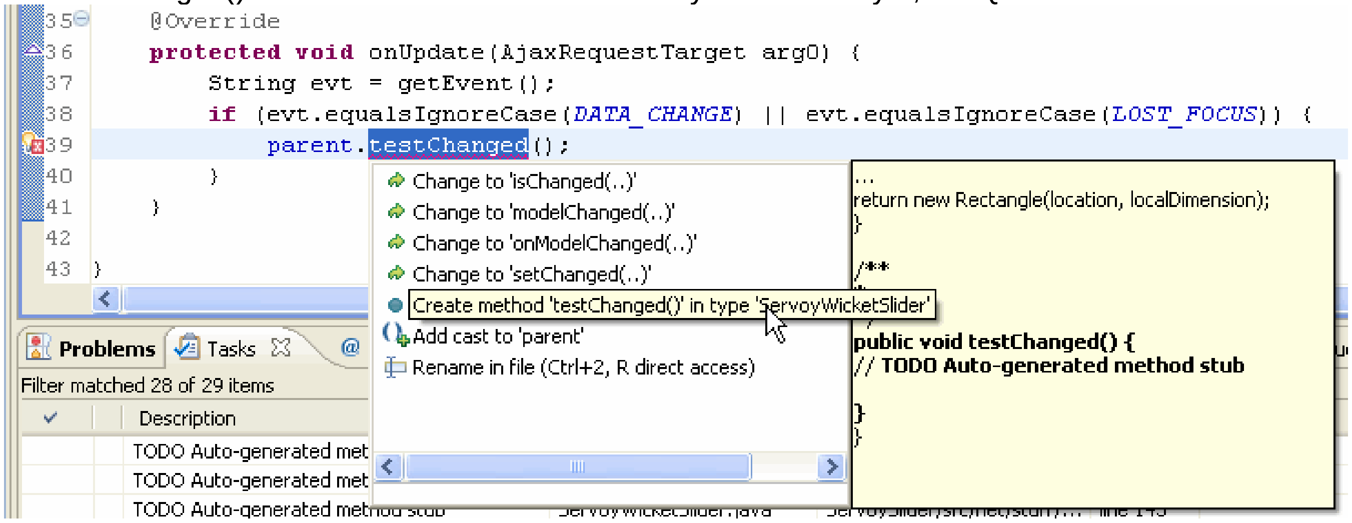
So that we can alter our constructor like so:

```
	public SliderUpdatingBehavior(String event, ServoyWicketSlider parent) {
		super(event);
		this.parent = parent;
	}
```

And code our unique onUpdate method like so:

```
@Override
protected void onUpdate(AjaxRequestTarget arg0) {
	String evt = getEvent();
	if (evt.equalsIgnoreCase(DATA_CHANGE) || evt.equalsIgnoreCase(LOST_FOCUS)) {
		parent.testChanged();
	}
}
```

The testChanged() method doesn't exist in the ServoyWicketSlider yet, so “Quick Fix” will add it for us.



```
    @Override
    protected void onUpdate(AjaxRequestTarget arg0) {
        String evt = getEvent();
        if (evt.equalsIgnoreCase(DATA_CHANGE) || evt.equalsIgnoreCase(LOST_FOCUS)) {
            parent.testChanged();
        }
    }
}
```

Change to 'isChanged(..)'
Change to 'modelChanged(..)'
Change to 'onModelChanged(..)'
Change to 'setChanged(..)'
Create method 'testChanged()' in type 'ServoyWicketSlider'
Add cast to 'parent'
Rename in file (Ctrl+2, R direct access)
Problems
Tasks
Filter matched 28 of 29 items
Description
TODO Auto-generated met
TODO Auto-generated met
TODO Auto-generated method stub

```
...
return new Rectangle(location, localDimension);
}

/**

public void testChanged() {
// TODO Auto-generated method stub

}
}
```

This testChanged() method in our ServoyWicketSlider class will be a simple test and a call to setChanged() if needed:

```
	/**
	 * Will trigger an update if the text has changed
	 */
	public void testChanged() {
		if (previousValue != getNumberValue()) {
			setChanged();
		}
	}
```

Now that we have our SliderUpdatingBehavior, it's only a matter of adding it to the constructor of our class, which will now be:

```
public ServoyWicketSlider(String id) {

      super(id);

      setNumberValue(0);

      add(StyleAttributeModifierModel.INSTANCE);
      add(TooltipAttributeModifier.INSTANCE);

      add(new SimpleAttributeModifier("class", getId()+" field"));

      setVersioned(false);
      setOutputMarkupId(true);

      TextField field = new TextField("slider-input",
            new PropertyModel(this, "numberValue"), Double.class
      );
      add(field);

      field.add(new SliderUpdatingBehavior(
            SliderUpdatingBehavior.DATA_CHANGE, this)
      );
      field.add(new SliderUpdatingBehavior(
            SliderUpdatingBehavior.LOST_FOCUS, this)
      );

}
```

That’s it for our implementation of a Wicket Panel and TextField as the equivalent of our Swing Slider.

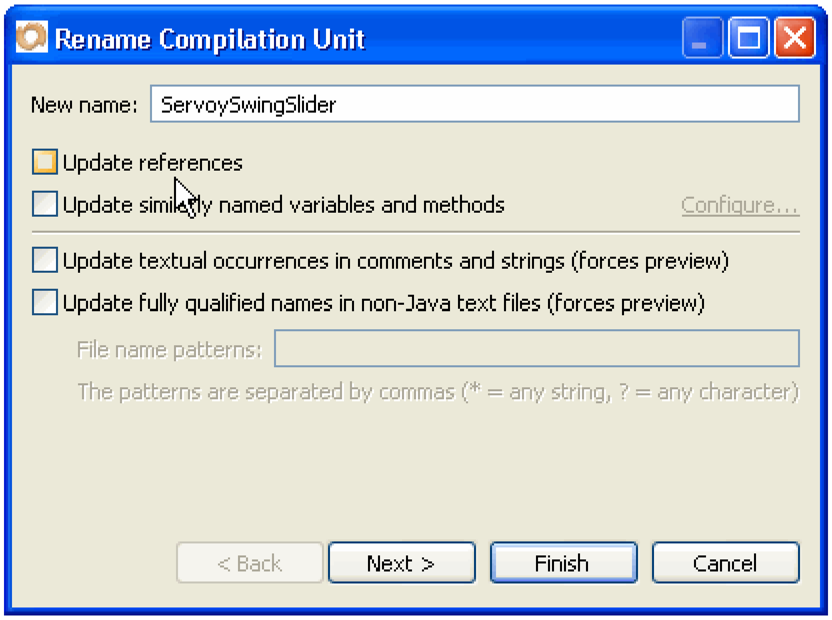
But we still have one very important step to do: we need to implement the IServoyBeanFactory that will tie our Wicket Slider to our Swing Slider and make them the two faces of the same coin.

DD. The ServoySlider as an IServoyBeanFactory

But we have a little problem. Right now we have a ServoySlider which is a Swing bean, and a ServoyWicketSlider which is the Wicket component. So which one is going to be our IServoyBeanFactory?

Actually it would be nice if our bean factory was the ServoySlider while the Swing bean would be a ServoySwingSlider.

Before we create a ServoySlider, let's rename our "ServoySlider" to "ServoySwingSlider", by using the "Refactor > Rename" menu (from a right click on the class in the Package Explorer or from the name of the class in its signature in the Java Editor):


Rename Compilation Unit

New name: ServoySwingSlider

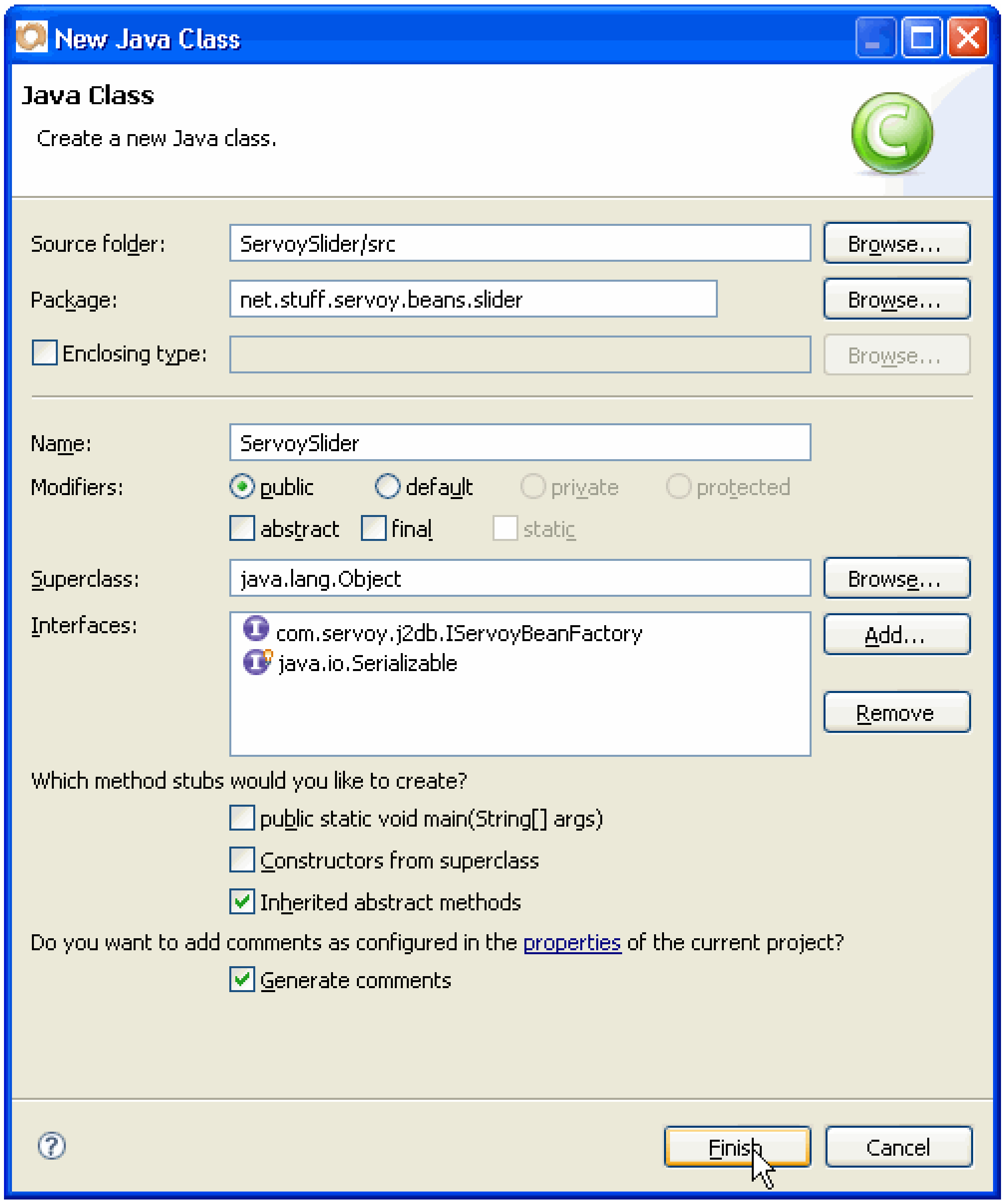
- [ ] Update references
- [ ] Update similarly named variables and methods
- [ ] Update textual occurrences in comments and strings (forces preview)
- [ ] Update fully qualified names in non-Java text files (forces preview)

File name patterns:

The patterns are separated by commas (* = any string, ? = any character)

Note that I asked Eclipse not to "Update references" (which is the default), because I don't want the ServoySliderBeanInfo class which refers to ServoySlider to be changed at all. It will show errors until we create a ServoySlider again, but that's fine since we want the bean to be the ServoySlider (it is the one that Servoy will use, and knowing that it is an IServoyBeanFactory it will be the one used to instantiate the proper implementation – Swing or Wicket).

So as soon as we have renamed our ServoySlider to ServoySwingSlider, we can create a new class ServoySlider implementing IServoyBeanFactory and java.io.Serializable to help Servoy persisting its properties:


New Java Class

Java Class

Create a new Java class.

Source folder: ServoySlider/src

Package: net.stuff.servoy.beans.slider

[ ] Enclosing type:

Name: ServoySlider

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:
com.servoy.j2db.IServoyBeanFactory
java.io.Serializable

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

[✓] Generate comments

Now it's only a matter of adding all the required properties, and the getters/setters.
First the properties:

```
private static final long serialVersionUID = 1L;

private IServoySliderBean component = new ServoySwingSlider();

private Color background;
private Border border;
private boolean focusable = true;
private Font font;
private Color foreground;
private boolean inverted;
private boolean opaque = true;
private int orientation;
private boolean paintLabels;
private boolean paintTicks = true;
private boolean paintTrack = true;
```

```
	private boolean snapToTicks = true;
	private Point location;
	private String name;
	private Dimension size = new Dimension(200,30);

	private String dataProviderID;
	private String toolTipMessage;

	private int majorTicks = 50;
	private int minorTicks = 10;
	private int minimumValue = 0;
	private int maximumValue = 100;
	private int currentValue = 0;

	private int precision = 2;
	private boolean usingDecimals;
```

Note that I use the IServoySliderBean interface to store a reference to the object that will be implemented (either Swing or Wicket)
Note that I have created a ServoySwingSlider right away. That's because it is going to be used in Developer as soon as you put it on a form, so it helps having it already created...
Note also that I have set a few default properties, these will be used when you place the bean on a form in developer.

Now all we have to do is to code all the getters and setters for these properties, and they will all have the exact same structure:

```
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
		if (component != null) {
			component.setName(name);
		}
	}
```

The gettter simply returns the instance property, while the setter updates the property and then passes the new value to the "real" component.

I leave you to code all these setters and getters, just ask Eclipse to "Generate Getters and Setters..." for you first from the properties, and then modify the setters.
The final code is in the Eclipse project that, as usual, will be available for download, if you are feeling lazy ;-)

We only have one method to code after that, and that is of course the getBeanInstance() method.

You will recognize its implementation from our last TestBean's factory:

```
public IComponent getBeanInstance(int appMode, IClientPluginAccess app,
                  Object[] paramArrayOfObject) {

      if (appMode == IClientPluginAccess.WEB_CLIENT) {
            String id = (String)paramArrayOfObject[0];
            component = new ServoyWicketSlider(id);
            setBackground((background == null) ? Color.white : background);
            setForeground((foreground == null) ? Color.black : foreground);
            setBorder((border == null) ? new EtchedBorder(1) : border);
      }
      initComponent(app);
      return component;
}
```

Remember that this method gets 3 parameters from Servoy, first an int containing the "applicationType" (the same value as in Servoy's application.getApplicationType(), second an interface to the client (IClientPluginAccess, which in the case of a Web client can be an IWebClientPluginAccess), and third an array of Object[], where the only object I ever found inside is the UUID of the element as a String, that we will pass to our ServoyWicketSlider constructor…

So we test if the mode is WEB_CLIENT (== 5), if so we retrieve the UUID and pass it to the constructor of our ServoyWicketSlider which then becomes our component property.
We set a few default values on our Wicket component (so that it appears like any other field components in the web page), and we call an initComponent() with the client object as parameter.

This calls for creating a new method initComponent() which will be:

```
      private void initComponent(IClientPluginAccess app) {
            component.initialize(app);
            setFocusable(focusable);
            setMinorTicks(minorTicks);
            setMajorTicks(majorTicks);
            setMaximumValue(maximumValue);
            setMinimumValue(minimumValue);
            setOpaque(opaque);
            setPaintTicks(paintTicks);
            setPaintTrack(paintTrack);
            setCurrentValue(currentValue);
            setSize(size);
            setPrecision(precision);
            setUsingDecimals(usingDecimals);
            setDataProviderID(dataProviderID);
            setToolTipMessage(toolTipMessage);
      }
```

In here, we send our app to the component via its initialize method (playing the role of Servoy when it instantiates an IScriptObject: a plugin) this is to ensure that our scripting methods (who might need this reference) is initialized correctly and then we set the values using our internal methods (this will pass the values of the Servoy Properties View to initialize our bean with the user's values).

Note that the ServoySliderBeanInfo is now happy because it has a ServoySlider class to refer too, and if you go into its getPropertyDescriptors() method you should have all the required properties referring to properties in your newly defined ServoySlider class.

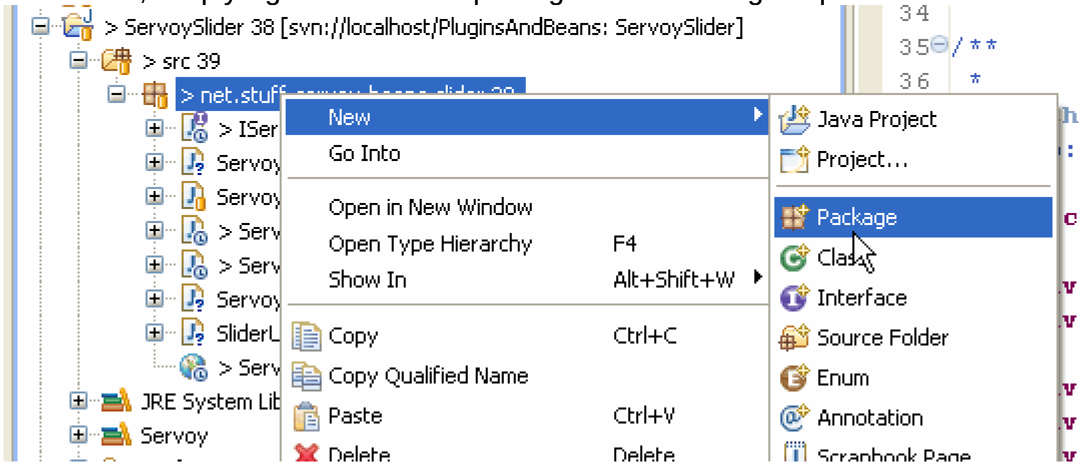
EE. Package clean up

So that will be it for our new implementation, a ServoySlider IServoyBeanFactory as the factory class that will return either a ServoySwingSlider for developer or the smart client and a ServoyWicketSlider for the web client.

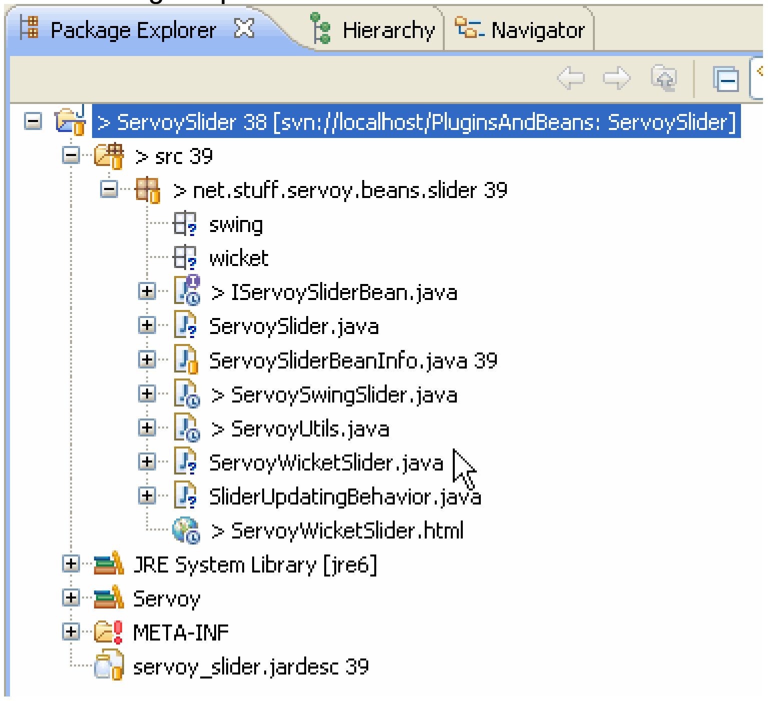
One last think that I would like you to do, is to clean up our package a bit.
Packages in java are supposed to hold logical classes and interfaces that share the same purpose and are used in the same context. Right now this is not really true, because we mix Swing and Wicket derived class: their purpose and context are clearly very different.

So to clean this up a little bit, what we can do is add 2 new sub-packages to our "net.stuff.servoy.beans.slider" package. We will need: net.stuff.servoy.beans.slider.swing and net.stuff.servoy.beans.slider.wicket.

To do that, simply right click on the package in the Package Explorer and choose "New > Package":

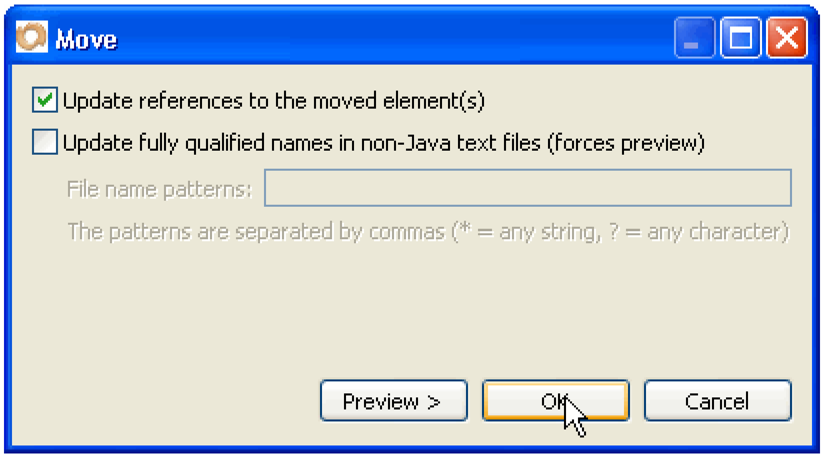


Your Package Explorer should now look like this:


Package Explorer
Hierarchy
Navigator
> ServoySlider 38 [svn://localhost/PluginsAndBeans: ServoySlider]
> src 39
> net.stuff.servoy.beans.slider 39
swing
wicket
> IServoySliderBean.java
ServoySlider.java
ServoySliderBeanInfo.java 39
> ServoySwingSlider.java
> ServoyUtils.java
ServoyWicketSlider.java
SliderUpdatingBehavior.java
> ServoyWicketSlider.html
JRE System Library [jre6]
Servoy
META-INF
servoy_slider.jardesc 39

And now, all you will have to do is to drag and drop the relevant files into the new packages.

First take the ServoySwingSlider and put it into the new “swing” package, Eclipse will show you this “Move” dialog to ask you if you want to “update references to the moved element(s)”, check that, and all the code that is referring to these class will be updated (the imports will be added, etc…):


Move

[✓] Update references to the moved element(s)

[ ] Update fully qualified names in non-Java text files (forces preview)

File name patterns:

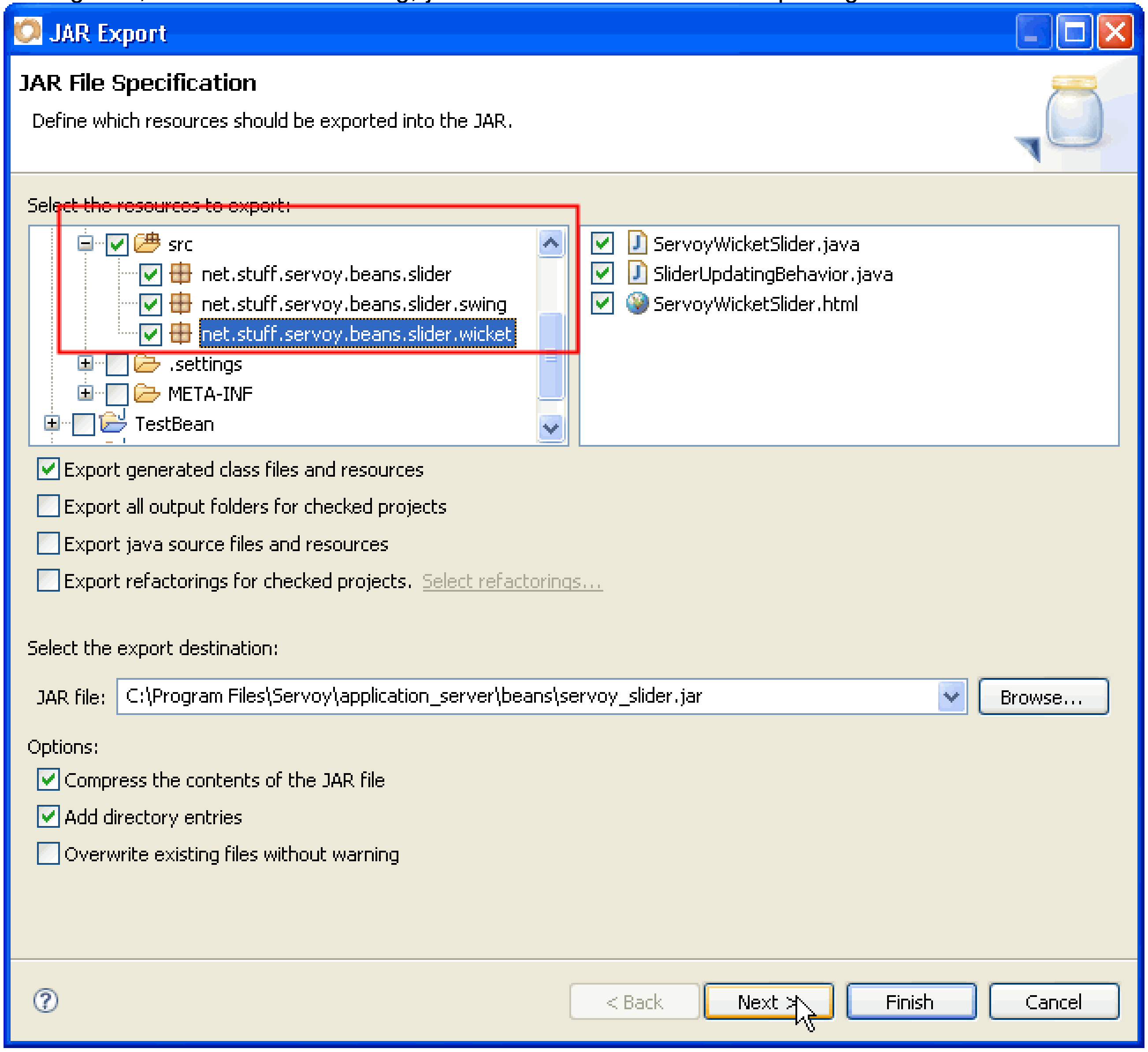
The patterns are separated by commas (* = any string, ? = any character)

Next, we put the ServoyWicketSlider, SliderUpdatingBehavior and ServoyWicketSlider.html files into the new “wicket” packages and the same dialog will show, to which you will also ask Eclipse to update the references.

If everything is OK, your project should contain no error, otherwise check the “Problems” view and see if there are some explanations. And get back in time in this tutorial (or check from the Eclipse project that you will find on the Servoy Stuff site to see what is different between your version and mine).

Now it’s time to publish our bean, but you need to update your servoy_slider.jardesc before you do so because by default it will not recognize the new packages automatically. You will need to tell him explicitly to export them otherwise your jar will not contains the “swing” and “wicket” beans, and you will have very bad errors in Servoy…

So right click on the "servoy_slider.jardesc" file in the Package Explorer and choose "Open JAR Packager…", then in the first Dialog, just make sure that the 2 new packages are selected:



JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- [✓] src
  - [✓] net.stuff.servoy.beans.slider
  - [✓] net.stuff.servoy.beans.slider.swing
  - [✓] net.stuff.servoy.beans.slider.wicket
- [ ] .settings
- [ ] META-INF
- [ ] TestBean

- [✓] ServoyWicketSlider.java
- [✓] SliderUpdatingBehavior.java
- [✓] ServoyWicketSlider.html

- [✓] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [ ] Export java source files and resources
- [ ] Export refactorings for checked projects. Select refactorings…

Select the export destination:

JAR file: C:\Program Files\Servoy\application_server\beans\servoy_slider.jar

Options:

- [✓] Compress the contents of the JAR file
- [✓] Add directory entries
- [ ] Overwrite existing files without warning

Then you can click "Finish" because the rest of the ".jardesc" is already properly configured.

Now, after restarting Servoy, if you put your ServoySlider bean on a form, give it a few properties like a dataProvider (coming from the form's related table) of type number, and a range (minimum and maximum) of acceptable values (0-100 for a percentage, with precisionFactor = 2, and usingDecimals = true for example), you should be able to use it in as usual in the Smart client but also in the Web client as a simple text input, but with real values updated from Servoy and working as any text input would do…

Only one last thing though that might be needed if we don't want problems is to add scripting to our Wicket bean, because the Swing bean is using scripting and has declared all the related js_XXX methods, so your users will be tempted to script your bean just like any Servoy elements and they will expect it to work in the Web client as well, except that the related js_XXX methods are not present there yet, and this will result in Exceptions. So let's add scripting to our Wicket component.

FF. Scripting the ServoyWicketSlider

Easily enough we are going to copy all of the js_XXX methods from our ServoySwingSlider class and paste them into our ServoyWicketSlider (that's not too bad here – for once! - because these methods have no real logic, they are only accessors' aliases).

There are only a few methods that will need tweaking for this to work, and they are:

```
	public void js_setLocation(int x, int y) {
		setLocation(new Point(x, y));
	}
	public void js_setSize(int width, int height) {
		setSize(new Dimension(width, height));
	}
```

Note that we simply forward to the already existing methods, creating the correct parameter.

```
	public void js_setValue(Number val) {
		if (val != null) {
			setNumberValue(val.doubleValue());
		} else {
			setNumberValue(0);
		}
	}
	public Number js_getValue() {
		return getNumberValue();
	}
```

Note here that we didn't perform a check with isUsingFactor() in this version of the bean, we will leave that to the next part of this tutorial when we will integrate a real Slider with real numeric values.

```
	public int js_getHeight() {
		return getWebBounds().height;
	}
	public int js_getWidth() {
		return getWebBounds().width;
	}
	public int js_getLocationX() {
		return getWebBounds().x;
	}
	public int js_getLocationY() {
		return getWebBounds().y;
	}
```

Note that we are using our getWebBounds (which returns a Rectangle) to return the height, width, x and y values accordingly.

Then we need to make sure that the js_setVisible() and js_setEnabled() methods are forwarding to the right methods, so that our ChangesRecorder class will update the markup properly, and we change them slightly to:

```
	public void js_setVisible(boolean visible) {
		setComponentVisible(visible);
	}
	public void js_setEnabled(boolean enabled) {
		setComponentEnabled(enabled);
	}
```

And we make sure that our js_setToolTipText is calling the setToolTipMessage that uses the Messages class for i18n:

```
	public void js_setToolTipText(String toolTipText) {
		setToolTipMessage(toolTipText);
	}
```

Now if you were to try it in the web client, you would find that the scripts are not executing correctly.

Because there is one thing needed by Wicket to allow for manipulation of the markup at runtime (which is basically what is happening when you change a property by scripting: an Ajax callback method updates the property on the server and then the result of the Ajax call changes the html markup in the web client with dynamic JavaScript), so to allow for this, you will need to add this line to your constructor:

```
		setOutputMarkupPlaceholderTag(true);
```

This basically tells Wicket that the markup as set in the html file is only a placeholder that can be replaced at runtime.

Now with this line added to the class constructor, the scripting will work for the basic methods and properties (background, visible etc…)!
Of course since we didn't implement the specialized Slider properties, all the related js_XXX will do nothing.

But this is something we will try to tackle in the next part of this tutorial, where we will add a DHTML Slider (and all its JavaScript and CSS dependencies) to our TextField.

As usual, you will find the complete Eclipse project on the Servoy Stuff web site, here:
http://www.servoy-stuff.net/tutorials/utils/t02/v4/ServoySlider_EclipseProject-v4.zip
(Get rid of the previous project of the same name and import in Eclipse)

The compiled bean (targeted for java 1.5) will be available here:
http://www.servoy-stuff.net/tutorials/utils/t02/v4/servoy_slider.jar
(Put in you /beans folder)

And the little "beans_tests" solution updated to use the new bean in situation will be available at:
http://www.servoy-stuff.net/tutorials/utils/t02/v4/beans_tests-v4.zip
(Unzip and import in Servoy 4.1.x)

Hope you enjoyed this tutorial and that you learned a few useful things from it…
Feel free to send comments, suggestions, questions, etc… to me on the Servoy Stuff site.

In the next part we will turn our pitiful TextField into an exciting Slider in the Web client, so stay tuned!

Patrick Talbot
Servoy Stuff
2009-07-27