Paper 179

Extending the Life of Your AF Application

Exploiting the Model-Viewer Paradigm

Gregory S. Barnes Nelson, STATPROBE, inc., Raleigh/ Durham, North Carolina

Abstract

SAS/AF FRAME technology has long been the mainstay of our visual entrée into traditional SAS applications. The World Wide Web has taken a front-seat to most application development efforts in an attempt to surface more and more information to entire enterprises. This interest, in part, has forced developers to look more critically at not only what technology we use to develop, but also how we can capitalize on our recent investments in applications development.

This paper attempts to dispel the notions that SAS/AF may be an outmoded technology. Rather, here we show how we can exploit the lessons learned from these applications by surfacing the interface on the web, while retaining the key business logic in SAS/AF. We will discuss the role that SAS/AF can still play in this new thin-centric world and how to surface this information across the enterprise.

Introduction

The Model-Viewer-Controller (MVC) paradigm originated in the late 1970's at Xerox PARC to support the Smalltalk-80 object-oriented programming interface (Simon 1995). It is used as a framework to help us understand and implement graphical user interfaces (GUI) and has been reused to varying degrees in other programming languages (Gamma, Erich et al. 1995). One of the most common applications of the MVC approach recently has been in Java. We typically think of Java as a platform-independent programming language for the Internet. However, in this paper, we will explore the use of a SAS/AF Model (i.e., class) to extend it's reach beyond the SAS-centric world into the Java world by recycling the models that we created in SAS/AF and surfacing them to a new Java Viewer.

First, however, we will discuss the main tenets of the MVC approach and give a real-world example of how it might be used. We also will discuss why MVC is important to us as SAS developers. We will then build on this knowledge by discussing how we an implement an AF Model that can used by a SAS/AF FRAME application, a SAS/IntrNet application and finally a Java version of our application.

Definitions

Client/ Server

In a traditional client/ server application, we typically define what the user sees and interact with as the client. The server, on the other hand, is the component in the application that either houses the data or does the majority of the processing. Client/ server applications can be two-tiered, three-tiered or *n*-tiered. This means that the application is divided logically into areas of responsibility. For example, in a classical mainframe application, the end-user computer was either a 3270 terminal or a PC with 3270 emulation (or similar). Here, the client (i.e. the local PC) handles the presentation while the remote system (or server) handles the application logic and data management.

Interestingly enough, web applications fall remarkably close to that model. Although with JavaScript (client-side programming structures) and Java (client and server based applications), the distinction become hazy. Given these types of applications, we refer to this as distributed logic or split logic. That is, the application's logic and business rules are defined both on the client and the server.

Fat vs. Thin Clients

When we refer to a ***thin*** client, we mean a software application that doesn't have a large footprint on the local PC (i.e., memory or required disk space). Heavy or fat clients, on the other hand, are applications that require substantial disk space (i.e., > 20 MB). Examples of thin clients include static web pages and server-side Java applications. An example of a heavy client would be a local installation of the SAS System. Even file-server-based installations of SAS would be considered a heavy client given its memory requirements (even though the local disk space requirements would be minimal.)

Overview of MVC

Model-View-Controller (MVC) or, as it is often referred to -- Model-View, is a way of developing applications whose primary purpose is to provide a clear separation between the presentation of the GUI and the application logic and data. This is the essence of the MVC paradigm. This differs from the separation of the physical architecture that we discussed earlier regarding client/ server architectures. There we were concerned about which physical environment was responsible for certain aspects of the application.

As we get closer to realizing this separation in our applications, we get closer to the pure goal in Object-Oriented Programming: ***reusability***. By building applications that are independent of the data, we can attach our model to new data without rewriting the code. By assigning responsibility to the Model, the Viewer and the Controller we take the tasks of modeling the external world, the visual feedback and user input respectively. In most discussions of MVC, the Viewer and the Controller are tightly coupled and often the latter gets dropped in favor of the Model-View perspective instead. Regardless, we see the clear separation of the areas of responsibilities for the application framework. Let’s show an example.

Example:

Let’s say we build an application that calculates the number of vacation days for the employees of a large company. We want to provide this information to the employees via the web so that the Human Resources personnel do not have to respond to phone calls or e-mails continually to provide this information. We also provide this and other information to the Human Resources staff via a SAS/AF application. The Payroll Department utilizes this information to verify vacation days earned through a biweekly batch SAS job that delivers static reports.

Problem

The Vice President of Human Resources has changed the effective rate at which people earn vacation. In 1998 the business rules were as follows:

- If an employee was hired after January 1, 1997, they earn .833 vacation days for every full month they worked. Once they have reached their 3-year anniversary, they earn vacation at 1.25 days for every full month they work.
- Prior to January 1, 1997, individuals were accrued vacation at different rates. This is provided in a lookup table.

Under the new plan (starting January 1, 1999), all newly hired employees (hired after October 1, 1998) will accrue 1 day of vacation for every month they work. Between October and January, all other employees with 1-3 years of service accrue at .833 days as before. After, January 1, 1999, all employees with 1-3 years of service accrue vacation days at a rate of 1 day per month. All other employees with three years of service or more after January 1, 1999 accrue their vacation at a rate of 1.5 days per month.

We now have to implement the system changes across three separate systems, test and rollout these new changes by month-end.

Solution

Since our application was developed in SAS/AF for the Human Resources department, Base SAS for the Payroll Department and SAS/IntrNet for the Employees to view on the Intranet, the application can easily be redesigned to take advantage of object-oriented programming techniques. The redesigned approach would then allow for all applications within the corporate framework to take advantage of the business rules contained in the application’s Model. New applications of the business rules can simply attach themselves to the Model components of the software application and implement new viewers. For example, if the CFO wanted to analyze the impact of a change in policy before its implementation she could tap into the existing business rules captured here for her research.

In Figure 1, we show a diagram of what we mean. We have three separate viewers and one model. The Model, represented here by the 3-D box, has an interface to the RDBMS system that stores the information that is separate from how we will view this information. Each of the viewers has a very specific, yet generalizable way of getting information from our object. In this case we can send messages to the Model and expect a consistent interface into our object regardless of the Viewer that is attached to the model.

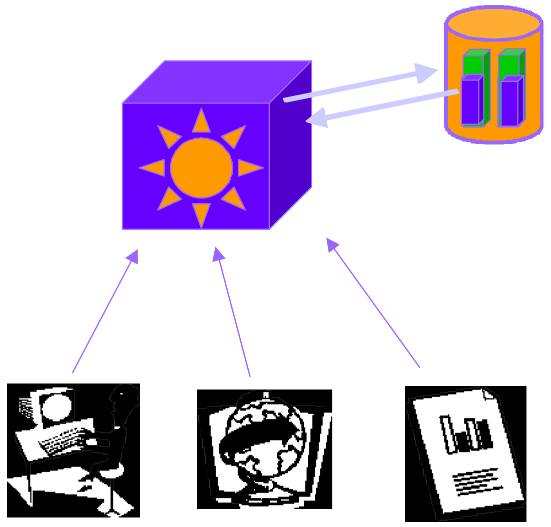


Figure 1. Multiple viewers attached to the object model.

This is contrasted with a system that for embeds the business rules directly in the Base SAS code for the static reports (DATA STEP and PROC REPORT); in the htmSQL code for the SAS/IntrNet application; and in SCL for the SAS/AF FRAME implementation. The Model and the Viewer are so closely linked, that it's hard not to impact one without affecting the other.

MVC Concepts

Here we have introduced a very real example of how this MVC paradigm could be applied to a SAS application. Let's take a step back and help define some of the concepts we've just explored.

As we said before, the essential goal of the MVC design pattern[1] is to provide the separation between the presentation of an application and the application logic and database access. That is, the business rules that define how our data is prepared and brought to the Viewer. The Viewer is concerned with the presentation of that information.

In our example, the **MODEL** is responsible for opening the data source in an appropriate mode (EDIT/BROWSE) and locking (RECORD/MEMBER). In addition, we want our data to read / write row(s) of data from / to the data source as needed. Further, our Model contains the business rules for our vacation object.

The **VIEWER** is responsible for displaying the data in some visual way (i.e., graphs and charts, tables or text output). It is also responsible for the visual properties of the display (colors and fonts) as well as forwarding user requests (scrolling, editing) to the Controller (or Model).

Finally, the **CONTROLLER** is responsible for "attaching" a Viewer object to a model object so they can communicate. It is also responsible for translating requests from the Viewer to appropriate requests on the Model as well as relaying data to the Viewer. Finally the Controller handles the "disconnection" of the Model from the Viewer.

[1] A design pattern is a methodology that is used to design object oriented application. A design pattern is to design what a class library is to coding. This includes documentation that addresses specific recurring problems and includes very general design prinicples all the way to laguage-specific constructs. Refer to (Kurotsuchi 1996) for a excellent overview of design patterns.

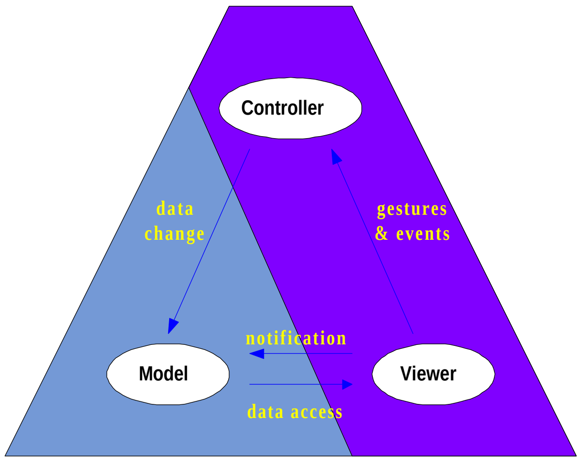


Figure 2. Separation of responsibilities for the Model and Viewer

Figure 2, adapted from (Potel 1996), shows us what the separation of responsibilities include. The left hand side of this illustration denotes the difference between Model-View-Controller and Model-View.

Purpose of MVC

If Models and Viewers remain independent and loosely coupled objects, they facilitate a myriad of benefits. Over time, we could replace the backend data structure from a SAS dataset to an Oracle database, for example. We could add other fields into the database, index the tables, and so forth, and still use the same presentation code without modification. We could develop many different kinds of Viewers without having to reinvent the proverbial wheel. We could, for example, add other information to our data store (RDBMS) which included anniversary date, date of birth, annual salary, etc. and enhance our Viewers to provide the added functionality without having to re-deploy each application that uses the data. In addition, we can have multiple developers working on the different models and different viewers in parallel and have these come together and work in different configurations.

By providing a consistent interface to the data and it's preparation (calculation of business rules), we achieve *encapsulation* in our programming. Encapsulation refers to the notion of information hiding. That is, we provide a method or an interface for applications to communicate with our object, while hiding how we do it and other unnecessary information about what's inside the object model.

Figure 3 gives a graphical representation of what we mean by encapsulation. The information (i.e., variables, rules) itself is hidden from the outside world, but is available through a published interface (i.e., methods).

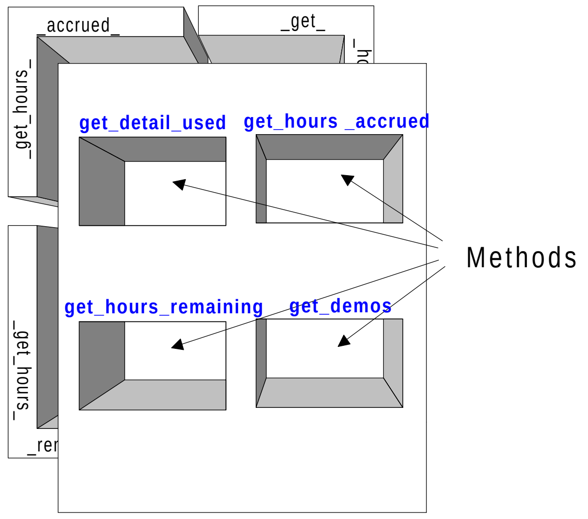


Figure 3. Vacation object Model provides an interface to the business logic which is hidden

Examples of MVC

In this example we will explore the concepts of MVC by building an application. Here will we build an application using a traditional SAS/AF approach. Next we will write this as an object-oriented application in SAS/AF. Then we will test our application by stripping out the Viewer and replacing it first with an HTML version using SAS/IntrNet and then a Java viewer. Our goal, remember, is to write an application whose logic and business rules can be front-ended with multiple viewers. In our case, we acknowledge that this test is a simple example, but the concepts should aid you in your own application development.

Our application is one that we described above. We want to write a module that might go into a larger Human Resources application that allows people to check how much vacation they have accrued and how much is available (based on how much they have used.) Functionally, we want to be able to enter our employee ID and request the following information:

- The number of hours we have accrued to date
- The number of vacation hours we have used
- Calculate vacation days remaining
- View the dates and the number of hours where we have already used up our vacation

From a design perspective, we approach this problem by trying to understand not only how to calculate these elements, but also what information the Viewers require as input: that is, what the interface has to look like to the Viewer. We know for example that we can display numeric information in a variety of widgets in SAS/AF FRAME (for example, Input Field, Extended Text Entry and the Extended Input Field). If we use SAS/IntrNet, we can use some of the common HTML tags to dynamically produce this information (for example, <INPUT> Tag). In Java, there are similar components that are available to us that allow us to show this kind of information (for example, TextField). This helps us understand what our Model will have to return back to the Viewer at run-time. The three numeric fields are relatively easy to get back from the Model.

The last piece of information we want to be able to get from our Model is the specific dates that were taken as well as how many hours were taken each day. We also may be interested in some additional information such as when the vacation was requested and who authorized or approved the time off. This information must be returned as a list or table of items.

In order to get a list of items back to the Viewer to display, we have to be able to display some kind of table viewer or list viewer. In SAS/AF and SAS/IntrNet, both have objects to achieve this. In Java, we will have to be able to send a potentially large list back to the Viewer. Here we are going to use webAF, which is an Integrated Java Development Environment from SAS Institute that has built-in components that can interact with SAS SCL lists (for example, ListBox through the com.sas.collection.hlist interface.)

Combined, we now understand the kinds of information we need to return, next let's focus on how we calculate them (i.e., what the business rules are.)

Building the Model

In Figure 3, presented earlier, we saw how we communicate with these remote objects and how they were "protected" by forcing us to use method calls to interact with our Model through its published interface. It is to these methods that we now turn.

Methods are analogous to *modules* in structured programming. They have very specific functions they perform and we communicate to our *objects* through these interfaces to act upon the variables contained within the object. Some methods can alter an object's current state while others can compute values that are returned to the caller (or sender of the message.) Our methods will produce some sort of returned value. The advantage of using this sort of design is that we can create a class in SAS/AF using SCL and utilize that *class* in interactive SAS applications, dynamic web applications or even in Java. In Java, we specifically exploit these classes by using SAS' Remote Object Class Factory (ROCF) to provide the interface between Java and SAS.

AF Model

Our Model can be characterized as a simple module that contains the business rules about our subject. It is relatively self-contained and contains information only about the subject of “Vacation”. Our Model, for example, contains the links to the datasets that are used to house the employee data, the individual vacation records as well as the code that calculates the accrual and the vacation hours remaining. We then provide method, which can be used to interact with our Model.

Figure 4 shows a graphical representation of this model, which now exposes the variables and methods to the programmer.

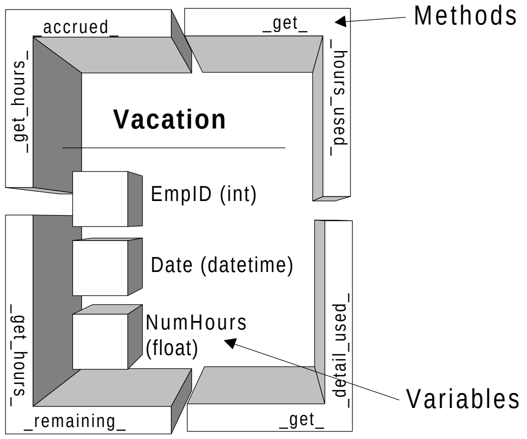


Figure 4. Model exposed by method calls

Building the Viewers

Once developed, the Model is ready for exploitation. We can now attach any of several Viewers that can take advantage of the information made available through the Model. Remember that the role of the Viewer is to handle what the user sees. Graphical User Interface objects such as list boxes and controlling events such as mouse presses and keyboard presses are interpreted by the Viewer (i.e., remember we condensed our definition of Model-Viewer-Controller into Model-Viewer). In theory we are ready to put on the lipstick and rouge as the bulk of the work (i.e., Model) has been defined and is doing most of the work.

SAS/AF FRAME

Because of the disk space required, memory it takes to load SAS and the CPU processing, a SAS/AF FRAME application is considered a Heavy, Fat or Thick client. Nevertheless, FRAME can be used very effectively as a client (Viewer) to our Model and, depending on the type of application, very appropriate for some interactive applications. Our vacation program can be accomplished simply by providing a screen that accepts user input for their employee ID and upon a mouse click on a button or pressing the ENTER key, a series of objects can be populated with the results of our Model. Our interface defines the parameters that we pass to the Model, as well as the type of data that is returned.

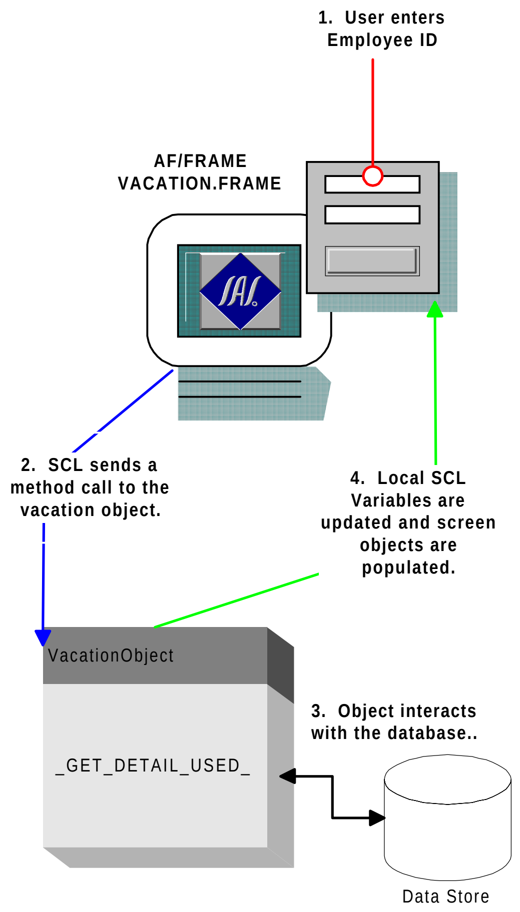

1. User enters Employee ID
AF/FRAME VACATION.FRAME
2. SCL sends a method call to the vacation object.
4. Local SCL Variables are updated and screen objects are populated.
VacationObject
_GET_DETAIL_USED_
3. Object interacts with the database..
Data Store

Figure 5. FRAME Model-Viewer

Figure 5 shows the flow of events:

1. The user enters their employee ID and presses enter
2. The Viewer/ Controller passes the information to the Model via its published interface (i.e., method calls)
3. The Model calculates the request (i.e., vacation hours accrued, taken and remaining)
4. The Model passes the information back to the Viewer for display

The VACATION class (Model) handles the business logic, while a SAS/AF FRAME application provides the GUI (Viewer).

Since FRAME objects are tightly integrated with the AF Model, when we return values (such as a SCL List or a

SCL variable), these can be automatically mapped to the screen elements like List Boxes and Input Fields. Not all AF objects have this automatic mapping.

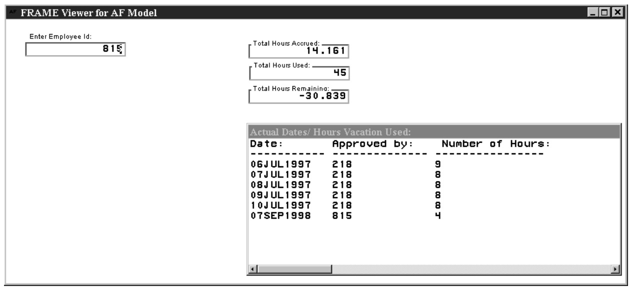
SAS/AF FRAME Interface

Since the Model and Viewer are so tightly integrated, the SCL code for the FRAME itself is fairly simple and controls only the GUI components and the calls to the instance of the VACATION object itself.

```
INIT:
     /* Control the screen stuff */
      cursor empid;
      call notify('actlist','_hide_');
      call notify('tot_rem','_hide_');
      call notify('tot_acc','_hide_');
      call notify('tot_sum','_hide_');
     /* Create an instance of the object */
      vacid=instance(loadclass('sasuser.models.vacation.class'));
RETURN;

MAIN:
   if empid then
      do;  /* Check to make sure empid has a value */

         empinfo=makelist();

/* Call the methods we need. Since Hours remaining has to   */
/* call the other methods, we can use just this one call    */
/* for the calculations. The second call is to return the   */
/* SCL List that contains the detail hours.                 */

         call send(vacid,'GET_HOURS_REMAINING',
                          empid, tot_rem, tot_acc, tot_sum);
         call send(vacid,'GET_DETAIL_USED',empid, empinfo);

      /* Control the screen stuff by unhiding objects */
         call notify('tot_rem','_unhide_');
         call notify('tot_acc','_unhide_');
         call notify('tot_sum','_unhide_');
         call notify('actlist','_unhide_');
         call notify('actlist','_repopulate_');
      end; /* Check to make sure empid has a value */

      else do;  /* Empid does not have a value */
         cursor empid;
         link init;
      end;      /* Empid does not have a value */
RETURN;
```

MVC.MVCDEMO.VAC_CALL.SCL

As you can see, the majority of this code is handling the screen controls and none of the business logic. We only send messages back and forth from the interface to the object through the CALL SEND statements. Figure 6 shows the results of the viewer being populated from the AF Model.



Enter Employee Id: 815

Total Hours Accrued: 14.161

Total Hours Used: 45

Total Hours Remaining: -30.839

Actual Dates/ Hours Vacation Used:

```
Date:        Approved by:   Number of Hours:
-----------  -------------  ----------------
06JUL1997    218            9
07JUL1997    218            8
08JUL1997    218            8
09JUL1997    218            8
10JUL1997    218            8
07SEP1998    815            4
```

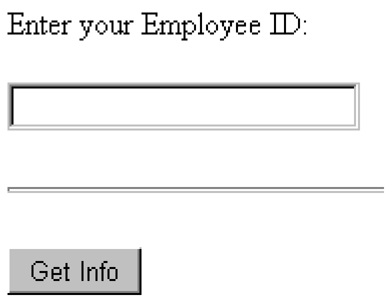
Figure 6. FRAME View of results

SAS/IntrNet

As we have seen above, our Model-View architecture has worked thus far in a SAS/AF application. Let's extend the example by providing a different perspective of the data by changing the Viewer. Since we are still using a SAS product (SAS/IntrNet), we wouldn't expect much of a challenge here. However, consider that the only components that we have to update are what the user sees. The Model remains untouched. The significance is clear: GUI implementations can be changed, enhanced, ported to different environments and the business rules continue to work as specified by the Model.

Here, we provide a similar view of the information by giving the user a chance to enter their employee ID, pressing SUBMIT, and having the results displayed back to the employee via their browser. We simply write the interface specifications differently. Although the parameters passed to the Model are the same, we expect the returned data be formatted as HTML using an SCL entry, which has been added to our VACATION class.

The HTML for the user input displays a simple text entry box with a submit button (Get Info):



The form action passes the information to the SAS/IntrNet broker and passes the following information:

```
<input type="hidden" name="_PROGRAM"
value="mvc.mvcdemo.webprog.scl">
<input type="hidden" name="_SERVICE" value="default">
```

The flow of information is similar to that presented above when we talked about the SAS/AF FRAME version of our viewer. However, it differs slightly and is illustrated below in Figure 7.

1. The user enters their employee ID and presses the Get Info button.
2. The broker converts the employee ID into a macro variable and calls the webprog.scl program.
3. The webprog.scl program makes two method calls to get the information needed.
4. The vacation object communicates as needed with the database.
5. Finally, the last method call to the vacation object creates an HTML formatted table, which is pumped back to the browser.

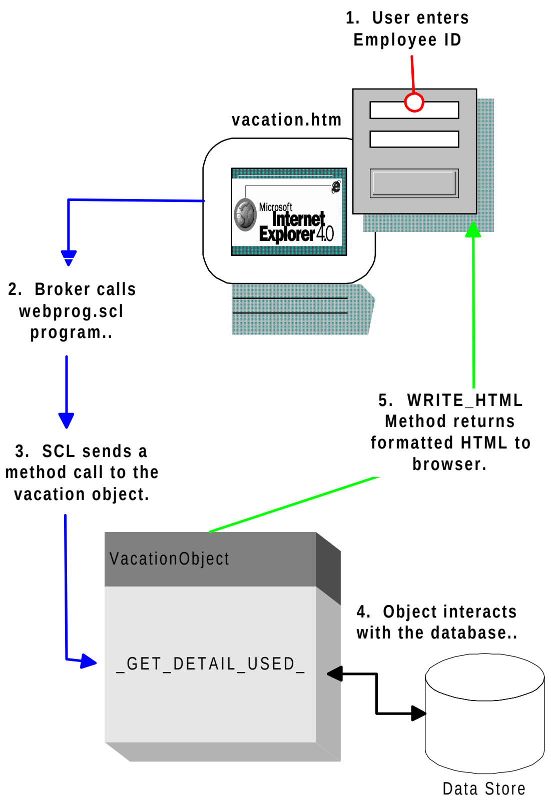

1. User enters Employee ID
vacation.htm
Microsoft Internet Explorer 4.0
2. Broker calls webprog.scl program..
3. SCL sends a method call to the vacation object.
5. WRITE_HTML Method returns formatted HTML to browser.
VacationObject
_GET_DETAIL_USED_
4. Object interacts with the database..
Data Store

Figure 7. SAS/IntrNet interaction with AF Model

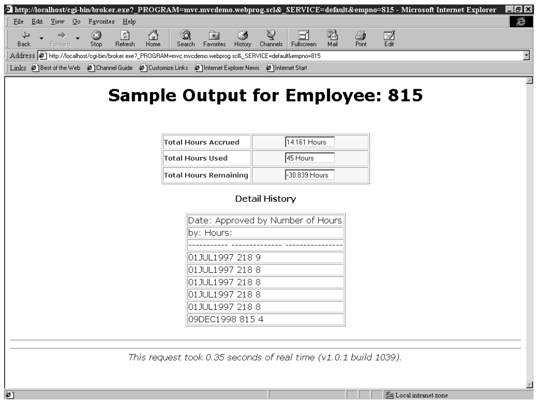
SAS/IntrNet Interface

As before, our Model requires an interface to communicate with our VACATION object. In this case, we have (a) the HTML, which drives the broker and (b) a SCL program that makes the method calls and formats the output. Think of this as providing an interface on both sides of the object.

The SCL program (webprog.scl) that we call with our HTML form is presented below:

```
INIT:
      /* Create an instance of the object */
      vacid=instance(loadclass('sasuser.models.vacation.class'));

      /* Get the employee id that SAS/IntrNet stores */
         empno=symgetn('empno');

   if empno then
      do; /* Check to make sure empid has a value */

         /* Make a list so that we can populate it later */
         empinfo=makelist();

         /* Call our methods to do the work */
         call send(vacid,'GET_HOURS_REMAINING',
                          empno, tot_rem, tot_acc, tot_sum);

         call send(vacid,'GET_DETAIL_USED',empno, empinfo);

         /* This method just writes the SCL List out into HTML */
         call send(vacid,'WRITE_HTML',
                   empinfo, tot_rem, tot_acc, tot_sum);

      end; /* Check to make sure empid has a value */

      else do;   /* Empid does not have a value */

        /* You could put some error HTML back out for the user */
         return;
      end;       /* Empid does not have a value */
RETURN;
```

The net result of the submission of the above HTML form is a formatted output as illustrated in Figure 8.


http://localhost/cgi-bin/broker.exe?_PROGRAM=mvc.mvcdemo.webprog.scl&_SERVICE=default&empno=815 - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print Edit
Address http://localhost/cgi-bin/broker.exe?_PROGRAM=mvc.mvcdemo.webprog.scl&_SERVICE=default&empno=815
Links Best of the Web Channel Guide Customize Links Internet Explorer News Internet Start
Local intranet zone

Sample Output for Employee: 815

Total Hours Accrued	14.161 Hours
Total Hours Used	45 Hours
Total Hours Remaining	-30.839 Hours

Detail History

```
Date: Approved by Number of Hours
by: Hours:
----------- ------------- ----------------
01JUL1997 218 9
01JUL1997 218 8
01JUL1997 218 8
01JUL1997 218 8
01JUL1997 218 8
09DEC1998 815 4
```

This request took 0.35 seconds of real time (v1.0.1 build 1039).

Figure 8. Dynamic output produced by SAS/IntrNet

Java

As was indicated previously, by using the SAS webAF product, we are able to take advantage of a SAS/AF Model to surface the "objects" we built in SAS to the Java programming language. The linkage is made as the client application is written using Java components which in turn make remote method calls to the AF objects. These objects, written in SCL, look just like another Java object from the perspective of Java. From a SAS perspective, they are a SAS class, which can similarly be called from a SAS/AF application through SCL method calls. By virtue of the fact that these can be remote objects, they can reside on the same machine as the Java client application, or on a remote host (using remote Java (RMI), remote SCL (JCONNECT), CORBA or DCOM. For more information on this, please refer to the SAS Institute White Paper entitled "Distributed Computing with Java and the SAS System" (Baily and Moss 1998).

The flow is, once again, slightly different than what we have seen before. Specifically, once downloaded to the browser, the Java applet can interact directly with the application server (and database server) without having to traverse the HTTP server.

1. The user requests the vacation.html page from our server.

2. The HTTP server acknowledges the request and downloads the Java applet to the browser.

3. Once the user enters their employee ID and presses return, the Java event handler (shown below as viewer.java) calls the SAS/AF SCL methods, which are defined by the interface program (modelviewInterface.java).

4. The Java applet makes a remote method call to the Vacation object.

5. The vacation object interacts directly with the database to retrieve the appropriate information.

6. The values are returned as variables in the Java applet and any screen updates are performed by the applet itself.

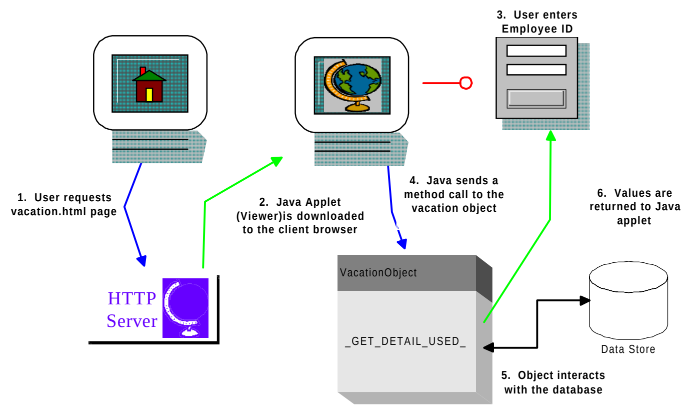

3. User enters Employee ID
1. User requests vacation.html page
2. Java Applet (Viewer)is downloaded to the client browser
4. Java sends a method call to the vacation object
6. Values are returned to Java applet
HTTP Server
VacationObject
_GET_DETAIL_USED_
Data Store
5. Object interacts with the database

Figure 9. Program Flow for Java applet.

The Java Applet

The Java applet itself contains two parts: the `viewer.java` and the `modelviewInterface.java`. The viewer provides the graphical user interface and the interface makes the linkage between the Java applet and the SAS/AF class. The GUI is shown in Figure 10 below.

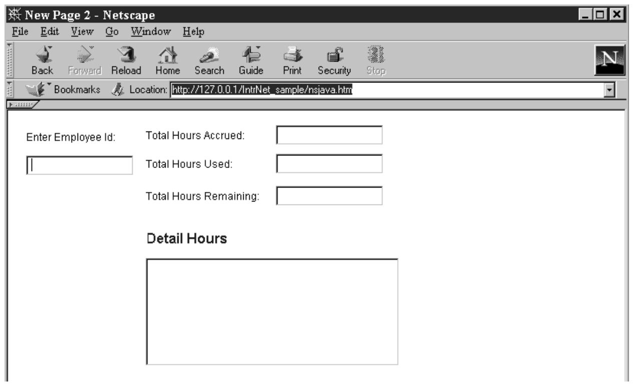

New Page 2 - Netscape
File Edit View Go Window Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://127.0.0.1/IntrNet_sample/nsjava.htm
Enter Employee Id:
Total Hours Accrued:
Total Hours Used:
Total Hours Remaining:
Detail Hours

Figure 10. The Java applet (Viewer).

Java Interface

In order to extend SAS/AF Models into Java you need to define an *interface* in Java. The interface is the set of rules that govern what is passed to the Model, what return types are expected and the names of the methods that we are to use. Once defined, we are ready to exploit the model. We define the model to Java by building the interface in the Java programming language. This is given below:

```
public interface modelviewInterface extends
com.sas.ComponentInterface
{
     String contextclasspath = "sasuser.models.vacation.class";
     public com.sas.collection.hlist.HListInterface
get_detail_used(double parm1);
     public double get_hours_accrued(double parm1, double parm2);
     public double get_hours_used(double parm1);
     public double get_hours_remaining(double parm1, double
parm2, double parm3);
     public double get_demos(double parm1);

}
```

modelviewInterface.java

Once defined, the remote proxies are built in the webAF environment (definitions about which automatically handle the client/ server issues for you), we can build the Java applet around this using method calls directly to our model within the Java environment. See the highlighted lines below in the `viewer.java` program.

```
// textFieldEmpnoActionPerformedHandler1
public void
textFieldEmpnoActionPerformedHandler1(java.awt.event.ActionEvent
event)
  {
 // IDVAL is the var that we grab out of the text box
   String item;
   double idval;
   idval=Double.valueOf(textFieldEmpno.getText()).doubleValue();

 // Here we start to calculate the variables using out AF Methods
 // Calculate hireDate
   double hireDate;
   hireDate=modelviewInterface8.get_demos(idval);

 // Calculate totalAccrued
   double totalAccrued;
   totalAccrued=modelviewInterface8.get_hours_accrued(idval,
                hireDate);
   textFieldTotacc.setText(""+totalAccrued);

 // Calculate totalUsed
   double totalUsed;
   totalUsed=modelviewInterface8.get_hours_used(idval);
   textFieldTotsum.setText(""+totalUsed);

 // Calculate totalRemaining
   double totalRemaining;
   totalRemaining=modelviewInterface8.get_hours_remaining(
               idval, totalAccrued,totalUsed);
   textFieldTotrem.setText(""+totalRemaining);

 // Get the SCL List returned and populate the box
   listBox1.removeAll();
   com.sas.collection.hlist.HListInterface textlist;
   textlist=modelviewInterface8.get_detail_used(idval);
   int listSize=textlist.count();
   for (int i=0; i < listSize; i++)
          {
            item=textlist.getString(i);
            listBox1.add(item);
          }
  }
```

viewer.java

Summary

In this application, we showed that we could take a SAS/AF application and, if built properly, separate the business logic and the presentation components of the application to create a Model, which can be attached to a multitude of Viewers. By creating this an application in this way, we have adhered to the Model-Viewer paradigm. By doing so, we have achieved the following object-oriented principles:

- Encapsulation
- Sharing

- Persistence
- Reusability

In his book David Taylor (Taylor 1990) talked about the potential benefits of the object-oriented approach. He talked about faster development, higher quality output, easier maintenance, reduced costs, increased scalability, better information structures, and increased adaptability. Some of these benefits are difficult to measure, with that said, in our small example, we have shown the following benefits:

View independence: multiple presentations can be built (HR, Payroll, Managers, and the CFO) as different views on the same underlying vacation calculation engine and data representation.

Model independence: Allows developers to change and evolve data structures or file formats without changing how the data is displayed or processed in the rest of the program, as well as introduce persistence, remote data bases, and sharing.

Reusable logic: Once implemented, different presenters can reuse these in multiple applications (e.g., vacation accrual is used in 3 different apps).

Input generality: Without changing the application logic or rendering of my application, we can support different menus, dialogs, and keyboard equivalents, gestures or handwriting/ pen input.

Finally, the overall benefit of using this framework to implement the MVC framework is that they facilitate portability across platforms, multiple standards (e.g., RMI, JCONNECT, CORBA and DCOM) distribution and multi-tier partitioning.

Perhaps most important is that from a personnel perspective, we can reposition our current AF programming staff to take advantage of this new technology. We allow the development of new viewers on top of our existing investment in SAS/AF applications. SAS/AF is not dead by any means. The real challenge is how and when to get started.

Appendix: Application Components

To summarize the component architecture for our example, we have included a visual representation of the architecture along with some reference points to specific code segments that were used in this article. This should aid the reader in building this application on your own.

SAS/AF FRAME Viewer

Component	Description
VAC_CALL. FRAME	SAS/AF FRAME entry which contains the viewer described in “Building the Viewers” presented earlier.
VAC_CALL. SCL	SCL entry for the VAC_CALL.FRAME

SAS/IntrNet Viewer

Component	Description
vacation.html	HTML file that collects the employee ID and passes the information onto SAS/IntrNet via the broker.
webprog.scl	The SCL program that is called from the broker (via vacation.html).

Java Viewer

Component	Description
modelviewInterface.java	The java source code built by the Jazz IDE which establishes the components required to talk to the SAS Model (in SAS/AF SCL).
viewer.java	The java source code that controls the way the information is presented to the user. This is the source code for the Viewer described in “Building the Viewers” presented earlier.
viewer.frame	In the Jazz IDE, this is the visual components where the user enters their information. This frame is controlled by viewer.java.
iejava.html	The HTML file that allows us to view the java applet. This contains JavaScript code used by Internet Explorer (4.0).

iejava.html

```
<html>
<head>   <title>New Page 2</title>   </head>
<body>
<p>
<object CLASSID="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
HEIGHT="400" WIDTH="600"
CODEBASE="http://java.sun.com/products/plugin/1.1.1/jinstall-111-
win32.cab#Version=1,1,1,0">
  <param name="CODE" value="sesug99.class">
  <param name="CODEBASE" value=".">
  <param name="ARCHIVE" value="sesug99.jar">
</object>
</p>
</body>
</html>
```

nsjava.html	The HTML file that allows us to view the java applet. This contains JavaScript code used by Netscape Navigator (4.0).

nsjava.html

```
<html>
<head>
<title>New Page 2</title> </head>
<body>
<p>
<embed
PLUGINSPAGE="http://java.sun.com/products/plugin/1.1.1/plugin-
install.html"
TYPE="application/x-java-applet;version=1.1"
java_CODE="sesug99.class" java_CODEBASE="."
java_ARCHIVE="sesug99.jar" HEIGHT="400" WIDTH="600"><NOEMBED>
</NOEMBED>
</EMBED>
</p>
</body>
</html>
```

SAS/AF SCL Model

Component	Description
vacation.class	The SAS/AF Class that controls the business logic defined as the Model. This is described in detail in “Building the Model”.
vacation.scl	This SCL entry contains all the source code for the methods used by the Vacation Class.

The Vacation Class contains six methods described earlier:

- GET_DEMOS
- GET_DETAIL_USED
- GET_HOURS_ACCRUED
- GET_HOURS_REMAINING
- WRITE_HTML

Each of these methods performs a very specific function and are available from the author along with the datasets used for these examples. The datasets that are needed include:

- GFATHER
- TRANS
- EMPLOYEE

Bibliography

Baily, C. and K. Moss (1998). Distributed Computing with Java and the SAS System, SAS Institute. 1998.

Gamma, Erich, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley.

Kurotsuchi, B. T. (1996). Welcome to the wonderful world of DESIGN PATTERNS. 1998.

Potel, M. (1996). MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java, Taligent, Inc. 1998.

Simon, L. (1995). The Art and Science of Smalltalk: An Introduction to Object-Oriented Programming Using VisualWorks. London, UK, Prentice-Hall.

Taylor, D. A. P. D. (1990). Object-Oriented Technology: A Manager's Guide. Reading, MA, Addison-Wesley.

Acknowledgements

SAS is a registered trademark or trademark 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.

Address

Questions pertaining to this article should be addressed to:

Gregory S. Barnes Nelson
STATPROBE, inc.
Internet: greg.barnesnelson@statprobe.com
Web: http://www.statprobe.com