Paper 018-2008

Web-Enable Your SAS® Applications

Teresia Arthur, SAS Institute Inc., Cary, NC
Mary Jafri, DRQ Solutions, Inc., Cary, NC

ABSTRACT

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

The objective of this paper is to demonstrate how a production SAS/AF® software application, which was originally written in 1993, on MVS, has become Web-enabled and serves more than 2,000 unique, global users; and its core application code has never been rewritten. This paper demonstrates how this application evolved using SAS® Component Language (SCL) (formerly called Screen Control Language) classes, and how SCL classes are now used in SAS/IntrNet® application dispatchers, e-mail gateways, and application programming interfaces (APIs).

This paper shows how SAS/IntrNet application dispatchers were used, along with SAS/SHARE® software, SAS/SHARE*NET™ software, and Base SAS® software, which supports structured query language (SQL). Using the SAS/SHARE® Driver for JDBC enables access and updates to SAS data directly through Java programs.

This paper is intended for intermediate and advanced application developers. It demonstrates the versatility of SAS software, its operating system independence, and how to design your applications to take advantage of this versatility and independence.

INTRODUCTION

SAS/AF software applications can grow on different operating systems and become Web-enabled without an expensive rewrite or extensive programming resources. In this paper, we describe our application experience, explain the evolutionary steps that we took, and demonstrate how we implemented SAS/IntrNet technologies. We share code examples and lessons that we learned in the process.

WHO WE ARE

We work at SAS in the Management Information Systems (MIS) department and the Research and Development tools team. We develop tools and applications which address internal, corporate, and enterprise business application needs, using SAS software. We develop and enhance applications to continually provide for internal user needs at minimal expense. We have created and implemented applications that remain useful and current more than 15 years after their creation. Just like you, we are constantly challenged to work smarter to meet our increasing user demands.

BACKGROUND

First, a brief background about our application to help you understand the examples in this presentation. Strategic Online Services (SOS) is an internal problem-and-request tracking system that is used worldwide by SAS employees to organize and manage their daily workload. For clarity, we focus here on SOS problem examples, instead of request examples, although both of these have similar components and they 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 e-mail to the Help Desk. Different teams have HTML forms that query their users for information. These forms use our SOS APIs to open and route problems.

In addition to its current data activity, SOS contains 15 years of archive data, an extensive answers database, and customized user profiles. Our users can interact with SOS through multiple interfaces, such as:

- SAS/AF interfaces
- Web interfaces
- E-mail gateways
- Web APIs
- Command-line APIs

The features and business rules in SOS are too numerous to list in this paper, but it should be obvious why we are anxious NOT to have to invest in a rewrite!

COMPONENT ARCHITECTURE

Component architecture has enabled us to maintain SOS easily as business rules change. It has enabled an easy transition to multiple interfaces and different operating systems, which has, in turn, enabled portability, extensibility, and maintainability.

BUSINESS-RULE ENCAPSULATION

In the beginning, SCL classes were written using object-oriented programming constructs that completely separated the model from the viewer. It was discovered later that the key to easy evolution and to Web-enabling an application is a solid-component architecture that uses business-rule encapsulation. The SOS user interface translates commands and relays data between SOS users and objects. As users request new functionality, the methods within the core SCL classes of SOS continue to evolve. All of the interfaces use the same core SCL classes, so as business rules change, we need to update only one location to serve all of the interfaces. It's important not to include business logic in any visual parts of the application. See the "Lessons Learned" section for more details.

MULTIPLE INTERFACES

The interfaces for SOS continue to grow as user needs grow. The interfaces started as SAS/AF frames, and then they expanded to include multiple Web interfaces, e-mail gateways, and command-line APIs. These additional interfaces provide great versatility and were possible because of the extended reuse of the core SCL classes in SOS that have business-rule encapsulation.

PORTABILITY

The host-specific portions of code are isolated in different parent classes, which are inherited by host-independent SCL classes. Host-specific code includes commands for sending e-mail, printing, and so on. The host-independent SCL classes are located in a separate host-specific tool catalog and are used by many applications. When migrating an application from one operating system to another operating system, we had to modify only the host-specific parent classes, which meant the SCL classes in SOS did not have to change. This amounted to only a few method updates.

As a result, there was no mystery about where to make updates in SOS for host-specific functions. All applications have the same API, regardless of the operating system, which is the key to portability.

THE EVOLUTION OVERVIEW

SOS originated in MVS on a mainframe and was originally coded using SAS version 6.06. It was ported to run on several versions of UNIX and was upgraded to various versions of SAS as they became available. It now runs on the SAS intranet, allowing worldwide access using SAS/IntrNet Application Dispatchers, an Apache Web server, and a Tomcat Java Virtual Machine (JVM).

Use of the SAS/IntrNet Application Dispatchers instantly made SOS a host-independent application accessible by anyone with a Web browser. Users around the globe with all types of hardware have no trouble accessing and using SOS.

The SOS evolution from MVS to UNIX to the SAS intranet was accomplished via SCL, SAS/IntrNet Application Dispatchers, and JavaServer Pages (JSP) and without rewriting the core application code. The SOS user base continues to grow, and user satisfaction is high. As noted, SOS has more than 2,500 unique global users that are owners within the system and who use it to manage their work. SOS has more than 10,000 users who are not owners, but access SOS to view status updates. SOS handles a million Web hits each week.

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 AND 2: OPERATING SYSTEMS AND E-MAIL GATEWAYS

UNIX

SOS was ported from MVS to UNIX. We used cross-domain access to SAS/SHARE servers to copy data to its new home on a UNIX file system. The CPORT and CIMPORT procedures were used to port the SAS/AF catalogs. Figure 1 and Figure 2 show the SOS user interface using SAS/AF frames on UNIX.

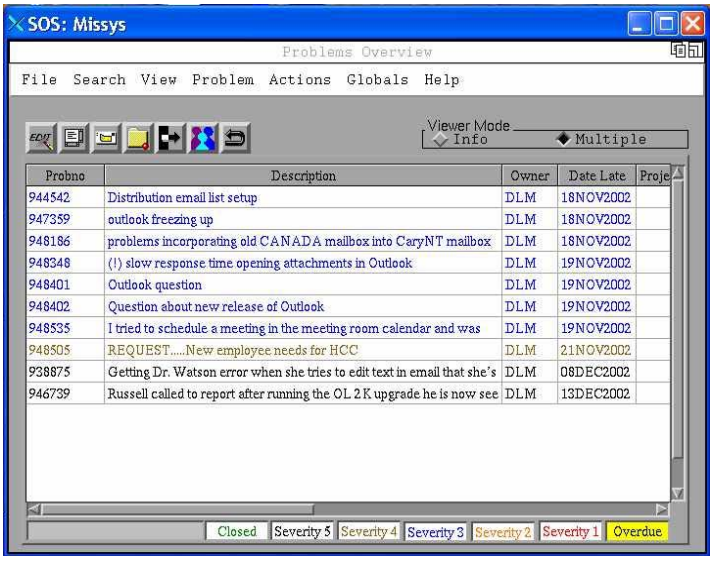

Figure 1. SOS Problem Overview (SAS/AF on UNIX)

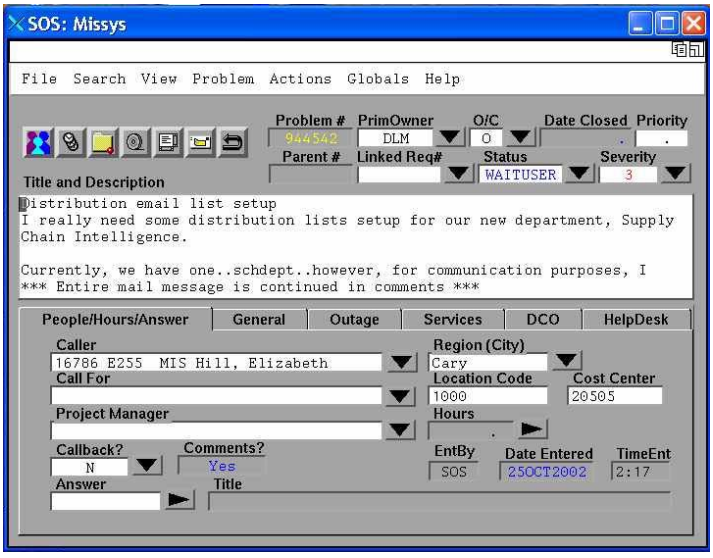

Figure 2. SOS Problem Detail (SAS/AF on UNIX)

E-MAIL GATEWAYS

We reused an existing SCL object, which added comments (unlimited number of lines of text) to each SOS problem, to create an interface that adds the text of an e-mail as a comment to an SOS problem. We created several e-mail gateways for automated approvals, problem opens, and so on. In this paper, we demonstrate the e-mail gateway for adding comments.

We used Elm filter rules on UNIX to trigger a small Korn shell (ksh) script, which invokes SAS and runs our SCL program in batch mode. (In the near future, we will be moving SOS to Sun hardware and will use Procmail instead of Elm.) The SCL program reads the e-mail's text from the standard input stream (STDIN) and then sends it to the comment class. Using the existing core SCL object simplified the procedure for us and ensured the enforcement of business rules.

EVOLUTION 3: WEB-ENABLEMENT

EASY INTRODUCTION TO THE WEB

In the early days of our Web development, the htmSQL component of SAS/IntrNet enabled us to provide a view of the detailed data information for each SOS problem. Because using htmSQL requires little more than basic SQL and HTML knowledge, the learning curve for using htmSQL technology is short. HtmSQL improved how information was sent in the e-mails that were generated by the SOS system.

Today, SOS sends an e-mail that contains only the latest information and includes a link to the Web view for more details. This is an improvement over sending old information and past details in e-mail, which is what SOS had been doing. Figure 3 and Figure 4 show the SOS user interface for the Web.

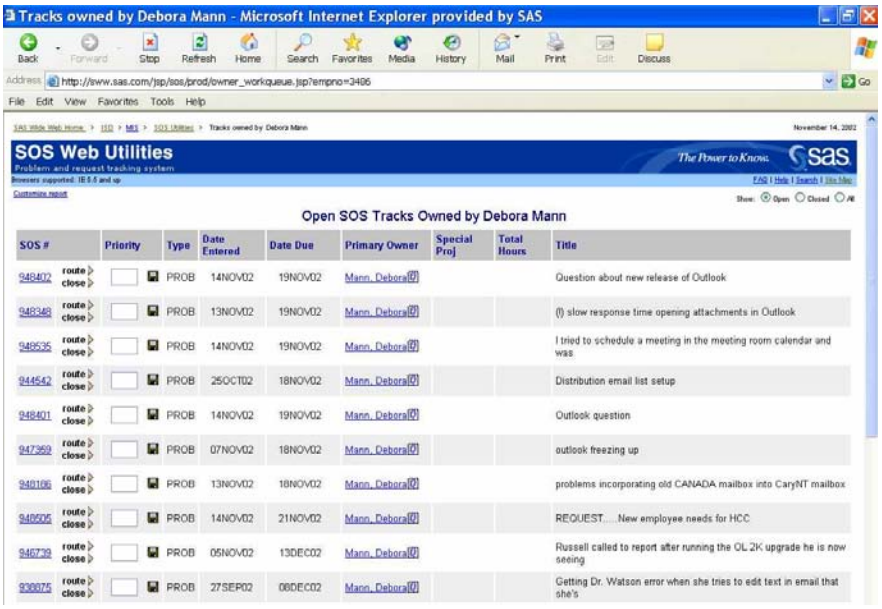

Figure 3. SOS Overview (JSP)

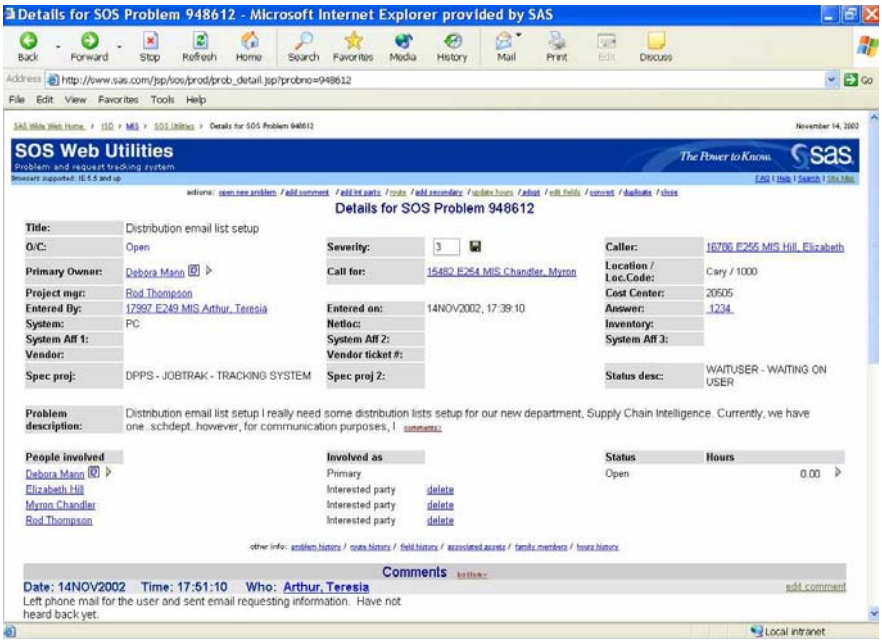

Figure 4. SOS Problem Detail (JSP)

E-MAIL GATEWAYS FROM THE WEB

After providing read-only views of SOS data, the next step was to provide the ability to update from the Web. We took advantage of our existing e-mail gateways from the Web. For example, we created a Common Gateway Interface (CGI) script to process the text from an HTML text area and send it to the e-mail gateway. This feature enables users to add comments to SOS problems from the Web. These e-mail gateways provided an easy segue into Web functionality because the e-mail gateways required simple HTML forms that send appropriately built e-mails. Figure 5 shows the Web view's Add comment page and Figure 6 shows the result page.

Figure 5. Add Comment Page That Also Provides Automatic E-mail Options

Figure 6. Result Page That the User Sees After Adding the Comment

SAS/INTRNET APPLICATION DISPATCHERS

We now needed a robust Web interface that used other application update objects. We did not have the time or the resources to rewrite SOS, and the SCL classes in SOS contained years of complex business rules. The SAS/IntrNet Application Dispatcher technology turned out to be the answer to our need.

By definition, the SAS/IntrNet 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. SAS software does not have to be installed on your machine!" (SAS/IntrNet 9.1: Application Dispatcher).

The SAS/IntrNet Application Dispatcher consists of the Application Server, the Application Broker, and other utilities that might be used. Basically, an Application Server is a SAS session that is listening on a port. Through this port, a CGI script (an Application Broker) is configured to send the submitted HTML form information to that SAS session. In SAS/IntrNet software, SAS provides this configurable CGI script, which you can run on your Web server.

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

Today, the SOS Web interface consists of a series of JSPs that use the following technology:

- JavaScript to create a dynamic user interface
- The SAS/SHARE Driver for JDBC for using SQL to query SOS 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/IntrNet Application Dispatcher technology to update SOS data with the information submitted from HTML forms

This combination of technologies enables all SAS and Java processing to take place on our servers, which means that no client installation is involved and only a Web browser is needed! This creates a host-independent environment. SOS is used globally, but it accesses servers located in the United States. The performance is sufficient for countries that have good network connections.

The SAS/IntrNet Application Dispatcher solution includes an Application Broker and multiple Application Servers. The Application Servers are defined as a socket service, where each server listens on a permanently defined port. We have multiple Load Managers that keep track of which Application Servers are available. When Web requests are made, the Load Managers send each Web request to an available Application Server for processing.

The use of a Load Manager is optional. You can define the Application Servers as a pool service or a launch service instead of a socket service, depending on which type of service serves your needs best. You can use a combination of these types of services. SOS is used 24/7, so the optimal solution for SOS is to have Application Servers defined as multiple socket services. The launch service or pool service might be a better choice for systems that occasionally run reports or for systems that do not access Application Servers continuously throughout the day.

Both launch and pool services invoke SAS sessions only when they are needed, which does not tie up server resources with idle SAS sessions. As suggested, a combination of these types of services might serve your needs best. For example, if you occasionally run reports that take more than a few seconds to run, then you might not want them interfering with your transactional system updates. In this case, you might want to define a launch service for running reports, so that it does not tie up your Application Servers defined as a socket service.

The SAS intranet hosts generic application dispatchers for general public use. The generic application dispatchers were defined until recently as socket services. They are now defined as pool services and they use a load manager. These application dispatchers are efficient and provide a simpler configuration on the server side.

Socket Services

Advantages

- The server is always running.
- The system administrator has explicit control of resources allocated to the socket service.
- An increasing load can be handled by adding more servers.

Disadvantages

- Servers must be started and stopped manually.
- There is no dynamic scaling to handle increasing loads.

Pool Services

Advantages

- Servers are started as needed.
- Once they are started, servers can be reused by new clients.
- Pool services can be on a different system than the Web server, and can be distributed across multiple server systems.

Disadvantages

- The load manager must be installed and the configuration is more complex.
- Client requests might have to wait for a server to be started.

Launch Services

Advantages

- Server start-up is automatic for each client request.
- Each client request runs on a separate server.
- Many client requests can run in parallel.
- Other applications will not affect client requests.

For more details about these types of services, see the SAS/IntrNet 9.1: Application Dispatcher product documentation. For this paper, we focus only on the socket service.

EXAMPLE LOAD MANAGER AND SOCKET SERVICE IMPLEMENTATION

Example of a SAS/IntrNet 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>
```

Example of a SAS/IntrNet 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 these examples is named sos and is defined as a socket service in the Application Broker configuration file. This file resides in the CGI bin for our Web server. The Application Broker executable, /bin/sosbroker, is called from a HTML 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 Application Broker configuration file definition determines to which port and server to send the data.

Example Application Broker configuration file 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 Load Manager is listening on port 5555 of the server.sas.com machine. It is 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 Application Broker configuration file.

Example Application 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 a time-out message is sent to the person that submitted the form. A time-out might occur if a request processes longer than expected, or if the Application Server is having trouble because of a program error. If the Application Server that the form requested is not running, then a message will be returned that indicates that the Application Server was not found. (In this case, it is not the time-out message.) In either case, the ServiceAdmin and ServiceAdminMail values are displayed as the support contact.

You can specify the Application Server name as the _service in the HTML form, which is necessary if you need to access a particular Application Server and omit the use of the Load Manager, or if you need to bypass the Load Manager in special circumstances. For example, specify _service=”sos2” instead of _service=”sos”. If you are using a Load Manager, the _service name in your HTML forms should be the name of the Load Manager.

Now that socket services have been defined in the Application Broker configuration file, you can start running the Load Manager and the Application Server as detailed.

Example of a Load Manager start command from a UNIX command line:

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

Example of a 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 can be retrieved from !SASROOT/utilities/bin. The previous examples are basic examples of starting and stopping a Load Manager. For more information on other start and stop command options, executables for other platforms, and options for viewing statistics regarding your Load Manager, see the SAS/IntrNet 9.1: Application Dispatcher product documentation.

Example of an Application Server start command:

SAS provides documentation for using the INETCFG utility to generate the PROC APPSRV code that is needed to start an Application Server. To access this information, see the SAS/IntrNet 9.1: Application Dispatcher product documentation.

Example of an Application Server stop command:

```
http://server/bin/broker?_service=sosgen1&_program=stop
```

Once you have your SAS/IntrNet Application Dispatcher processes configured and running, then you can communicate with them from the Web to run your batch SAS programs. When you call an Application Server from an HTML form, the name and value of each HTML form element is passed as a name/value pair both as macro variables and in an SCL list. Our example focuses on using the SCL list.

The previous form example produces a SCL list like the following:

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

The _RMTUSER element is not a name/value pair that is passed from the HTML form. There are many elements that start with an underscore that are automatically created by SAS/IntrNet Application Dispatcher technology. _RMTUSER has the user ID value that was used for authentication. HTREFER is another automatic element; it holds the URL of the page from which the HTML form was submitted.

Because our SCL objects were non-visual, the only thing we had to do was create new SCL entries to use the SCL objects from the Web application interface. These entries take the information in the SCL list passed to the SAS program, then process and pass this information to an SCL object in the form that the object expects.

Example SCL code that interfaces between an HTML form and an SCL 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 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 in */
/* 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 the SOS design that we created multiple child applications. Each child application inherited SOS classes and interfaces, which was done to provide custom tracking systems for different functional areas. We added methods to migrate problems and requests between the child applications when needed. We did not copy and duplicate the SOS code. All of the child applications inherited code from SOS; some methods in SOS were overridden for the specific business rules that were required by the child application.

While we were able to provide new applications in a short time frame, we learned that this is not always such a great thing to do. It served our application needs at the time, but ultimately, it costs us in maintenance. We have since then found a better way to meet our needs. Now, added functionality to the core SOS classes, and incorporated the child application functionality into SOS then retired the children. Long live the parent!

All of the child applications would also have needed to be Web-enabled, which would have required an overwhelming amount of work for us in creating and maintaining web interfaces for them. By incorporating the child application functionality into SOS, we have one system (SOS) to maintain/enhance and all of the child application users benefit.

STAY TRUE TO YOUR CLASSES!

During tremendously overworked moments, we took a few shortcuts when adding new functionality. For example, we added business logic to a few SAS/AF frames, instead of adding it appropriately to the class methods. This has cost us more than it saved us. Ultimately, we had to remove the business logic from the SAS/AF frames and add it to the class methods. Therefore, we highly recommend putting all business logic into your class methods, without exception.

EVEN HARDWARE MIGRATION CAN BE EASIER

Migrating to new server hardware is inevitable, as hardware is improving and our global SOS user base grows. Putting our data and programs on Network Attached Storage (NAS) simplifies the task of migration. We run our processes on newer servers, but the data and programs do not need to move. Assigning alias names to your server hardware is helpful. As you deploy on new servers, you do not have to change LIBNAME statements if you reassign the aliases to the new servers.

Web-enabling SOS made it platform-independent for our users. We have users on multiple operating systems, but all they need to access SOS is a Web browser. Because SOS is simply a URL from the user's perspective, any server hardware changes are transparent.

CONCLUSION

Today, the Web is the platform of choice for application access, but what will it be tomorrow? As soon as you master an operating system, your users want a different one. Being able to succeed in this environment depends on extensible and portable application architectures and software. SAS applications can be Web-enabled and have multiple interfaces with reduced effort and without rewriting the core application.

The architecture in SOS makes it possible to take an iterative approach to interface development. There were extended periods of time when no developers were assigned to SOS because of higher priority projects and limited programming resources. During those times, users helped themselves by using SOS APIs, which ensured progress and use of appropriate business rules. They were able to create their own customized Web forms that fed data into SOS APIs.

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

REFERENCES

SAS/IntrNet 9.1: Application Dispatcher. 2007. SAS Institute Inc. Available at: http://support.sas.com/rnd/web/intrnet/dispatch.

SAS/IntrNet 8.2: Application Dispatcher. 2007. SAS Institute Inc. Available http://support.sas.com/rnd/web/intrnet/dispatch82.

RESOURCES

SAS/IntrNet Software Documentation. SAS Institute Inc. Available online: http://support.sas.com/documentation/onlinedoc/intrnet.

SAS/SHARE Documentation. SAS Institute Inc. Available online: http://support.sas.com/documentation/onlinedoc/share.

AS/AF Documentation. SAS Institute Inc. Available online: http://support.sas.com/documentation/onlinedoc/af/.

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 17 years, and who have contributed code and designed architecture that is still productive and in use today.
- Joe O'Brien of SAS for contributing valuable input based on his hands-on experience with SAS/IntrNet products.
- All SAS product developers who have contributed to the products mentioned 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:

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

Mary Jafri
DRQ Solutions, Inc.
Email: maryjafri@gmail.com