Creating Alteryx Tools in .NET

Introduction

With the release of Alteryx 6.1, we have added the ability for .NET developers to create their own custom tools for Alteryx. Custom tools reside in assemblies (called Plugins) that expose specific interfaces that Alteryx uses to communicate with them. A plugin assembly may contain any number custom tools.

Before we look at the process of building tools, let's first look at how they are deployed.

Deploying Plugins

Plugins require no registration in order to be recognized by Alteryx. All that is needed is for them to be placed into the Plugins folder.

You need to add a new .ini file in <alteryx install>\Settings\AdditionalPlugins. (The default is: C:\Program Files\Alteryx.)

The CustomPlugin.ini should look like:

```
[Settings]
x64Path=C:\PROGRAM FILES\Alteryx\<directory for x64 plugin>
x86Path=C:\PROGRAM FILES\Alteryx\<directory for x86 plugin>
ToolGroup=Developer
```

(ToolGroup is where you want it to show up in Alteryx as a tool.)

Simply create the appropriate key and add a string value for each folder that Alteryx should check in for custom plugins. The name of the string value may be anything, but the value should be the path to check. If Alteryx doesn't find any plugins in the specified folder, it will search for a **debug** subfolder if you are running via a debugger or a **release** folder if you are running normally. This can be useful for setting up a debug environment for testing your tools.

When Alteryx runs, it examines each DLL in these folders and looks for the .NET classes which implement the Alteryx IPlugin interface (see below). This interface, in turn, tells Alteryx the name of the assembly and the class for the corresponding engine implementation. It is important to note that this same information (assembly and class) is contained in each module which uses the given tool. This allows the Alteryx Engine to locate and instantiate each tool directly, without having to rely on the Alteryx GUI to provide this mapping at execution time.

The Samples

There is a sample project that goes along with this document called **CustomDotNetTools**. It includes the implementation of three custom tools:

1. **XmlInputTool** – a tool that reads data from an XML file. The data must be formatted in the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<SomeContainingElement>
    <SomeField1>Some Data</SomeField1>
    <SomeField2>Some Data</SomeField2>
    …
</SomeContainingElement>
<SomeContainingElement>
    <SomeField1>Some Data</SomeField1>
    <SomeField2>Some Data</SomeField2>
    …
</SomeContainingElement>
etc.,
```

2. **XmlOutputTool** – a tool that writes data into an XML file. The output XML format will be similar to the format supported by the XmlInputTool.

3. **XmlTransformTool** – a tool that takes an input stream with a text field that contains XML data in the format supported by the XmlInputTool and extracts that data into the outgoing connections. It takes an optional second input stream that, if provided, will be used for the structure of the data to be read from the input stream.

Also included with the sample project are a sample Alteryx module and data that demonstrate how the custom tools work. In order to run the module, you will first have to build the project and deploy the generated assembly to your Plugins folder. Make sure you build the project for the correct platform (32 or 64 bit) for your Alteryx installation.

Building Alteryx Tools in .NET

In order to work properly, Alteryx needs three things from a tool built in .NET:

1. A .NET class that tells Alteryx where to find the implementation of the tool. This must implement the **AlteryxGuiToolkit.Plugins.IPlugin** interface.

2. A .NET UserControl that displays the tool's configuration controls in the Property window in Alteryx. This must implement the **AlteryxGuiToolkit.Plugins.IPluginConfiguration** interface.

3. A .NET class that exposes the functionality of the tool. This must implement the **AlteryxRecordInfoNet.INetPlugin** interface.

A fourth item will also be necessary if your tool accepts incoming connections from other tools:

4. One or more .NET classes that can handle incoming connections from other tools. These must implement the **AlteryxRecordInfoNet.IIncomingConnectionInterface** interface.

All of these items can be built into the same .NET assembly. This will make it easier to work with and deploy.

Start by creating a new project in Visual Studio. Because Alteryx is built on top of the .NET framework version 4.0, you will probably need to use Visual Studio 2010. The type of project you want to create is a Class Library.

With that out of the way, we can look at each of the above requirements in turn.

Step 1: Implementing AlteryxGuiToolkit.Plugins.IPlugin

The AlteryxGuiToolkit.Plugins.IPlugin interface can be found in the AlteryxGuiToolkit.dll assembly which will be located in the Engine folder in your Alteryx install. Simply add this assembly to your project's references in order to get access to it. ***Note: When adding this assembly, make sure you set the Copy Local property for it to False. If you fail to do this, you will an encounter an error when running the tool:*** `Could not find INetPlugin.`

Next, create a new class for your tool. For the purposes of this example, I am going to create an input tool that can read data from an XML format. Therefore, I will call my class XmlInputTool. Make sure you make your class public (or Alteryx won't be able to access it), and add the declaration for the AlteryxGuiToolkit.Plugins.IPlugin interface. Your class should look something like this:

```
public class XmlInputTool : AlteryxGuiToolkit.Plugins.IPlugin
{
}
```

Through the magic of Visual Studio, you can have the methods for IPlugin added to your class automatically by right-clicking on the interface name and selecting "Implement Interface". If you do this, you will see five new methods added to your class. They are:

1. GetConfigurationGui()

2. GetEngineEntryPoint()

3. GetIcon()

4. GetInputConnections()

5. GetOutputConnections()

Don't be overwhelmed by this as they are generally trivial to implement.

GetConfigurationGui()

This method is called by Alteryx in order to obtain the UserControl to display in the Properties window for the tool. This will be the class that satisfies the second requirement that was mentioned above. You simply need to create a new instance of your UserControl and return it. The XmlInputTool sample's implementation looks like this:

```
    public AlteryxGuiToolkit.Plugins.IPluginConfiguration GetConfigurationGui()
    {
        // Return a new instance of our GUI control.
        return new XmlInputToolGui();
    }
```

GetEngineEntryPoint()

This method is called by Alteryx in order to determine what class to use when running the tool. This will be the class that satisfies the third requirement that was mentioned above. You must create a new AlteryxGuiToolkit.Plugins.EntryPoint object, passing in the name of the assembly that your tool is implemented in, the namespace-qualified name of the class that implements the tool's functionality, and a parameter telling Alteryx that this is a .NET-implemented tool. The XmlInputTool sample's implementation looks like this:

```
    public AlteryxGuiToolkit.Plugins.EntryPoint GetEngineEntryPoint()
    {
        return new AlteryxGuiToolkit.Plugins.EntryPoint("CustomDotNetTools.dll",
            "CustomDotNetTools.XmlInputToolEngine", true);
    }
```

GetIcon()

This method is called by Alteryx to obtain the image that will be used as the tool's icon in the ToolBox and Flowchart windows in Alteryx. The image should be a PNG file that is 171 x 171 pixels in size and 32 bit color. It should use the same arrow standards that we use (green arrows for required connections, white arrows for optional connections). I also recommend adding your PNG file to your project as an Embedded Resource so that you don't need to distribute separate image files. The XmlInputTool sample's implementation looks like this:

```
    // For performance reasons, we cache the bitmap.
    private System.Drawing.Bitmap m_icon;
    public System.Drawing.Image GetIcon()
    {
        // NOTE: If you follow this example and include the bitmap in the dll,
        // be sure to specify its "Build Action" as "Embedded Resource", or this
        // code will not be able to locate it!
        if (m_icon == null)
        {
            // Get the assembly we are built into.
            System.IO.Stream s =
                typeof(XmlInputTool).Assembly.GetManifestResourceStream(
                    "CustomDotNetTools.XmlInputTool.png");

            // Load the bitmap from the stream.
            m_icon = (System.Drawing.Bitmap)System.Drawing.Bitmap.FromStream(s);
            m_icon.MakeTransparent();
        }

        return m_icon;
    }
```

GetInputConnections()

This method is called by Alteryx in order to determine what (if any) input connections your tool allows. You must return an array of AlteryxGuiToolkit.Plugins.Connection objects that describe each connection you want to allow, or an empty array if you don't allow any input connections. Since The XmlInputTool sample is an input tool, it doesn't allow any input connections and the code looks like this:

```
public AlteryxGuiToolkit.Plugins.Connection[] GetInputConnections()
{
    // Since this example is an input tool and has no input connections,
    // we will return an empty array.
    return new AlteryxGuiToolkit.Plugins.Connection[] { };
}
```

If you were building a tool that accepted one connection, such as the XmlOutputTool sample, the code would look like this:

```
public AlteryxGuiToolkit.Plugins.Connection[] GetInputConnections()
{
    // In this example, we only have one input connection called "Input".
    return new AlteryxGuiToolkit.Plugins.Connection[] {
            new AlteryxGuiToolkit.Plugins.Connection("Input")
        };
}
```

GetOutputConnections()

This method is called by Alteryx in order to determine what (if any) output connections your tool allows. You must return an array of AlteryxGuiToolkit.Plugins.Connection objects that describe each connection you want to allow, or an empty array if you don't allow any output connections. The XmlInputTool sample only supports one output connection, and the code looks like this:

```
public AlteryxGuiToolkit.Plugins.Connection[] GetOutputConnections()
{
    // In this example, we only have one output connection called "Output".
    return new AlteryxGuiToolkit.Plugins.Connection[] {
            new AlteryxGuiToolkit.Plugins.Connection("Output")
        };
}
```

Step 2: Implementing AlteryxGuiToolkit.Plugins.IPluginConfiguration

The AlteryxGuiToolkit.Plugins.IPluginConfiguration interface is also located in the AlteryxGuiToolkit.dll assembly. To implement it, add a new UserControl to your project and in the code for it, add the declaration for the interface. The XmlInputTool sample's implementation looks like this:

```
public partial class XmlInputToolGui  : UserControl,
    AlteryxGuiToolkit.Plugins.IPluginConfiguration
{
}
```

You can add the methods for IPluginConfiguration the same way you did for IPlugin. But before you do this, you should stop and think about what you want your configuration control to do. This is the code that will allow the user to specify any information that you will need to make your tool work.

For the XmlInputTool sample, we want the user to be able to browse for an XML file and have the tool parse it and output data. Since there really is no standard way to store data in XML, we are going to limit the tool to being able to read data in the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<SomeContainingElement>
    <SomeField1>Some Data</SomeField1>
    <SomeField2>Some Data</SomeField2>
    …
</SomeContainingElement>
<SomeContainingElement>
    <SomeField1>Some Data</SomeField1>
    <SomeField2>Some Data</SomeField2>
    …
</SomeContainingElement>
etc.,
```

In order to support this format, we will also need the user to specify the name of the containing element. Additionally, since we don't want to deal with XSD, we will want the user to be able to specify the field types and sizes for each of the data fields.

Therefore, the XmlInputTool sample's configuration should allow the user to specify the following:

- The path to the XML file to read.
- The name of the container element for the data elements in the XML file.
- The data type, size, and scale for each field, by name.

To avoid boring you with the details of implementing all of that – you can take a look at the sample if you are interested. What you do need to know, however, is how all of that information gets stored in the Alteryx module document and passed into your tool at runtime.

Alteryx uses an XML format of its own to store this information. It is up to you to determine how you want to store your configuration information in this XML format. It really doesn't matter how you do it, so long as it is compatible with XML. For the XmlInputTool sample's implementation, the configuration looks like this (the actual values will obviously depend on the data chosen by the user):

```
<Configuration>
  <XmlFile>C:\temp\NWProducts.xml</XmlFile>
  <XmlElement>Products</XmlElement>
  <Fields>
    <Field Name="ProductID" Type="E_FT_String" Size="10" Scale="-1" />
    <Field Name="ProductName" Type="E_FT_String" Size="100" Scale="-1" />
    <Field Name="SupplierID" Type="E_FT_Int16" Size="10" Scale="-1" />
    …
  </Fields>
</Configuration>
```

Now that we've determined what the configuration should look like, we can implement it. In terms of getting the information from the user on the UserControl, that is really up to you to figure out. You have access to all of the standard .NET WinForms controls that you might want to use. In terms of interacting with Alteryx, though, you simply need to implement the two functions that are exposed in the AlteryxGuiToolkit.Plugins.IPluginConfiguration interface:

1. GetConfigurationControl()

2. SaveResultsToXml()

These will be discussed below, but you should take a look at the sample code for the implementation details. It isn't difficult, but it's a little too big to include here, mostly due to the code for filling the sample data and the field grid.

GetConfigurationControl()

The GetConfigurationControl() method is called by Alteryx when the tool becomes active and the Property window needs to be displayed. The method should return `this` since `this` is the class that implements the UserControl.

The method is passed several pieces of information from Alteryx to help the UserControl configure itself:

- AlteryxGuiToolkit.Document.Properties docProperties – information about the document that the tool is in.

- XmlElement eConfig – the Configuration XML described above that contains the saved configuration for the tool. Note that often this will be empty if the tool has not yet been configured.

- XmlElement[] eIncomingMetaInfo – the field descriptions for any incoming connections that are attached to the tool. These will be in the same order as they were specified in your implementation of IPlugin.GetInputConnections(). The actual values in the array may be null if no connections have been made.

- int nToolId – the unique ID number for the tool that is being configured.

- string strToolName – the name of the tool.

You should use this information to configure your UserControl. The XmlInput sample's implementation uses the eConfig parameter to get the XmlFile element and the XmlElement elements to populate text boxes, and the Field elements to display a data table that allows the user to modify the field types.

SaveResultsToXml()
This method is called by Alteryx when the user has finished modifying the properties in your UserControl. You use this to save the configuration XML and set the default annotation for the tool. If you've taken the time to define your XML configuration format as we did above, then this should be fairly easy to implement.

Step 3: Implementing AlteryxRecordInfoNet.INetPlugin
The AlteryxRecordInfoNet.INetPlugin interface can be found in the AlteryxRecordInfo.Net.dll assembly which can also be found in the Engine folder in your Alteryx install. Simply add this assembly to your project's references in order to get access to it. ***Note: When adding this assembly, make sure you set the Copy Local property for it to <u>False</u>. If you fail to do this, you will an encounter an error when running the tool:*** `Could not find INetPlugin.`

INetPlugin is the interface that Alteryx uses t o pass data into your tool and get data out of it at run time. It exposes the following methods:

1. PI_Init()
2. PI_AddIncomingConnection()
3. PI_AddOutgoingConnection()
4. PI_PushAllRecords()
5. PI_Close()
6. ShowDebugMessage()

How complicated these are to implement really depends on how complicated the functionality of your tool is.

PI_Init()
This method is called by Alteryx in order to initialize the tool. It provides the tool with its unique ID number, an EngineInterface which the tool can use to send messages to Alteryx, and an XmlElement that contain the configuration properties for the tool. The tool should hold onto this information and use the properties to configure itself for whatever functionality it is going to do. The XmlInputTool sample's implementation looks like this:

```
public void PI_Init(int nToolID, AlteryxRecordInfoNet.EngineInterface
    engineInterface, System.Xml.XmlElement pXmlProperties)
{
    // Save the incoming information.
    m_nToolId = nToolID;
    m_engineInterface = engineInterface;
```

```
        m_xmlProperties = pXmlProperties;

        // Create our PluginOutputConnectionHelper that will be used to manage
        // any outgoing connections.
        m_outputHelper = new AlteryxRecordInfoNet.
            PluginOutputConnectionHelper(m_nToolId, m_engineInterface);
    }
```

PI_AddIncomingConnection()

This method is called by Alteryx when a new incoming connection is added to the tool. It provides the tool with the type and name of the connection. The type will be the name that you gave the connection in the IPlugin.GetInputConnections() method. The name will be the name assigned to the connection by the user in the Alteryx GUI. You can use this information to decide how to handle the connection. The method returns an object that implements the AlteryxRecordInfoNet.IIncomingConnectionInterface. Since the the XmlInputTool sample doesn't support any incoming connections, its implementation looks like this:

```
    public AlteryxRecordInfoNet.IIncomingConnectionInterface PI_AddIncomingConnection(
        string pIncomingConnectionType, string pIncomingConnectionName)
    {
        // Since we are an input tool, we don't accept any incoming connections.
        // In our implementation of IPlugin (in XmlInputTool.cs) we specified no
        // incoming connections, so this method should never be called.
        throw new NotImplementedException("Input tools cannot accept incoming
connections.");
    }
```

A tool that does support an incoming connection, such as the XmlOutputTool sample, would look like this:

```
    public AlteryxRecordInfoNet.IIncomingConnectionInterface PI_AddIncomingConnection(
        string pIncomingConnectionType, string pIncomingConnectionName)
    {
        // Since we only accept one incoming connection and we have implemented the
        // IIncomingConnectionInterface in this class to handle it, return this.
        return this;
    }
```

And a tool that supports multiple incoming connections, such as the XmlTransformTool sample, looks like this:

```
    public AlteryxRecordInfoNet.IIncomingConnectionInterface PI_AddIncomingConnection(
        string pIncomingConnectionType, string pIncomingConnectionName)
    {
        // Since we accept two different incoming connections,
        // we must handle each of the separately.
        if (pIncomingConnectionType == "Input")
        {
            // The XmlTransformToolEngine class will handle the "Input" connection.
            return this;
        }
        else // "LayoutSource"
        {
            // Create a new RecordLayoutSourceInput to handle
            // the "LayoutSource" connection.
            m_layoutSourceInput = new RecordLayoutSourceInput(this,
                m_engineInterface, m_nToolId);
            return m_layoutSourceInput;
        }
    }
```

PI_AddOutgoingConnection()

This method is called by Alteryx when a new outgoing connection is being added to the tool. It provides the tool with the name of the outgoing connection and an AlteryxRecordInfoNet.OutgoingConnection object that represents the connection. The name will be the name that you gave the connection in the IPlugin.GetOutputConnections() method. You will need to use the OutgoingConnection object to send your data downstream. You should use the AlteryxRecordInfoNet.PluginOutputConnectionHelper to manage all of your outgoing connections as it takes care of things like closing connections that have stopped accepting data and notifying Alteryx of the amount of data being sent. The XmlInputTool sample's implementation looks like this:

```
    public bool PI_AddOutgoingConnection(string pOutgoingConnectionName,
        AlteryxRecordInfoNet.OutgoingConnection outgoingConnection)
    {
        // Add the outgoing connection to our PluginOutputConnectionHelper
        // so it can manage it.
        m_outputHelper.AddOutgoingConnection(outgoingConnection);
        return true;
    }
```

PI_PushAllRecords()

This method is called by Alteryx when it is requesting that an Input tool send all of its data downstream. This method will only be called on tools that have no incoming connections. The method is provided with a parameter that specifies the maximum number of records that Alteryx wants the tool to provide. This value will be -1 if Alteryx wants the tool to provide all of its records. The implementation for the XmlInputTool sample is somewhat lengthy as it deals with all of the XML parsing, so it is not included here.

PI_Close()

This method is called by Alteryx after the tool has notified it that it has finished processing its data. It provides a parameter that indicates whether there were any errors so that the tool can respond appropriately. You should use this method to clean up any resources that you may have used during the processing. Since the XmlTransformTool sample uses two incoming connections and it can't rely on one finishing before the other, it closes its PluginOutputConnectionHelper in this method. The implementation looks like this:

```
    public void PI_Close(bool bHasErrors)
    {
        // Since we can't close our output until both the "Input" and
        // "LayoutSource" incoming  connections have closed, we have to
        // close our PluginOutputConnectionHelper here.  This method
        // shouldn't be called by Alteryx until we have told it that we
        // are finished processing, which won't happen until both incoming
        // connections have closed.
        m_outputHelper.Close();
    }
```

ShowDebugMessages()

This method is called by Alteryx whenever an unhandled exception in your .NET code is caught. You should return true if you want Alteryx to display detailed error information for debugging purposes, or false if you don't want those details displayed. The XmlInputTool sample's implementation looks like this:

```
    public bool ShowDebugMessages()
    {
        // Return true to help us debug our tool.
        // This should be set to false for general distribution.
        return true;
    }
```

Step 4: Implementing AlteryxRecordInfoNet.IIncomingConnectionInterface

The AlteryxRecordInfoNet.IIncomingConnectionInterface interface is also located in the AlteryxRecordInfo.Net.dll assembly. Alteryx uses this interface to provide data from upstream tools that are connected to your tool. You only need to implement this interface if your tool will accept incoming connections. However, you must implement this interface separately for each incoming connection that your tool accepts. The exception is if your tool accepts an undefined number of incoming connections (such as the Union tool). In this case, you would handle all of those connections through the same implementation, but you would need to manage how to deal with them using the name parameter in the PI_AddIncomingConnection() method.

The IIncomingConnectionInterface exposes these methods:

1. II_GetPresortXml()
2. II_Init()
3. II_PushRecord()
4. II_UpdateProgress()
5. II_Close()
6. ShowDebugMessages()

As with the INetPlugin interface, the complexity of the implementation of these methods is based upon how complicated your tool's functionality is.

II_GetPresortXml()

This method is called by Alteryx immediately after the incoming connection is added to the tool by the PI_AddIncomingConnection() method. It is used to determine if you want the data coming into the tool to be presorted or if you only want a specific subset of fields to be provided. Alteryx will perform this sorting and field filtering for you automatically. To take advantage of this, you must return an XmlElement that contains the sorting and field filtering information. If you don't want any sorting or filtering, simply return null. The method is provided with the tool's configuration XML properties to assist you in determining how to handle the request. The format of the returned XML should look like this:

```
<SortInfo> <!--specifies the fields to sort by-->
    <Field field="SortField1" order="Asc" />
    <Field field="SortField2" order="Desc" />
    ...
</SortInfo>

<FieldFilterList> <!--specifies what fields to include in the input-->
    <Field field="FilterField1" />
    <Field field="FilterField2" />
    ...
</FieldFilterList>
```

II_Init()

This method is called by Alteryx when it is initializing the incoming connection. It will be called before data is passed in from the upstream tool. It provides the tool with a RecordInfo object that describes the data structure of the incoming records. You can use this information to prepare your tool for accepting data. In the XmlOutputTool sample, we create a new XmlTextWriter to handle writing our data to an XML file. The implementation looks like this:

```
public bool II_Init(AlteryxRecordInfoNet.RecordInfo  recordInfo)
{
    // Save the provided RecordInfo which describes the incoming data structure.
    m_recordInfoIn = recordInfo;

    // Create the XmlTextWriter that we will use to output our data.
    m_xmlTextWriter = new XmlTextWriter(m_xmlFile, Encoding.UTF8);
    m_xmlTextWriter.Formatting = Formatting.Indented;

    // Write the start of the document.
    m_xmlTextWriter.WriteStartDocument();

    // Begin a tag called AlteryxData that will contain our data container elements.
    m_xmlTextWriter.WriteStartElement("AlteryxData");

    // Initialize our record counter.
    m_nRecords = 0;

    return true;
}
```

II_PushRecord()

This method is called by Alteryx when the upstream tool is providing a data record to our tool. It is provided a RecordData object that contains the data for the incoming record. You will need to use the RecordInfo object that was provided in II_Init() to retrieve the data from this object. If the XmlOutputTool sample, we create a new container XML element for each record, and add to it an XML element for each field in the RecordInfo object. Each field's XML element's inner text is set to the data value for that field from the RecordData object. The implementation looks like this:

```
public bool II_PushRecord(AlteryxRecordInfoNet.RecordData  pRecord)
{
    // Start the element that will contain the data elements.
    m_xmlTextWriter.WriteStartElement(m_containerElementName);

    // Iterate through the fields in the incoming record and
    // write them out as child elements of the container element.
    for (int i = 0; i < m_recordInfoIn.NumFields();  i++)
    {
        // Get the FieldBase for the field.
        AlteryxRecordInfoNet.FieldBase  fieldBase = m_recordInfoIn[i];

        // Write out the data element, using the field's name as the
        // element's name, and the field's value as the element's inner text.
        m_xmlTextWriter.WriteElementString(fieldBase.GetFieldName(),
            fieldBase.GetAsString(pRecord));
    }

    // End the container element.
    m_xmlTextWriter.WriteEndElement();

    // Increment our record counter.
    m_nRecords++;

    // Return true to indicate that we successfully processed the
    // incoming record.
    return true;
}
```

II_UpdateProgress()

This method is called by Alteryx when it is requesting our progress status. It is passed the percent completion of the upstream tool for us to use in calculating our own progress. The XMLOutputTool sample's implementation looks like this:

```
public void II_UpdateProgress(double dPercent)
{
    // Since our progress is directly proportional to the progress of the
    // upstream tool, we can simply output it's percentage as our own.
    if (m_engineInterface.OutputToolProgress(m_nToolId,  dPercent) != 0)
    {
        // If this returns anything but 0, then the user has canceled the operation.
        throw new AlteryxRecordInfoNet.UserCanceledException();
    }
}
```

II_Close()

This method is called by Alteryx when the upstream tool has finished sending us its data. You should use this method to clean up any resources that were created for use with this connection. If you are implementing a tool that supports multiple input connections, take care not to destroy any resources that the other connections may depend upon. In the XmlOutputTool sample, we use this method to finish writing our XML file and tell Alteryx that we are finished processing (since we don't have any other incoming connections to worry about). The implementation looks like this:

```
public void II_Close()
{
    // If we started writing to our output, clean it up and close it.
    if (m_xmlTextWriter != null)
    {
        // End the AlteryxData element.
        m_xmlTextWriter.WriteEndElement();

        // End the document.
        m_xmlTextWriter.WriteEndDocumen
        t();

        // Close the writer.
        m_xmlTextWriter.Clos
        e();
    }

    // Send Alteryx a message that indicates how many records were
    written. m_engineInterface.OutputMessage(m_nToolId,
        AlteryxRecordInfoNet.MessageStatus.STATUS_Info,
        m_nRecords.ToString() + " records output to " +
        m_xmlFile);

    // Tell Alteryx that we are finished writing data so it can close
    // the plugin.
    m_engineInterface.OutputMessage(m_nToo
    lId,
         AlteryxRecordInfoNet.MessageStatus.STATUS_Complete,
                                ""); 
}
```

ShowDebugMessages()

This method is exactly the same as the INetPlugin.ShowDebugMessages. In fact, if you are implementing both the INetPlugin and the IIncomingConnectionInterface interfaces in the same class, you only need to implement this method once. However, if you are implementing the IIncomingConnectionInterface interface in a separate class, as in the XmlTransformTool sample, you will need to also implement this method.

Conclusion

While this may seem like a lot of effort, implementing the Alteryx plugin interfaces is really not that difficult. Once you become familiar with how the data flows through tools, and what Alteryx expects your implementation to do, it should become fairly simple to start writing your own tools. As always, if you have any technical issues you need help with, please contact us at dev_support@alteryx.com.