Paper 015-2007

Extending SAS® Data Integration Studio with Java: Custom GUIs and SAS Server Interaction

Christopher Treglio, Bayer HealthCare Pharmaceuticals, Montville, NJ

ABSTRACT

This paper introduces the concept of extending the functionality of the SAS Data Integration Studio client application through its Java plugin facility. Making use of this facility gives programmers access to significant features built into the SAS Data Integration Studio API, which enable full interaction with the SAS Metadata and Workspace servers. Both metadata objects and SAS data sets can be created, read, deleted, and modified. Data written by the SAS Data Integration Studio application can be read and reported on, and extended metadata can be authored and manipulated.

This paper will explain the advantages of building a Java client upon SAS Data Integration Studio over building an independent standalone application, and give a tour of the basic API made available to its plugins. Additionally, it will cover topics like creating new objects within the metadata, editing existing SAS metadata, and modifying SAS data sets with either SAS or SQL code.

Finally, the paper will profile three working applications which have been built and deployed on the SAS Data Integration Studio platform to standardize the generation of CDISC SDTM 3.1 submission data sets. The applications make extensive use of SAS Data Integration Studio GUI classes and backend connection APIs. They include a variety of innovative features, such as our Java/SAS persistence layer built with O/R mapping techniques that saves Java objects in SAS data sets over the workspace server connection.

CORE FUNCTIONALITY

Developing within the application framework means that certain core client application functions are built for you. Additionally, utilities and APIs are available for some SAS OMA interaction that would be complex to code independently.

Among the core application components that SAS Data Integration Studio provides are

- Logging and Error Display
- Properties
- GUI Base Classes

LOGGING AND ERROR DISPLAY

By turning on the “debug” log with *MdObjectFactory.getInstance().setDebug(true)*, you can output error messages to the message window in the DI Studio application, with *Util.printOutputln("message").*

Also, with the *MdObjectFactory.getInstance().setLogginEnabled(true)*, you can output the metadata queries that SAS Data Integration Studio issues to the Metadata Server for debugging.

PROPERTIES

SAS provides the com.sas.plugins.PluginResourceBundle class for properties and other resources. Simply instantiate the resource bundle with the Class of a class in the package where the “PropertyBundle.properties” file exists, and your String and int properties, as well as images, will be available with simple getter methods. Note that there is a special property – “ImageLocation.notrans” – that holds the base directory for image files gotten this way.

SAS’s PluginResourceBundle, like the standard Java one, can be internationalized with the addition of the appropriate country code to the file name, like “PropertyBundle_DE.properties”.

The typical way to use this PluginResourceBundle class is to put properties files in the same package as the class that uses them, and distribute them within the .jar file of your plugin. For example:

```
PluginResourceBundle rb = new PluginResourceBundle(GuiExample.class);
```

In this case, the PropertyBundle.properties file should be deployed within the same package as the GuiExample.class file.

However, sometimes it's convenient to have properties files outside the .jar file, so they can be changed without rebuilding and redeploying your plugin. The PluginResourceBundle class offers an alternative constructor that takes an InputStream. You can construct a simple FileInputStream as follows

```
FileInputStream externalStream = new FileInputStream("plugins"
  +System.getProperty("file.separator")+ "external.properties");

PluginResourceBundle rb = new PluginResourceBundle(externalStream);
```

This resource bundle would load properties from the "external.properties" file, found in the "plugins" directory of your DI Studio installation.

GUI BASE CLASSES

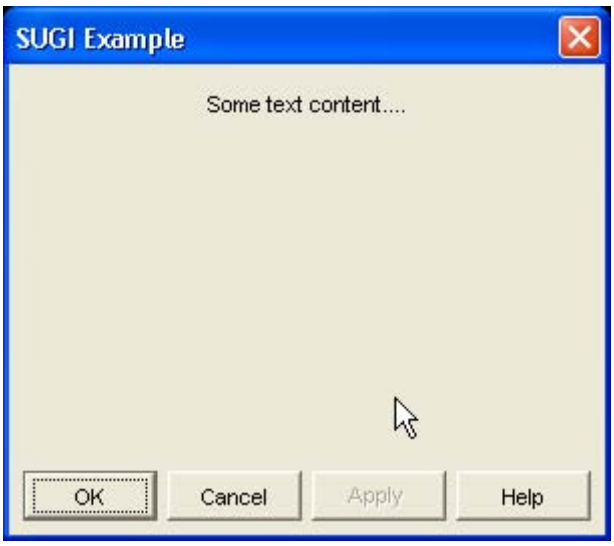
The SAS Data Integration Studio GUI is built with Java Swing, and all Swing and AWT classes can be used within plugins. To give your plugins the same GUI look-and-feel as the rest of the application environment, you may want to use the base GUI classes SAS provides. Two GUI layouts in particular are easy to build with SAS GUI base classes

- Standard Dialogs, with the WAStandardDialog
- Wizards, with the WAWizardDialog and with WAPropertyTab pages

The simpler is the standard dialog. Here's a example WAStandardDialog subclass:

```
public class ExampleDialog extends WAStandardDialog {
    public ExampleDialog(Frame frame, String title) {
        super(frame, title);
        WAPanel mainPanel = new WAPanel();
        JLabel someLabel = new JLabel("Some text content....");
        mainPanel.add(someLabel);
        // add more components as appropriate here ...
        mainPanel.setPreferredSize(new Dimension(200, 200));
        setMainPanel(mainPanel);
}
```

This code creates a dialog like this:

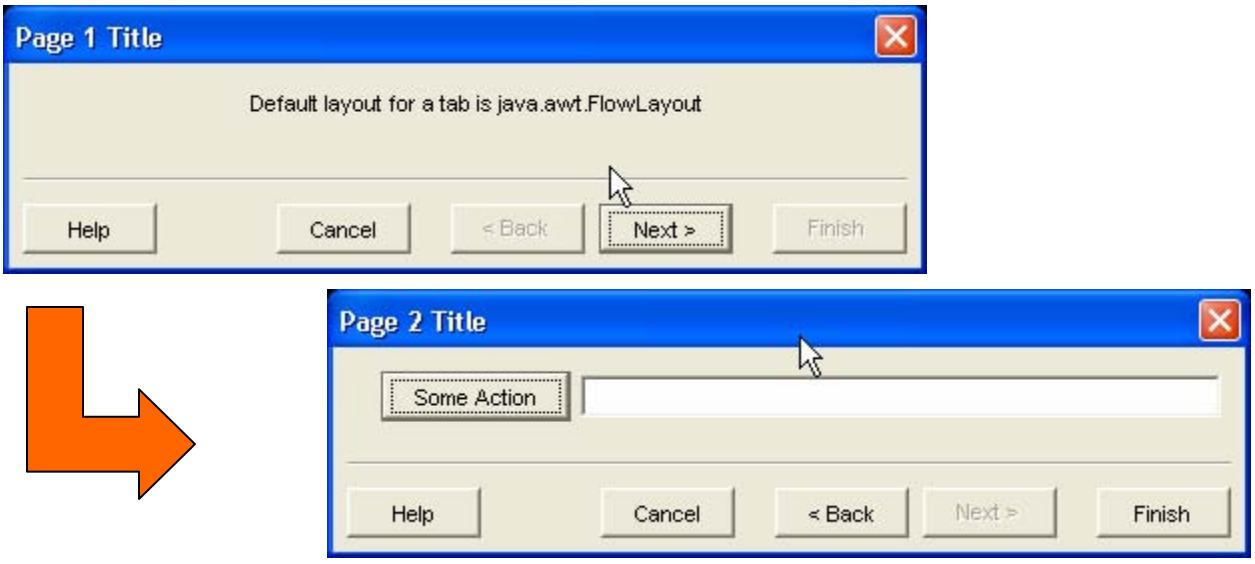


WAStandardDialog offers methods you can override to provide functionality for the four buttons displayed by default. Similarly, the WAWizardDialog can be used as follows:

```
public class ExWizard extends WAWizardDialog {

    public ExWizard(Frame frame) {
        super(frame);
        WAPropertyTab tab1 = new WAPropertyTab();
        tab1.add(new JLabel("Default layout for a tab is " +
          tab1.getLayout().getClass().getName()));
        this.addTab("Page 1 Title" ,"PAGE_ONE",tab1, false);
```

```
        WAPropertyTab tab2 = new WAPropertyTab();
        tab2.add(new JButton("Some Action"));
        tab2.add(new JTextField(40));
        this.addTab("Page 2 Title" ,"PAGE_TWO",tab2, true);
    }
}
```

Which results in:



Notice that the “Finish” button is available on the second page – this is controlled by the last boolean argument of the WAPropertyTab constructor.

In real use, these WAPropertyTab objects would probably be subclassed to perform more substantial work. Also, these WAPropertyTab objects are WAPanels, just like the panel set as the “main panel” in the first example. The WAPanel can be much more sophisticated than either example shown above, providing functionality for validating data and moving data between GUI components and the “model”. These functions are documented more fully in SAS’s online API docs.

ASSEMBLING A PLUGIN.JAR FILE

Of course, somehow we have to deploy your code within SAS Data Integration Studio. To accomplish this, you will need to put a .jar file in the “plugins” subdirectory of your installation (which at the time of this writing is still called “ETLStudio”). That .jar file needs to contain a MANIFEST.MF file that has these entries, minimally:

```
Manifest-Version: 1.0
Created-By: 1.4.2_06-b03 (Sun Microsystems Inc.)
Plugin-Init: examplePlugin.ExLauncher.class
Copyright: Copyright (c) 2006 Yourcompany Inc.
```

The highlighted entry above is the important one. The Plugin-Init field must name the class in your plugin that implements the com.sas.wadmin.plugins.ShortcutInterface, which provides the following methods.

```
public void onSelected() {
    //this method is called when the user invokes the plugin
}
public Icon getLargeIcon() {
    // the 32 x 32 icon shown in the “shortcut” bar
}
public JMenuItem getMenuItem() {
    // complex menu items can be constructed, or just new JMenuItem()
}
public int getLocations() {
    // return SHOW_ON_SHORTCUT, SHOW_ON_MENU, or SHOW_ON_ALL
}
```

```
public void initPlugin() {
    // initialization method called when the plugin is loaded
}
public void dispose() {
    // use this method to clean up any resources used
}
public String getDescription() {
    // description of the plugin
}
public Icon getIcon() {
    // smaller icon used for the Menu
}
public String getName() {
    // name displayed in the shortcut bar
}
```

Our customary practice is to build a “Launcher” class that implements this ShortcutInterface, which then kicks off our GUI dialog within its onSelected method. This arrangement allows the GUI dialogs (which are typically subclasses of javax.swing.JPanel) to be run independent of the SAS Data Integration Studio environment, for debugging purposes.

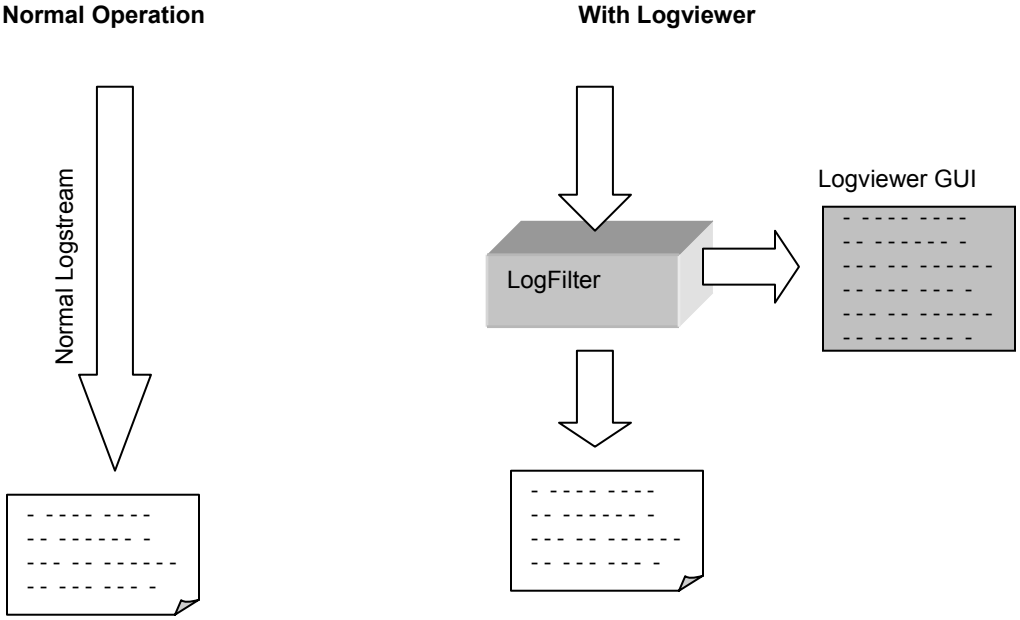
EXAMPLE: THE DI LOGVIEWER

With only the above information, we can already code a useful application. The plugin application demonstrated below is called the “Logviewer.” This plugin does three things:

- It turns on metadata conversation log on, with the MdObjectFactory.getInstance().setLoggingEnabled(true), discussed above.
- It filters the logging stream to provide for appropriate formatting for Windows clients.
- It displays a simple GUI window showing the log as it is written.

The Logviewer is comprised of 3 classes

- Viewerlauncher, implementing Shortcutinterface
- LogViewDialog, extending JFrame
- LogFilter, extending FilterOutputStream



The Logviewer plugin intercepts the standard logging stream, and writes its contents to both the viewer window and to the file for which it was originally destined. (In the process, it corrects a “quirk” in the format of the log stream to ensure that the log contents has OS-appropriate line breaks – the normal log is not legible for Windows clients.)

THE MANIFEST.MF FILE

The Manifest for the Logviewer jar file is as follows:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.4.2_06-b03 (Sun Microsystems Inc.)
Plugin-Init: diplugin.example.logviewer.ViewerLauncher.class
Copyright: Copyright (c) 2006 Bayer HealthCare Pharmaceuticals
```

This manifest was created with an Ant script as follows:

```
<property name="jar_location" location="."/>
<property name="logviewer_jar" value="logviewerPlugin.jar"/>
<jar destfile="${jar_location}\${logviewer_jar}" basedir=".\antTemp">
 <manifest>
  <attribute name="Plugin-Init"
value="diplugin.example.logviewer.ViewerLauncher.class"/>
  <attribute name="Copyright" value="Copyright (c) 2007 Bayer HealthCare
Pharmaceuticals"/>
  <!--
  the jar manifest spec allows arbitrary name-value pairs to be    included...
  <attribute name="Name" value="Value"/>
  -->
 </manifest>
</jar>
```

THE VIEWERLAUNCHER

The `diplugin.example.logviewer.ViewerLauncher.class` listed above in the “Plugin-Init” attribute refers to the class in your plugin that implements the ShortcutInterface class. Ours looks like this:

```
public class ViewerLauncher implements ShortcutInterface {
    private String pluginTitle;
    protected static PluginResourceBundle rb = new
        PluginResourceBundle(ViewerLauncher.class);
    private LogViewDialog dialog;
    /**
     * Notice that this method is capable of both opening and closing the dialog.
     */
    public void onSelected() {
        if (dialog.isVisible())
        {
                // close window
                dialog.setVisible(false);

        } else if (!dialog.isVisible())
        {
                // open window
                dialog.setVisible(true);
        }
    }
    /**
     * This method gets the image named in the log.plugin.icon entry in the property
       file, and returns it as an Icon.
     *
     */
    public Icon getLargeIcon() {
        ImageIcon i = rb.getImageIcon("log.plugin.icon");
        return i;
    }
    /**
```

```
     * This can't be null if the getLocations indicates that the plugin should be shown
       in the menu.
     */
    public JMenuItem getMenuItem() {
       return null;
    }
    /**
     * We're only showing the plugin on the Shortcut menu sidebar
     */
    public int getLocations() {

       return SHOW_ON_SHORTCUT;
    }
    /**
     * This method runs when the plugin class is loaded. Here, it gets the plugin title
       from a property file, and
     * instantiates the dialog that will be shown when the plugin icon is clicked
     */
    public void initPlugin() {

       pluginTitle = rb.getString("log.plugin.title");
       dialog = new LogViewDialog(pluginTitle);
    }
    /**
     * Nothing done here.
     */
    public void dispose() { }
    /**
     * This description is unused.
     */
    public String getDescription() {
       return "DESCRIPTION";
    }
    /**
     * This method can return null, because we're only showing the plugin in the
       shortcut menu. The icon we use is
     * gotten from the {@link #getLargeIcon()} method.
     */
    public Icon getIcon() {
       return null;
    }
    /**
     * This becomes the String shown in the shortcut menu.
     *
     */
    public String getName() {

       return "Toggle Log Window On/Off";
    }
}
```

Perhaps the most important method above is the onSelected, which determines what happens when the plugin icon is clicked. Also note that uncaught exceptions thrown in the init method will hang SAS Data Integration Studio.

THE LOGVIEWDIALOG

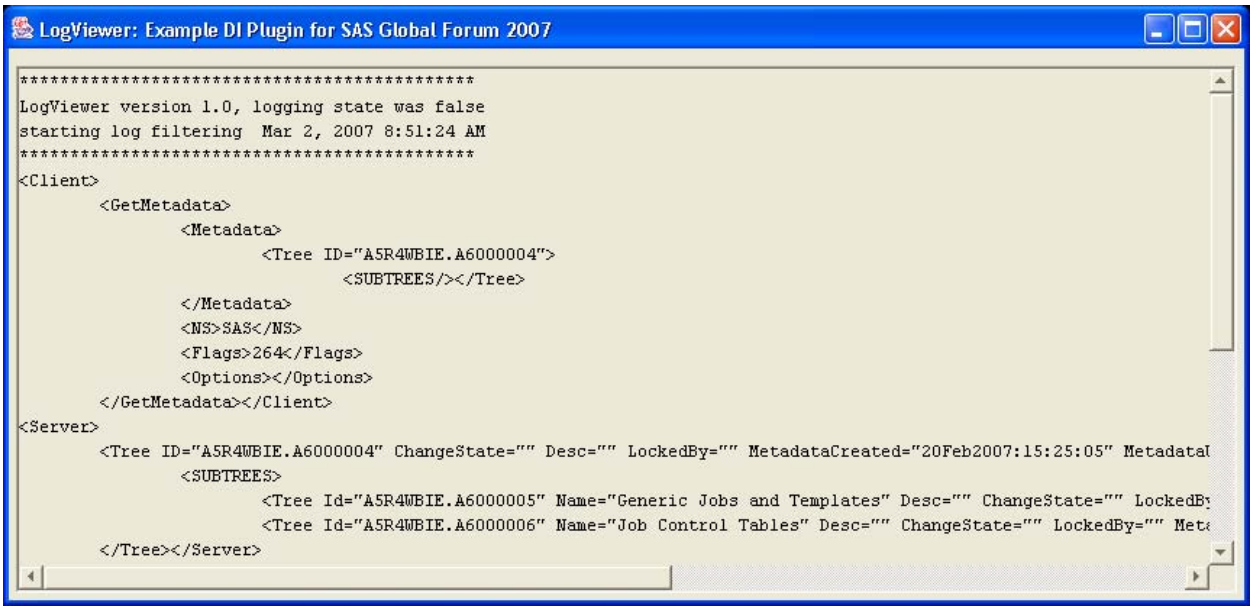
Above we saw the init method instantiating – and the onSelected method making visible – an object of the LogViewDialog class. This is what that class looks like:

```
/**
 * When instantiated, this JFrame subclass lays itself out. Note that
 * it is still invisible, until the ViewerLauncher makes the window
 * appear.
 *
 * @param title
 */
public class LogViewDialog extends JFrame {
```

```
OutputStream logToFile;
JTextArea window;
LogFilter filter;

public LogViewDialog(String title) {

    super(title);
    JPanel contentPane = new JPanel(new BorderLayout());

    // if a Log stream already exists, take note of it
    if (Util.getLogStream() != null)
    {
        logToFile = Util.getLogStream();
    }
    // if no log stream exists, use System.out
    else
    {
        logToFile = System.out;
    }
    window = new JTextArea();
    LookAndFeel laf = UIManager.getLookAndFeel();
    window.setBackground(
        laf.getDefaults().getColor("Label.background"));

    JScrollPane scrollingResult = new JScrollPane(window);
    scrollingResult.setPreferredSize(new Dimension(800, 300));
    contentPane.setBorder(BorderFactory.createEmptyBorder(12,5,5,5));
    contentPane.add(scrollingResult, BorderLayout.CENTER);
    this.setContentPane(contentPane);
    pack();
}
/**
 * This method instantiates the LogFilter, which splices the
 * application's output stream. The content is routed to both the
 * "window" object and to its original output location.
 *
 */
private void startLogToWindow()
{
    filter = new LogFilter(logToFile, window);
    Util.setLogStream(filter);
    boolean oldLogState = MdObjectFactory.getInstance().getLoggingEnabled();
    MdObjectFactory.getInstance().setLoggingEnabled(true);
    // tag the beginning of the log window with this stamp
    window.append("*****************************************\n");
    window.append("LogViewer version 1.0, logging state was " + oldLogState);
    window.append("\nstarting log filtering  " +
    DateFormat.getDateTimeInstance().format(new Date())  + "\n");
    window.append("*****************************************\n");
}
/**
 * This method removes the LogFilter from the OutputStream,
 * reattaching the original log directly to the application's
 * output log.
 *
 * @throws IOException
 */
private void endLogToWindow() throws IOException
{
    // empty whatever might remain.
    filter.flush();
    // reset original output stream
    Util.setLogStream(logToFile);
}
/**
```

```
     * This method is called by the onSelected method of our
     * ShortcutInterface.
     */
    public void setVisible(boolean open) {

        if (open)
        {
            startLogToWindow();
        }
        else
        {
            try {
                endLogToWindow();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        super.setVisible(open);
    }
}
```

This log window JFrame displays the content it sees being logged by the application by instantiating a LogFilter filter stream (which is not shown in detail here). This subclass of FilterOutputStream corrects the line breaks for the client OS, prints the content to this dialog's window, and passes it along to the original log stream. The whole application looks like this:



When you interact with SAS Data Integration Studio's interface, the metadata conversation between the client and the server appears in the window. Above, we see the query and response from opening a Tree with the ID "A5R4WBIE.A6000004." The server responds with a list of subtrees, one named "Generic Jobs and one named Templates" and "Job Control Tables."

Upon close, this dialog disconnects the filter stream and reconnects the original log stream back to the application. Note that only one of these dialogs will be instantiated by the ViewerLauncher class – subsequent clicks to the shortcut icon will simply redisplay the one dialog, so you won't lose previously logged content.

QUERYING, UPDATING, AND SAVING METADATA OBJECTS

Within the DI Studio architecture, interacting with the Metadata Server (often still called the "OMR" in SAS API documentation) is simple. The following methods – the same methods that are available in typical SAS code – are available from either the MetadataUtil or the MdOMIUtil utility classes:

- AddMetadata
- CheckinMetadata/CheckoutMetadata

- CopyMetadata
- DeleteMetadata
- GetMetadata
- GetMetadataObjects/ GetMetadataObjectsSubset
- GetRepositories
- UpdateMetadata
- Etc.

There are two distinct ways of interacting with the Metadata Server – through the MetadataUtil for single connection applications, and through the MdOMIUtil for mid tier applications. The methods in these classes are almost identical, but through interacting with the MetadatUtil, you will retrieve and operate on objects in the com.sas.metadata package, while the MdOMIUtil will get you objects in com.sas.metadata.remote. For simplicity, this paper will deal with single connection applications working with the MetadataUtil.

Additionally, there's generic method "DoRequest" which can execute any of the above methods, formatted as XML.

QUERYING OBJECTS

Let's say, for example, that we want to query the Metadata Server about SAS Libraries that have been defined there. Because we're interested in multiple metadata objects, we want to use the GetMetadataObjectsSubset method above. The code looks something like this:

```
String id= Workspace.getWorkspace().getDefaultRepository().getFQID();
MdStore mdstore = MdObjectFactory.createObjectStore();

List objects  = MetadataUtil.getMetadataObjectsSubset(
    mdstore,
    id,
    MdFactory.SASLIBRARY,
    0, // this would be any of the MetadataUtil.OMI_ ... flags
    "" // options, like an XMLSELECT filter
    );

for (Iterator iter = objects.iterator(); iter.hasNext();) {
    SASLibrary library = (SASLibrary) iter.next();
    // do something with the library
}
```

Note that the GetMetadataObjectsSubset method returns fully-fleshed out metadata objects, while the GetMetadataObjects returns only "root" objects, which are less useful (but faster to retrieve).

The query above gets a list of all the SASLibrary objects within the specified repository, but this is rarely the kind of query you will want. More often, you will need to retrieve objects based on some criteria, and those criteria can be based on an object's *attributes*, or its *associations*.

- Attributes are simple pieces of data about a metadata object: strings, integers, or dates.
- Associations are connections between metadata objects

Criteria-based queries are built using the XMLSELECT element. For example, instead of getting all the SASLibrary objects in the repository, perhaps we only want to retrieve the one library with the libref of "TEST". The XMLSELECT for that would define a simple attribute-based query, as follows:

```
<XMLSELECT search="@Libref='TEST'"></XMLSELECT>
```

Another identical way of expressing the same query would be:

```
<XMLSELECT search="SASLibrary[@Libref='TEST']"></XMLSELECT>
```

which can be read as "all SASLibrary objects with the Libref attribute of 'TEST'". Either of these XMLSELECT can be used to query the Metadata Server.

Because we're now looking for just one object, you might be inclined to think that we can use the getMetadata method. However, experiment with that method and you'll realize that it requires an Object ID, which we don't have (we only have a libref String). Besides, we don't *know* that we're going to get one object – only the object ID is considered a unique identifier, there may very well be multiple libraries in the Metadata Server with the libref of 'TEST.' As a result, we will need to use the same getMetadataObjectsSubset we used before, and check to make sure that the List returned contains only one object. That code looks like this:

```
MdStore store = MdObjectFactory.createObjectStore();
String select = "<XMLSELECT search=\"@Libref='TEST'\"></XMLSELECT>";

List objects  = MetadataUtil.getMetadataObjectsSubset(store,
    Workspace.getWorkspace().getDefaultRepository().getFQID(),
    MdFactory.SASLIBRARY,
    MetadataUtil.OMI_XMLSELECT, // this flag needs to be set now
    select);
// check to ensure that 1 and only 1 object was returned.
if (objects.size() != 1)
{
    throw new RuntimeException(objects.size() + " libraries with libref " + libref + "
       returned!");
}
SASLibrary library = (SASLibrary)objects.get(0);
```

Where before we had sent a "0" as a flag to the getMetadataObjectsSubset method, we now need to send the OMI_XMLSELECT flag. (This indicates that an XMLSELECT element is coming – without the OMI_XMLSELECT flag, our search would be ignored!)

These queries are getting more useful, but let's say we wanted to query for the path that the Metadata Server has associated with the libref TEST. To do so, we are going to need to query for more than attributes – now we need to query for *associations*.

The Metadata Server stores the path associated with a library in a separate object, appropriately called a Directory. The path of a library (the Directory object attached to the SASLibrary object) is connected through the UsedByPackages association. So, to get the actual path string, we're looking for a Directory object, but the only information we have about that Directory object is that it's attached to a SASLibrary with the libref TEST. This is expressed in the XMLSELECT search as follows:

```
<XMLSELECT search="Directory[UsedByPackages/SASLibrary[@Libref='TEST']]">
</XMLSELECT>
```

Notice that the Directory object is the object named outside the outermost brackets. That means that we're going to get Directory objects back from our query. Put it all together like this:

```
MdStore store = MdObjectFactory.createObjectStore();
String select = <XMLSELECT
       search=\"Directory[UsedByPackages/SASLibrary[@Libref='TEST']]\">
    </XMLSELECT>";

List objects  = MetadataUtil.getMetadataObjectsSubset(store,
    Workspace.getWorkspace().getDefaultRepository().getFQID(),
    MdFactory.DIRECTORY, // <- notice this has changed!
    MetadataUtil.OMI_XMLSELECT,
    select);
// check to ensure that 1 and only 1 object was returned.
if (objects.size() != 1)
{
    throw new RuntimeException(objects.size() + " Directories with SASLibrary libref "
       + libref + " returned!");
}
Directory dir = (Directory)objects.get(0);
String path = dir.getDirectoryName();// <- this is our answer
```

N.B. If we wanted to reverse the query – look for SASLibraries that had a known path string – we'd have to change the association name. The UsedByPackages association shown above is the name of the association *from the Directory to the SASLibrary*. The reciprocal association *from the SASLibrary to the Directory* is called "UsingPackages."

MODIFYING OBJECTS

Once you have retrieved objects from the Metadata Server, modifying objects is also possible simply by changing the object's attributes and asking the object to "update its metadata".

```
library.setDesc("new description . . .");
library.updateMetadataAll();
```

Creating wholly new objects is also possible. Simply ask the MdObjectFactory for a blank copy of an object, modify it as necessary, and update it as above.

```
SASLibrary newLibrary =(SASLibrary)
    MdObjectFactory.createComplexMetadataObject(
           mdstore,
           null,
           "NewLibName",
           MdObjectFactory.SASLIBRARY,
           reposID);

newLibrary.setDesc("new description . . . ");
newLibrary.updateMetadataAll();
```

INTERACTING WITH SAS DATASETS

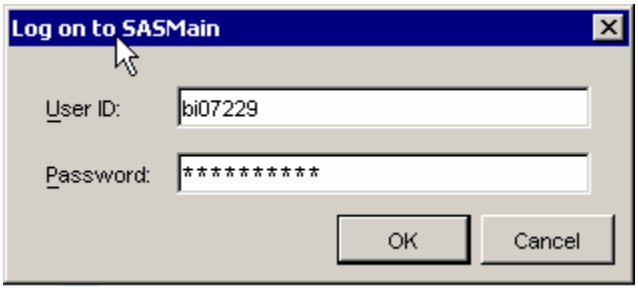
Above we were interacting with the Metadata Server, retrieving and modifying metadata objects. Of course, a time will come when you want to actually work with SAS data as well. This can be done through the Workspace Server.

CONNECTING TO THE WORKSPACE SERVER

The easiest way to make a connection to the Workspace Server within SAS Data Integration Studio is to let the Workspace (the singleton object that offers access to the SAS Data Integration Studio GUI application running behind your plugin) to “make” you an application server, with the simple code

```
AppServer appserver = Workspace.getWorkspace().makeAppServer();
```

Because the username and password to access these servers may be different from the username and password used to log in to DI Studio itself, this call will display login windows as follows:



The AppServer object offers two useful functions: allowing access to data sets both via SQL and via standard SAS code.

QUERYING, UPDATING, AND SAVING DATA WITH SQL SYNTAX

Once an AppServer object is “made” for you by your Workspace object, building JDBC-style interactions with the worskspace server is quite easy. First, set the SAS library you would like to interact with the SASLibrary metadata object (remember, we retrieved SASLibrary objects above, while working with the Metadata Server) as follows

```
appserver.assignLibref(saslibrary);
```

and get a java.sql.Connection object as follows:

```
appserver.makeSQLConnection();
java.sql.Connection conn = appserver.getSQLConnection();
```

Now, use any SAS standard SQL (as you would within a PROC SQL) with typical Java SQL techniques. Building on the previous example, this code snippet returns the names and last modified dates of the tables in our library.

```
AppServer appserver = Workspace.getWorkspace().makeAppServer();
appserver.assignLibref(saslibrary);
appserver.makeSQLConnection();
java.sql.Connection conn = appserver.getSQLConnection();
```

```
StringBuffer statement = new StringBuffer();
statement.append("Select memname, modate from DICTIONARY.TABLES");
statement.append(" where libname='");
statement.append(saslibrary.getLibref());
statement.append("';");

try {
    Statement sqlstatement = conn.createStatement();
    sqlstatement.execute(statement.toString());
    ResultSet rs = sqlstatement.getResultSet();
    while(rs.next())
    {
        String tablename = rs.getString(1);
        long moddate = rs.getTimestamp(2).getNanos();
        // etc. . . .
    }

} catch (SQLException e) {
    System.out.println("Error in query: " + statement.toString());
    e.printStackTrace();
}
```

Why would we want to query the dictionary for information like this, since the members of a SAS Library are available through the Metadata Server? Usually, we wouldn't, but there could be a difference between the data sets in the library directory and those registered in the Metadata Server. Furthermore, data like "last modified data" changes frequently and without the Metadata Server's knowledge, and is not available there.

QUERYING, UPDATING, AND SAVING DATA WITH SAS SYNTAX

SQL isn't the only way to interact with SAS data sets through the AppServer object. Standard SAS code can be submitted to the AppServer with a call like this:

```
int resultCode = appserver.submitSrc("SAS CODE");
```

This method returns a 0 result code for a normal completion, or throws an "AppServerException" if it encounters an error. Log output is available from the "getLog()" method, as follows:

```
String log = appserver.getLog().toString()
```

Output is also available, but it comes back as a String array. You can work with it as like this:

```
String[] out = appserver.getListLinesHolder().value;
StringBuffer outBuffer = new StringBuffer();
for (int i = 0; i < out.length; i++) {
    outBuffer.append(out[i] + "\n");
}
```

At this point you are logged on to the "appserver" object as the user who supplied their login information in the dialog box shows above, not the user who logged in to DI Studio. Permissions to libraries, etc. are determined based on this login.

APPLICATION EXAMPLES

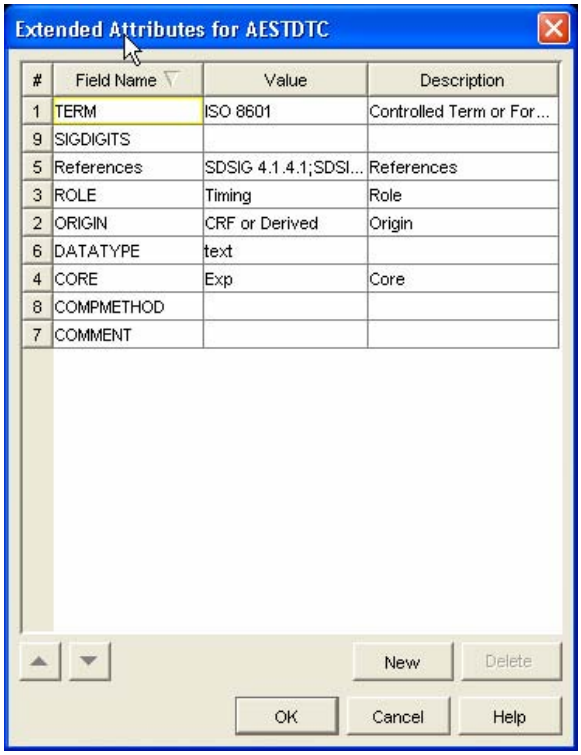
The techniques described above have been used in production applications for approximately a year now at Bayer. Three DI Studio plugins are now deployed.

DEFINE XML PLUGIN

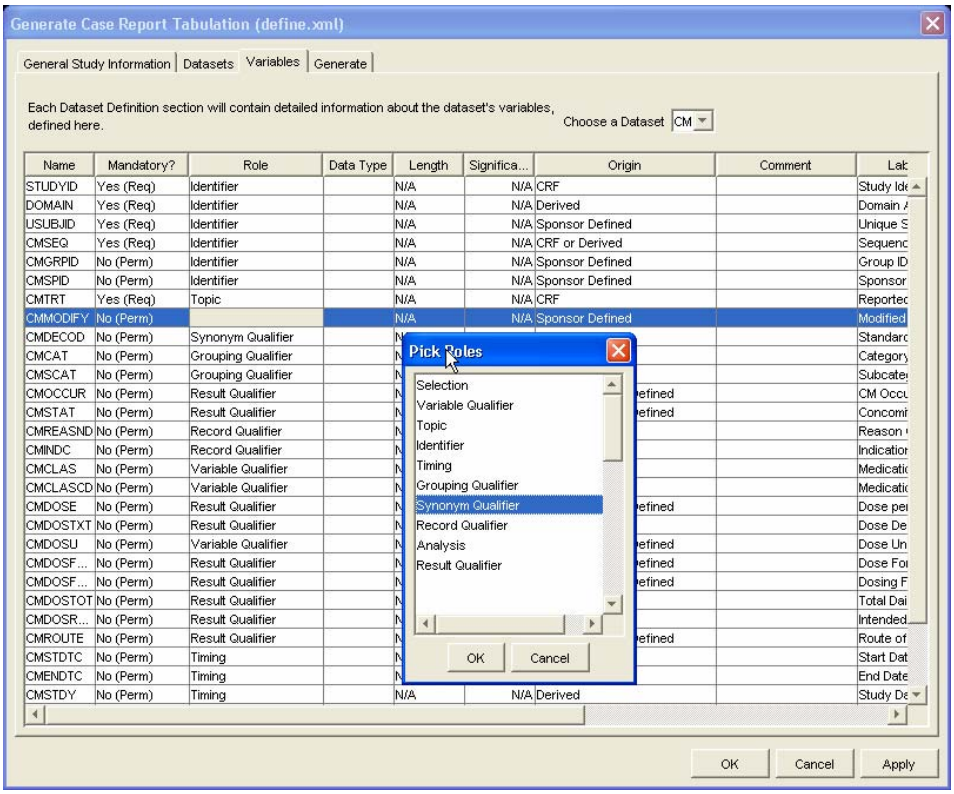
One of the standard documents required by the CDISC (Clinical Data Interchange Standards Consortium) for clinical submissions is the "define.xml". The define.xml is a table-of-contents style document detailing each data set (and its variables) included in a submission to the FDA. Since we've used SAS Data Integration Studio to create our submission data sets, all the tables and variables in our submission are registered in our SAS Metadata Repository, and generating the define.xml is simply a matter of systematically querying the repository, and writing the XML file to the local disk.

Since some of the fields required for the define.xml file are beyond the typical metadata registered in the repository, this application makes heavy use of "ExtendedAttribute" objects. ExtendedAttribute objects can be added to any metadata object stored by SAS, and provide a simple and powerful way to add custom name-value annotations to

your repository. ExtendedAttributes are shown through the “Extended Attributes” tab in the properties window, for all metadata objects.



Since entering our sometimes-complex annotations in these windows would be inconvenient for our users, we’ve deployed a plugin which offers a more user-friendly way of editing them.

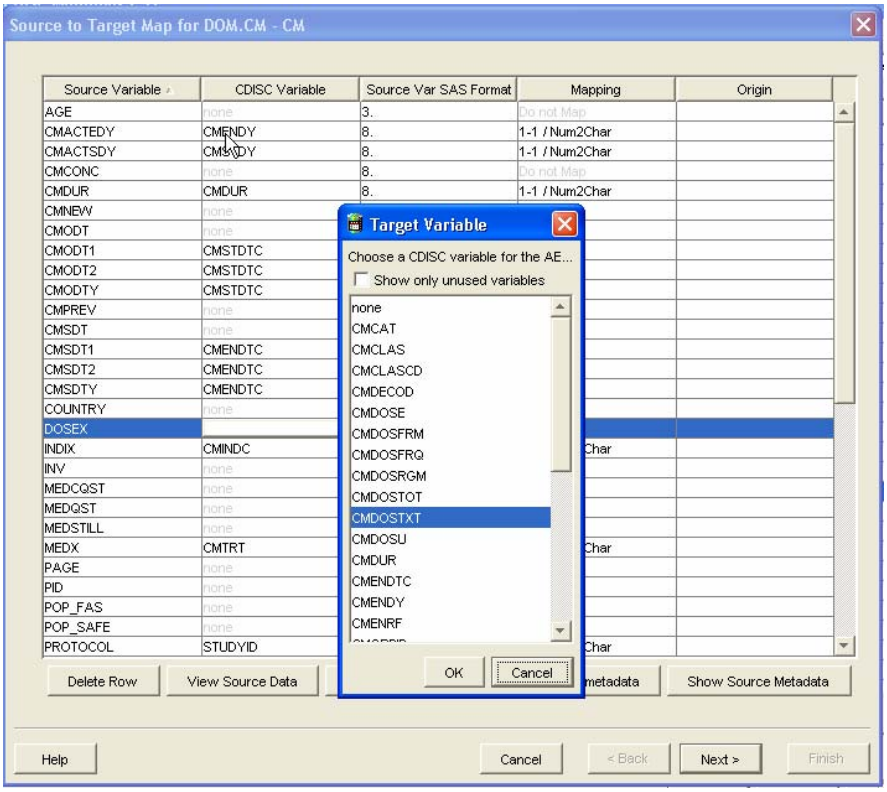


In the above screen capture, we see the Define.xml GUI modifying the extended attributes of a SAS variable (represented in the Metadata Server by a Column object). The GUI is built with a

com.sas.workspace.WATabbedPanel object, with each tab an instance of com.sas.workspace.WAPropertyTab. The table and all the editing features are standard javax.swing classes.

SDTM MAPPING PLUGIN

To generate our submission data sets, we use a custom "mapping" technique whose logic is held in a series of SAS data sets we call "Control Tables." These control tables are read by our SAS Data Integration Studio jobs, which have been enhanced with SAS Macros capable of translating their contents into programmatic logic. The SDTM Mapping plugin is responsible for giving our users a friendly GUI to write this logic into our control tables. Unlike the Define XML plugin above, the SDTM Mapping plugin primarily interacts with raw SAS data sets over the Workspace Server, rather then SAS metadata objects through the Metadata Server.



To read and write the Control Table data sets, we could have used simple SAS or SQL as described above. But to simplify our interaction, and to avoid repetitive SQL, our SDTM Mapping Plugin instead uses a "persistence layer" capable of automatically building SQL to read, write, and delete data within these data sets. Like an Object/Relational mapping framework (like the popular Hibernate), it links fields in our Java objects with variables in our data sets. Because this "Object/SAS" persistence layer can automatically load and unload data to and from memory, it frees us from manually coding SQL.

ADMINISTRATION PLUGIN

Our third application is call the Administration Plugin. Certain administrative tasks, like initializing our Metadata Repositories for use with our other plugins, would be tedious and error-prone without some kind of automation. Certain users, identified as "administrators" by our business, have access to our Administration plugin, which handles tasks like initializing our repositories, locking our repositories when an FDA submission is complete, and copying metadata from one repository to another for re-use in later projects.

Use of our Administration plugin takes some of the burden off our administrative support group, and ensures that administrative tasks are performed by appropriate staff, in a controlled way.

CONCLUSION

Data Integration Studio offers a good foundation for developing custom functionality for the SAS environment. As the Metadata Server becomes the core part of many SAS architectures, being able to interact programmatically with it becomes an invaluable skill, both for administration and for custom extensions to our business users' applications.

REFERENCES

The API for all the SAS based GUI classes mentioned above can be found here:

http://support.sas.com/rnd/gendoc/bi/api/workspace/index.html

The API for Metadata Server interaction can be found here:

http://support.sas.com/rnd/gendoc/bi/api/metadata/index.html

More general information about metadata structure can be found in SAS Help, under the heading “SAS Open Metadata Architecture.”

CONTACT INFORMATION

Your comments and questions are valued and encouraged. For a copy of the code shown above, please contact me at:

Christopher Treglio
Bayer HealthCare Pharmaceuticals
Montville, NJ USA
E-mail: chris_treglio@berlex.com