

Spatial Operations Tutorial for uDig Framework

Reusing Existent Spatial Process

Integrating Sextante

| <i>Author</i> | <i>Details</i> | <i>Date</i> |
|----------------|-------------------------|-------------|
| Aritz Davila | Initial version | 03/15/2010 |
| Mauricio Pazos | General document review | 04/29/2010 |
| | | |

Table of Contents

| | |
|--|----|
| 1 Introduction | 4 |
| 2 Goal..... | 5 |
| 3 Setting the Development Environment..... | 6 |
| 3.1 Setting Eclipse..... | 6 |
| 3.2 Adding the Spatial Operation Framework..... | 8 |
| 3.3 Testing the Development Environment..... | 9 |
| 4 Creating a New Plugin Project..... | 12 |
| 5 Extension for Spatial Operations..... | 14 |
| 6 Developing the LineToPolygon Spatial Operation | 16 |
| 7 Writing the LineToPolygon Operation..... | 18 |
| 7.1 SOLineToPolygonComposite class..... | 22 |
| 7.2 LineToPolygonImages class..... | 23 |
| 7.3 LineToPolygonCommand class..... | 24 |
| 7.4 Adding Sextante..... | 32 |
| 7.5 LineToPolygonTask class..... | 33 |
| 8 Executing the LineToPolygon Spatial Operations | 36 |
| 9 Contact..... | 38 |

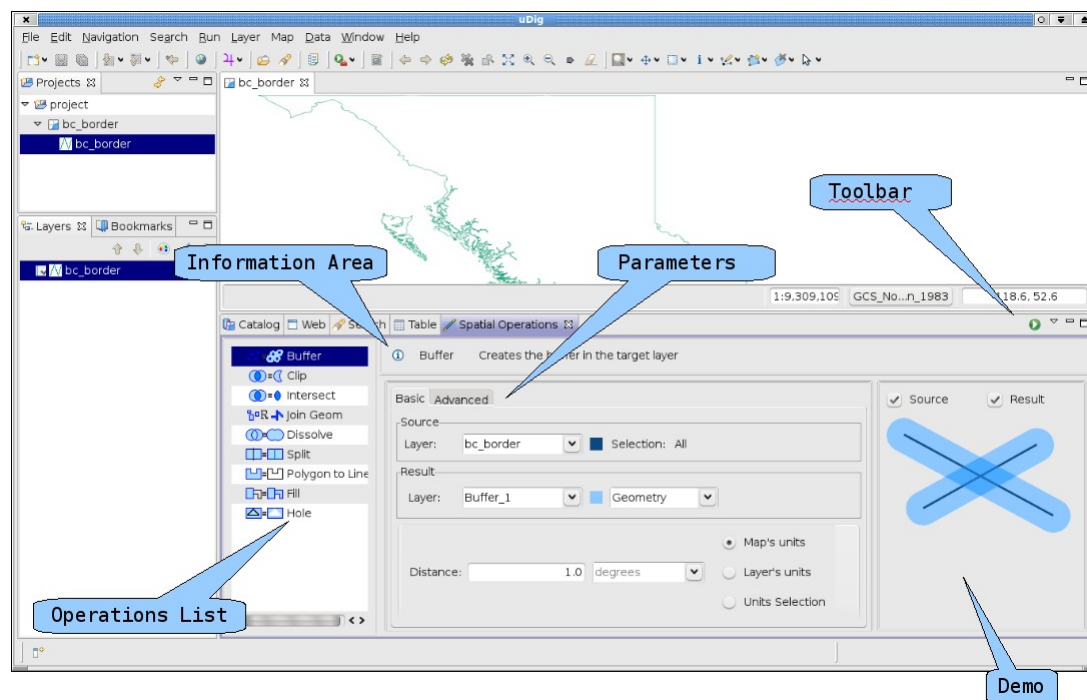
1 Introduction

This document is a description of how third parties can contribute to the Spatial Operations framework.

The Spatial Operations framework, promoted by [Gipuzkoa Provincial Council - Mobility and Land Planning Department](#), provides a template where the spatial operations can be displayed and executed. It is integrated into [uDig](#).

In this tutorial it is assumed that you know the uDig framework and your [Eclipse](#) environment is configured with **uDig SDK**. You can find good documents in the [uDig Developers](#) page.

The following figure presents the Spatial Operations view.

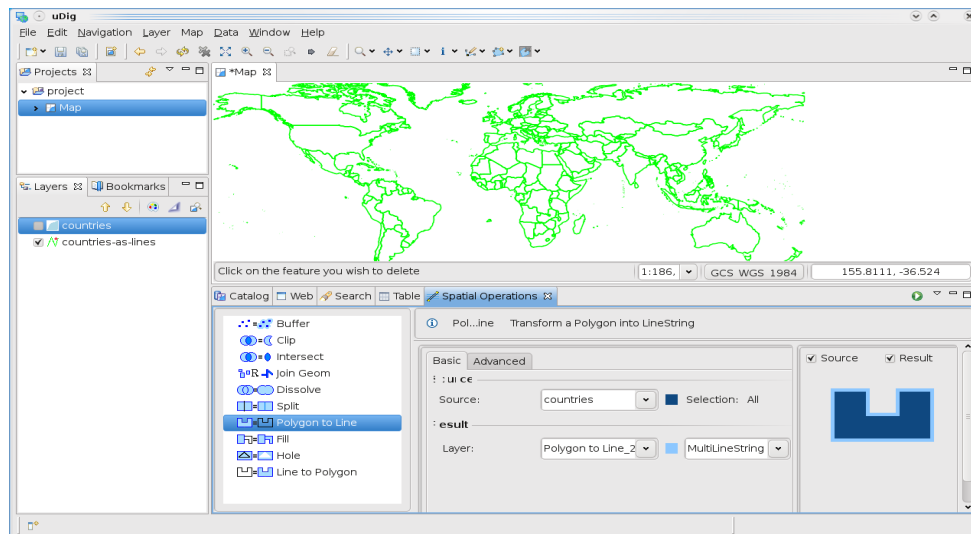


We can see the following panels:

- **Operations List:** shows a set of available spatial operations.
- **Information Area:** displays a message in order to guide in the spatial operation settings.
- **Parameters:** This panel presents the parameters required by the selected spatial operation in the **Operations List**.
- **Demo:** Provides feedback about the result of the selected spatial operation based in the parameters value.
- **Tool Bar:** Provides the actions that can be applied.

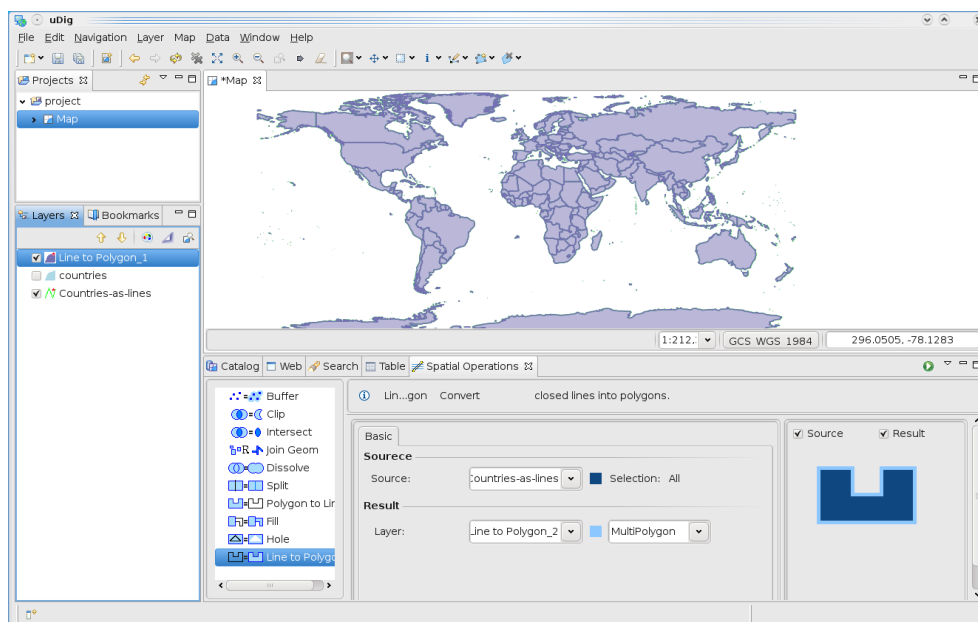
2 Goal

In this tutorial we will reuse the spatial operation **LineToPolygon** from www.sextantegis.com project. Sextante was developed under the auspices of the government of the Spanish autonomous region of Extremadura. We will use the default behavior implemented by **SimplePresenter** and **SimpleCommand** which are provided by the Spatial Operations framework.



Suppose that you have the country boundaries as shown below:

We will include the **LineToPolygon** operation as shown in the following screenshot, which could be used to create a polygon layer based in the country boundaries.



3 Setting the Development Environment

This tutorial was written for the following platform.

- uDig 1.2.x SDK
- Spatial Operations SDK 1.3.x
- Eclipse Platform 3.5.2

3.1 Setting Eclipse

1. Include the **uDig 1.2 SDK** in your Eclipse environment as described in the [uDig SDK Quick Start](#). It's a good idea to execute the uDig product (in net.refraction.udig) in order to test your environment.

2. Download **es.axios.udig.spatialoperations-1.3.x-sdk.zip** from our **download page** in www.axios.es

3. Create a new directory or folder like:

```
mkdir INSTALL_DIR/java/target/udig-spatialoperations-sdk
```

4. Unpack the spatial operations SDK in that directory.

```
unzip es.axios.udig.spatialoperations-sdk-1.3.x.zip
```

You should have the following files:

INSTALL_DIR/java/target/udig-spatialoperations-sdk/features/

es.axios.udig.spatialoperations.sdk_1.3.0.m4.jar

es.axios.udig.spatialoperations.sdk.source_1.3.0.m4.jar

INSTALL_DIR/java/target/udig-spatialoperations-sdk/plugins/

es.axios.udig.extensions_1.3.0.m4.jar

es.axios.udig.extensions.source_1.3.0.m4.jar

es.axios.udig.ui.common_1.3.0.m4.jar

es.axios.udig.ui.common.source_1.3.0.m4.jar

es.axios.udig.ui.spatialoperations_1.3.0.m4.jar

es.axios.udig.ui.spatialoperations.source_1.3.0.m4.jar

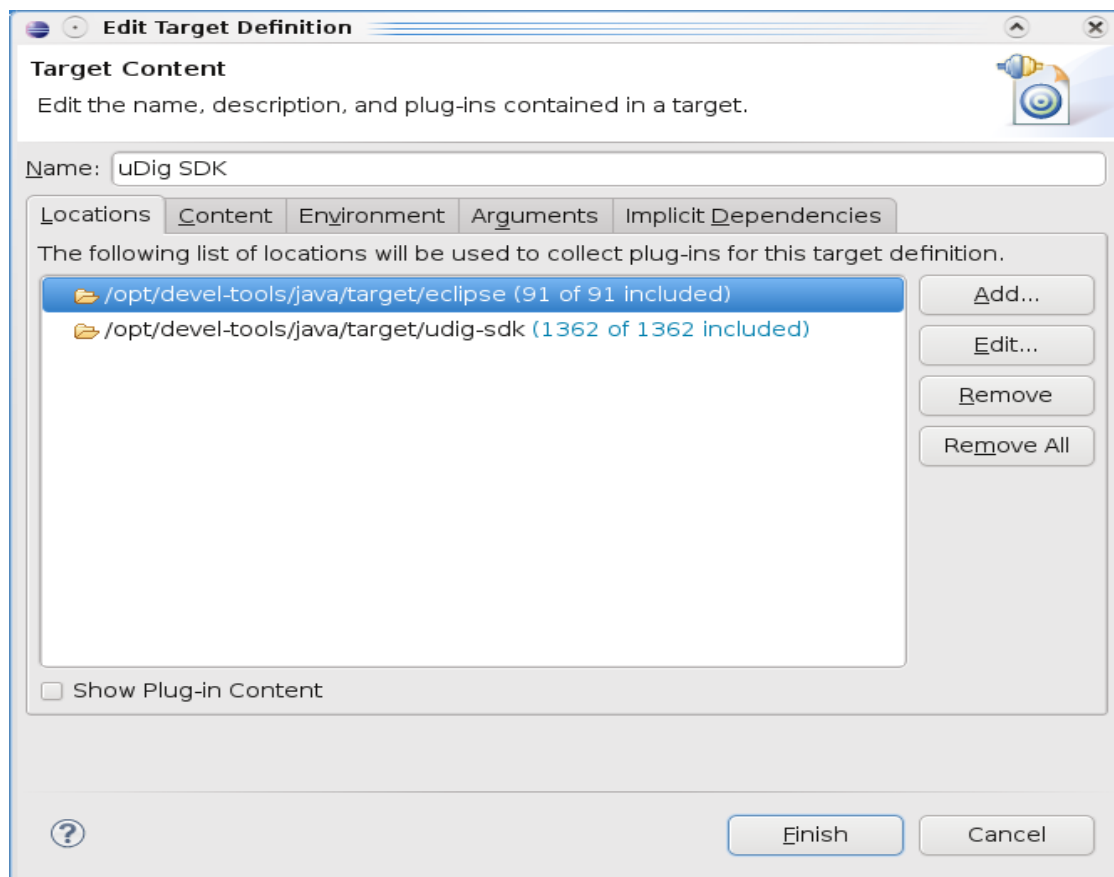
5. This document describes the most relevant function, but it does not provide details about some utility methods. Thus,

the source code for this tutorial is provided to avoid you to write that code. To accomplish this tutorial without pains, you can download the **axios-sextante plug-in sources** from the mentioned download page.

3.2 Adding the Spatial Operation Framework

Open the Eclipse Preferences Dialog: **Window > Preferences > Plug-in Development > Target Platform**.

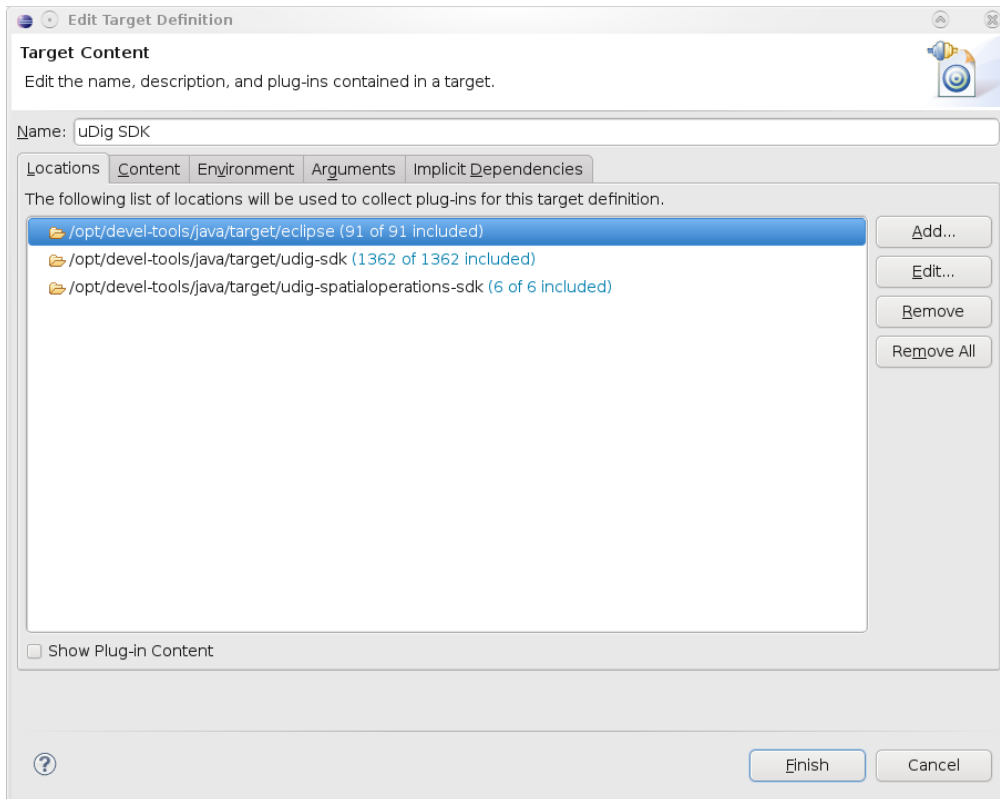
If you have followed the **uDig SDK Quick Start** document you should have **uDig SDK** as Target Platform, so select it and press the **Edit** button. In the **Locations** tab you should have the following target definition.



To add the Spatial Operations SDK, press the **Add** button, then select the directory, where it was installed. In this case:

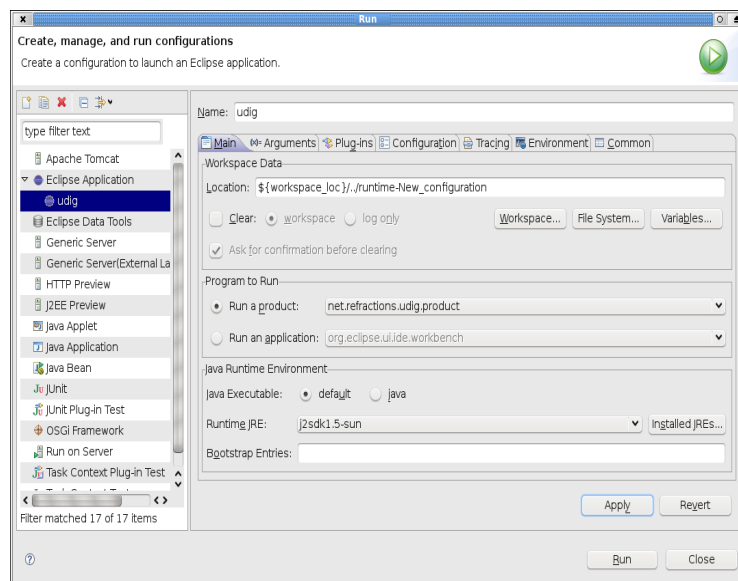
```
INSTALL_DIR/java/target/udig-spatialoperations-sdk
```

Now, you will get the following target configuration:

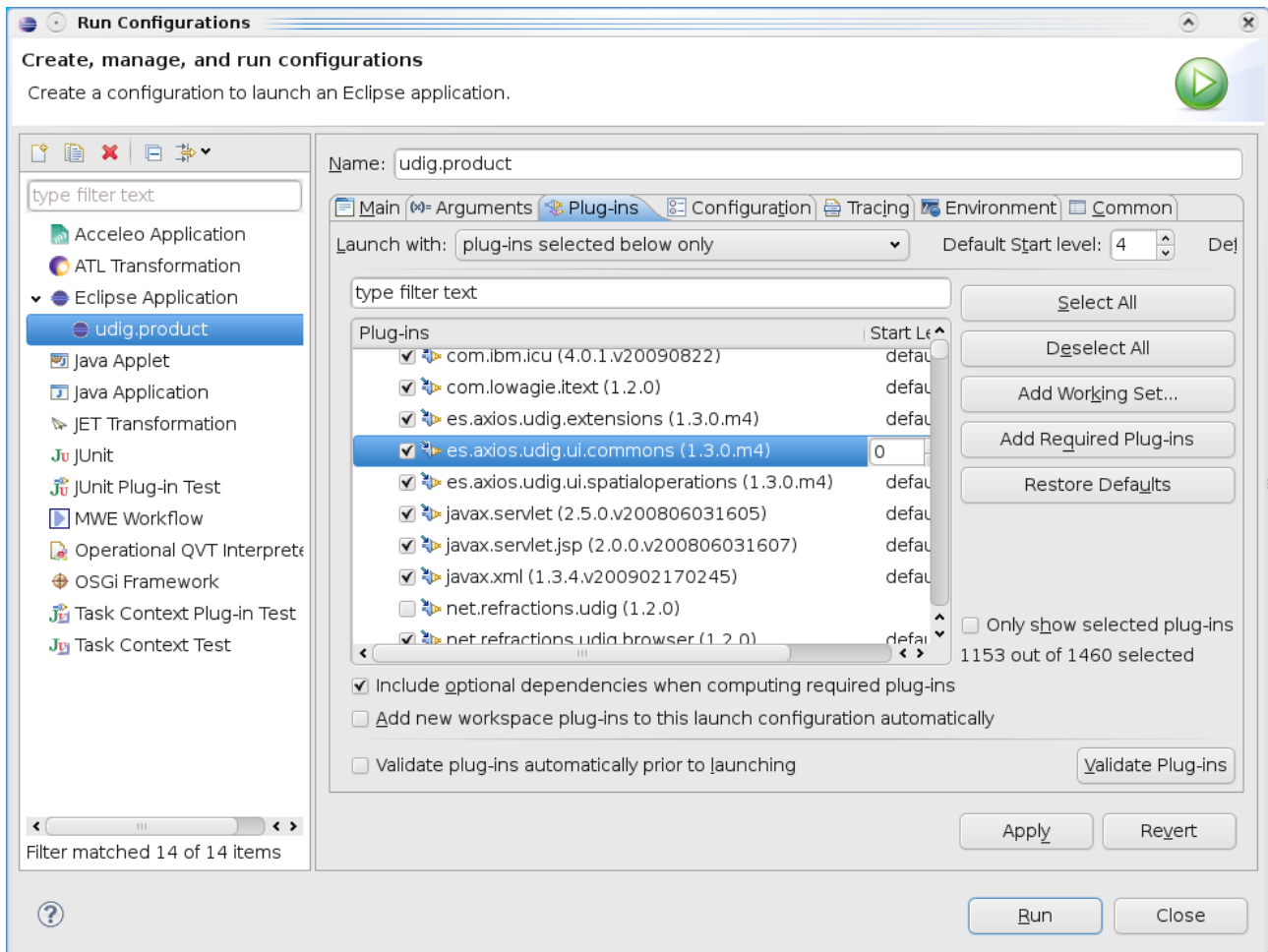


3.3 Testing the Development Environment

To test your environment you need to open the **Eclipse Run Dialog**, and create a run configuration in Eclipse Application. Be sure that the **Run a Product** radio button is checked and **net.refractions.udig.product** is selected.

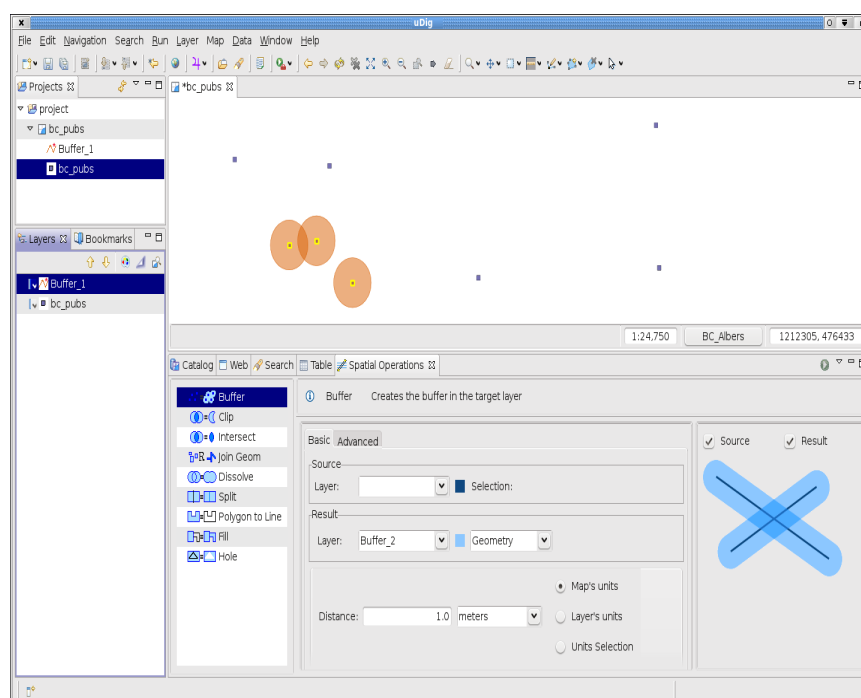
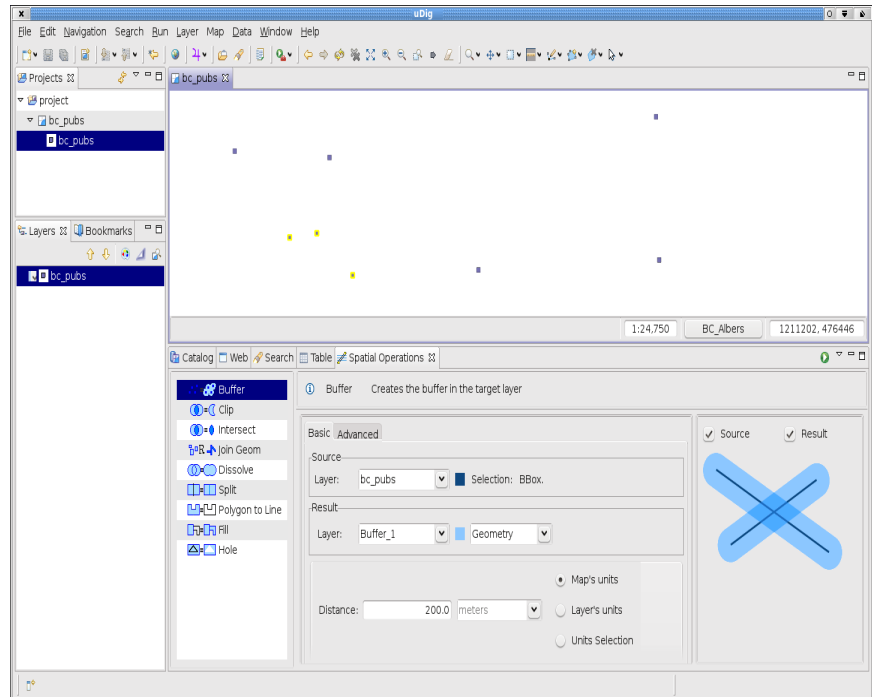


Be sure that the `es.axios.udig.*` plugins are checked in the **Plug-ins** tab in the uDig Run Configurations.



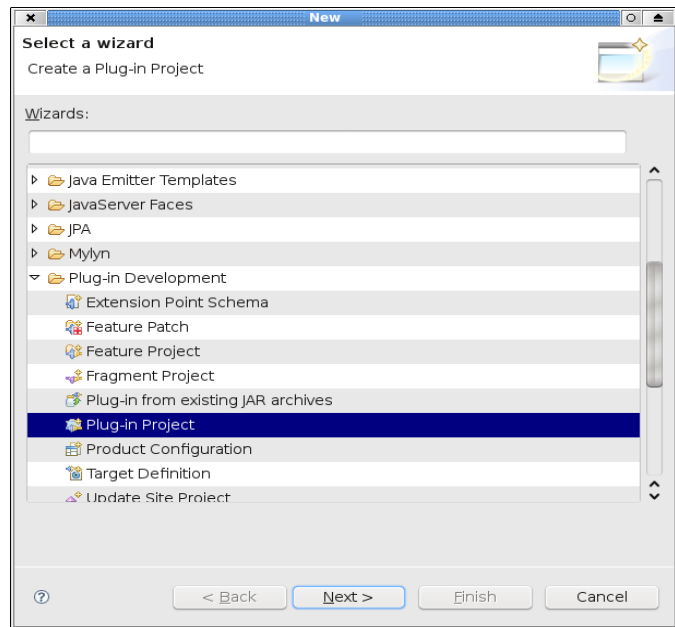
At this point everything is ready to test uDig with spatial operations.

1. Press the **Run** button. The uDig main window should be shown.
2. Add some layers to test your development environment with the default spatial operations. By example: **bc_pubs.shp**.
3. Select a layer.
4. Open the **pop up menu** (right click on **bc_pub** layer).
5. Select **Operations> Spatial Operations** to open the Spatial Operations View.
6. Finally, three Pubs are selected to execute the buffer operation. The result is shown in the next screenshot.

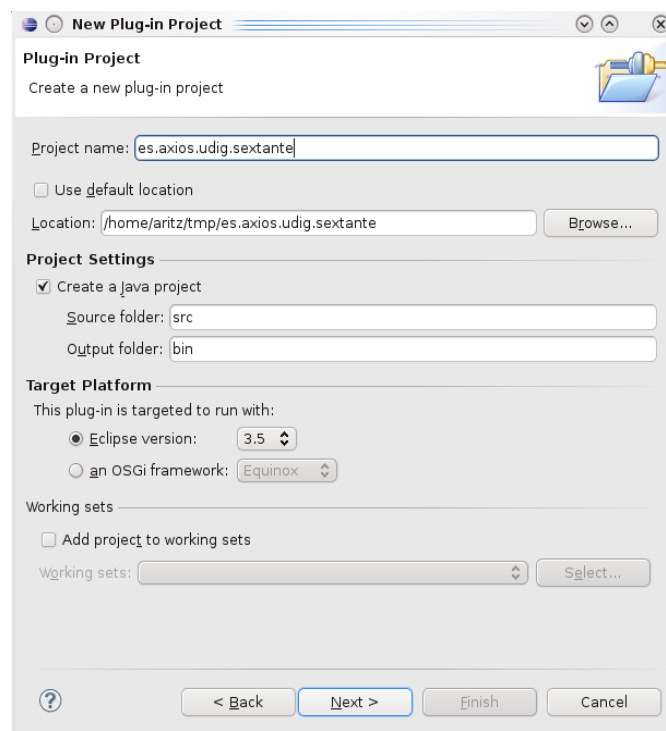


4 Creating a New Plugin Project

1. Our first step will be to create a new plug-in project using the Eclipse wizard. **File > New > Other > Plug in Project**



2. Write the project name: **es.axios.udig.sextante** in the next dialog.



3. In the next step it's important to select **No** in the **Rich Client Application** question.

4. Now, press the **Finish** button, so the project template for the new plug-in can be created.



New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID: es.axios.udig.sextante

Version: 1.0.0.qualifier

Name: Spatial operation with sextante

Provider:

Execution Environment: J2SE-1.5

Options

☒ Generate an activator, a Java class that controls the plug-in's life cycle

Activator: es.axios.udig.sextante.Activator

☒ This plug-in will make contributions to the UI

☐ Enable API Analysis

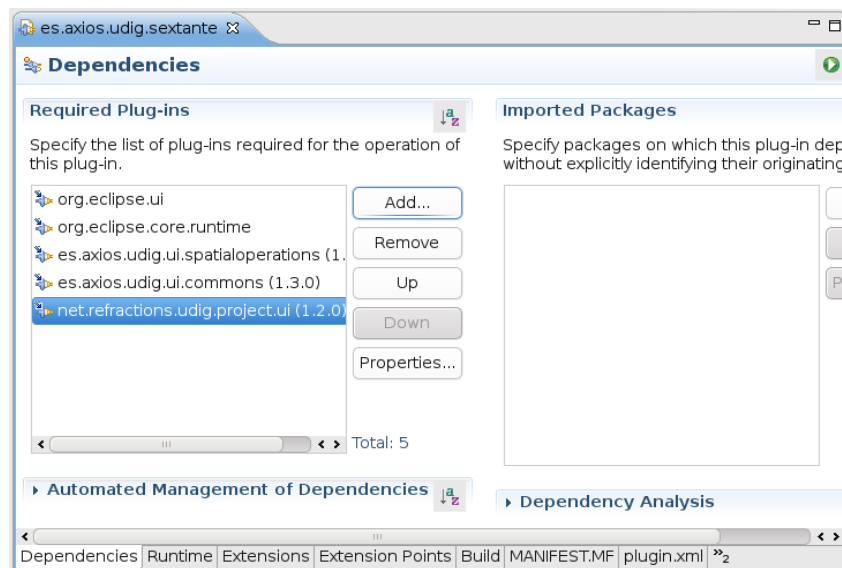
Rich Client Application

Would you like to create a rich client application? ☐ Yes ☒ No

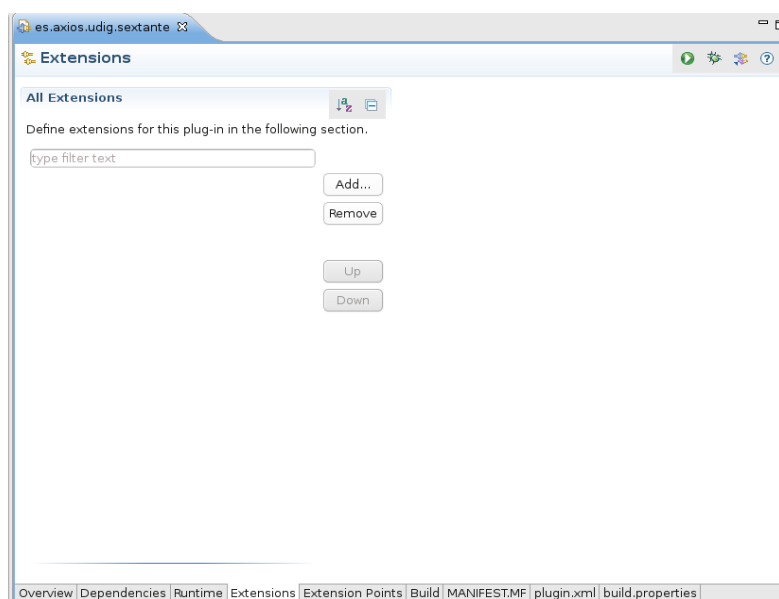
< Back Next > Finish Cancel

5 Extension for Spatial Operations

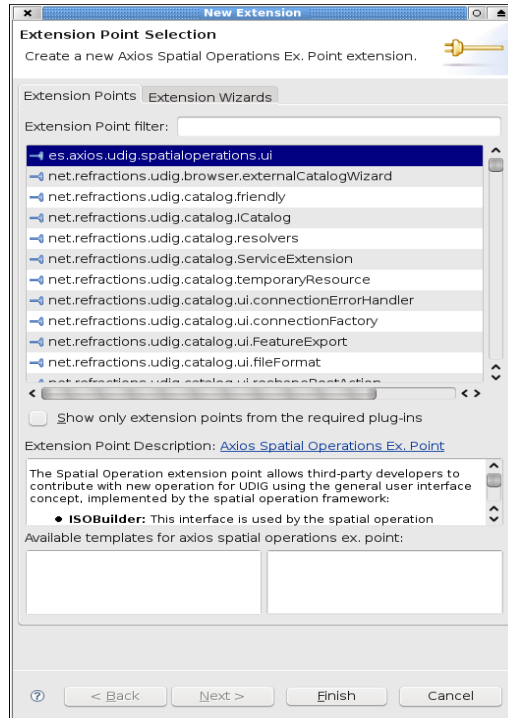
1. In order to define the new extension you must declare it in the **MANIFEST.MF**.
2. Switch to the **Dependencies** page and click on the **Add** button under Required Plug-ins panel.
3. Select the **es.axios.udig.ui.spatialoperations**, **es.axios.udig.ui.commons** and **net.refractions.udig.project.ui** plug-ins, then **save** the manifest.



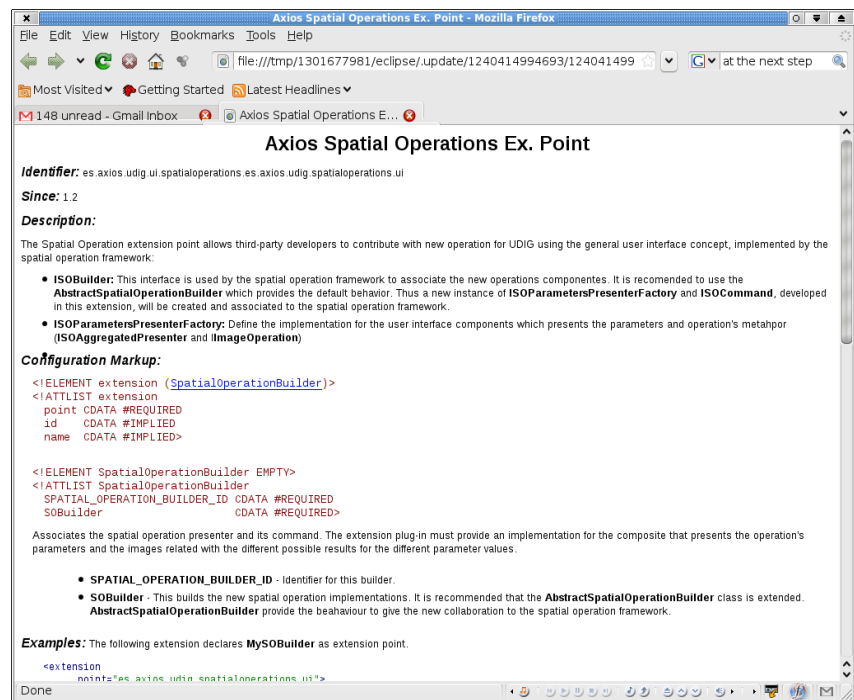
4. Select the **Extensions** page.



5. Press the **Add** button, and select the extension point **es.axios.udig.spatialoperations.ui**.



6. Additional information about this extension point is available in the **Extension Point Description** link shown in the previous image.



7. Finally, press the **finish** button and **save the MANIFEST.MF**

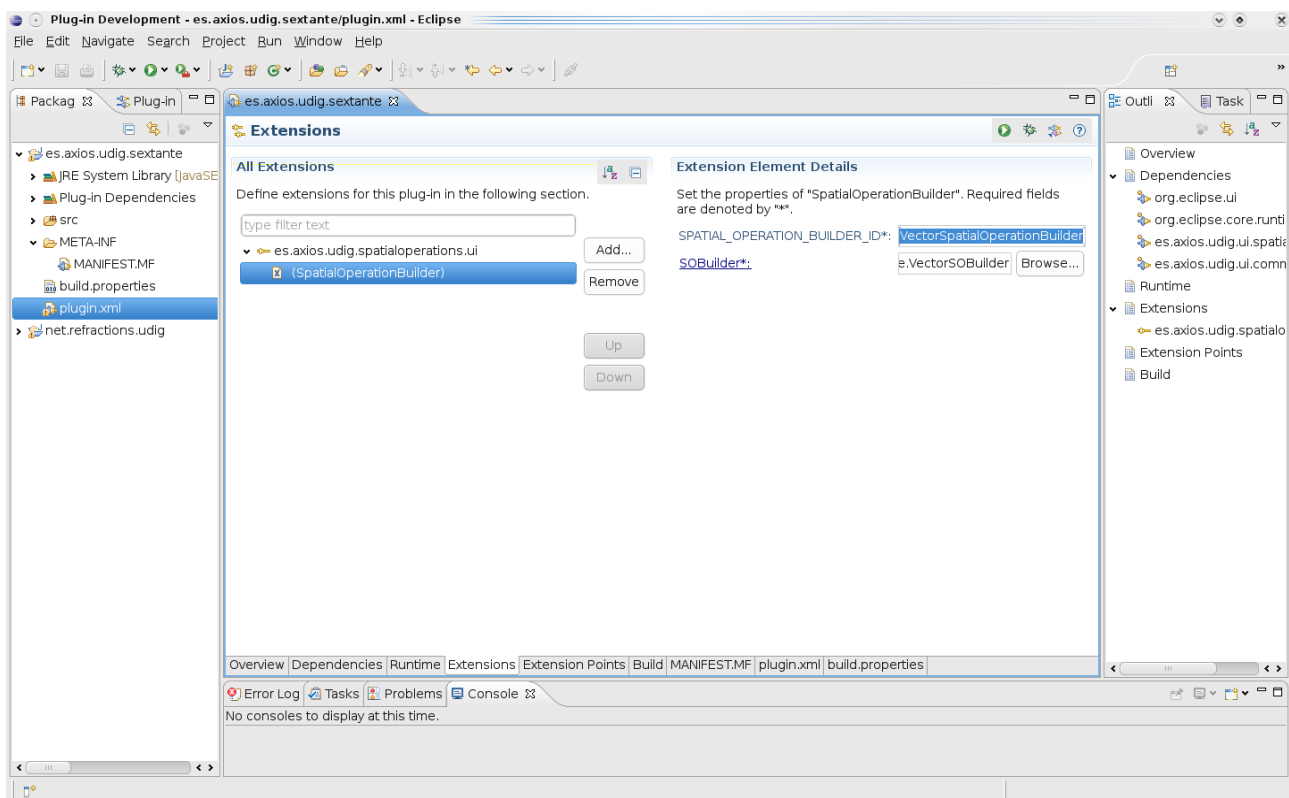
6 Developing the LineToPolygon Spatial Operation

Now, that the new extension was defined we are going to develop the new spatial operation presentation.

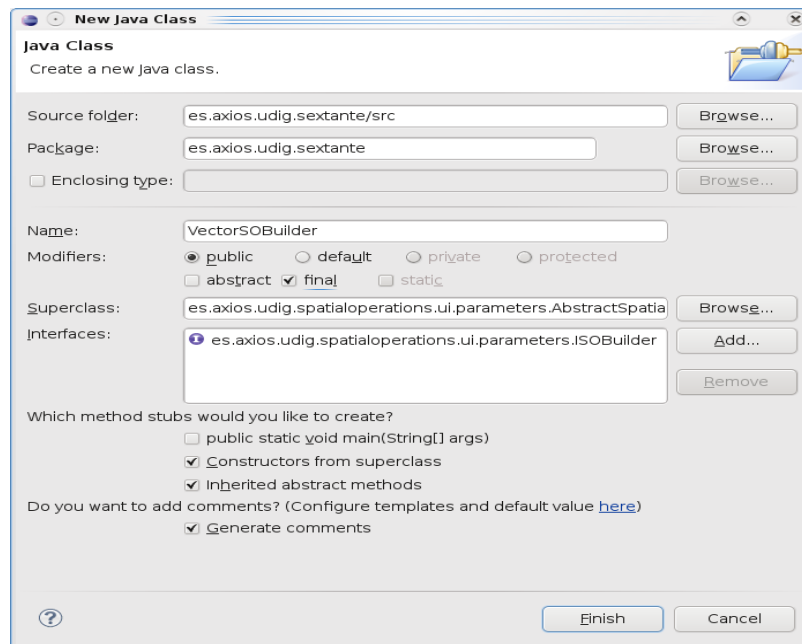
The framework requires an implementation for **ISOBuilder**. To add the implementation class follow these steps:

1. Open the plugin.xml and select the extension **es.axios.udig.spatialoperations.ui**.
2. Use the right button to open the pop up menu and select:
New > SpatialOperationBuilder
3. Select the **SpatialOperationBuilder** and fill in the right panel with the **ID** and the name of the implementation class for this builder. We typed in the ID field:

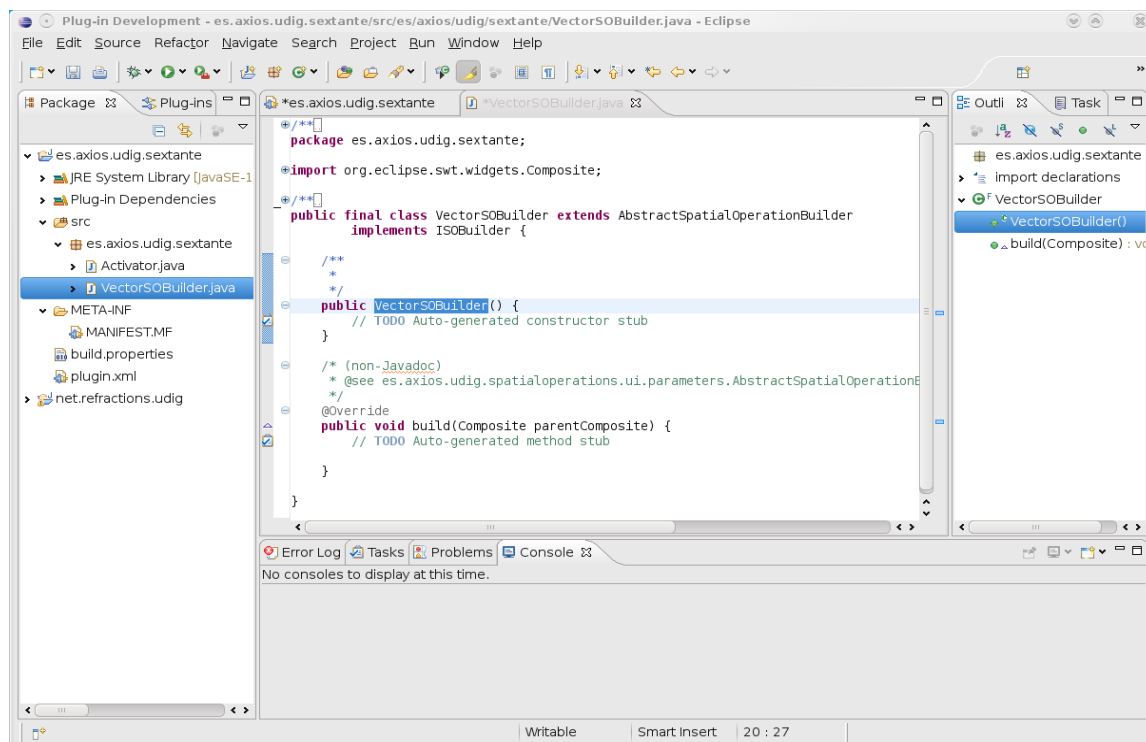
es.axios.udig.sextante.VectorSpatialOperationBuilder



- To add the builder implementation class, click on **S0Builder*** and fill the **New Java Class** dialog as it is shown in the next screen. We will take advantage of the **AbstractSpatialOperationBuilder** class declared as superclass of **SextanteS0Builder**, that implements a common behavior for this sort of builders.



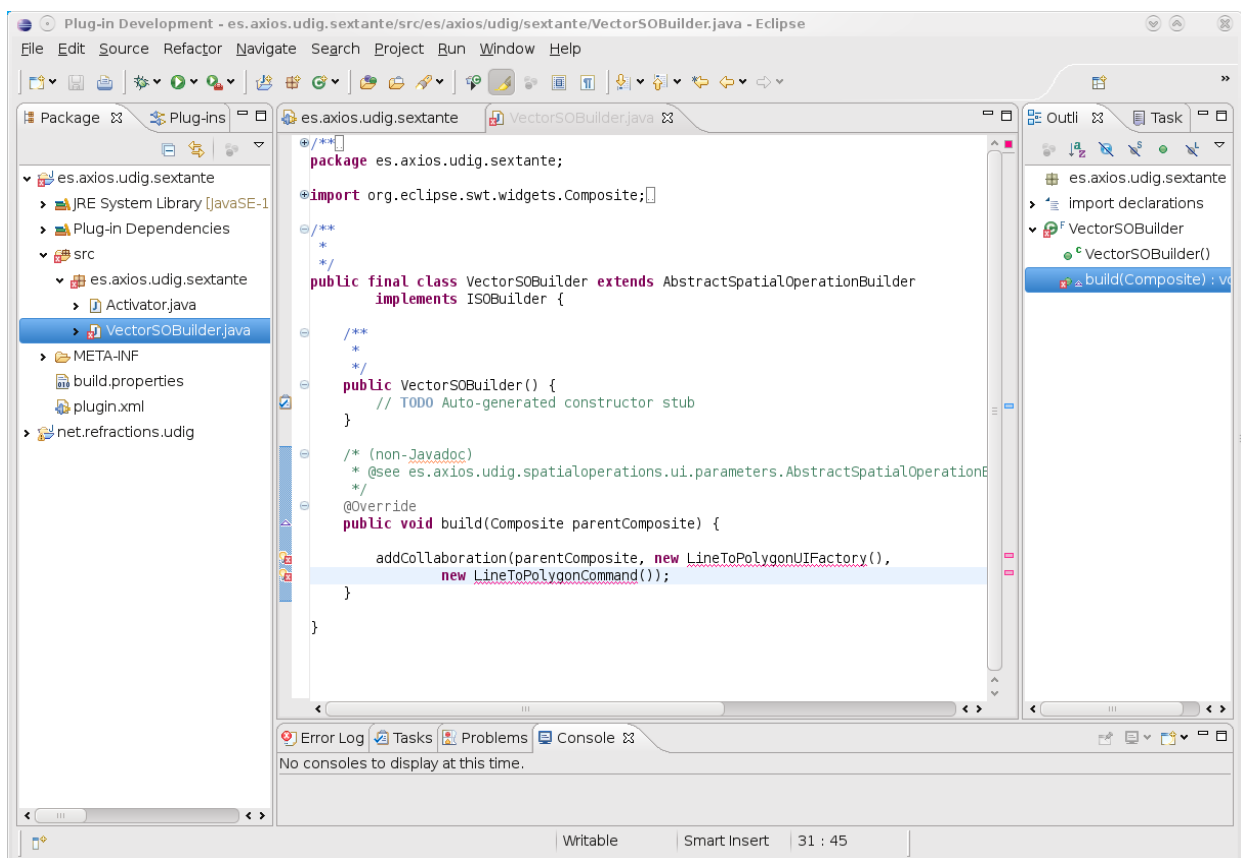
- The wizard will create the new Builder class, **SextanteS0Builder**.



7 Writing the LineToPolygon Operation

In the previous section we have created the **SextanteSOBuilder** class which provides the user interface components required by the framework. In this section we are going to develop the user interface for the **LineToPolygon** operation with the help of **SimplePresenter**.

1. In order to add the new spatial operation you should use the build method to create the component required by the spatial operations framework. The **AbstractSpatialOperationBuilder** provides the **addCollaboration** default method which can be used in the **build()** method to create one or more collaborations. We need to create only the **LineToPolygon** spatial operation.

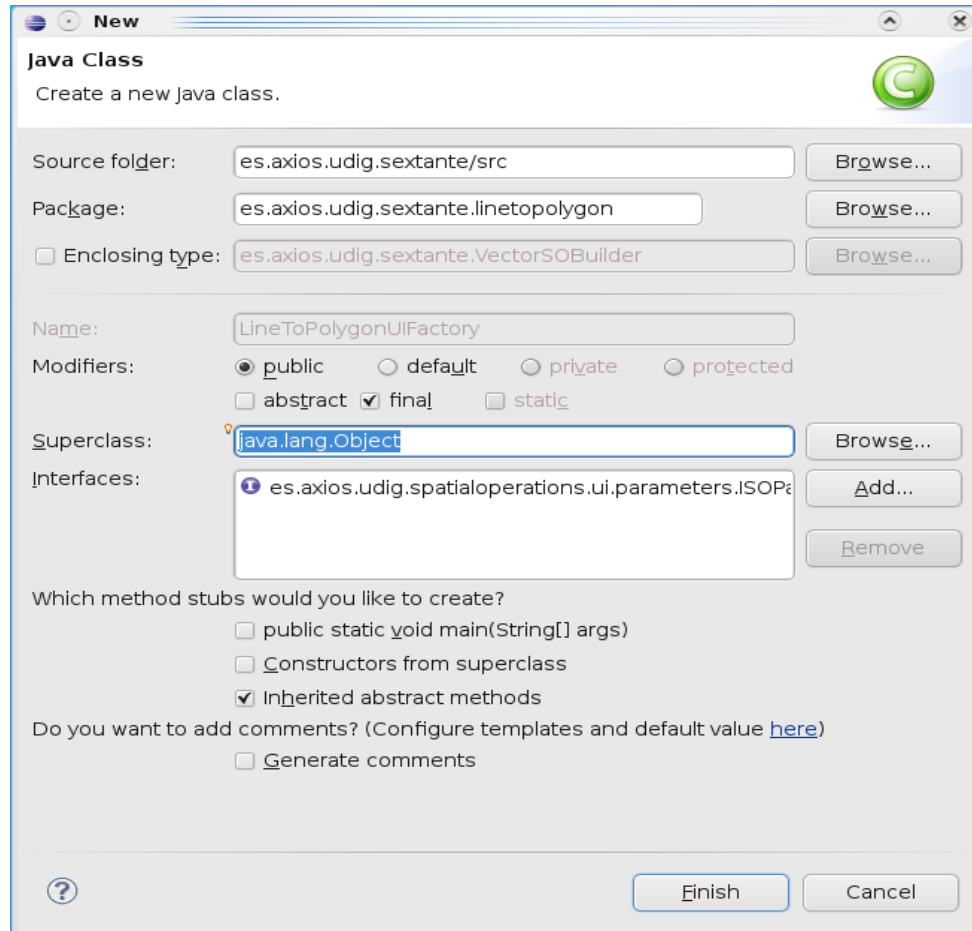


```

@Override
public void build(Composite parentComposite) {
    addCollaboration(parentComposite, new LineToPolygonUIFactory(),
        new LineToPolygonCommand());
}

```

2. Create an implementation class called **LineToPolygonUIFactory** into the **es.axios.udig.sextante.linetopolygon** package, that implements the **ISOParametersPresenterFactory** interface.



3. The framework will get this implementation to compose the **LineToPolygon** spatial operation in the spatial operations view. The factory requires an implementation to get the spatial operation parameters (**createDataComposite**);

```
public ISOAggregatedPresenter createDataComposite(
    ScrolledComposite dataParent, int style) {
    return return new SOLineToPolygonComposite(dataParent, style, cmd);
}
```

the command used on the composite and its method;

```
private ISOCommand cmd = null;
public void setCommand(ISOCommand cmd) {
    assert cmd != null : "Can't be null!"; //$NON-NLS-1$
    this.cmd = cmd;
}
```

the demo images (**createDemoImages**);

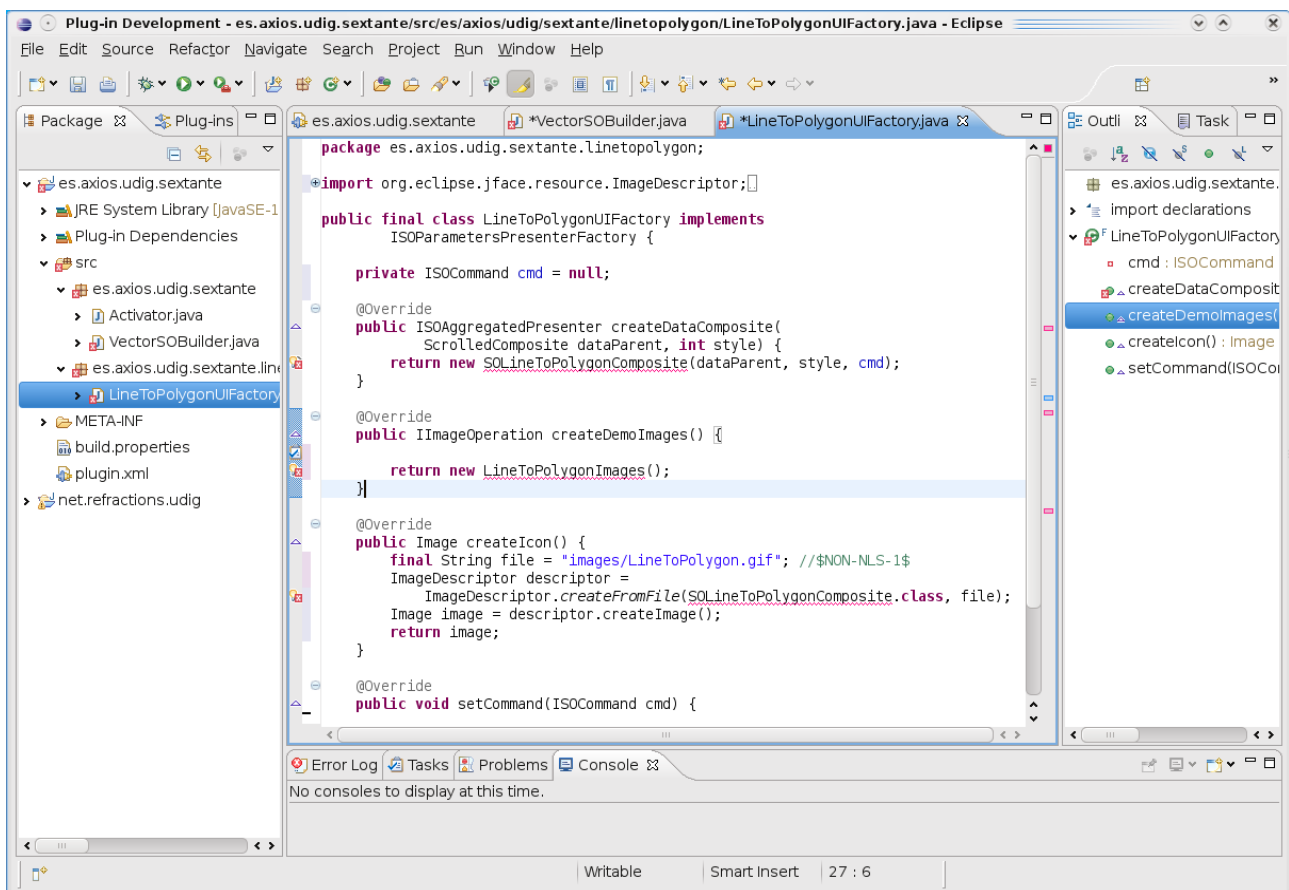
```
public IImageOperation createDemoImages() {
    return new LineToPolygonImages();
}
```

and the icon for the Centroid operation (**createIcon**).

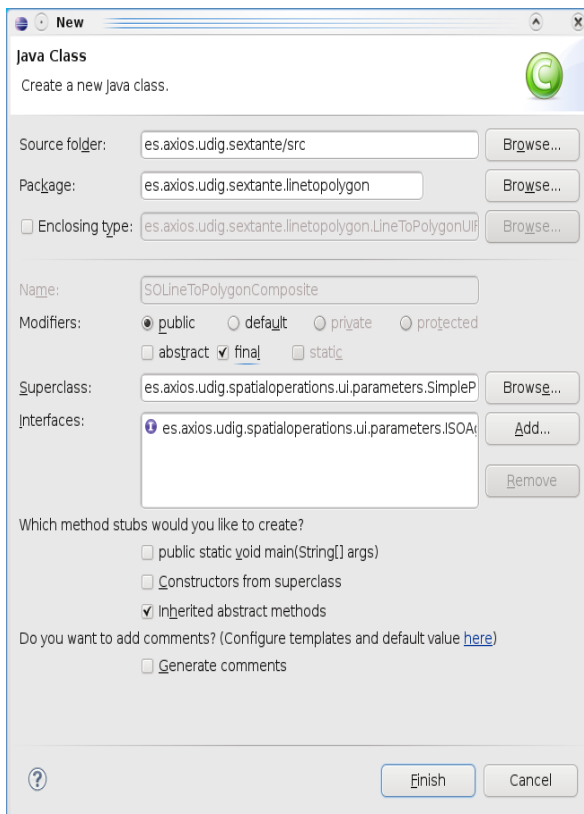
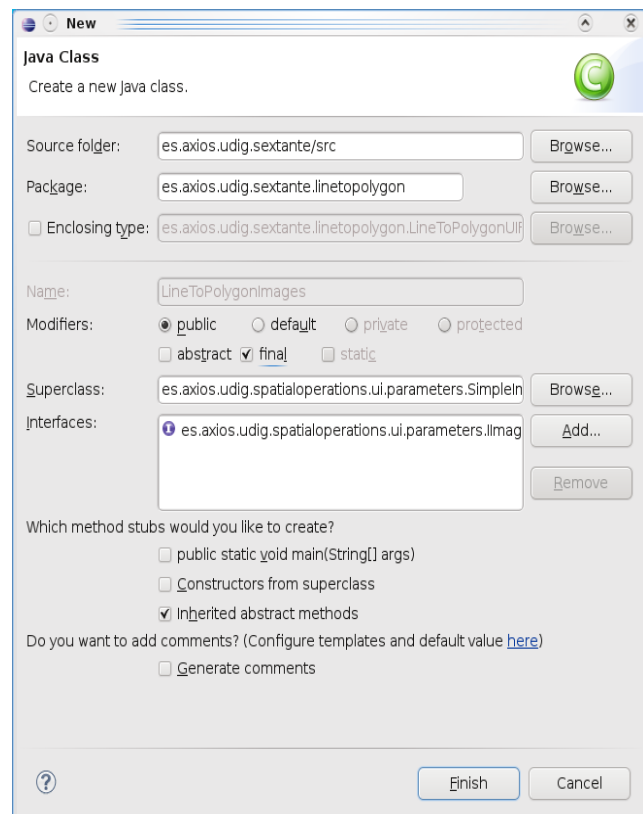
```
public Image createIcon() {
    final String file = "images/LineToPolygon.gif"; //$NON-NLS-1$
    ImageDescriptor descriptor = ImageDescriptor.
        createFromFile(SOLineToPolygonComposite.class, file);
    Image image = descriptor.createImage();
    return image;
}
```

Finally add this imports:

```
import org.eclipse.jface.resource.ImageDescriptor;
```

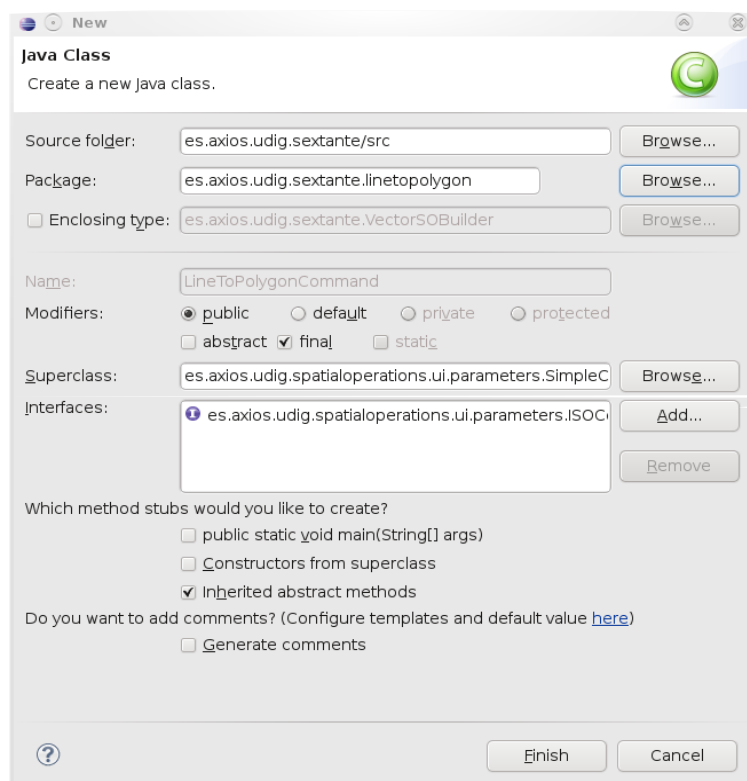


4. We have to create the **SOLineToPolygonComposite** (as subclass of **SimplePresenter**) and **LineToPolygonImages** (as subclass of **SimpleImages**) classes into the same package.

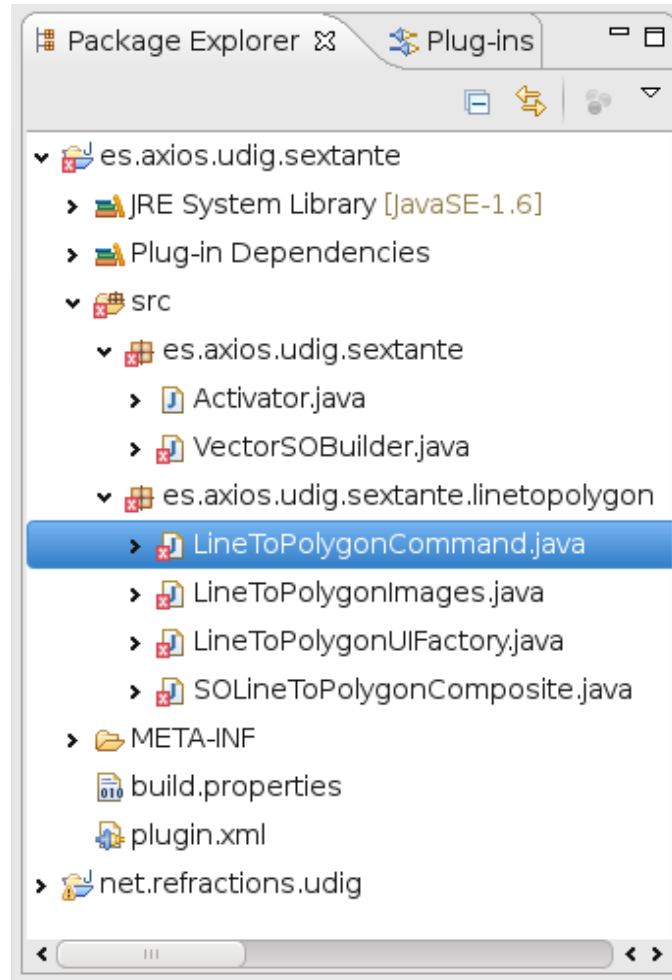



5. Then we will create the **LineToPolygonCommand** class, which is a subclass of **SimpleCommand**. The **SimpleCommand** class is a default implementation for common spatial operation cases, it will help saving programming time.

6. Fill the fields and press **Finish**. In the class editor you can add the method required by the interface. We are going to program this class later.



7. Let's check that you have the following package structure:



In the next sections we will write the code for the created classes.

7.1 *SOLineToPolygonComposite class*

This class is responsible of presenting the widgets that will be used to provide the parameter values for the operation. The following image shows the idea.

The constructor must call the **initialize()** method defined in **AbstractParamsPresenter** class. Thus the framework will



execute the operations to create the composite widgets, populate with the initial parameter values, and add some common listeners for spatial operations (map listener and layer listener).

But, we will take the advantage of the **SimplePresenter**, **SimpleCommand**, etc superclasses. It's only needed to define the constructor.

```
public S0LineToPolygonComposite(Composite parent, int style, ISOCommand cmd) {  
    super(parent, style, cmd);  
}
```

The **SimplePresenter** is responsible of the presentation, its methods can be redefined and its fields are protected so it could be used by the developers.

7.2 *LineToPolygonImages class*

This presentation is very simple, it is responsible of displaying the demo images. To take advantage of framework abstract implementation this class will extend from **SimpleImages**. The most important methods are **getDefaultImage()**, which is called when the default demo image is presented, and the **getImage(...)**, that is invoked when the user requires the source image, the result image or the **LineToPolygon** operation final image. Those methods are implemented on the superclass, so the developers only have to define the name of the images and call the superconstructor.

The code is shown below.

```
private static final String IMAGE_SOURCE = "source"; //$NON-NLS-1$  
private static final String IMAGE_RESULT = "result"; //$NON-NLS-1$  
private static final String IMAGE_FINAL = "final"; //$NON-NLS-1$  
  
public LineToPolygonImages() {  
  
    super(LineToPolygonImages.class, IMAGE_SOURCE, IMAGE_RESULT,  
        IMAGE_FINAL);  
}
```

Finally, it's necessary to provide the images to display. You can copy all images required for this tutorial from

```
es.axios.udig.sextante/src/es/axios/udig/sextante/linetopolygon/images
```

provided in the sources file. These files should be in the

es.axios.udig.sextante.linetopolygon.images package, so you will need to create it.

7.3 *LineToPolygonCommand* class

The **LineToPolygonCommand** class extends the **SimpleCommand** abstract class. This **SimpleCommand** is responsible of executing the task, store the parameters and verify them. Also, it has the methods to populate the **SimplePresenter** but we will see this later.

The developers will define on this command the “polygon to line” description shown on the information panel and the method to execute the Sextante task. Moreover, they could redefine some methods, for example, to validate additional parameters, to restrict the geometry types of the layers, etc.

First, we will create the constructor and call the superclass constructor. We need to declare an initial message describing the operation, as is done below:

```
protected static InfoMessage INITIAL_MESSAGE = new InfoMessage("Convert  
closed lines into polygons.", InfoMessage.Type.IMPORTANT_INFO);  
  
public LineToPolygonCommand() {  
    super(INITIAL_MESSAGE);  
}
```

The abstract methods from the **SimpleCommand** forces the developers to set the values which will be presented on the **SOLineToPolygonComposite** that defines this command. Use the fields hosted by the **SimpleCommand** to fill that methods.

```
@Override  
public void setGroupSourceText() {  
    this.groupSourceText = "Source";  
}  
  
@Override  
public void setGroupTargetInputsText() {  
    this.groupTargetInputsText = "Result";  
}  
  
@Override  
public void setLabelSourceLayerText() {  
    this.labelSourceLayerText = "Source";  
}  
  
@Override
```



```
public void setTabItemAdvancedText() {  
    this.tabItemAdvancedText = "Advanced";  
}  
  
@Override  
public void setTabItemBasicText() {  
    this.tabItemBasicText = "Basic";  
}  
  
@Override  
public void setTargetLabelText() {  
    this.targetLabelText = "Layer";  
}  
  
@Override  
public void setTargetLabelToolTipText() {  
    this.targetLabelToolTipText = "The layer where the result will go. Select  
an existent layer or write a new one.";  
}
```

The framework requires an **ID**, **Name** and **tooltip** text for each spatial operation, this is the purpose of the following methods:

```
@Override  
public void setOperationID() {  
    this.operationID = "LineToPolygon";  
}  
  
@Override  
public void setOperationName() {  
    this.operationName = "LineToPolygon";  
}  
  
@Override  
public void setToolTipText() {  
    this.tooltipText = "Convert lines to polygon.";  
}
```

The valid geometries for the **LineToPolygon** result layer are Polygon or MultiPolygon, they should be provided by the **getResultLayerGeometry** method. Redefine it:

```
@Override
protected Object[] getResultLayerGeometry() {
    Object[] obj;
    if (sourceLayer != null) {
        obj = new Object[1];
        Class<? extends Geometry> sourceGeom =
            LayerUtil.getGeometryClass(this.sourceLayer);
        // is a geometry collection
        if (sourceGeom.getSuperclass().equals(GeometryCollection.class)) {
            obj[0] = MultiPolygon.class;
        } else {
            obj[0] = Polygon.class;
        }
    } else {
        obj = new Object[2];
        obj[0] = Polygon.class;
        obj[1] = MultiPolygon.class;
    }
    return obj;
}
```

The required geometries for the source layer are: line or multiLine. The **getSourceGeometryClass()** is responsible of providing this information to the user interface.

```
@Override
protected Object[] getSourceGeometryClass() {
    Object[] obj = new Object[] {
        LineString.class,
        MultiLineString.class };
    return obj;
}
```

The framework will call the **validateParameters** method in order to check if the parameter values are OK before executing the spatial operation. The **SimpleCommand** has a **validateParameters** which is designed for validate the parameters from the **SimplePresenter**, if the developers wish to validate more information, they just need to override the method.

Note: you can see the details of each private method in the source code provided with this tutorial.

When the **validateParameters()** predicate returns **true**, the framework enables the **execute** button which allows executing the operation. We will override the **executeOperation()** to create and execute the **LineToPolygon** task.

```
@Override
public void executeOperation() throws SOCommandException {
    final NullProgressMonitor progress = new NullProgressMonitor();
    this.map = sourceLayer.getMap();
    final FeatureStore<SimpleFeatureType, SimpleFeature> source =
        getFeatureStore(sourceLayer);
    try {
        progress.setTaskName("LinesToPolygon Spatial Operation");
        String msg = MessageFormat.format("Doing lines to polygon of {0}",
            sourceLayer.getName());
        progress.beginTask(msg, IProgressMonitor.UNKNOWN);
        IRunnableWithProgress runner = new IRunnableWithProgress() {
            public void run(IProgressMonitor monitor) throws
                InvocationTargetException,
                InterruptedException {
                final FeatureStore<SimpleFeatureType, SimpleFeature>
                    resultStore;

                LineToPolygonTask task = new LineToPolygonTask(source);
                try {
                    task.run();
                    resultStore = task.getResult();
                    addFeaturesToTargetStore(resultStore);
                } catch (GeoAlgorithmExecutionException e){
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (IOException ioe){
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (SOCommandException soe){
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } finally {
                    this.map.getRenderManager().refresh(
                        this.targetLayer,
                        this.map.getBounds(progress
                            progress.done());
                }
            }
        };
    }
```

```

    };
    DialogUtil.runInProgressDialog("Executing LinesToPolygon Operation",
        true, runner, true, true);
} catch (Exception e) {
    throw new SOCommandException(e.getMessage());
}
}

private final FeatureStore<SimpleFeatureType, SimpleFeature> getFeatureStore(final
    ILayer layer) throws SOCommandException {

    assert layer != null;
    IGeoResource resource = layer.getGeoResource();
    if (resource == null) {
        throw new SOCommandException("The layer does not have GeoResource");
    }
    FeatureStore<SimpleFeatureType, SimpleFeature> store;
    try {
        store = resource.resolve(FeatureStore.class, new NullProgressMonitor());
        return store;
    } catch (IOException e) {
        throw new SOCommandException(e.getMessage());
    }
}

private final void addFeaturesToTargetStore(FeatureStore<SimpleFeatureType,
    SimpleFeature> resultStore)
    throws SOCommandException, IOException {
    FeatureStore<SimpleFeatureType, SimpleFeature> emptyTargetStore = null;
    if (this.targetLayer != null) {
        emptyTargetStore = this.getFeatureStore(this.targetLayer);
    } else { // create a new layer
        this.targetLayer = createNewLayer();
        IGeoResource targetGeoResource = targetLayer.getGeoResource();
        try {
            emptyTargetStore =
                targetGeoResource.resolve(FeatureStore.class, null);
        } catch (IOException e) {
            throw new SOCommandException(e.getMessage());
        }
    }
    Transaction transactionOld =
        ((net.refractions.udig.project.internal.Map) this.map)

```

```

        .getEditManagerInternal().getTransaction();
        S0FeatureStore soStore = new S0FeatureStore(emptyTargetStore,
        transactionOld);

        // add to target store.
        addToStore(soStore, resultStore);
    }

    private void addToStore(S0FeatureStore soStore, FeatureStore<SimpleFeatureType,
        SimpleFeature> resultStore)
        throws IOException {
        FeatureCollection<SimpleFeatureType, SimpleFeature> collection =
            resultStore.getFeatures();

        FeatureIterator<SimpleFeature> iter = collection.features();
        SimpleFeatureType newFeatureType = soStore.getSchema();
        Transaction transaction = soStore.getTransaction();
        try {
            while (iter.hasNext()) {
                SimpleFeature sextanteFeature = iter.next();
                SimpleFeature featureToAdd =
                    FeatureUtil.createFeatureUsing(
                        sextanteFeature, newFeatureType,
                        (Geometry) sextanteFeature.getDefaultGeometry());
                soStore.addFeatures(
                    DataUtilities.collection(
                        new SimpleFeature[] { featureToAdd }));
                transaction.commit();
            }
        } catch (Exception ex) {
            transaction.rollback();
            throw new IOException(ex.getMessage());
        } finally {
            collection.close(iter);
            transaction.close();
        }
    }

    private final CoordinateReferenceSystem getTargetCRS() {
        if (this.targetLayer != null) {
            return this.targetLayer.getCRS();
        } else {
            return getMapCRS();
        }
    }

```

```

    }
}

private CoordinateReferenceSystem getMapCRS() {
    return MapUtil.getCRS(map);
}

private final ILayer createNewLayer()
    throws SOCommandException {

    FeatureStore<SimpleFeatureType, SimpleFeature> targetStore;
    try {
        SimpleFeatureType type = FeatureUtil.createFeatureType(
            this.sourceLayer.getSchema(), targetLayerName,
            getTargetCRS(), this.targetGeometryClass);

        IGeoResource targetGeoResource =
            AppGISMediator.createTempGeoResource(type);
        assert targetGeoResource != null;
        targetStore = targetGeoResource.resolve(FeatureStore.class, null);
        assert targetStore != null;
        ILayer newLayer = MapUtil.addLayerToMap(
            (IMap) this.map, targetGeoResource);

        return newLayer;
    } catch (Exception e) {
        e.printStackTrace();
        throw new SOCommandException(e);
    }
}

```

To solve the compilation errors, write the following import declarations: (Note: not all the errors will be corrected yet)

```

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.text.MessageFormat;

import net.refractions.udig.catalog.IGeoResource;
import net.refractions.udig.project.ILayer;
import net.refractions.udig.project.IMap;

```

```
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.NullProgressMonitor;
import org.eclipse.jface.operation.IRunnableWithProgress;
import org.geotools.data.DataUtilities;
import org.geotools.data.FeatureStore;
import org.geotools.data.Transaction;
import org.geotools.feature.FeatureCollection;
import org.geotools.feature.FeatureIterator;
import org.opengis.feature.simple.SimpleFeature;
import org.opengis.feature.simple.SimpleFeatureType;
import org.opengis.referencing.crs.CoordinateReferenceSystem;

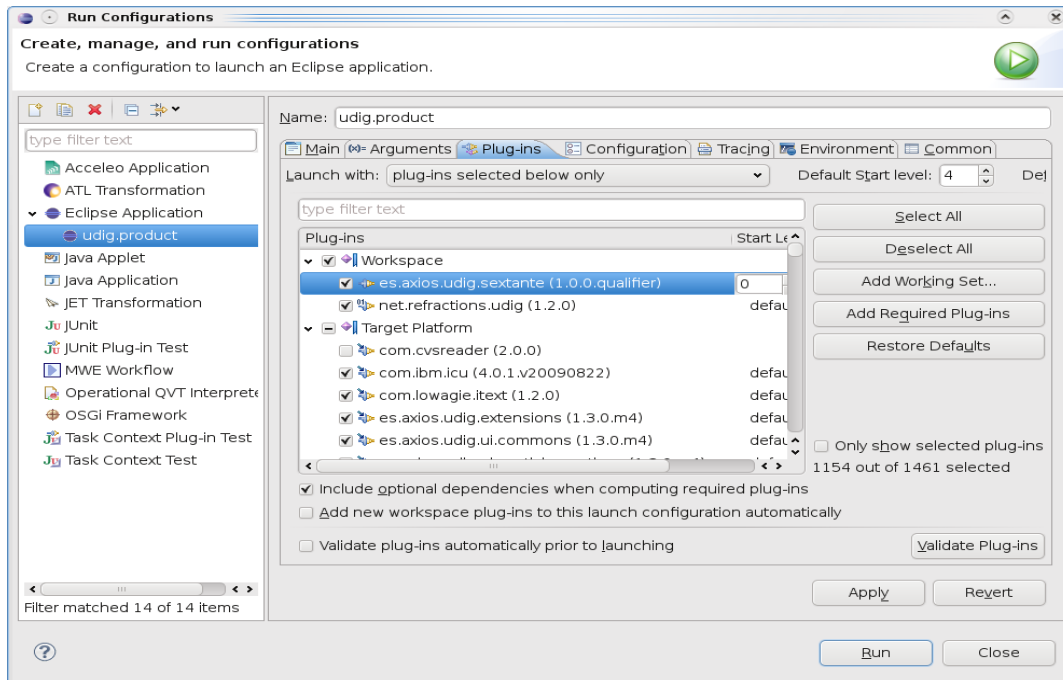
import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.geom.GeometryCollection;
import com.vividsolutions.jts.geom.LineString;
import com.vividsolutions.jts.geom.MultiLineString;
import com.vividsolutions.jts.geom.MultiPolygon;
import com.vividsolutions.jts.geom.Polygon;

import es.axios.udig.sextante.task.LineToPolygonTask;
import es.axios.udig.spatialoperations.ui.parameters.S0CommandException;
import es.axios.udig.spatialoperations.ui.parameters.SimpleCommand;
import es.axios.udig.spatialoperations.ui.taskmanager.S0FeatureStore;
import es.axios.udig.ui.commons.mediator.AppGISMediator;
import es.axios.udig.ui.commons.message.InfoMessage;
import es.axios.udig.ui.commons.util.DialogUtil;
import es.axios.udig.ui.commons.util.FeatureUtil;
import es.axios.udig.ui.commons.util.LayerUtil;
import es.axios.udig.ui.commons.util.MapUtil;
import es.unex.sextante.exceptions.GeoAlgorithmExecutionException;
```

It could be a good idea to try the **LineToPolygon** user interface before writing the **LineToPolygonTask**. Thus, you should comment imports that aren't compiled and the previous code from the `executeOperation` and put this instead:

```
System.out.println("LineToPolygon Task should be implemented");
```

Before executing the **uDig product**, have a look in the **Run Dialog** to be sure that the **es.axios.udig.sextante** plug-in is checked to launch.



After opening the spatial operations view you should see the new **LineToPolygon** operation.

Note: don't forget to revert this hack before continuing this tutorial.

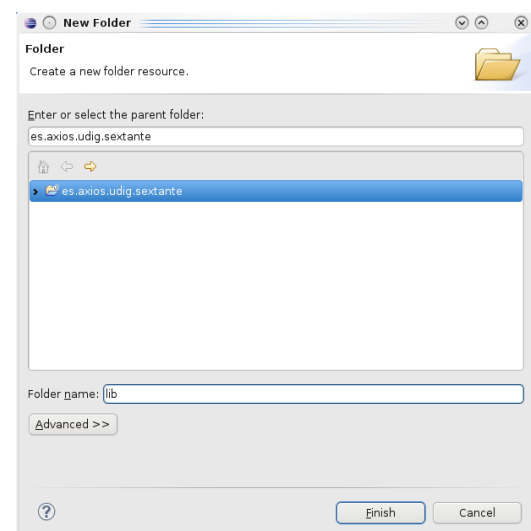
7.4 Adding Sextante

Before starting with the **LineToPolygonTask**, we are going to configure the **plugin.xml**. First we create a folder called "lib" where the .jars will be stored.

Right click on **es.axios.udig.sextante** > New > Folder.

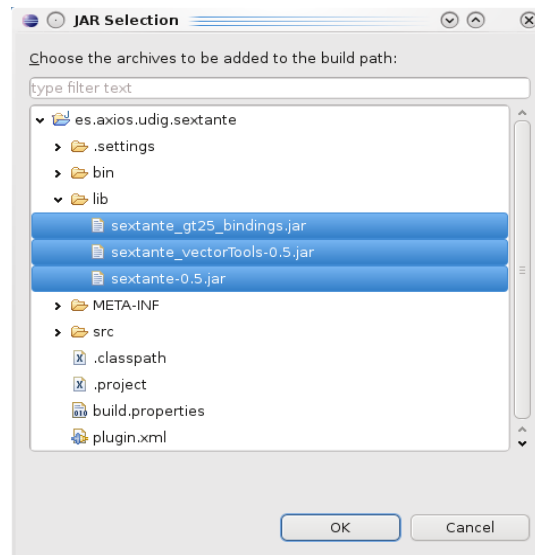
After creating the folder, we will put in the next .jars:

- sextante-0.5.jar
- sextante_gt25_bindings.jar
- sextante_vectorTools-0.5.jar

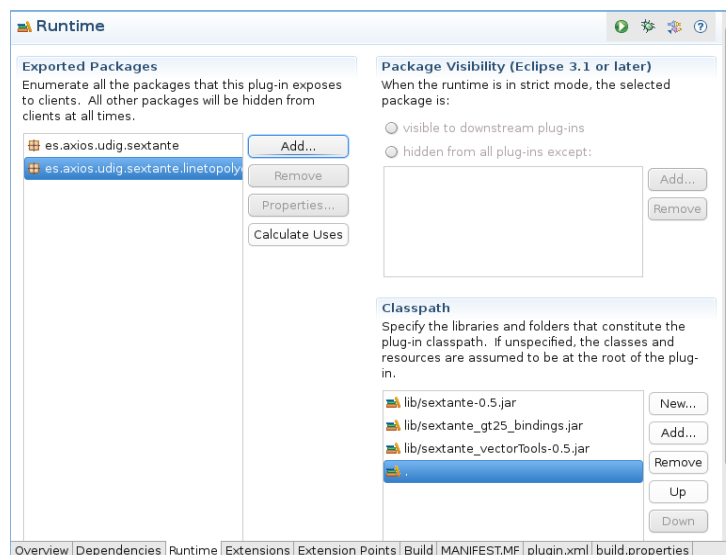


These jars are provided with the source code in the folder:
es.axios.udig.sextante/lib

After refreshing the workspace, we are going to add the jars to the project. Right click on **es.axios.udig.sextante** > Properties > Java Build Path > Libraries tab > Add Jars... And then select these 3 jars.



Go to the plugin.xml "Runtime" tab and add these jars and save it. Also, don't forget to add the "." as it's shown on the image. For adding the ".", click on New... and write it.



7.5 LineToPolygonTask class

Finally, it's time to provide the **LineToPolygonTask**. This task will call **Sextante** algorithm **PolyLinesToPolygonsAlgorithm**. To simplify this tutorial we will create a simple class. Interested people could have a look at the default spatial operation model

(buffer, intersect, clip, etc).

First, create the package where the **LineToPolygonTask** will be:

es.axios.udig.sextante.task

After that, you can start to develop the **LineToPolygonTask**.

```
public final class LineToPolygonTask {

    private final FeatureStore<SimpleFeatureType, SimpleFeature> sourceStore;
    private FeatureStore<SimpleFeatureType, SimpleFeature> targetStore = null;

    public LineToPolygonTask(
        final FeatureStore<SimpleFeatureType, SimpleFeature> sourceStore) {
        assert sourceStore != null :
            "Illegal argument. Expects sourceStore != null";
        this.sourceStore = sourceStore;
    }

    public void run() throws GeoAlgorithmExecutionException, IOException {
        PolylinesToPolygonsAlgorithm alg = new PolylinesToPolygonsAlgorithm();
        // set the inputs.
        DataStore ds = (DataStore) sourceStore.getDataStore();
        GTVectorLayer layer = GTVectorLayer.createLayer(ds, ds.getNames()
            .get(0).getLocalPart());
        ParametersSet params = alg.getParameters();
        params.getParameter(PolylinesToPolygonsAlgorithm.LAYER)
            .setParameterValue(layer);
        // set the outputs.
        OutputFactory outputFactory = new GTOutputFactory();
        OutputObjectsSet outputs = alg.getOutputObjects();
        Output contours = outputs
            .getOutput(PolylinesToPolygonsAlgorithm.RESULT);
        alg.execute(null, outputFactory);
        IVectorLayer result = (IVectorLayer) contours.getOutputObject();
        targetStore = (FeatureStore) result.getBaseDataObject();
    }

    public FeatureStore<SimpleFeatureType, SimpleFeature> getResult() {
        return this.targetStore;
    }
}
```

To solve the compilation errors, write the following import declarations

```
import java.io.IOException;

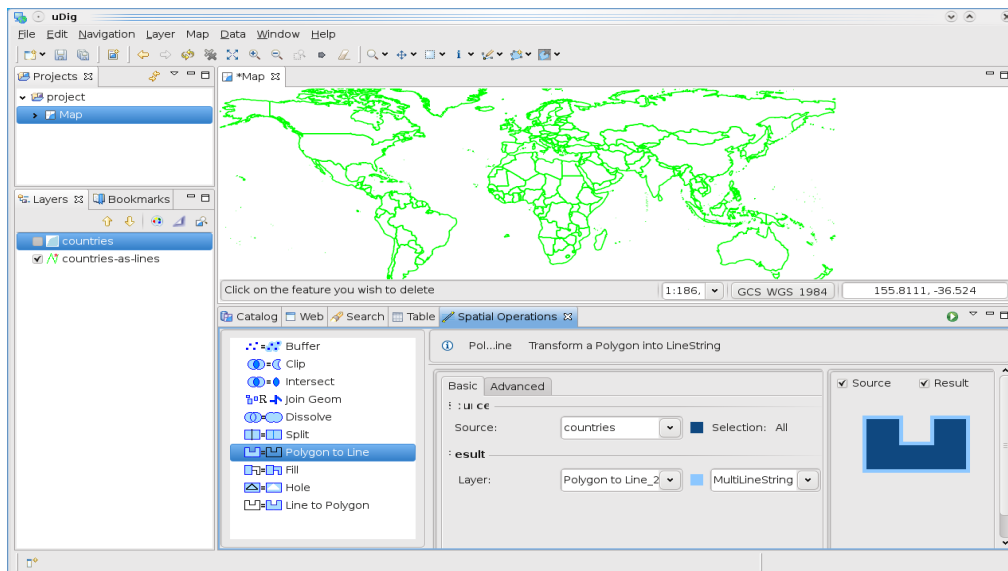
import org.geotools.data.DataStore;
import org.geotools.data.FeatureStore;
import org.opengis.feature.simple.SimpleFeature;
import org.opengis.feature.simple.SimpleFeatureType;

import es.unex.sextante.core.OutputFactory;
import es.unex.sextante.core.OutputObjectsSet;
import es.unex.sextante.core.ParametersSet;
import es.unex.sextante.dataObjects.IVectorLayer;
import es.unex.sextante.exceptions.GeoAlgorithmExecutionException;
import es.unex.sextante.geotools.GTOutputFactory;
import es.unex.sextante.geotools.GTVectorLayer;
import es.unex.sextante.outputs.Output;
import es.unex.sextante.vectorTools.polylinesToPolygons.PolylinesToPolygonsAlgorithm;
```

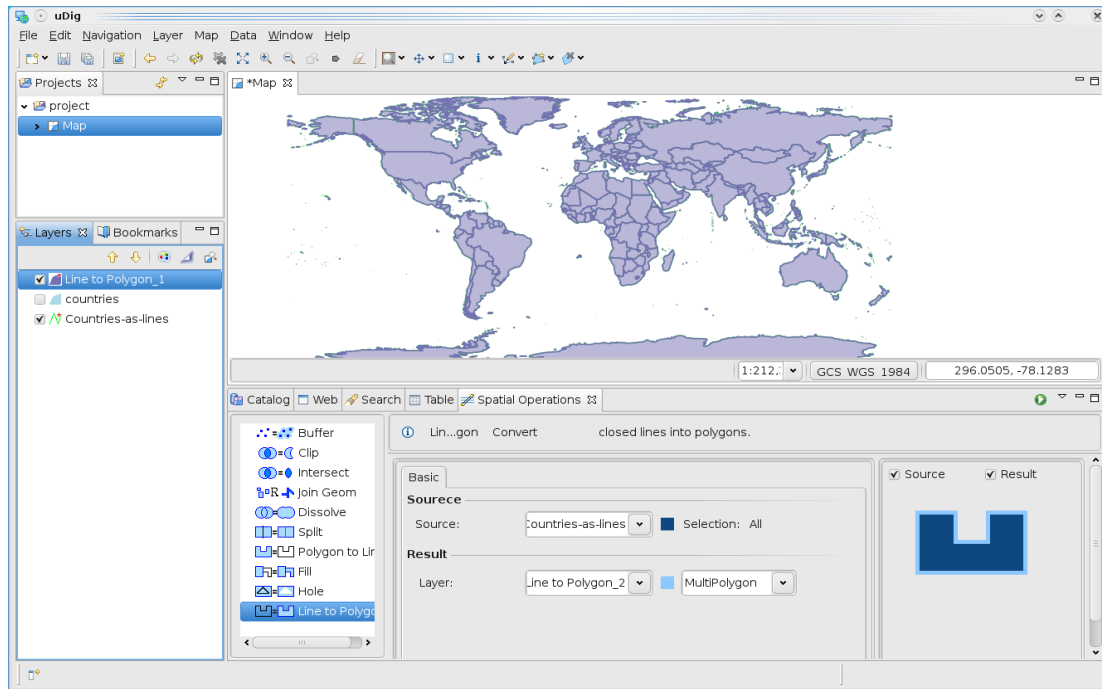
8 Executing the LineToPolygon Spatial Operations

At this point, you should not have any compilation errors. If all is OK, you can execute the **uDig product**, and you should see the newest **LineToPolygon** operation in the spatial operations list. To test it, you need a LineString layer with some rings. We are going to generate a LineString layer using the country layer applying the PolygonToLine operation, as you can see in below:

- Load the shapefile countries.shp.
- Execute the operation **PolygonToLine** using as source layer **countries.shp**, and as result layer **Countries-as-lines**. The result is shown in the following screenshot.



- Now, we have the required input for LineToPolygon. So, execute the operation LineToPolygon using as source layer **Countries-as-lines** and result layer **LineToPolygon_1**.



9 Contact

For any suggestions or doubts about this tutorial, please don't hesitate to contact us:

www.axios.es

info@axios.es

or

Mauricio Pazos

Axios Director

mauricio.pazos@axios.es