SAS® Software and Java for Interactive Graphics

Andrew A. Norton, Trilogy Consulting Corporation, Kalamazoo, Michigan

INTERACTIVE GRAPHICS

I have long wanted to be able to write my own interactive graphics programs. SAS/INSIGHT® does an admirable job of providing canned facilities for interactively exploring data, but it is not intended for applications development.

The key element I am seeking is to be able to draw my own elements on the screen, and then associate clicks or drags upon the screen with particular elements. When the user takes an action, I can respond by drilling down, excluding or including, or whatever else I wish. I then draw a new screen and repeat the process.

INTERACTIVE GRAPHICS WITH SAS/AF

I used to provide these capabilities using SAS/AF®. SAS/GRAPH® vector graphics are made up of individual graphical segments such as lines, points, and text strings. These segments are sequentially numbered, with the first segment drawn being number 1.

The SAS/GRAPH Output class provides a _get_info_ method that tells the coordinates of a click upon the screen. The SCL list that is returned also contains an element named SPOTID that corresponds to the SAS/GRAPH graph segment number.

One way to identify a click on a graph is by checking the coordinates against the location of elements written to the screen. But a more direct means is to make use of the graph segment number itself. The only issue (which can be substantial) is to figure out which element corresponds to the given graph segment number.

Luckily, the segment numbers are not assigned at random, but are assigned in sequence as the elements are written to the screen. So if we know what elements appear on the screen and in what order they were drawn, we can translate a segment number back into a given element.

Procedure Graphics

For graphs produced by SAS/GRAPH procedures, we do not have access to the internal program code. We can tell by inspection, however, that the procedure may draw the X-axis, then the Y-axis, then the titles, then the bars from left to right. Given this information we can devise a formula to figure out which segment number corresponds to which segment number.

Customized Graphics

I also write custom graphs upon the screen using the Data Step Graphics Interface (DSGI) of SAS/GRAPH (SAS Institute, 1990.) DSGI lets you individually write whatever elements you wish upon the screen. It would be nice if DSGI would provide you with the segment number of each element as it wrote it, but it doesn’t.

So instead, as I write each element to the screen, I keep a list. Then when the user clicks upon segment number 12, I can check my list and find that that object corresponds to Ohio.

I have long made use of the object-oriented features of SCL. Management of graphics through composite objects is a well-understood field (see Gamma, et. al. 1995.) So in practice I often will have non-visual objects (representing data points) that can draw a graphical representation by invoking methods in a Graph Drawer class. The Graph Drawer executes the DSGI commands (in SCL) and at the same time maintains a table connecting each graphical element number with the content object that is being represented.

For example, I might have an Experimental Trial class. This class would support a method draw(graph_drawer). So when the draw method was invoked on a particular point, I might get a sequence such as the following.

1. `call send (trial1, ‘draw’, graph_drawer);`
   Trial1 is object id 37.

2. The trial1 object sends the message
   ```
   call send (graph_drawer, 'draw_point',
      _self_, 23, 34);
   ```
   The graph drawer records that segment 1 was requested by object 37.

Later, the user clicks on the graph widget, with the following sequence:

1. The system invokes the _object_label_ method on the graph_widget object.
2. The _object_label_ method obtains the clicked segment number:
```
call send (graph_widget,
'_get_info_', info_l);
spotid = getnitemn(info_l,'spotid');
```
3. The _object_label_ method notifies the graph_drawer that an object was clicked:
```
call send (graph_drawer, 'clicked',
spotid);
```
4. The graph drawer looks up segment number 1 and finds that it corresponds to object 37.
5. The graph drawer notifies object 37 that it has been clicked.
```
call send (37, 'clicked');
```

Thus content objects can recognize when their corresponding graphical representation has been clicked and take appropriate action.

Systems constructed using these techniques work well but tend to run slowly. One problem is that every change on the screen requires an entire redraw using the DSGI commands.

INTERACTIVE GRAPHICS IN JAVA

Java provides advantages over SAS for customized graphics programming.

Speed

Java draws graphics on the screen much faster than the DSGI method I had been using. It is also possible to selectively repaint sections of the screen.

Animation

Animated graphics are a key feature of Java. The multi-threading capabilities make it easy to simultaneously monitor actions by the user (keyboard and mouse) while animating the screen.

Modeless Dialog Boxes

In Java, you can have multiple windows open at a time and communicating with each other. This is especially nice for interactive graphics, as you can select points on one window and edit their properties on another.

Dynamic Downloading

Java programs can be moved from one machine to another at runtime without recompilation. It is therefore possible to deliver content together with the rendering engine for that content, so the user can interact locally with the display.

In addition, the Java security model allows you to run other people's programs without fear of viruses.

Cost

Java runtime virtual machines are available for free.

SAS AND JAVA TOGETHER

Java provides a superb platform for object-oriented graphical programming, but of course does not have the data management and analysis capabilities of SAS. So I would like to combine SAS on the back end with Java as the front end. This lets us use existing programs and expertise, and take advantage of the SAS System's extensive library of tools developed over many years.

SAS/INTRNET® allows connections from Java to SAS data via ODBC, or submission of SAS batch language from Java. But the most flexible way to connect Java to the SAS System is to connect to SCL objects which can then obtain data or submit batch SAS code. This also lets us stay within an object-oriented framework: If we are using object-oriented Java and object-oriented SCL, what we would really like is to send messages directly from Java objects to SCL objects. So when a user clicks on a glyph on a Java screen, a 'click' method will execute on the corresponding SCL object.

CONNECTING JAVA TO SCL

This paper will now present a method for sending messages from Java objects to SCL objects. Because it uses sockets (a base SAS capability) it requires no more than base SAS and SAS/AF. In fact, even the SAS/AF license could be dispensed with, because no license is required merely to execute a SAS/AF application.

CORBA

At the time of this writing, SAS Institute has not announced a mechanism for communicating with SCL objects from Java. There is an existing standard for communicating with objects across machines and across languages, called CORBA (Common Object Resource Broker Architecture.) If the SAS System supported CORBA, or even a proprietary alternative, then this would be the way to go. In the meantime, it is still possible to connect Java objects to SCL objects using sockets.

Sockets

Both Java and SAS support the ‘socket’ protocol, which is a low-level means of sending data from one program to another. The socket has an address made up of a TCP/IP address and a port number. So messages can be sent through the socket from one machine to another, or on a single machine (for example to use Java and SAS together.)

On the Java side, the socket simply appears to be a stream that you can write text to. On the SAS side, the socket appears to be an infile that you can read text from. So you can transmit information from Java to SAS by having Java write into the socket and SAS read from it. Similarly, the same socket can be used in reverse to send data from SAS to Java.

Java-to-SCL Gateway: An Overview

The interesting part is that by developing our own conventions for interpreting strings sent across the socket, we can send messages to SAS objects rather than just exchange data. This is possible because SAS allows messages to be dynamically constructed. For example, suppose we want to send the message ‘get_name’ to SCL object number 37, returning a string. If we were in SCL, we could simply execute

```
call send (37, 'get_name', name);
```

But to send this from Java, we need to come up with both an equivalent call in Java and a way to convert such a call to a string. Using the Java gateway I have developed, this example could be invoked as

```
   String name = MySAS.sendc(37,
         'get_name', '#C');
```

The sendc method of the SASConnection class would convert this message to a string

```
   '37 get_name #C'
```

And when the SCL server program receives this string, it would execute:

```
   call send (37, 'get_name', arg_l);
```

The result would be pulled from the argument list and sent back along the socket to the Java session, which would return it as if the method had been directly executed in Java.

SETTING UP THE SOCKET

Java side of the socket

The java.net.Socket class lets you create a socket with a statement such as

```
Socket theSocket = new Socket(“127.0.0.1”,
                              5050);
```

where the first argument is the remote host TCP/IP address and the second argument is an arbitrary port number.

Once the socket is created, Java output and input streams can be established. When Java is sending information into the socket, the toSAS output stream (java.io.PrintStream) is used (as if writing output to a file). When Java is reading information from the socket, the fromSAS input stream (java.io.DataInputStream) is used.

```
try {
toSAS = new PrintStream
      (theSocket.getOutputStream(), true);
fromSAS = new DataInputStream
      (theSocket.getInputStream());
} catch (java.io.IOException e) {}
```

SAS side of the socket

Sockets are defined in SAS using the SOCKET option of the FILENAME statement, specifying the port number instead of a file specification.

```
filename fromjava socket ‘:5050’ server;
```

This filename can be opened in SCL or a DATA step.

When you attempt to read a line from the filename, SAS will wait until a line is sent. Then you can treat it like an ordinary input process. In SCL this looks like:

```
submit;
   filename javasock socket ':5050'
```

```
    server reconn=0;
endsubmit;
javasock = fopen('javasock', 'w');
do while (1);
   rc = fread(javasock);
   do while (fget(javasock,token) eq 0;
      /* process token */
   end;
end;
```

Similarly, you can write back to the filename, and Java will be able to read from the socket what you write.

INTERCOMMUNICATION PROTOCOL

Now that we have a way of sending strings back and forth between Java and SCL, we need to be able to translate Java messages to strings, and then strings to SCL messages.

Return values

At this moment we encounter differences between Java messages and SCL messages. First of all, Java methods have a return value and SCL methods do not. Secondly, Java parameters are passed-by-value while SCL parameters are copied in and out. So a Java call to retrieve a value might look like

```
        value = objref.getValue();
```

while the equivalent SCL call might look like

```
        call   send   (objref,   ‘get_value’,
                    value);
```

I allow Java to specify zero or one output parameter to be returned from SAS. When Java assembles the string to be sent to SAS representing the method call, the output parameter is designated with a special placeholder (‘#N’ for integer, ‘#F’ for floating point, ‘#C’ for character). So for the current example, if objref=37 then the string set across the socket would be

```
37 get_value #N
```

and the first parameter (an integer) would be returned to Java. If an SCL class supported a method

```
call  send  (objref,  'get_zip',  'Trilogy',
           zip);
```

then the string sent would be

```
37 get_zip 'Trilogy' #N
```

and the second parameter (an integer) would be returned to Java. The SCL server program returns whatever value is designated by the special symbol.

Object Identifiers

How does the Java user identify which SCL object is to be the target of the message?

Every SCL object is identified by an integer. So once you have one object ID, you can send it methods to have it return associated object ID’s that it knows about. For example, you might have a directory object that can return object ID’s by name. Once you know the SCL object id of that object (in this case stored in the variable directory), you can use the directory:

```
int  person  =  mySASsession.sendn(directory,
             "get_person", "Andy", "#N");
String companyName=mySASsession.sendc(person,
             "get_company", "#C");
```

But how do you get started? How do you get that first object reference when object reference numbers are unpredictably assigned? For this purpose the gateway provides the ability to access a SAS class object by name. Because SAS classes are themselves objects, they can have methods of their own. When the first argument is a string, the SCL server treats it as a class name, loads the class, and sends the message. For example, you can create a new object of the class SASHELP.FSP.OBJECT and obtain its id with the call:

```
int objref =
mySASsession.sendn("SASHELP.FSP.OBJECT",
                   "_new_", "#N");
```

SCL Lists

In SCL, method parameters may be (and in fact often are) lists of many different elements of varying types. The current version of this gateway provides no support for SCL lists.

JAVA INTERFACE

As you may have already noticed, Java is a typed language. Return values from SAS arrive in the form of strings (because they were sent over a socket) yet they may have originated from SAS as numbers. Furthermore, Java distinguishes between integers and floating point numbers (in fact, there are further distinctions that need not concern us now).

So sending a method to SAS may result in a return value of void (no return), character, float, or integer. I have written different send methods to be used in each of these cases.

Putting it all together

The SASConnection class provides access to SAS method calls. There are four method names:

`sendv` returns **`void`**
`sendc` returns **`string`**
`sendf` returns **`double`**
`sendn` returns **`int`**

For each of these method names, there are two ways of specifying the SCL target object: a classname or an SCL object ID. So there are eight methods in total.

Two sample method signatures are:

```
public String sendc (String classname, String
method, String parms) throws IOException;

public int sendn (int objID, String method,
String parms) throws IO Exception;
```

Implementation

Here is the implementation of one of the methods. The others are very similar. A string is assembled containing the target, method name, and parameters. After the string is sent to SCL via the socket, SCL sends back a return value, which is then cast to the appropriate Java type.

```
public    String    sendc    (int    objID,
 String   method,   String   parms)  throws
 IOException                              {
   toSocket().println(String.valueOf(objID)+
                  " "+"'"+method+"' "+parms);
   String token = fromSocket().readLine();
   return                  token.substring(1,
                          token.length()-1);
}
```

SAS SERVER

Now let’s look at what happens on the SCL side when a message is received.

The file is opened and a record is read from the file. The program will wait on this statement until a record is available or the connection is dropped.

The incoming string contains an identifier of the target, the method name, and the parameters (as a single string.) This is parsed using the scan() function. Characters are contained within single quotes so they can be distinguished from numbers.

The first token identifies the target. If it is numeric, this is the SCL object ID. If it is character, the object ID of the class is obtained with the loadclass() function.

The second token should always be character, and identifies the method name.

The remainder of the string contains parameters (if there are no parameters it will be blank). These parameters may be mixed character and numeric. There also may be one of the return value symbols ‘#N’, ‘#C’, or ‘#F’. The parameters are placed on an SCL list and passed to the method using the call apply routine. The return value symbols need special processing. They are changed to an appropriate missing value (numeric or character) after noting the position.

The method is called using the call apply routine, such as

```
   call apply (target, method, parm_l);
```

After the method has been executed, the return value (if any) is retrieved from the specified position in the parameter list, and sent back to Java via the socket.

SAMPLE APPLICATION

The capabilities of the Java to SCL gateway can be demonstrated with an interactive regression program.

A Java applet displays a canvas showing data points with a regression line through them, together with the slope and intercept of the line. Clicking on a point opens a dialog window showing the properties of the point. This dialog box is nonmodal; You can change from one point to another by clicking in the graph window while the dialog box remains open.

The dialog box shows the coordinates of the point and lets you select or deselect the point (for regression calculations.) When you deselect the point, it is removed from the regression. The regression line and statistics change accordingly. If you select the point again, the regression line and statistics will change back to what they were.

Behind the scenes, all of the computations are being done by SAS. The data points are also stored within SAS. Each data point has its own associated non-visual object in SCL. Whenever a data point changes status (from included to excluded or vice versa) the statistics are recalculated.

When the connection to SAS is established, Java requests information about every point in the data set: the coordinates and the SCL object ID representing the point. When a point is excluded or included, a message is sent from Java to the point object. This causes the statistics to be recalculated on the SAS side.

There must be explicit programming on the Java side to redraw the regression line and statistics when a point changes status. It is not possible for SCL objects to notify Java objects that their state has changed. Java programs must know when to check for a possible change in a derived value. This is further discussed in the Limitations section below.

LIMITATIONS

Performance

The speed of individual method calls is acceptable (perhaps 100 per second.) But when there are many data points there need to be many method calls to set up the initial graph.

The trick to speeding up the application is to reduce the number of method calls. I had started out with three method calls for each SCL point object:

- getX()
- getY()
- getStatus()

I reduced the number of method calls simply by adding another method getInfo() that returned X, Y, and Status concatenated into a string. This improved performance proportionately.

In general it is often useful to add a few extra methods that would not be elegant in a local object context, but serve to improve performance in a distributed context.

No Callbacks

A common technique in object-oriented programming is to follow the Observer pattern (Gamma, 1995) in which objects can request to be notified of changes in other objects.

Unfortunately, such an architecture is not possible in the current system. Suppose Java sends a message changing the value of SCL object X, and this change causes a change to SCL object Y. It is not possible for object Y to send an unsolicited message back to Java with a change notification.

In fact, SCL can initiate no messages to be sent to Java at all. The SCL server I have developed waits for messages and responds to them. It is a mechanism for messages to be sent from Java to SCL, not the other way around.

The root cause of this limitation is that the current version of SCL is single threaded. When the server is listening for messages coming in on the socket, it is waiting for input and not doing anything else. So you can't send out messages while simultaneously listening for messages coming in.

Firewalls

Firewalls are programs that reduce the risk of hostile activity limit the type of connections made to a server. They inspect the requests made from the Internet and determine which to allow by checking the source and nature of the requests.

Many firewalls block sockets because the flexibility of sockets allows them to be used for hostile purposes. Firewalls can cause problems for this Java-to-SCL gateway because it is based upon sockets.

Firewalls usually allow HTTP (web) connections, so it is possible to send strings to servers in the guise of HTTP requests. The strings can then be relayed to SCL by a program (Java or otherwise) on the server. It's slower than using sockets directly, but it works.

CONCLUSION

The ability to send messages from Java objects to SCL objects lets us unite two different object-oriented systems within the object-oriented paradigm. A central tenet of object-oriented programming is the encapsulation of implementation within a minimal and stable interface. Using this system, we can access SCL objects from Java **as objects**.

A typical approach is to have a Java object acting as a proxy for each SCL object of interest, forwarding messages from the Java proxy to SCL. With this approach the use of the SASConnection class is deliberately concealed from the body of the application code, permitting enhancements and migrations to other approaches in the future.

This encapsulation of the means by which messages are sent to SCL is a key point, as my intention is to use these techniques as a transitional stage to what I hope will be SAS software support of industry-standard object connectivity solutions such as CORBA in the future.

ACKNOWLEDGEMENTS

SAS, SAS/AF, SAS/CONNECT, SAS/GRAPH, SAS/INSIGHT, and SAS/INTRNET are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

REFERENCES

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley.

SAS Institute, Inc. (1990), *SAS/GRAPH Software: Reference, Version 6, First Edition, Volume 1*, Cary, NC: SAS Institute Inc.

AUTHOR CONTACT

If you would like further information please contact:

Andrew A. Norton
Research and Development Strategist
Trilogy Consulting Corporation
5278 Lovers Lane
Kalamazoo MI 49002
(616) 344-4100
aanorton@trilogyusa.com