Paper 35-28

Web Enable Your SAS® Applications

Teresia Arthur, SAS®, Cary, NC
Mary Jafri, SAS®, Cary, NC

ABSTRACT

How many times do we write applications only to rewrite them later because a new operating system comes online, customers want a friendlier interface or customers just want "the latest cool stuff"? Wouldn't it be wonderful if we could write core application code that remained in use throughout the evolution of an application and if that application remained in service for many years? Who said applications have to become antiquated so quickly and who has time and resources for application rewrites?

The objective of this paper is to demonstrate how a production SAS/AF® software application, originally written eleven years ago on MVS, has become Web-enabled and serves 1300+ global customers, without rewriting the core application. We will demonstrate how this application evolved using SAS Component Language (formerly called Screen Control Language or SCL) classes, and how they are now utilized through SAS/IntrNet® software application dispatchers, email gateways, and Application Programming Interfaces (API).

We'll demonstrate how SAS Application Dispatchers were used along with SAS/SHARE® software, SAS/SHARE*NET™ software and Base SAS software, which supports Structured Query Language (SQL). By using the SAS/SHARE*NET driver for JDBC we are also able to access and update our SAS data directly through Java programs.

This paper is intended for intermediate and advanced application developers. We will demonstrate the versatility of SAS software, operating system independence, and how best to design your applications to take advantage of these benefits.

INTRODUCTION

SAS/AF software applications can evolve on different operating systems and become Web-enabled without an expensive rewrite and with very limited programming resources. We will describe our application experience, explain the evolutionary steps we took, and demonstrate how we implemented SAS/IntrNet technologies. We will also share code samples and some "lessons learned" in the process.

WHO WE ARE

We work in the SAS Management Information Systems department at SAS, which addresses internal corporate business application needs using SAS software. We are tasked with developing and evolving applications to continually provide for internal customer needs at minimal expense. We have created and implemented some applications that remain useful and current 10+ years after their creation and even longer. Just like you, we are constantly challenged to work smarter to meet our ever-increasing customer demands.

BACKGROUND

First, a very brief background about our application should be helpful for your understanding of the upcoming examples. Strategic Online Services (SOS) is an internal problem and request tracking system that is utilized worldwide by SAS employees to organize and manage their daily workload. For clarity in this paper, we will focus on SOS "problem" examples rather than problems and requests although both have similar components and share the same parent class.

Our internal help desk employees use SOS to manage problems that are automatically opened and routed when any SAS employee sends email to the help desk. Different teams have Web forms that query their customers for information. They use our SOS APIs to open and route problems from these forms.

Along with current data activity, SOS contains 11 years of archive data, and an extensive answers database along with customized user profiles. Our customers can use multiple interfaces to interact with SOS, such as:

- SAS/AF interfaces
- Web interfaces
- Email gateways
- Web APIs
- Command line APIs.

The features and business rules in this application are too numerous to list here, but it should already be clear to the reader why we are very anxious NOT to invest in a rewrite!

COMPONENT ARCHITECTURE

Component architecture has allowed us to easily maintain our system as business rules have evolved. It also has allowed for an easy transition to multiple interfaces and different operating systems. This allows for portability, extensibility, and maintainability.

BUSINESS RULE ENCAPSULATION

The key to easy evolution and to Web-enabling an application is a solid component architecture that utilizes business rule encapsulation. At the very beginning, SCL classes were written using object oriented constructs that completely separated the model from the viewer. The user interface translates commands and relays data between SOS customers and objects. As customers request new functionality, the methods within the core classes of SOS continue to evolve. All interfaces take advantage of the same core SCL classes, so as business rules evolve, we only have one location to update in order to serve all of the interfaces. It is essential not to include business logic in any visual parts of the application. See the lessons learned section later in this paper for more details.

MULTIPLE INTERFACES

The interfaces for SOS continue to grow as customer needs grow. The interfaces started as SAS/AF frames then expanded to include multiple Web interfaces, email gateways, and command line APIs. These additional interfaces provide great versatility and were made possible because of the extended reuse of the core SOS classes.

PORTABILITY

The host-specific portions of code are isolated in separate parent classes, which are inherited by host-independent classes. Examples are host-specific commands for sending email or printing. These classes reside in a separate host-specific tool catalog and are utilized by many applications. When migrating an application from one operating system to another, we modified only the host specific parent classes. The SOS application classes themselves did not have to change. This amounted to just a few method updates, which is a very minimal effort.

As a result, there was no mystery about where to make updates in the SOS application for host specific functions. All applications reaped the benefits of keeping the same API, regardless of the operating system. This concept is key to maintaining portability.

THE EVOLUTION OVERVIEW

The SOS application originated on an MVS mainframe. It was then ported to run on several versions of Unix and now runs on the SAS intranet allowing worldwide access using SAS application dispatchers, an Apache Web server, and a Tomcat Java application server (JVM).

All the SOS evolution from MVS to Unix to the SAS intranet has been accomplished via SCL, SAS application dispatchers, and JSP, and all without rewriting the core application. The SOS customer base continues to grow and customer satisfaction is high. The data is stored in SAS data sets and is made available through SAS/SHARE servers which provide multiple user capability and JDBC access from the Web.

EVOLUTION #1 & #2: OPERATING SYSTEMS AND EMAIL GATEWAYS

UNIX

First, we ported the application from SAS/AF version 6.06 on MVS to SAS/AF version 6.09 on Unix. We used cross-domain access to SAS/SHARE servers to easily copy data to its new home on a Unix file system. PROC CPORT and PROC CIMPORT were used to port the SAS/AF catalogs. Figure 1 and figure 2 show example SOS user interface AF frames on Unix.

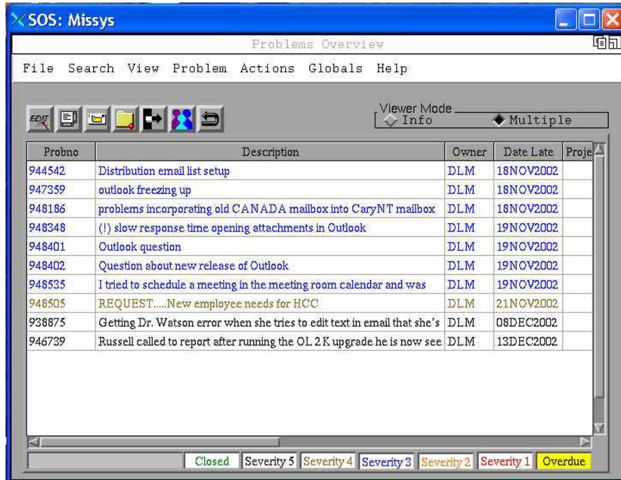

Figure 1. SOS Problem Overview (SAS/AF on Unix)

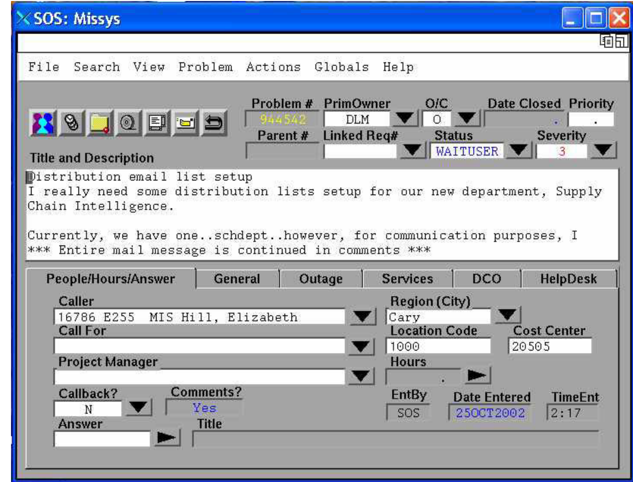

Figure 2. SOS Problem Detail (SAS/AF on Unix)

EMAIL GATEWAYS

We reused our existing SCL object for adding comments (unlimited number of lines of text) to each problem by creating an interface that adds the text of an email as a comment to a problem. We created several email gateways for automated approvals, problem opens, etc. For this example, we demonstrate only the adding comments gateway.

We used elm filter rules on Unix to trigger a small ksh script, which invokes SAS and runs our SCL program in batch. This program reads the email text from STDIN then sends it to the comment class. Using the core SCL objects made it simpler on us and ensured that business rules were enforced.

SAS VERSION 6.12

When SAS Version 6.12 became available it offered some great new UI features such as the data table and tab objects, so with our upgrade from SAS version 6.09 to SAS version 6.12 on Unix we were able to provide a faster and easier-to-use interface with little or no changes to the underlying classes that drive the system.

EVOLUTION #3: WEB ENABLEMENT

EASY INTRODUCTION TO THE WEB

In the early days of our Web development, the htmSQL component of the SAS/IntrNet product enabled us to provide a view of the detailed data information for each problem. Since using htmSQL requires little more than basic SQL and HTML knowledge, the learning curve for using this technology is very short. This new feature opened the door to also evolve the information sent in the emails that are generated by the system. The application now only sends an email with the latest information, along with a link to the Web view for more details. This replaces the sending of all of the past details in email, as was the case before the Web view was a possibility. Figure 3 and figure 4 show SOS user interface pages for Web SOS.

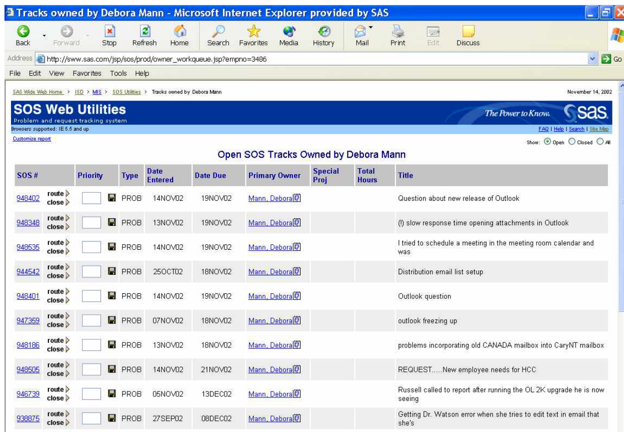

Figure 3. SOS Overview (JSP)

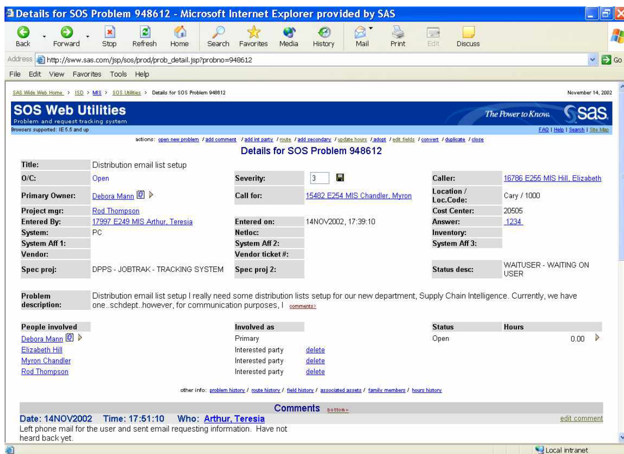

Figure 4. SOS Problem Detail (JSP)

EMAIL GATEWAYS FROM THE WEB

After providing read-only views of the system data, the next step was to provide update ability from the Web. It was easy to take advantage of our existing email gateways from the Web. For example, we created a CGI script to process the text from an HTML text area and send it to the email gateway. This allows users to add comments to problems from the Web as well. These email gateways provided an easy introduction to Web functionality because it only required simple Web forms that send appropriately built emails. Figure 5 and figure 6 show the Web SOS add comments page and the result page.

Figure 5. Comments edit page which also provides some automatic email options.

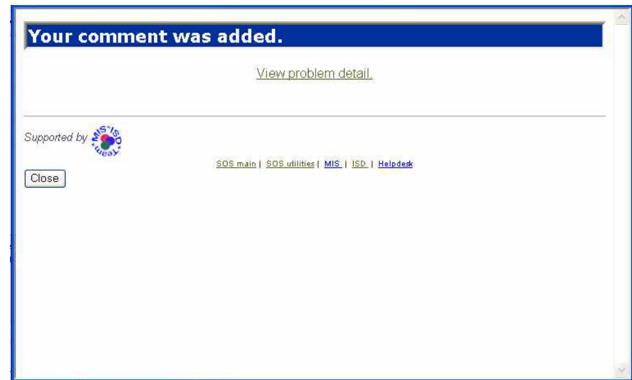

Figure 6. Message that the customer sees in their browser after adding the comment.

APPLICATION DISPATCHERS

The email gateways are a fine solution for adding comments, but are not suitable for everything. We needed to provide a robust Web application interface that utilized other application update objects. We did not have the time or resources to rewrite the system and the SOS classes already contained years of complex business rules. The SAS Application Dispatcher technology turned out to be the answer to our needs.

By definition, the SAS Application Dispatcher is:
"A SAS/IntrNet component, which is a Web gateway from your Web browser to the power of SAS processing. This gateway, written by using the Common Gateway Interface (CGI), provides access to data in combination with a powerful array of analysis and presentation procedures."

Essentially, an "Application Server" is a SAS session that is listening on a port. Through this port, a CGI script ("Application Broker") is configured to send the submitted HTML form information to that SAS session. Along with the SAS/IntrNet product, SAS provides this configurable CGI script, which you can run on your Web server. The "Application Dispatcher" solution is comprised of the Application Server and Application Broker, along with other utilities that may optionally be used.

You can process any batch SAS program in an Application Dispatcher session. The reserved filename, _WEBOUT, can be used to send output and messages back to the Web browser. The SAS ODS (Output Delivery System) component integrates nicely with the Application Dispatcher component. There are many other powerful ways to take advantage of the Application Dispatcher technology, but for this paper we are only focusing on SCL processing via Application Dispatchers.

Today our application Web interface is comprised of a series of Java Server Pages which use:

- JavaScript to help create a dynamic user interface
- The SAS/SHARE*NET driver for using SQL to query our application data served by a SAS/SHARE server
- Java scriptlets and classes to process the query results and to store information at the application and session levels
- SAS Application Dispatcher technology to update our application data with the information submitted from HTML forms.

Notice that this combination of technologies allows us to have all SAS and Java processing take place on our servers. No client install is involved. Only a Web browser is needed! SOS is used globally, and accesses servers located only in the U.S. The performance is sufficient for those countries that have good network connections.

Our Application Dispatcher solution includes an Application

Broker and multiple Application Servers that are each listening on a permanently defined port using the "Socket Service" option. We then have multiple "Load Managers" that keep track of which Application Servers are available. When Web requests come through, the Load Managers send each request to an available Application Server for processing.

The use of a Load Manager is optional. You may use the "Pool Service" or "Launch Service" instead of a Socket Service, depending on which one serves your needs best. You may also use a combination of these services. SOS is used continuously throughout all hours of the day, so we have found the optimal solution for SOS to be multiple Socket Services. The Launch Service or Pool Service might be a better choice for occasionally running reports or for systems that will not be accessing application servers continuously throughout the day.

Both Launch and Pool Services only invoke SAS sessions when needed. This way they do not unnecessarily tie up server resources with idle sessions. You might even find that you need a combination of these services to best serve your system. For example, if you occasionally have reports that take more than a few seconds to run, then you might not want them interfering with your more transactional system updates. In this case, you might want to use the Launch Service for those reports rather than tie up your dispatchers configured with a Socket Service.

For more details about these options, please see the product documentation at: http://www.sas.com/rnd/web/intrnet/dispatch/servtype.html . For this discussion we will focus only on the Socket Service solution.

EXAMPLE LOAD MANAGER AND SOCKET SERVICE IMPLEMENTATION

Sample Application Dispatcher call from an HTML form:

```
<form action="/bin/sosbroker/"
name="commform" method="post">
<input type="hidden" name="_service"
    value="sos">
<input type="hidden" name="_program"
    value="soscat.master.comment.scl">
<input type="hidden" name="_debug"
    value="0">
<input type="hidden" name="NUMBER"
    value="12345">
<textarea name="COMMENTS" cols=80 rows=12
    wrap="HARD">
This is comment line 1.
This is comment line 2.
This is comment line 3.
</textarea>
<input type=submit value=" Submit ">
</form>
```

Sample Application Dispatcher call from a Unix command line:

```
EXPORT REMOTE_USER=userid
/bin/sosbroker
'_service=sos&_program=soscat.master.comment.scl
&_debug=0&NUMBER=12345&COMMENTS=This is comment
line 1.&COMMENTS0=3&COMMENTS1=This is comment
line1.& COMMENTS2=This is comment line2.&
COMMENTS3=This is comment line3.'
```

The Load Manager in our example is named **sos**. This is defined as a Socket Service in the Broker configuration file. This file resides in the CGI bin for our Web server. The Broker executable, /bin/sosbroker, is called from a Web form by setting **/bin/sosbroker** as the form action. The Load Manager is called by setting the form input field named _SERVICE to a value of **sos**. The Broker configuration file definition is used to determine which port and server to send the data to.

Sample Broker configuration definition:

```
SocketService sos "SOS Generic"
   ServiceDescription "SOS Generic"
   ServiceAdmin "SOS Team"
   ServiceAdminMail "xxxxx@sas.com"
   Server  server.sas.com
   Port 1111 2222 3333
   ServiceTimeout 90
   ServiceLoadManager server.sas.com:5555
```

In this example, the SOS Load Manager is expected to be listening on port 5555 of the server.sas.com machine. It should be monitoring the status of the Application Servers that are listening on ports 1111, 2222, and 3333 on the same machine. Each application server is named sos1, sos2, and sos3 respectively, via their definitions in the Broker configuration file.

Sample Broker configuration definition:

```
SocketService sos1 "SOS Generic 1"
   ServiceDescription "SOS Generic 1"
   ServiceAdmin "SOS Team"
   ServiceAdminMail "xxxxx@sas.com"
   Server  server.sas.com
   Port 1111
   ServiceTimeout 90

SocketService sos2 "SOS Generic 2"
   ServiceDescription "SOS Generic 2"
   ServiceAdmin "SOS Team"
   ServiceAdminMail "xxxxx@sas.com"
   Server  server.sas.com
   Port 2222
   ServiceTimeout 90

SocketService sos3 "SOS Generic 3"
   ServiceDescription "SOS Generic 3"
   ServiceAdmin "SOS Team"
   ServiceAdminMail "xxxxx@sas.com"
   Server  server.sas.com
   Port 3333
   ServiceTimeout 90
```

The **ServiceTimeout** period is the number of seconds to wait before sending a timeout message to the person submitting the form. A timeout might occur when a request processes longer than expected, or if the Application Server is having trouble due to a program error. If the Application Server requested by the form submission is not running, then a message will be returned indicating that it was not found. (The timeout message is not returned in this case.) In either case, the ServiceAdmin and ServiceAdminMail values will be displayed as the support contact.

You can specify the individual Application Server name as the _SERVICE in the Web form. This would be necessary if you needed to access only one Application Server in particular and omit the use of the Load Manager entirely. You could also use it to bypass the Load Manager if needed in special circumstances. For example, specifying **_service="sos2"** instead of **_service="sos"** would accomplish this. If you are using a Load Manager, the _SERVICE name in your application forms should be the name of the Load Manager.

Now that the Socket Services have been defined in the broker configuration file, you can start running the Load Manager and the Application Server as detailed below.

Example Load Manager start command (Unix):

```
/bin/loadmgr -port="5555"
  -log=loadmgr.5555.log &
```

Example Load Manager stop command (from a Web browser):

```
http://server/bin/broker?_service=5555
   &_program=endloadmgr
```

In a Unix SAS installation, the /bin/loadmgr executable may be retrieved from !SASROOT/utilities/bin. These are just basic

examples of starting and stopping a Load Manager. For more information on other start and stop options, executables for other platforms, and options for viewing statistics regarding your Load Manager see:
http://www.sas.com/rnd/web/intrnet/dispatch/loadcmd.html

Example Application Server start command:
SAS provides documentation for using the INETCFG utility to generate the PROC APPSRV code needed to start an Application Server. Documentation can be found at:
http://www.sas.com/rnd/web/intrnet/dispatch82/appsrv.html

Example Application Server stop command:
http:/server/bin/broker?_service=sosgen1&_program=stop

Once you have your Application Dispatcher processes configured and running, then you can start communicating with them from the Web to run your SAS batch programs. When you call an Application Server from a Web form, the name and value of each Web form element is passed in as a name/value pair in an SCL list.

Our earlier Web form example would produce a list like:

```
(_SERVICE="SOS"
_PROGRAM="SOSCAT.MASTER.COMMENT.SCL"
_DEBUG="0"
NUMBER="12345"
COMMENTS="This comment line 1."
COMMENTS0="3"
COMMENTS1="This comment line 1."
COMMENTS2="This comment line 2."
COMMENTS3="This comment line 3."
_RMTUSER="userid";
)
```

Notice that the _RMTUSER item is not one that we passed in from the form. There are many items that all start with an underscore that are automatically created by the Application Dispatcher technology. _RMTUSER will have the user ID that was used to authenticate to the Web browser as a value. Another automatic one that we often use is _HTREFER which holds the URL of the page from which the form was submitted.

Since our SCL objects are non-visual, in order to use them from the web application interface, we only had to create some new SCL entries. These entries take the information in the list passed to the program, then process and pass this information to an object in the form that the object expects.

Example SCL code that interfaces between a Web form and an object: (Our example instantiates the comment.class .)

```
entry paramLst 8;
/* paramLst = parameters passed from Web page*/

init:
outlist=makelist();
if (paramLst le 0) then do;
  rc=insertc(outlist,"No parameter list was
    passed.",-1);
  haserror=1;
 end;
if (^haserror) then do;

/*************************************/
/* read each item on the parameter   */
/* list into a  variable or another  */
/* list.                             */
/*************************************/
number=getnitemc(paramLst, "NUMBER",1,1,"");

/*************************************/
/* Special processing for HTML       */
/* textarea elements:                */
/* There is a list item that holds   */
/* the number of lines in the        */
/* textarea if there is more than    */
/* one.  Its name is the name of the */
/* textarea appended with a 0 (zero).*/
/* Each line of a the textarea is    */
/* passed in as a separate list item.*/
/* Each line's name is the name of   */
/* the textarea appended with the    */
/* line number. The first line of the*/
/* textarea simply has the name of   */
/* the textarea assigned to it.      */
/*                                   */
/* In our example the textarea name  */
/* is COMMENTS.  Our comment         */
/* update object expects the comment */
/* to be passed in a list, so        */
/* populate that list.               */
/*************************************/
comlist=makelist();
templine=getnitemc(paramLst, "COMMENTS",1,1,"");
numlines=getnitemc(paramLst,
    "COMMENTS0",1,1,'0');
if (numlines gt 1) then do;
  do i=1 to numlines;
  templine=

    getnitemc(paramLst,'COMMENTS'||left(trim(put(
    i,best.))),1,1,"");
    rc=insertc(comlist,templine,-1);
  end;
end;  /* if more than one comment line */

/*******************************************/
/* Read each line from comments textarea    */
/* If there is just one line then put it on */
/* the list.                                */
/*******************************************/
else rc=insertc(comlist,templine,1);

/*******************************************/
/* _RMTUSER passes the userid the customer  */
/* used to authenticate to the web site.    */
/*******************************************/
 rmtuser=trim(lowcase(getnitemc(paramLst,
    '_RMTUSER',1,1,"")));
/*********************************************/
/* We are finished parsing the paramLst that  */
/* was passed in on entry, now instantiate the*/
/* comment object.                            */
/*********************************************/
 cmnt=instance(loadclass(
  'soscat.master.comment.class'));

/*******************************************/
/* The object needs the comment to be in a  */
/* source file so save the comlist contents.*/
/*******************************************/
 rc=savelist('CATALOG',
   'WORK.TEMP.COMMENT.SOURCE',comlist);
 tempin=makelist();
 tempout=makelist();
 rc=setnitemc(tempin,number,'KEY');
 /*load comment object values; */
 call send(cmnt,'LOAD',tempin,tempout);
 rc=setnitemn(tempin,-1,'NUMBER');
 /*add the new comment */
 rc=setnitemc(tempin,'WORK.TEMP.COMMENT.SOURCE',
      'FROMSRC');
 rc=setnitemc(tempin,rmtuser,'WHO');
 call send(cmnt,'ADD',tempin,tempout);
 rc=dellist(tempin);
 rc=dellist(tempout);

/*********************************************/
/* Let the customer know that the comment was */
/* added.                                     */
/*********************************************/
 rc=insertc(outlist,'Your comment was
    successfully added.',-1);

 end; /* ^haserror */
```

```
return;  /* init */

main:
return;

term:
/********************************************/
/* Done with the object to add comments,    */
/* now send a message back to the browser.  */
/********************************************/
/* Write the output to the _WEBOUT file     */
/* reference. _WEBOUT is a special file     */
/* reference defined in SAS for Application */
/* Dispatcher output.                       */
/********************************************/
 fid=fopen('_webout','O');
 rc = fput(fid, 'Content-type: text/html');
 rc = fwrite(fid);
 rc=fput(fid,"<head>");
 rc=fwrite(fid);
 rc = fput(fid,'<LINK REL="stylesheet"
               HREF="style.css" '||
              'TYPE="text/css">');
  rc = fwrite(fid);
  rc=fput(fid,"</head><body>");
  rc=fwrite(fid);

/********************************************/
/* Write custom messages generated in this  */
/* scl.                                     */
/********************************************/
  max = listlen(outlist);
  do i = 1 to max;
   text = getitemc(outlist, i);
   rc = fput(fid, text||'<BR>');
   rc = fwrite(fid);
  end;

  rc  = fput(fid, '</body></html>');
  rc  = fwrite(fid);

rc=fclose(fid);
comlist=dellist(comlist);
outlist=dellist(outlist);
return;  /* term */
```

LESSONS LEARNED:

EVEN GREAT PARENTS CAN HAVE TOO MANY CHILDREN!

Alas, we are not perfect. We got so excited about the extensibility of this design that we created multiple child applications. Each child inherited SOS classes and interfaces. This was done in order to provide custom tracking systems for different functional areas. We even added methods to migrate problems and requests between the child applications when needed. We did not copy and duplicate the SOS code. All the child tracking systems simply inherited from SOS and we overrode a few methods as necessary for the specific business rules required.

While we were able to provide whole new applications in a very short time frame, we eventually learned this is not such a great thing to do. It served the needs at the time but ultimately it cost us in maintenance. We have since found a better way. We are adding functionality to the core SOS classes and incorporating the child application functionality into SOS itself. Upon further analysis, it became apparent that the needs are not all that different.

All the child tracking systems wanted to be Web enabled, which would have required an overwhelming amount of work for us. By incorporating those tracking systems back into SOS, we can move forward with one system to maintain/enhance. All tracking system customers benefit.

STAY TRUE TO YOUR CLASSES!

During overworked moments in the past, a few shortcuts were taken when adding new functionality. In just a few SAS/AF frames, some business logic was added instead of adding it appropriately to the class methods. This has cost us more than it saved us. In the long run, this code had to be removed from the frames and added to the class methods appropriately. We thought it would save time at the moment but in the end it cost us more. We highly recommend putting all business logic into your class methods without exception.

EVEN HARDWARE MIGRATION CAN BE EASIER

We found that migrating to new server hardware is also inevitable as the industry hardware tends to improve and as our global SOS customer base continues to grow. We've found putting our data and programs on a Network Appliance simplifies the task. We run our processes on newer servers but the data and programs don't need to move. Assigning alias names to your server hardware also is helpful. As you deploy on new servers, you don't have to change LIBNAME statements if you reassign the alias to the new server.

THE FUTURE

VERSION 9 SAS® OLAP SERVER

We are implementing a Version 9 SAS OLAP (Online Analytical Processing) server with data summary and statistics that will enable customers to make decisions based on data in SOS. Customers will be able to use Enterprise Guide® software for quick access to the analytic power of SAS against SOS data in a SAS OLAP server. This will provide reports, summaries, data analyses, graphics, etc. that can be shared among our users.

VERSION 9 OF SAS AND REDHAT LINUX

We are upgrading SOS and its various parts to Version 9 SAS. We are also migrating to Redhat Linux and more current server hardware.

MORE FUNCTIONALITY

New functionality continues to be requested. We will continue to grow the core SOS classes and methods in order to accommodate these requests. The additional functionality will therefore be accessible from all SOS APIs.

CONCLUSION

Today the Web is the platform of choice for application access but what will it be tomorrow? As soon as one operating system is mastered your customers want something else entirely. Being able to succeed in this environment depends on extensible and portable application architectures as well as the software used to implement it. We've demonstrated here that SAS applications can be Web-enabled and have multiple interfaces with reduced effort and no rewriting of the core application.

The architecture demonstrated here also makes it possible to take an iterative approach to interface development. There were many extended periods of time where no developers were assigned to SOS due to other higher priority projects and limited programming resources. Even during those times some customers helped themselves by using SOS APIs, ensuring their progress and the utilization of appropriate business rules. They were able to create their own customized Web forms that fed data into SOS APIs, which guaranteed the business rules were enforced.

The investment made to develop an application has much greater returns when the application classes remain in service for many years and continue to service the growing needs of its customers.

We found SAS software to be strategic in operating system portability as well as application flexibility, code reuse, ease of evolution and model/view separation in component architectures which makes all this possible.

REFERENCES

SAS/IntrNet: http://www.sas.com/products/intrnet/index.html
Enterprise Guide: http://www.sas.com/products/guide/index.html
SAS/Share: http://www.sas.com/products/share/index.html
SAS/AF: http://www.sas.com/products/af/index.html
OLAP Server: http://www.sas.com/technologies/olap/
Application Dispatcher:
http://www.sas.com/rnd/web/intrnet/dispatch/

ACKNOWLEDGMENTS

We are grateful to and acknowledge the contributions of:

All individuals who have been involved in tools and SOS application development over the past 11 years who have contributed code and designed architecture that is still productive and in use today.

All SAS product developers who have contributed to the products mentioned here that provide extensible and portable application frameworks that make Web-enabling SAS applications possible.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Mary Jafri
SAS Institute Inc.
Cary, NC 27513
Email: Mary.Jafri@sas.com

Teresia Arthur
SAS Institute Inc.
Cary, NC 27513
Email: Teresia.Arthur@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

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