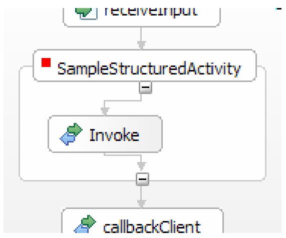
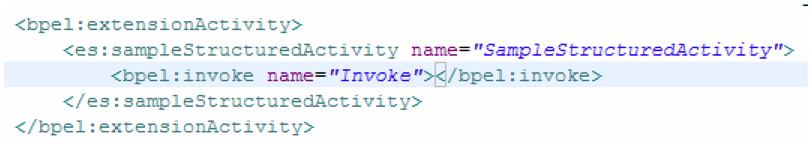
Extending the Eclipse BPEL Designer with custom Activities

Benjamin Höhensteiger (hoehenbn@studi.informatik.uni-stuttgart.de)
Michael Illiger (milliger@de.ibm.com)
Simon Moser (smoser@de.ibm.com)

1 Introduction

Purpose of this tutorial is to demonstrate how the Eclipse BPEL[1] Designer can be extended, so that it can handle custom activities of its own as a BPEL 2.0 ExtensionActivity. We will show how a custom activity can be added to the BPEL Designer so that it can be properly used in BPEL processes, both in the graphical and in the textual mode. The custom activity we create in this tutorial will be able to contain any other BPEL activity as a child element.





```
<bpel:extensionActivity>
    <es:sampleStructuredActivity name="SampleStructuredActivity">
        <bpel:invoke name="Invoke"></bpel:invoke>
    </es:sampleStructuredActivity>
</bpel:extensionActivity>
```

Extending the BPEL Designer with a custom activity is basically a two step procedure. The first part is creating an EMF[2] model representation of your activity, taking care of serialization and deserialization and plugging your custom model into the BPEL model. The second part is visualizing the custom activity in the UI. This also includes to bring it onto the palette and to allow it to be dropped onto the canvas. The structure of this tutorial follows this two step paradigm. In chapter 2 we show how the EMF model is created and in chapter 3 we care about the UI. It is also good practice to have these two tasks separated from each from a code point of view. This means that two separated Eclipse plug-ins will be created.

Reference implementations of the model and the UI plug-in described in this tutorial can be found in the BPEL Designer CVS in the '/examples/plugins' directory.

- org.eclipse.bpel.extensionsample.model
- org.eclipse.bpel.extensionsample.ui

[1] Business Process Execution Language
[2] Eclipse Modelling Framework

2 The model plug-in

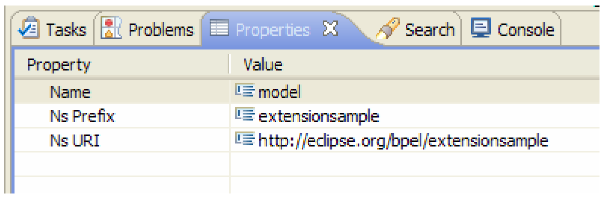
- Create a new plug-in project and name it **org.eclipse.extensionsample.model**. On the second page of the "New Plug-in Project" wizard deselect the **Generate an activator …** and the **This plug-in will make contributions to the UI** checkboxes.

- Create a new folder inside the model plug-in and call it **model**.

2.1 Create an ecore model

- To create the EMF model for our custom activity we start with creating an ecore model. Right click on the **model** folder and select **New** ⇨ **Other** ⇨ **Ecore Model**. Enter **extensionsmaple.ecore** as the filename and select **EPackage** as the Model Object on the last wizard page.

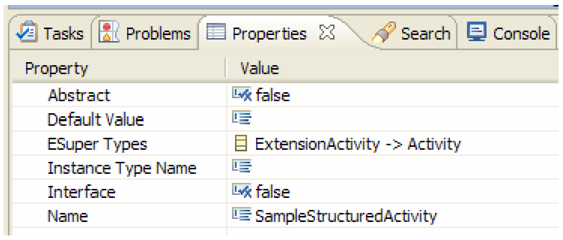
- Your newly created ecore model should now be opened in the Ecore Model Editor. Select the "empty" package and set N**ame**, **Ns Prefix** and **Ns URI** to the following values in the properties view.



Property	Value
Name	model
Ns Prefix	extensionsample
Ns URI	http://eclipse.org/bpel/extensionsample

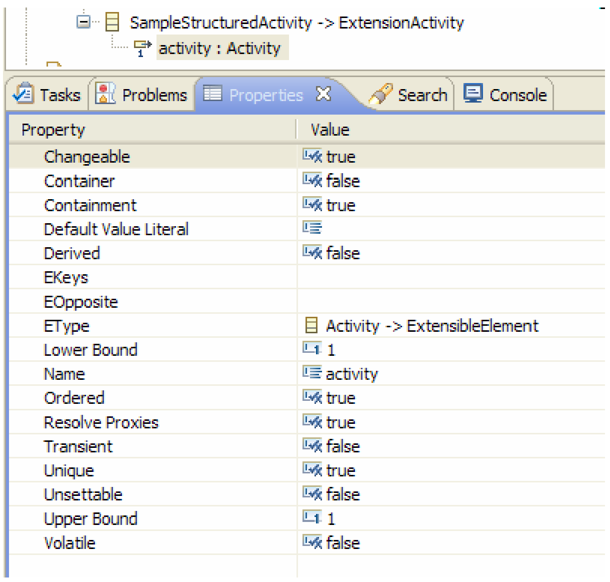
- Now we are ready to create our activity, but first we need the BPEL model information in our custom model, so that we can extend our Activitiy from ExtensionActivity. Right click on the model, select **Load Ressource** and add the **bpel.ecore** model from the workspace.

- To add our custom activity right click on the **model** package and select **New Child** ⇨ **EClass**. In the properties view set the new EClass' Name to **SampleStructuredActivity** and its ESuperType to **ExtensionActivity** from the BPEL model.



Property	Value
Abstract	false
Default Value	
ESuper Types	ExtensionActivity -> Activity
Instance Type Name	
Interface	false
Name	SampleStructuredActivity

- The SampleStructuredActivity should contain a child that can be any BPEL activity. Right click on the SampleStructuredActivity EClass and select **New Child** ⇨ **EReference**. Set the Name to be **activity** and the EType to be **Activity** from the BPEL model. In order to make our child activity mandatory set the Lower Bound attribute to **1**. Make sure the **Containment** attribute is set to **true**.



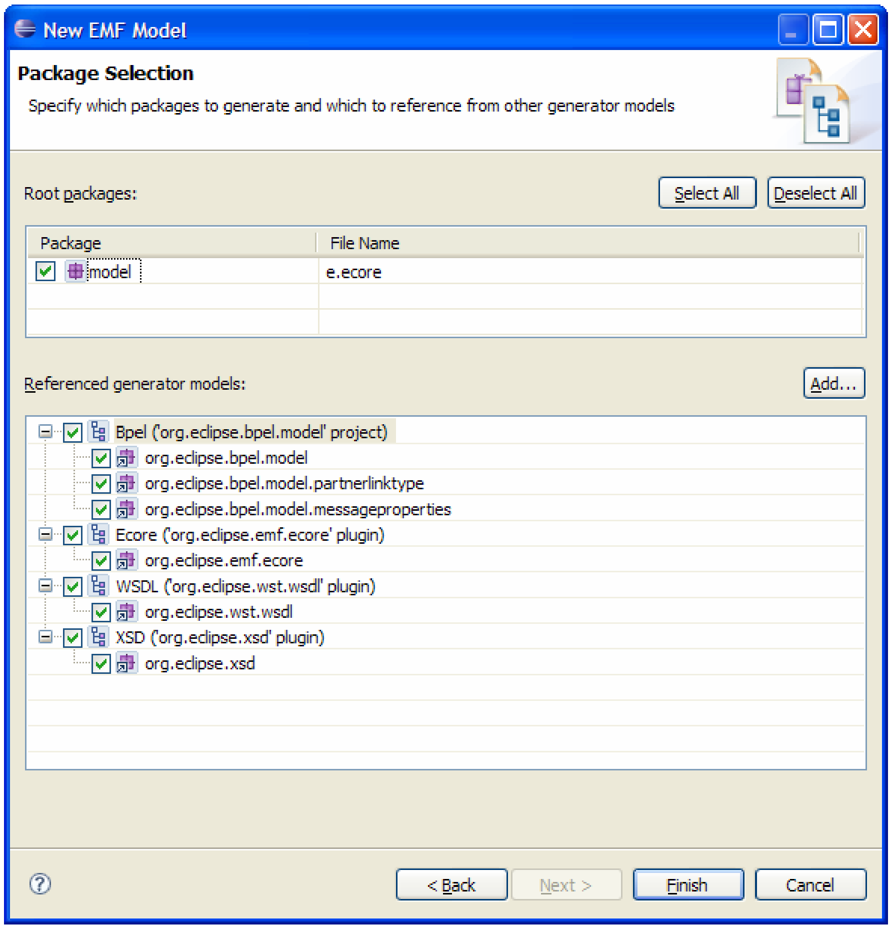
Property	Value
Changeable	true
Container	false
Containment	true
Default Value Literal	
Derived	false
EKeys	
EOpposite	
EType	Activity -> ExtensibleElement
Lower Bound	1
Name	activity
Ordered	true
Resolve Proxies	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1
Volatile	false

- Save and close the ecore model editor.

2.2 Generate the model code

The next step is to create an EMF genmodel out of our ecore model and to generate the model Java code.

- Right click the **model** folder and select **New** ⇨ **EMF** ⇨ **EMF Model**.
- Enter **extensionsample.genmodel** as the filename.
- Click **Next** and select **Ecore model**. Click **Next**.
- Select **Browse Workspace…** and choose the **extensionsample.ecore** file we just created. Click **Next**.
- Click the **Add** button and add **bpel.genmodel** from the **org.eclipse.bpel.model** plug-in to the list of referenced generator models.
- Check all the checkboxes as shown in the screenshot and select **Finish**.



Package Selection

Specify which packages to generate and which to reference from other generator models

Root packages:

Package	File Name
✓ model	e.ecore

Referenced generator models:

- ✓ Bpel ('org.eclipse.bpel.model' project)
  - ✓ org.eclipse.bpel.model
  - ✓ org.eclipse.bpel.model.partnerlinktype
  - ✓ org.eclipse.bpel.model.messageproperties
- ✓ Ecore ('org.eclipse.emf.ecore' plugin)
  - ✓ org.eclipse.emf.ecore
- ✓ WSDL ('org.eclipse.wst.wsdl' plugin)
  - ✓ org.eclipse.wst.wsdl
- ✓ XSD ('org.eclipse.xsd' plugin)
  - ✓ org.eclipse.xsd

- The **extensionsample.genmodel** is opened. Change the **Base Package** value of to **org.eclipse.bpel.extensionsample** and save the genmodel.
- Now do a right click on the **Model** package and select **Generate Model Code**.
- EMF generates your model's Java classes in the src folder of your model plug-in. If there is a compile error that mentions that the type javax.wsdl.extensions.ExtensibilityElement cannot be resolved, please add **javax.wsdl** to the list of required plug-ins in your plug-in's MANIFEST.MF file.

2.3 Modify the generated code to handle reconciling

The mechanism that updates the source tab of the Eclipse BPEL Designer as soon as the BPEL model is changed is known as "reconciling". This mechanism isn't auto-generated by EMF and needs to be implemented in the *Impl model classes manually.

- In order to implement reconciling for our generated custom activity open the **SampleStructuredActivityImpl** class and add the following lines to the **basicSetActivity** method.

```
    if (!isReconciling) {
       ReconciliationHelper.replaceChild(this, oldActivity, newActivity);
    }
```

- To prevent EMF from overwriting the changes you made to the basicSetActivity method the next time our model is re-generated, change the **@generated** annotation to something like **@customized**.

- Since our SampleStructuredActivity can have a nested child activity we also need to override the **adoptContent()** and the **orphanContent()** methods in the **SampleStructuredActivityImpl** class. They are needed for proper setting, replacing and deleting the child activity.

```
@Override
protected void adoptContent(EReference reference, Object object) {
   if (object instanceof Activity) {
      ReconciliationHelper.replaceChild(this, activity, (Activity) object);
   }
   super.adoptContent(reference, object);

}

@Override
protected void orphanContent(EReference reference, Object obj) {
   if (obj instanceof Activity) {
      ReconciliationHelper.orphanChild(this, (Activity)obj);
   }
   super.orphanContent(reference, obj);
}
```

Congratulations, the model of your new activity is finished. The next parts of this tutorial describe what classes need to be adapted and what you have to write to get your activity run in the BPEL Designer.

2.4 Write your custom Deserializer

The BPELReader is responsible for reading .bpel files and building up its object representation. Of course it does not know how to create an instance of our SampleStructuredActivity activity by default. Therefore we have to write a custom deserializer that implements the **BPELActivityDeserializer** interface. In the unmarshall method we analyze the passed in DOM node and create a new instance of our SampleStructuredActivity.

```
public Activity unmarshall(QName elementType, Node node, Process process,
      Map nsMap, ExtensionRegistry extReg, URI uri, BPELReader bpelReader) {

   /*
    * SampleStructuredActivity
    */
   if ("sampleStructuredActivity".equals(elementType.getLocalPart())) {

      Element sampleStrucutredActivityElement = (Element) node;

      // create a new SampleStructuredActivity model object
      SampleStructuredActivity activity = ModelFactory.eINSTANCE
               .createSampleStructuredActivity();

      // attach the DOM node to our new activity
      activity.setElement(sampleStrucutredActivityElement);

      // handle the child activity
      NodeList childElements = sampleStructuredActivityElement.getChildNodes();
      Element activityElement = null;
      if (childElements != null && childElements.getLength() > 0) {
         for (int i = 0; i < childElements.getLength(); i++) {

            // the only element node is the child activity
            if ((childElements.item(i).getNodeType() == Node.ELEMENT_NODE)) {
               activityElement = (Element) childElements.item(i);
               Activity child = bpelReader.xml2Activity(activityElement);
               if (child != null) {
                  activity.setActivity(child);
               }
```

```
                }
            }
        }

        return activity;
    }

    System.out.println("Cannot handle this kind of element ");
    return null;
}
```

2.5 Write your custom Serializer

As the BPELReader reads the file, the BPELWriter serializes the object representation to an xml file. To write our activity we have to create a BPELActivitySerializer class that does this for us.

The following code snippet creates a new DOM element and fills it with the values from the model object. It also registers the namespace URI of our model at the process' namespace map.

```
public void marshall(QName elementType, Activity activity, Node parentNode,
     Process process, BPELWriter bpelWriter) {

   Document document = parentNode.getOwnerDocument();

   /*
    * SampleStructuredActivity
    */
   if (activity instanceof SampleStructuredActivity) {

      // register the namespace
      String nsPrefix = ModelPackage.eINSTANCE.getNsPrefix();
      String nsURI = ModelPackage.eINSTANCE.getNsURI();
      INamespaceMap<String, String> nsMap = BPELUtils.getNamespaceMap(process);
      nsMap.put(nsPrefix, nsURI);

      // create a new DOM element for our Activity
      Element activityElement = document.createElementNS(elementType.getNamespaceURI(),
            "sampleStructuredActivity");
      activityElement.setPrefix(nsPrefix);

      // handle child activity
      Activity childActivity = ((SampleStructuredActivity) activity).getActivity();
      if (childActivity != null) {
         activityElement.appendChild(bpelWriter.activity2XML(childActivity));
      }

      // insert the DOM element into the DOM tree
      parentNode.appendChild(activityElement);

   } else {
      System.out.println("Cannot handle this kind of Activity");
   }
}
```

2.6 Register your Serializer and Deserializer

Well, we now have classes that transform the new activity from their xml representation to the EMF object representation and vice versa, but they are never called! How can we tell the BPEL Designer to use our classes? This is very easy, because our activity is an instance of **ExtensionActivity**. Take a look at the BPELReaders **xml2ExtensionActivity** method. There you can see that it tries to unmarshall an “unknown” activity from a class like the one we just created. To inform the BPELReader and BPELWriter about our newly written Deserializer and Serializer we register them at the BPEL Designers extension registry. This is best done at an early loaded class like the ModelPackage. Therefore we create the new method **registerSerializerAndDeserializer** and call it from the ModelPackageImpl’s **init**() method. (Don’t forget to modify the @generated annotation of the init method!)

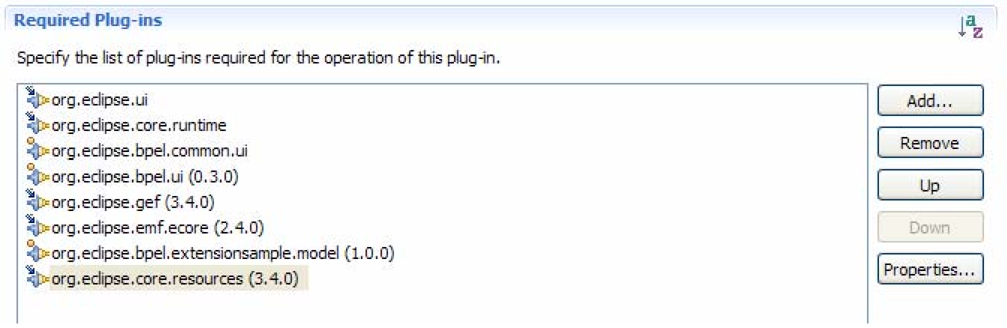
```
private static void registerSerializerAndDeserializer() {
    // Initialize BPEL extension registry
    BPELExtensionRegistry extensionRegistry = BPELExtensionRegistry.getInstance();

    // Initialize our own serializers and deserializers
    ExtensionSampleActivityDeserializer deserializer = new
          ExtensionSampleActivityDeserializer();
    ExtensionSampleActivitySerializer serializer = new ExtensionSampleActivitySerializer();

    // SampleStructuredActivity
    String name = SampleStructuredActivity.class.getSimpleName();
    extensionRegistry.registerActivityDeserializer(new QName(ModelPackage.eNS_URI,
          "sampleStructuredActivity"), deserializer);
    extensionRegistry.registerActivitySerializer(new QName(ModelPackage.eNS_URI, name),
          serializer);
 }
```

3 The UI plug-in

So far we have created an EMF model of the custom activity and took care about serializing and deserializing it. Now we come to the graphical part that adds our activity to the BPEL Designer's palette and the user to drag it onto the editor's canvas. As already mentioned is it good style to have the model separated from the UI parts. Therefore we start with a brand-new UI plug-in.

- Create a new plug-in project and name it **org.eclipse.extensionsample.ui**. On the second page of the "New Plug-in Project" wizard make sure the **Generate an activator …** and the **This plug-in will make contributions to the UI** checkboxes are checked.

- Add the following plug-ins to the list of required plug-ins in **MANIFEST.MF** file of your newly created plug-in.


Required Plug-ins

Specify the list of plug-ins required for the operation of this plug-in.

org.eclipse.ui
org.eclipse.core.runtime
org.eclipse.bpel.common.ui
org.eclipse.bpel.ui (0.3.0)
org.eclipse.gef (3.4.0)
org.eclipse.emf.ecore (2.4.0)
org.eclipse.bpel.extensionsample.model (1.0.0)
org.eclipse.core.resources (3.4.0)

3.1 Write your custom UIObjectFactory

The first step that we have to do in our new UI plug-in is the creation of our own **UIObjectFactory** (a factory that knows how to create conceptual types of UI objects) and to register it at the **org.eclipse.bpel.ui.uiObjectFactories** extension point.

```
<extension id="ExtensionSampleUIObjectFactory" name="test"
   point="org.eclipse.bpel.ui.uiObjectFactories">
   <factory
      class="org.eclipse.bpel.extensionssample.ui.factories.ExtensionSampleUIObjectFactory"
      specCompliant="false" categoryId="not.used"
      id="org.eclipse.bpel.extensionssample.ui.factories.ExtensionSampleUIObjectFactory" />
</extension>
```

```
public class ExtensionSampleUIObjectFactory extends AbstractUIObjectFactory implements
   IExtensionUIObjectFactory {

   private EClass modelType;
   private EClass[] classArray = { ModelPackage.eINSTANCE.getSampleStructuredActivity() };

   public ExtensionSampleUIObjectFactory(EClass modelType) {
      super();
      this.modelType = modelType;
   }

   public ExtensionSampleUIObjectFactory() {
      super();
   }

   @Override
   public Image getLargeImage() {
      return Activator.getDefault().getImageRegistry().get("obj32/default.gif");
   }

   @Override
   public ImageDescriptor getLargeImageDescriptor() {
      return Activator.getDefault().getImageDescriptor("obj32/default.gif");
   }

   @Override
   public EClass getModelType() {
      return this.modelType;
   }

   @Override
   public Image getSmallImage() {
      return Activator.getDefault().getImageRegistry().get("obj16/default.gif");
   }

   @Override
   public ImageDescriptor getSmallImageDescriptor() {
      return Activator.getDefault().getImageDescriptor("obj16/default.gif");
   }

   @Override
   public String getTypeLabel() {
      return getModelType().getName();
   }

   @Override
   public EClass[] getClassArray() {
      return this.classArray;
   }

   @Override
   public void setModelType(EClass modelType) {
      this.modelType = modelType;
   }

}
```

3.2 Write your custom activity adapter

Activity adapters are a common concept of the Eclipse BPEL Designer and are used for many different purposes. To follow this pattern we create our own **SampleStructuredActivityAdapter** and extend it from **ContainerActivityAdapter**. The only two methods that need to be implemented are **createConatinerDelegate()** which should return an instance of an IContainer that is aware of how our child activity can be accessed. And **createEditPart()** that returns a GEF[3] EditPart representing our custom activity. In our example we reuse the existing SequenceEditPart which will make our custom activity look like a BPEL Sequence.

```
public class SampleStructuredActivityAdapter extends ContainerActivityAdapter {

   @Override
   protected IContainer createContainerDelegate() {
      return new ActivityContainer(
            ModelPackage.eINSTANCE.getSampleStructuredActivity_Activity());
   }

   @Override
```

[3] Graphical Editing Framework

```
   public EditPart createEditPart(EditPart context, Object model) {
      EditPart result = new SequenceEditPart();
      result.setModel(model);
      return result;
   }

}
```

3.3 Subclass your model's AdapterFactory and register it in the Activator

Now that we have defined our Adapter, we can create the **ExtensionSampleUIAdapterFactory**. Extend the AdapterFactory that was automatically created by EMF. In our example this is the **ModelAdapterFactory** class in the **org.eclipse.bpel.extensionsample.model.util** package. Override the **createSampleStructuredActivityAdapter()** and return an instance of our **SampleStructuredActivityAdapter**.

```
   @Override
   public Adapter createSampleStructuredActivityAdapter() {
      if (this.sampleStructuredActivityAdapter == null) {
         this.sampleStructuredActivityAdapter = new SampleStructuredActivityAdapter();
      }
      return this.sampleStructuredActivityAdapter;
   }
```

Once your ExtensionSampleUIAdapterFactory is ready, you need to register it for your custom model. The best place to do this is during plug-in startup. Add the following line to the **start()** method in your plug-in's **Activator** class.

```
   BPELUtil.registerAdapterFactory(ModelPackage.eINSTANCE,
      new ExtensionSampleUIAdapterFactory());
```

3.4 Add your activity to the palette

To add your custom activity to the Eclipse BPEL Designer's palette you need to extend the **org.eclipse.bpel.common.ui.paletteAdditions** extension point and implement your own **IPaletteProvider**. The code example below demonstrates how a new palette category that contains a **BPELCreationToolEntry** can be created.

```
<extension point="org.eclipse.bpel.common.ui.paletteAdditions">
   <additions targetEditor="org.eclipse.bpel.ui.bpeleditor"
      provider="org.eclipse.bpel.extensionssample.ui.palette.ExtensionSamplePaletteProvider">
   </additions>
</extension>
```

```
public class ExtensionSamplePaletteProvider implements IPaletteProvider {

   @Override
   public void contributeItems(PaletteRoot paletteRoot) {

      PaletteCategory category = new PaletteCategory("ExtensionSample");
      category.setCategoryId("extensionsample");
      category.setOrder(40);

      category.add(new BPELCreationToolEntry("Sample Structured Activity",
            "Creates a new Sample Structured Activity", new ExtensionSampleUIObjectFactory(
                  ModelPackage.eINSTANCE.getSampleStructuredActivity())));

      paletteRoot.add(category);
   }

}
```

It's not a must to create a new palette category. You can also add the BPELCreationTool for your custom activity to one of the existing categories.

4 Next Steps

The next steps would be to create some sections or other GUI elements to set the attributes and the children of the SampleStructuredActivity. New activities can be added by declaring them in the model, generating the code and then doing all the stuff described above again. You can also use the same De-/Serializers (insert else ifs for according proceeding).