Let SAS® Do the Work for You: Interface SCL Classes from webAF ™ Applications

Yvonne Selby, SAS Institute Inc., Cary, NC

ABSTRACT

With the new Java based webAF tools provided with AppDev Studio™, it is possible to create dynamic applications that use the power of SAS/AF® software classes to interface with your applets or applications. If you're familiar with creating CLASS entries in Screen Component Language (SCL), you can write interfaces to your classes to take advantage of those classes within your Java application. This paper will demonstrate the process of creating and executing remote SCL interfaces using webAF.

INTRODUCTION

webAF provides components which allow you to view various types of SAS files such as data sets and MDDB files. Other components provided in webAF include models (or interfaces) to non-visual SAS structures such as library lists, catalogs, and data sets. Interfaces are also provided to allow you to submit code to SAS and retrieve the LOG and OUTPUT window results. Another interface is the connection component which allows you to connect to your SAS server and is needed whenever your applet contains components or interfaces which will utilize SAS in any way.

webAF also provides functionality for generating remote SCL interfaces which ultimately allow you to invoke your SCL class methods from within your Java applet. All of this basic webAF functionality mentioned above is defined in the com.sas.ComponentInterface interface which is implemented in the base class com.sas.Component.

Another feature provided in webAF is automatic Java code generation. As you drag and drop components on your applet/application frame, webAF generates much of the Java source code for you. In addition, it automatically inserts comments to indicate where it is appropropriate for you to add your own code. Source code that cannot be modified is protected so required code cannot be accidentally removed. The source code editor in webAF features color coding as well as automatic indention to make writing your applets much easier.

This article will use an example application as a basis for discussing generating remote proxies, adding event handlers, and ultimately modifying the Java source to add the functionality required in your application.

THE APPLICATION

The application displays a company's divisional organization chart such as one that may be used for technical support. You can select a node or person from the tree to display more information about the selected employee. After selecting an employee, it displays the employee's name, department and position within that department. You can optionally view more detailed information about the employee by selecting a pushButton labeled View Support Info. You can also view a bar chart of the problems the employee has worked on for the current month.

The first step in developing a webAF application is to create a project. A project is a container for a set of files that webAF uses to create an applet or application. It includes a list of all files used by the project and the project's properties. The name of this project is `proxydemo`. Figure 1 shows the applet frame as it appears in the build environment.

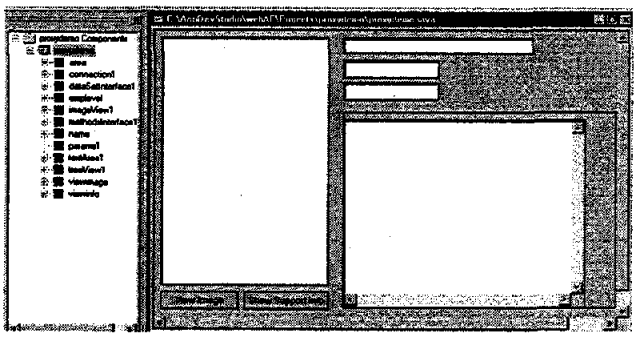

Figure 1 proxydemo Frame in Build Environment

The components of the `proxydemo` project are:

treeView1	a treeView component which will display an organizational chart.
imageView1	an imageView component which will display the contents of a stored GRSEG entry.
textArea1	a scrollable textArea component which will display the contents of a SAS catalog SOURCE Entry.
viewimage	a pushButton component which will fire an event to display the GRSEG entry. The text property has been changed to View Image.
viewinfo	a pushButton component which will fire an event to display the SOURCE entry contents. The text property has been changed to View Support Info.
area	a textField component that displays the name of the area in which the employee works.
name	a textField component that displays the name of the selected employee.
emplevel	a textField component that displays the level of the employee.
connection1	a connection component that allows the Java Applet to communicate with SAS
dataSetInterface1	a dataSetInterface model which is dropped

on the treeView component. EMPLOYEE.TECHSUP has been assigned to the dataSet property. For the `textColumn` property, number 1 has been assigned to point to the first variable, NAME, in the dataset. Number 2 has been assigned to the `levelColumn` property to point to the second column in the data set, LEVEL. Number 3 has been assigned to the `indexColumn` property to point to the third column in the data set called INDEX. The rootText property has been assigned the value "Technical Support Organization." These properties are automatically understood by the treeView component and are used to display the appropriate diagram.

methodsInterface1 a methodsInterface model which allows Java to invoke remote SCL class methods. You must build this interface and add it to your project.

To build this project, you can drag and drop all of the visual components onto your frame and arrange them in any order you like.

The `dataSetInterface` model is dropped on the treeView component automatically linking it to the model property of the `treeView1` component. The treeView component then recognizes the property for `textColumn` as the text to display for the nodes in the tree. The `levelColumn` property determines at which level in the tree diagram the current node is displayed. The `indexColumn` defines where in the data set this level exists. The `dataSetInterface` and the treeView component do not recognize other variables in the data set EMPLOYEE.TECHSUP, therefore you need to be able to query the selected node and retrieve the other variable values from the data set. For example, you may want to display the NAME of the employee, the AREA in which the employee works and the level of the employee, EMPLEVEL. The data set includes a variable which points to the location of a catalog SOURCE entry containing additional information about the products the employee supports. Also included in the data set is a variable which contains the location for a GRSEG entry that provides additional feedback regarding the types of problems the employee has been working on. To retrieve this information from the EMPLOYEE.TECHSUP data set when you select a node, you can write and invoke an SCL method.

Two additional remote SCL methods are needed to display the GRSEG entry and the SOURCE entry contents when you select the `viewimage` and `viewinfo` components respectively.

THE SCL CLASS

The SCL class, CUSTOM.ROCF.METHODS.CLASS, has been defined as a subclass of SASHELP.FSP.OBJECT.CLASS. Since remote interfaces cannot contain visual components, your SCL CLASS methods will normally be subclasses of SASHELP.FSP.OBJECT.CLASS.

The following methods are defined in this class:

Method Name	SCL Entry	Label
GETROOTVALUE	CUSTOM.ROCF.METHODS.SCL	GETROOT
GETOTHERVALUE	CUSTOM.ROCF.METHODS.SCL	GETOTHER
GETPICTURE	CUSTOM.ROCF.METHODS.SCL	GETPIC
DISPLAYSOURCE	CUSTOM.ROCF.METHODS.SCL	DISPSRC

The GETROOTVALUE method will be called when the first real node value is clicked on in the `treeView1` component. The GETOTHERVALUE method will be called whenever any node other than the first node is clicked on in the `treeView1` component. Input parameters common to both of these methods are the `dataSetName`, the `levelColumn` number, the `textColumn` number and the `indexColumn` number assigned in `dataSetInterface1`. The parameter returned by both methods is an SCL list.

The GETOTHERVALUE method requires additional input parameters. These include values for the text of the parent node, the index position of the selected node in relation to its siblings, the depth or level of the parent node, as well as the depth of the current node.

The GETPICTURE method has two input parameters, the file reference for a SOCKET connection and the location of the stored GRSEG entry. A SOCKET connection exists between the SAS server and the client application. The server opens a connection and waits for the client to connect to it. The connection allows data to flow back and forth between the client and server as needed. A GRSEG entry is a vector graphics file produced by a SAS/Graph® procedure. Before it can be displayed in the Java applet, it must be converted to a bitmap. The SOCKET connection allows the GRSEG output on the SAS host to be streamed directly to the Java client, eliminating the need to create and store a bitmap file on the server.

Input to the DISPLAYSOURCE method is the location of the SOURCE entry associated with the selected employee. The method returns an SCL list containing the contents of that source entry.

GETROOTVALUE Method

The return value for a remote SCL method must be listed first in the METHOD statement. As mentioned earlier, this method returns an SCL list to the Java client. Using the values received for the input parameters, the SCL program opens the data set, determines the column name for the number passed in from the retrieved `levelColumn` property, and determines the column names for the text column and the index column. It fetches the first record in the data set and uses the GETVARC function to retrieve the values for the RANK, AREA, IMAGE, and INFO columns in the data set. These values are then inserted into the SCL list using the INSERTC function.

The SCL source code for the GETROOTVALUE method follows.

```
getroot: method list 8 dsetname $  levelvar
         textvar indexvar 8;

  list=makelist();
  dsid=open(dsetname);
  levelcol=varname(dsid,levelvar);
  textcol=varname(dsid,textvar);
  indexcol=varname(dsid,indexvar);
  rc=where(dsid,levelcol||'=1');
  rc=fetch(dsid);
  desc=getvarc(dsid,textvar);
  rank=getvarc(dsid,varnum(dsid,'rank'));
  area=getvarc(dsid,varnum(dsid,'area'));
  imageloc=getvarc(dsid,varnum(dsid,'image'));
  infoloc=getvarc(dsid,varnum(dsid,'info'));
  rc=insertc(list,desc,-1);
  rc=insertc(list,rank,-1);
  rc=insertc(list,area,-1);
  rc=insertc(list,imageloc,-1);
  rc=insertc(list,infoloc,-1);
  dsid=close(dsid);
endmethod;
```

The SCL source code for the GETOTHERVALUE method follows.

```
getother: method list 8 dsetname $ levelvar 8
          textvar 8 indexvar 8 pnodetext $
          index 8 pdepth 8 depth 8 ;
  list=makelist();
  dsid=open(dsetname);
  levelcol=varname(dsid,levelvar);
  textcol=varname(dsid,textvar);
  indexcol=varname(dsid,indexvar);
  rc=where(dsid,textcol||'='||
     quote(pnodetext));
  rc=fetch(dsid);
  startindex=getvarn(dsid,indexvar);
  if depth > 2 then
    rc=where(dsid,levelcol||' ='||pdepth||
    'and '||indexcol||' ge '|| startindex);
  else if depth = 2 then
    rc=where(dsid,levelcol||'='||depth||' and '
    ||indexcol||' ge '|| startindex);
  rowcount=0;
  do while(fetch(dsid)=0);
      rowcount+1;
  end;
  if rowcount > 1 then do;
       rc=fetchobs(dsid,rowcount);
       stopindex=getvarn(dsid,indexvar);
       wherestr=levelcol||"="||depth||" and ("
                ||startindex||" <=" ||
                indexcol||" <= "||
                stopindex||")";
  end;
  else if rowcount = 1 then
       wherestr=levelcol||"="||depth||" and "
       || indexcol|| ">= "|| startindex;
   rc=where(dsid,wherestr);
   rc=fetchobs(dsid,index+1);
   desc=getvarc(dsid,textvar);
   level=getvarn(dsid,levelvar);
   indexval=getvarn(dsid,indexvar);
   rank=getvarc(dsid,varnum(dsid,'rank'));
   area=getvarc(dsid,varnum(dsid,'area'));
   imageloc=getvarc(dsid,varnum(dsid,'image'));
   infoloc=getvarc(dsid,varnum(dsid,'info'));
   rc=insertc(list,desc,-1);
   rc=insertc(list,rank,-1);
   rc=insertc(list,area,-1);
   rc=insertc(list,imageloc,-1);
   rc=insertc(list,infoloc,-1);
   dsid=close(dsid);
endmethod;
```

The purpose of the GETOTHERVALUE method is to ultimately build a where clause that will return the same records as those displayed in the treeView for the selected node and that node's siblings. In this method, the SCL program searches for the text of the parent node and fetches that record. It then uses the GETVARN function to get the value of the INDEX variable. This will be the starting index for the final WHERE subset that is applied to retrieve the records for the siblings of the selected node.

If the depth or level of the selected node is greater than 2, a WHERE clause is applied to subset the data set. The `stopindex` value isn't needed. It instead retrieves records where the INDEX variable is GE to the `startindex` value. The final WHERE clause is assigned to the variable WHERESTR .

If the `depth` equals 2, a subset is applied to count the records where the LEVEL value is equal to 2.

If there is more than 1 row in the data set where LEVEL=2, the last record in that WHERE subset is fetched and GETVARN is again used to retrieve the value of the INDEX variable. This gives you the `stopindex` value for the final WHERE clause. The appropriate where clause is assigned to the SCL variable, WHERESTR.

If the value of ROWCOUNT equals 1, the final where clause is defined and assigned to the WHERESTR variable.

Once the final and appropriate where clause has been built, it is applied to the data set. The index value passed to the SCL method from the Java applet indicates the position of the selected node in relation to its siblings. It is important to note that, unlike the typical SCL index, the index for the Java treeView component begins at zero. It is therefore necessary to add one (1) to the index value to insure that the appropriate record in the subset is fetched. The GETVARC function retrieves the values for the RANK, AREA, IMAGE, and INFO columns in the data set. These values are then inserted in the SCL list using the INSERTC function.

GETPICTURE Method

The SCL source code for the GETPICTURE method follows.

```
getpic:
  method fileref $ imageloc $;
  libref=scan(imageloc,1,'.');
  catalog=scan(imageloc,2,'.');
  entry=scan(imageloc,3,'.');
  submit continue;
    goptions gsfname=&fileref gaccess=gsasfile
             device=gif373
             cback=white nodisplay;
    proc greplay igout=&libref.&catalog nofs;
       play &entry;
    quit;
    goptions reset=all;
  endsubmit;
endmethod;
```

The GETPICTURE method accepts as parameters a file reference and an image location string. The IMAGELOC variable contains the fully qualified name of the GRSEG entry such as EMPLOYEE.INFO.T9.GRSEG. Using the SCAN function the LIBREF, CATALOG, and ENTRY variables are assigned the appropriate values. A SUBMIT block then executes with the values of the SCL variables substituted for the ampersand references in the submitted code.

The GSFNAME option is assigned the file reference received from the calling method. This file reference points to a SOCKET connection described earlier. Using the GOPTIONS GSFNAME, GACCESS=GSASFILE, and DEVICE=GIF373, the results of the replayed graph are automatically streamed directly to the Java client. By streaming your graphics output, you don't have to generate a physical file such as a GIF file and store it on your WEB server.

PROC GREPLAY executes to replay the specified GRSEG entry.

DISPLAYSOURCE Method

The SCL source code for the DISPLAYSOURCE method follows.

```
dispsrc:
  method list 8 source $;
  list=makelist();
  rc=fillist('catalog',source,list);

endmethod;
```

The DISPLAYSOURCE method builds an SCL list using the MAKELIST function and then uses the FILLIST function to populate it with the contents of the SOURCE entry. This SCL list is returned to the calling Java method.

DEFINING THE REMOTE PROXY INTERFACE

Once you have defined your SCL class and tested the methods from within your SAS session, you can build your proxy interface to the class. A proxy is an entity that performs a function on your behalf. In the context of this application, a proxy is a Java class that adheres to the SAS model's interface and understands how to talk to the remote SCL object. webAF has a Remote Interface Wizard and a Proxy Generation Wizard to facilitate the creation and implementation of these proxies.

The first step is to open your project and select Remote Interface from the FILE menu shown in Figure 2.

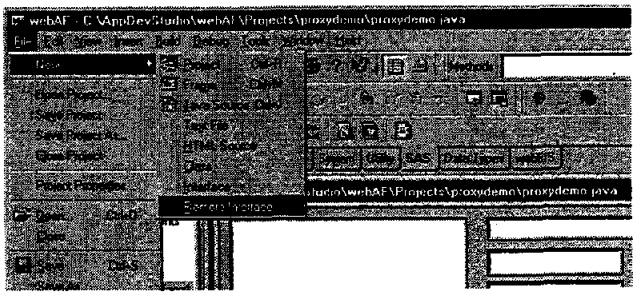

Figure 2 Selecting Remote Interface

The New Remote Interface Wizard shown in Figure 3, is invoked to help in the creation of the interface.

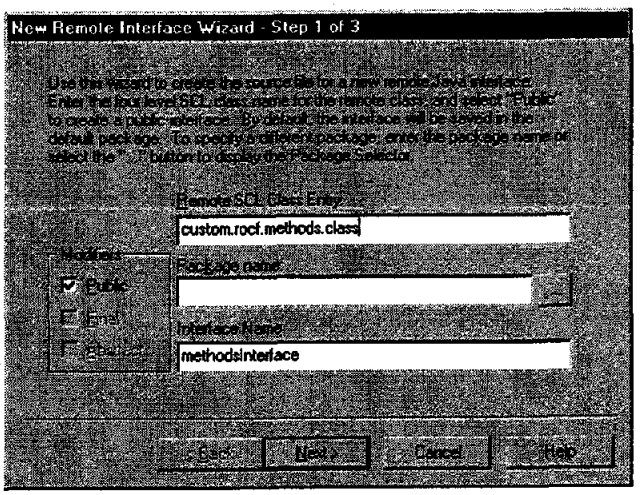

Figure 3 New Remote Interface Wizard – Step 1 of 3

For the Remote SCL Class Entry, specify the four level name of your CLASS entry. The interface name is automatically filled in for you. By default, the interface name is given the same name as your CLASS entry. You can change the name of the interface to be something more descriptive if you choose. For this example, you will use the default value.

When you click Next, the wizard displays the next window, Figure 4, automatically filling in the value for Extends. As noted earlier, com.sas.ComponentsInterface contains all of the basic webAF functionality, so your remote SCL interface will be an extension of this.

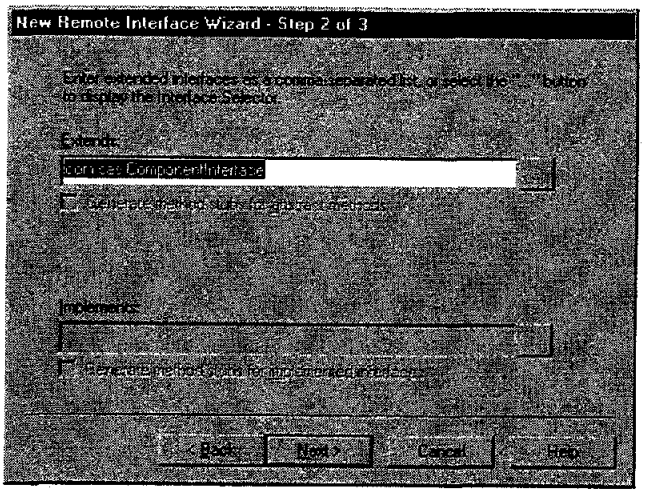

Figure 4 New Remote Interface Wizard - Step 2 of 3

Selecting Next will display the final window shown in Figure 5.

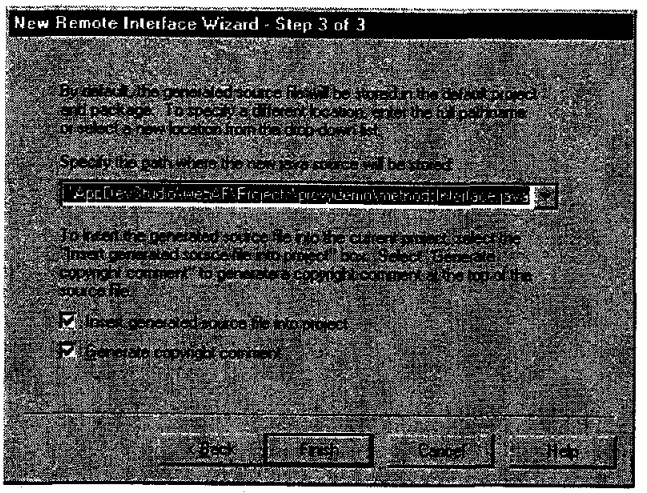

Figure 5 New Remote Interface Wizard - Step 3 of 3

In this window, you have the opportunity to change the location of the generated source file. You can also select whether or not to insert the source into the current project. In this example, the defaults are accepted and the new interface is inserted into the current project.

When you select Finish, a new window source entry displays the Java code shown in Figure 6.

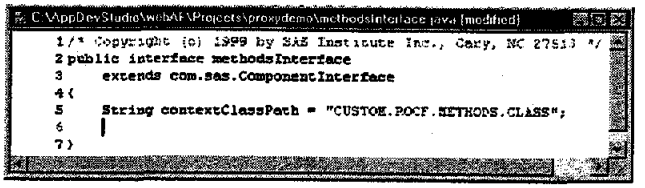

Figure 6 methodsInterface Source Code

The contextClassPath variable specifies the name of the CLASS entry. This is used to indicate where to locate the methods for this interface. Now you must modify the source code to include your method definitions. In this case, the following code is added to the methodsInterface.Java source file:

```
com.sas.collection.hlist.HListInterface
  getrootvalue(String dataset, int
               levelvar, int textvar,
               int indexvar);

com.sas.collection.hlist.HListInterface
  getothervalue(String dataset, int
                levelvar, int textvar, int
                indexvar, String
                pnodetext,
                int index, int pdepth, int
                depth);
void getpicture (String fileref, String
                entryname);
com.sas.collection.hlist.HListInterface
  displaysource( String source);
```

The first method, GETROOTVALUE, specifies com.sas.collection.hlist.HListInterface as the return type. This construct provides the necessary interface for an SCL list to be passed to a Java application. The method name follows the return type. Within the parenthesis, the data types for the parameters that will be passed to the SCL class are specified.

The GETOTHERVALUE method also has a return type of com.sas.collection.hlist.HlistInterface. This method accepts a number of parameters. The first, `dataset`, is of type String. The parameters `levelvar`, `textvar`, and `indexvar` are of type int. The `pnodetext` parameter is of type String and all remaining parameters are of type int.

The GETPICTURE method has no return type, so void is specified. This method accepts two parameters `fileref` and `entryname`, both of type String.

Again, com.sas.collection.hlist.HlistInterface is specified as the return type for the DISPLAYSOURCE method. This method accepts one parameter, `source`, of type String.

Once you have completed specifying the source code for your methods, you can compile them and generate the remote proxies. This can be accomplished in a single step by right clicking in the source window and selecting Generate remote proxies from the popup list. The ProxyWizard window, Figure 7, is displayed allowing you to select the protocols to use to communicate with your remote SCL object. In this case, the default is SAS.

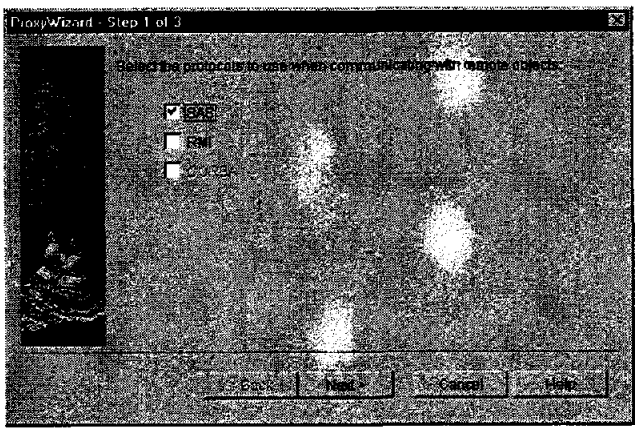

Figure 7 ProxyWizard - Step 1 of 3

Select Next to display the second window (Figure 8). Here you must decide whether on not to insert the proxies into the current project. By default, No is selected. For this example, change the selection to Yes.

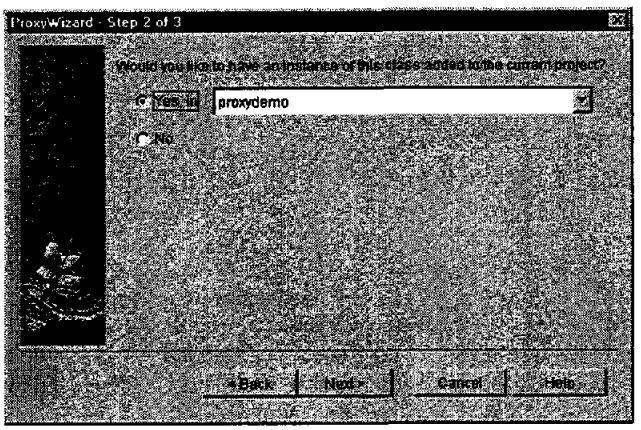

Figure 8 ProxyWizard - Step 2 of 3

Selecting Next takes you to the final window of the ProxyWizard, Figure 9, which simply displays all of the information about your proxy and your selections.

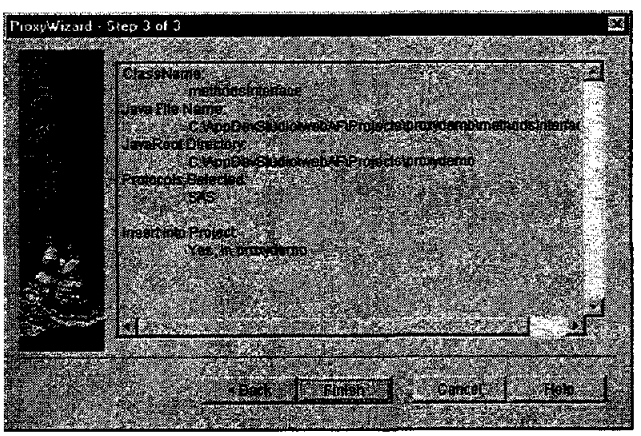

Figure 9 ProxyWizard - Step 3 of 3

Upon selecting Finish, the proxy files are generated and a component is added to the current project called `methodsInterface1`.

With the proxy generated, the functionality of your remote SCL classes can be accessed from the Java application. Before doing that, however, some of the components need to have event handlers defined for them. webAF allows you to define an event on a component by selecting Handle Event from the popup list for the currently selected component.

EVENTS

In webAF an event is the link between an action (such as a mouse click, a window opening or closing, or a value being changed) and a Java program that provides instructions on how the applet or application should respond. The Java program that responds to the event is called the event handler.

Defining an event handler on a source component such as the `treeView1` component in this example, allows the source component to generate an event which can then be received and acted upon by a second component called the listener.

In this example, you need to define 3 event handlers, one for each of the `treeView1` , `viewimage`, and `viewinfo` components. These three components will each be a source component for its respective handler. When you select the Handle Event item from the popup, it opens the New Event Handler window. For each of the three components in turn, select "Write your own code" when prompted with "Which type of handler do you want to create?". Let it default to "Within the frame class" for "How do you want to implement the hander?".

For the `treeView1` component, select "when the event occurs on `treeView1`". This opens the Select Source Component and Event window. Select `treeView1` for the source component, and `actionPerformed` for the method and select OK. webAF automatically adds the event handler classes for the Java source.

For the `imageview` component, select "when the event occurs on `viewimage`" and then select `viewimage` for the source component and select `actionPerformed` for the method. Do the same thing for the `viewinfo` component.

The `actionPerformed` event is fired when you make a selection in the `treeView1` component or when the pushButton components, `viewimage` or `viewinfo`, are selected. So, when a selection is made, the application needs to ultimately execute one of the four methods defined in the remote interface, `methodsInterface`. To accomplish this, the Java source code must be modified.

ADDING JAVA CODE TO THE APPLET

With the event handlers added to the Java source code, additional code can be added to complete the application. First some import statements are needed to import the appropriate classes or packages of classes. In this case you need to include the following import statements.

```
import com.sas.collection.hlist.*;
import com.sas.visuals.NodeView;
import Java.awt.Image;
import Java.awt.Toolkit;
import Java.io.ByteArrayOutputStream;
import
com.sas.sasserver.inputstream.SreamInterface;
import
com.sas.sasserver.inputstream.ReadExitInterface;
```

The first import statement imports the `hlist` package which includes `HlistInterface`. This is needed because several of the methods return this type. The com.sas.visuals.NodeView class is needed to determine which node is selected in the `treeView1` component. The java.awt.Image class and the java.awt.ToolKit class are needed to construct an image from the output received from the GREPLAY procedure. The com.sas.sasserver.inputstream.StreamInterface and com.sas.sasserver.inputstream.ReadExitInterface classes are needed to create the SOCKET connection so the output from the GREPLAY procedure can be streamed directly to the Java client.

Since the `ReadExitInterface` is an abstract interface that cannot be instantiated so the project must implement it. Add the following statement to the class definition as shown in figure 10.

```
.com.sas.sasserver.inputstream.ReadExitInterface
```

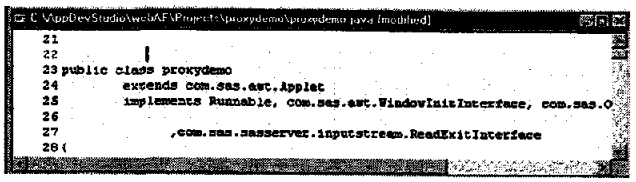

Figure 10 Adding ReadExitInterface implementation

To implement the `ReadExitInterface`, you must also include an implementation for each of the methods defined in it. In this particular interface, only one method exists, the `processReadbuffer` method. So the following code is added.

```
public void processReadBuffer(byte[] buffer,
                              int len)
{
    out.write(buffer,0,len);
}
```

Also, add the following fields defining them as transient. The transient modifier defines the fields so they will not be part of the class's persistent state.

```
transient String infoloc, imageloc;
transient proxydemo currentapplet;
transient ByteArrayOutputStream out;
```

At initialization of the application, the `textAreal` and `imageView1` components should be hidden. To do this, the `setVisibility` method can be used to set the visibility to false. The following code is added to the `postInit` method.

```
public void postInit(){
        // Note: Add initialization code here
        imageView1.setVisible(false);
        textAreal.setVisible(false);
        currentapplet=this;
}
```

In addition, the `currentapplet` variable is assigned the value referenced by `this` which is an implicit argument that refers to the current object. In this example, `this` refers to the `proxydemo` applet object. You can relate the implicit argument, `this`, to _SELF_ in SCL.

EXECUTING THE EVENTS

Remember that when a node is selected in the `treeView1` component, an event should be fired to execute our code. The event handler code for this component should determine the selected node, find that node in the data set, and fetch the variable values related to that node. Figure 11 shows the frame after a node has been selected.

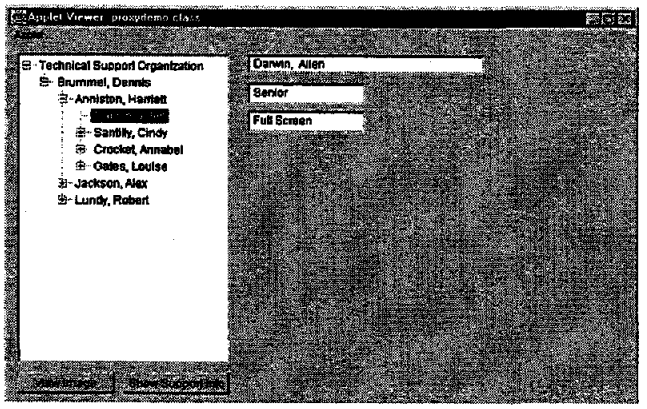

Figure 11 After execution of event handler for `treeView1` component

The code for the `actionPerformed` event handler needs to be entered for the `treeView1` component.

```
public void treeView1ActionPerformedHandler1
(Java.awt.event.ActionEvent event){
  // NOTE: Add new code here
  textArea1.setVisible(false);
  imageView1.setVisible(false);
  NodeView currentnode=
       treeView1.getSelectedNode();
  NodeView rootnode=treeView1.getRoot();
  NodeView parent;
  int depth, index, parentdepth;
  String datasetname=
       dataSetInterface1.getDataSetName();
  int indexcolumn=
       dataSetInterface1.getIndexColumn();
  int levelcolumn=
       dataSetInterface1.getLevelColumn();
  int textcolumn=
       dataSetInterface1.getTextColumn();
  HListInterface textlist=null;
  if (currentnode != rootnode)
  {
    depth=currentnode.getDepth();
    parent=currentnode.getParent();
    index=parent.getIndex(currentnode,0);
    if (depth == 1)
     textlist=methodsInterface1.getrootvalue
       (datasetname, levelcolumn,textcolumn,
        indexcolumn);
  }
  else if ( depth > 1)
   {
     parentdepth=parent.getDepth();
     textlist=methodsInterface1.getothervalue(

     datasetname,levelcolumn,textcolumn,
              indexcolumn,
     parent.getNodeText(),
              index,parentdepth,depth);
   }
   name.setText(""+textlist.getItem(0));
   emplevel.setText(""+textlist.getItem(1));
   area.setText(""+textlist.getItem(2));
   imageloc=""+textlist.getItem(3);
   infoloc =""+textlist.getItem(4);
   }
}
```

The event handler code for the `treeView1` component must perform a number of tasks.

- It hides the `textArea1` and `imageView1` components. These should only display after selecting a node and then clicking on the perspective pushButton components, `viewinfo` or `viewimage`.
- The selected node is retrieved and assigned to the NodeView field, `currentnode`, using the `getSelectedNode` method of the treeView class. Also, the rootNode is retrieved and assigned to the `rootnode` field.
- Other fields are defined for later use. These include `parent`, `depth`, `index`, and `parentdepth`.
- The value of the `dataSetName` property of `dataSetInterface1` is queried and assigned to the datasetname field. Likewise, the values of the `indexColumn`, `levelColumn`, and `textColumn` properties are assigned to the fields, `indexColumn`, `levelColumn`, and `textColumn` respectively. Remember that these properties contain the position of the variable within the corresponding SAS data file.
- Next an `HlistInterface` field, called `textlist`, is defined for later use. The `textlist` field will ultimately be assigned the contents of the SCL list returned by the `methodsInterface1.displaysource` method call.
- If the `currentnode` is the same as the `rootnode`, no action should occur. This is because the `rootnode` contains descriptive text and does not correspond to a record in the SAS data file.
- However, if `currentnode` is not the same as the `rootnode`, invoke the `getDepth` method to return the value of the `levelColumn` variable in the SAS data file. Get the `parentnode` using the `getParent` method. Then retrieve the relative index location of the `currentnode` within the current list of children for the parent node. This is necessary to determine which record to fetch in the SCL method when the final WHERE clause is applied.
- If the depth of the current node is 1, you clicked on the first real node displayed in the `treeView1` component. This should correspond to the first record in the SAS data file. Invoke the methodsInterface1.getrootvalue method passing the values for the datasetname, `levelColumn`, `textColumn`, and `indexColumn`. The SCL proxy method executes and returns a list containing the variable values for the selected record.
- If the selected node depth is anything other than 1, get the depth of the parent node. Invoke the methodsInterface1.getothervalue method passing the values for datasetname, `levelColumn`, `textColumn`,`indexColumn`, `parentnode` text, `index`, `parentdepth`, and `depth`. When the SCL proxy method executes, it returns a list of the variable values for the selected record.
- The list is assigned to `textlist` and from the `textlist` `HlistInterface` item, the individual `HlistItem` values are retrieved and assigned to the corresponding fields. The `getItem` method of the `listInterface` class is used to retrieve the individual `HlistItem` values. The first three items are assigned to the text field components, `name`, `area`, and `emplevel` using the `setText` method. The nonvisual fields `imageloc` and `infoloc` are assigned their values.

This completes the code for the `textView1` event handler.

When the `viewimage` pushButton component is selected, the event will trigger execution of our event handler code. This will create a SOCKET connection and invoke the `methodsInterface1.getpicture` method. Figure 12 shows the frame after the `viewimage` button has been selected.

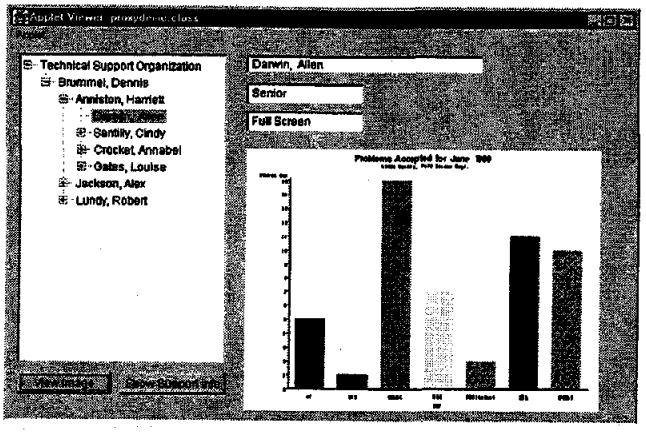

Figure 12 After selecting the `viewimage` component

Event Handler code for viewimage component

```
public void viewimageActionPerformedHandler2
   (Java.awt.event.ActionEvent event)
{
  // NOTE: Add new code here
  InputStreamInterface isinterface;
  try {
   isinterface=(InputStreamInterface)
   __rocf.newInstance(InputStreamInterface.class,
   connection1);
   String fileref=
     isinterface.makeInputStream(currentapplet);
   out= new ByteArrayOutputStream();

   methodsInterface1.getpicture(fileref,imageloc);
   byte[] buffer=out.toByteArray();
   out.close();
   Image img=
Toolkit.getDefaultToolkit().createImage(buffer);
   imageView1.setImage(img);
   imageView1.setVisible(true);
   textArea1.setVisible(false);
  } catch (Exception e) {
       System.out.println("Exception: " + e);
  }
}
```

When the `viewimage` component is selected, the following sequence takes places.

- An `InputStreamInterface` called `isinterface` is defined for later use.
- A `try/catch` is used to capture any exception that might occur.
- Interfaces themselves can't be instantiated but since they are data types, in the same manner as classes are data types, you can define a field as a data type of an interface by casting the object to the specified interface. In this case, `isinterface` is assigned a new instance of an input stream for `connection1`.
- The String variable, `fileref`, contains a reference to a SOCKET connection which is created using the `makeInputStream` method. This actually creates a `ReadExitInterface` so you can ultimately read the information that is returned to the Java client from the remote SCL proxy method, methodsInterface1.getpicture, via this SOCKET connection.
- The `out` field is instantianted as a new `ByteArrayOutputStream`.
- The remote proxy method, `methodsInterface1.getpicture` executes passing the value of the `fileref` and `imageloc` fields. The method executes and using the GOPTIONS specified, the GRSEG entry is replayed and sent as a graphics stream file to back to the Java client.
- A byte array is created from the stream file and using the Java Toolkit class, the buffer is converted to an Image object.
- The image, `img`, is assigned to the `imageView1` component using the `setImage` method.
- The `imageView1` component is made visible and the `textArea1` component is hidden.

Finally we must add code for the event handler associated with selection of the viewImage pushbutton. When this button is pushed, an event should be fired to invoke the remote SCL proxy method methodsInterface1.displaysource. This will ultimately retrieve the contents of a SOURCE entry in a SAS catalog and display the text in the `textArea1` component shown in Figure 13.

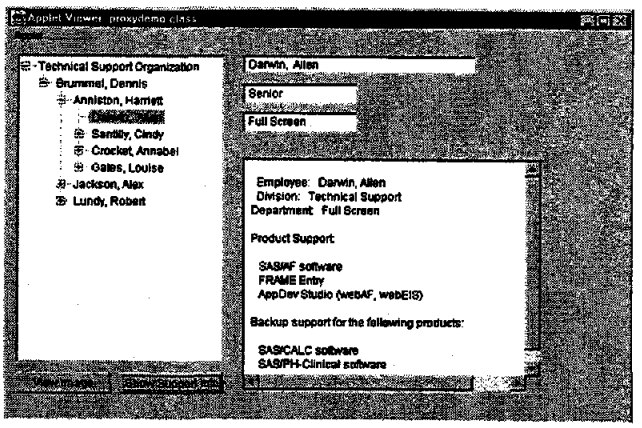

Figure 13 After selecting the `viewinfo` component

Event handler code for viewinfo component

```
public void viewinfoActionPerformedHandler3
   (Java.awt.event.ActionEvent event)
{
  // NOTE: Add new code here
  HListInterface textlist =
     methodsInterface1.displaysource(infoloc);
  textArea1.setModelInterface(
     (com.sas.ModelInterface) textlist);
  textArea1.setVisible(true);
  imageView1.setVisible(false);
}
```

For this event, the method is invoked passing the value of the `infoloc` field. The results of this method invocation returns an SCL list and the contents are assigned to the

`HlistInterface` item `textlist`. The `textlist` is then defined to be the model for the `textArea1` component. Using the `setModelInterface` method, the `textlist` is cast to `com.sas.ModelInterface` object and the model is assigned to the `textArea1` component and it automatically displays the contents. The `textArea1` component is made visible and the `imageView1` component is hidden.

This completes the additional code needed for this example project.

SUMMARY

webAF generates much of the Java code for you as you build your projects and define your remote proxies. In this example, some Java code is needed to full implement the functions the application. The amount of Java code necessary for any given application be greatly reduced by substituting SCL methods for Java code when accessing SAS data.

Building the remote proxy requires some knowledge of the parameter types you will be passing to your remote SCL method and also an understanding of the return types. Being familiar with writing classes in SAS/AF software and Object Oriented Programming (OOP) can speed up the process.

You can also write remote SCL classes that can be reused by many projects. While the `getrootvalue` and `getothervalue` methods are defined specifically to return a list of column values, they could be modified to make them more generic and therefore available for use in other projects.

The `getpicture` method could be expanded to not only replay stored graphics but maybe to even execute SAS/GRAPH software procedures. The `displaysource` method could be modified to retrieve the contents of stored flat files as well as OUTPUT, LOG, and SOURCE entries in catalogs.

Using remote SCL methods can make your Java application much more robust especially when working with SAS procedures and exploiting the power of the SAS system.

CONCLUSION

Using webAF in conjunction with SCL methods can make not only coding your Java application easier, since it may cut down on the amount of Java source code, it may also speed up your coding process if you're very familiar with SAS and SCL but have limited knowledge of Java.

The wizards provided in webAF also make writing your applications much easier. The wizards help build the basic code for you which you can modify to fit your needs.

For more information regarding webAF and AppDev Studio, refer to the online documentation and help. You may also find additional information on the World Wide Web at www.sas.com/appdev_studio.

REFERENCES

Flanagan, David (1997) *Java in a Nutshell, Second Edition,* O'Reilly & Associates, Inc.