Application Development in SAS/AF® Software Using Class Libraries

Carl R. Haske, Ph.D., STATPROBE, Inc., Ann Arbor, Michigan

ABSTRACT

Proper applications are logically divided into functional layers. Classes promote the logical organization of components into generic and application-specific elements. This division allows a developer to derive generic classes that are reusable in different types of software systems. Classes also enhance usability because they give applications a consistent presentation. For the developer, the advantages of using classes are rapid application development and consistency across multiple applications. The consistency provided by classes contributes to a reduced learning curve for end users as they become familiar with many applications developed with use of the same class libraries.

Standard classes handle system-level details like registering users, maintaining a data dictionary, tracking system activity with an error log, typical database management functions, and typical dialogs with the user. Classes also provide default behavior in an application. An application can completely override default behavior for any special application needs, or it can perform tailored behavior and branch to the default behavior.

In SAS/AF, classes are organized in SAS® catalogs. An application prototype can be quickly developed in SAS by assembling class entries that are stored in SAS catalog files. This tutorial describes techniques to organize class libraries efficiently as part of an application development framework.

INTRODUCTION

A SAS/AF application uses catalogs to organize the various dialogs and associated code comprised in the application. The nature of the catalog file structure supports an object-oriented, "layered" approach to application development within the SAS/AF system. This tutorial shows how to assemble a well-developed system to support application development in SAS/AF.

The first section of this paper discusses application layers and the concept of breaking an application into components that can be reused in many applications. Not all components in this system are completely "plug and play"; however, it is possible to design the application constituents so that they require minimal customization for use in any application.

The next section discusses an application framework. This section includes several examples of how to develop standard frames and classes for the application front end. In particular, this section shows how to develop subclasses of widgets and frames to support the application framework. The section concludes by showing how to develop a general method to navigate through the menu system of any application and how to standardize and control the environment within SAS so that any application developed within the system can run with a small amount of customization.

In the third section, we discuss how to organize standard application components into separate catalogs and provide examples of how to develop some of these components, such as a login facility and generic dialogs.

The final section shows how to pull all the application components together and link the customized elements with the reusable components. We will see how to prototype and design the top-level components of the application, customize dialogs, and write a compiler for the application using Proc Build.

SAS/AF is a useful development platform to provide the non–SAS user with applications to access SAS data and analyses. SAS/AF provides many classes that are useful in developing applications. However, in cases where many applications need to be developed or recompiled for implementing custom features, it is important to develop new classes, standard frames, and a standard development environment to support this effort.

APPLICATION LAYERS

Application development involves several meetings between developers and users to understand what the application must do. During the early meetings, the developer learns the basic requirements and can begin assembling prototypes. To get started, the developer must get answers to such questions as:

- Is the application a single user or multiuser system?
- What is the intended environment—standalone or client-server?
- What type of security is required? Is a login facility required?
- What types of system administration are required? A user registry? Other types of registries?
- Is error logging required?
- Is this a database application? What is the nature of the database?
- What types of user dialogs will be required?

To a large extent, the answers to these questions determine the architecture of the system. Moreover, it is possible to develop standard classes that address these questions, so that the initial stages of development involve an "off the shelf" approach of selecting these components from predeveloped libraries rather than expending significant effort in new development.

Because software applications, especially database applications, often have many common features, the developer of a new software application need not start from scratch. It is advantageous to view an application as being constructed in layers, which can be classified and can often be developed independent of the overall thrust of the application. Application components can then be abstracted into class libraries, which provide the building blocks for an application development environment. Applications can be constructed quickly by selecting from libraries of standard components. The developer is left to add custom features to fine-tune the final implementation of the application.

Diagram 1 on the next page illustrates the application layer concept and the architecture used to build an application upon standard layers. The lower level layers of an application can often be based on reusable code, whereas the higher level layers of the application require more customization. For example, a login facility is a generic process that requires a standard login dialog and a standard system user table. On the other hand, an asset tracking database may require the implementation of depreciation formulas and highly customized dialogs to interact with the user.

As we proceed through this tutorial, we will learn how to set up an application development system for SAS/AF in which all application layers up to and including the System Database Layer are completely standardized. To develop a new application, all that will be necessary will be to design the elements of the main menu, the application database, and the application dialogs. In the next section, we discuss a standard framework for the menu driver component of applications and lay the groundwork for the application development system.
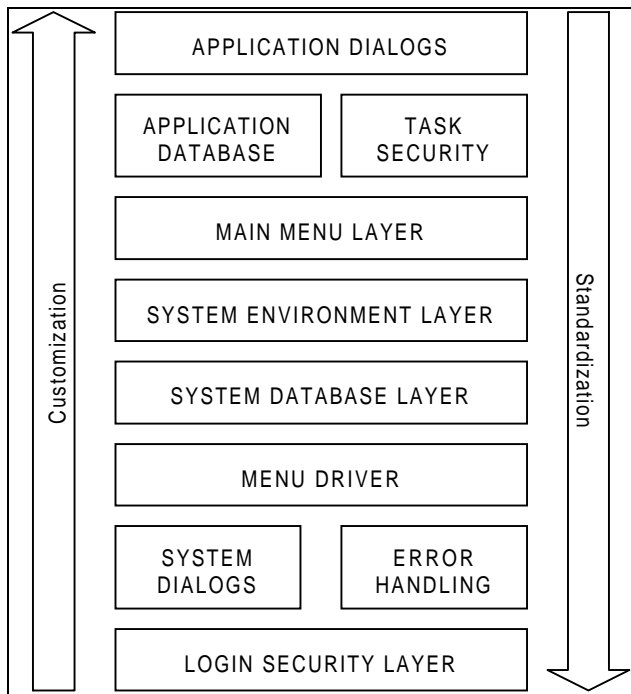
Diagram 1. Logical layers of an application

APPLICATION FRAMEWORK

Application Switchboard

An application framework consists of a set of logical tasks, organized hierarchically so that higher-level tasks can branch to subtasks. For example, an application task called "Database Administration" may consist of several tasks, including "Setup Database," "Modify Database," "Copy Database," "Browse Database," and "Print Database." Therefore, when the user tells the application to perform a specific task, the application will either branch to a set of subtasks or complete the execution of the task via a series of application-specific dialogs.

The natural structure for this task management is a tree (see diagram 2). Each node represents a task. Each node or task, when selected, either branches to additional tasks or performs a specific task. In the diagram, one task can branch to as many as four levels in the task hierarchy.
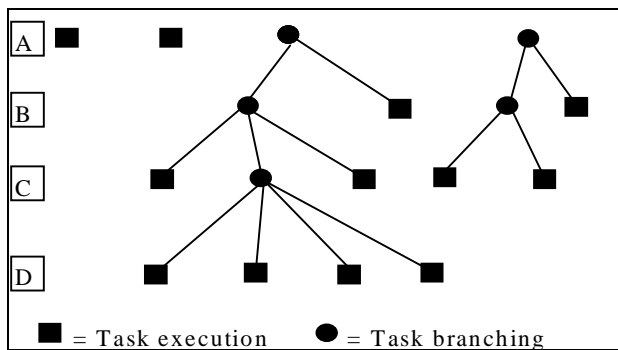

Diagram 2. Application schematic

Note that the navigation through such a system, i.e., branching from node to node and executing actions, is independent of the actual content at each node. A general system that models this process

has a user interface commonly referred to as a "switchboard." In our application framework, the switchboard provides the standardized menu driver layer of the application.

Menu Image Icon Class

To model this application structure in SAS/AF, we can develop a subclass of the Image Icon class. The subclass is referred to as the Menu Image Icon class. A switchboard in an application consists of a set of menu image icon objects. Every application is prototyped by defining a set of switchboards as parameters in the system. To complete an application, it is necessary only to specify the menu image icon objects that execute a specific function as opposed to branching to another switchboard. The Menu Image Icon class has four new instance variables, listed in table 1.

Table 1. Menu Image Icon Class Instance Variables

Instance Variable	Description
DISP_APP	Catalog entry to display
SLIST	Slist entry corresponding to a Menu Image Icon set
SYSLEV	Display attribute for system administration
TASKLEV	Display attribute for task levels

The DISP_APP instance variable is a text string that contains the name of the catalog frame or SCL entry to execute when the menu image icon is activated to perform a specific task. For example, an application catalog MYLIB.MYAPP may contain a dialog called SELDB.FRAME that allows the user to select a SAS database with the intention of performing some data processing. In this case, the contents of the DISP_APP instance variable would be "MYLIB.MYAPP.SELDB.FRAME."

The SLIST instance variable is a text string that contains the name of the catalog. A slist entry defines a switchboard to display when the menu image icon is activated to branch to lower level tasks. For example, a menu image icon in an application could be "User Administration," which may branch to the subtasks "Add User," "Delete User," and "Modify User," with each subtask accessible via a menu image icon.

The remaining instance variables, SYSLEV and TASKLEV, are defined for security reasons. The SYSLEV variable allows various security levels to be defined for administrative activities within the application. The TASKLEV variable allows various security levels to be defined for application-specific tasks. For example, administrative activities in an application involve managing the user registry and system database registry. The menu image icons that access these activities within the application will use the SYSLEV variable and remain hidden if the user does not have the proper system level security clearance. Any menu image icon that has SYSLEV 2 or lower would be visible to a user who has system level clearance 2 or higher.

Using a switchboard menu system has many advantages. Designing the front end of an application is reduced to a matter of minutes. The developer can produce several prototypes and meet with the users to discuss the direction of the application, helping ensure that development time is not misspent. Another advantage of this framework arises during application development. In many cases, the specific frame behind a menu image icon object is general enough that the frame can be saved as a template and reused in other applications. The switchboard is also very easy and intuitive for the user, reducing the time needed to master an application.

When a switchboard is displayed, the application is in a wait state until the user selects one of the menu image icons object of the switchboard. The _select_ method for the menu image icon has been overridden. Listing 1 shows the code for the select method.

```
SELECT:
 Method;
   If length(disp_app)>0 then do;
     call method('sashelp.fsp.parse.scl','namediv',disp_app,
              lib,cat,entry,type,rc);

    If rc=0 and lib=_BLANK_ then do;
      call send(_FRAME_,'_GET_NAME_',name);
      call method('sashelp.fsp.parse.scl','namediv',name,
              lib,cat,dummy1,dummy2,rc);
      If rc=0 then do;
       name=lib||'.'||cat||'.'||disp_app;
       If exist(name) then disp_app=name;
        Else do;
         name=searchpath(disp_app);
         If exist(name) then disp_app=name;
        End;
      End;
    End;

    If cexist(disp_app) then do;
      If type='FRAME' or type='SCL' then
        call display(disp_app);
    End;
    Else call send(_FRAME_,'_SET_MSG_',
               'WARNING: Entry '||disp_app||' does not exist');
   End;

  Call super(_self_,_METHOD_);
 Endmethod;
```

Listing 1. _SELECT_ method for menu image icon

The key line in this method code is:

```
                    call display(disp_app);
```

This line causes the catalog entry specified in the instance variable DISP_APP to execute. Note that the catalog entry should be of type Frame or SCL; this class was designed this way because all applications developed at STATPROBE use Frame entries as opposed to Program entries. The menu image icon class could easily be modified to allow branching to other types of entries.

The remainder of the code in the _select_ method is for error checking. The code first splits the text in DISP_APP to get library, catalog, entry name, and type, the components of the four-level name. The existence of the entry is then verified prior to proceeding.
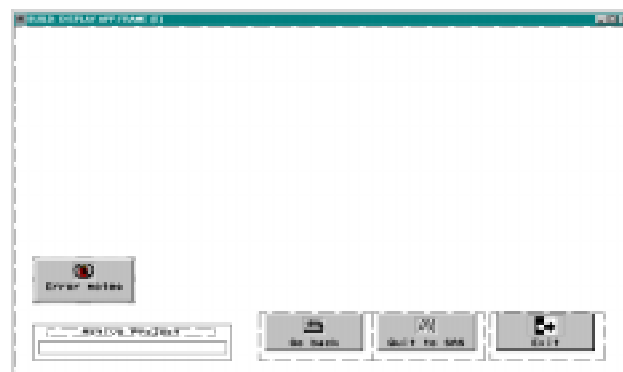
Application Node Navigator

If the menu image icon object should branch to another array of switchboard items, the application node navigator handles this action. The application node navigator controls the application session and causes the proper switchboard to display. The application node navigator is displayed in figure 1.

APP.FRAME is the main application screen; it supplies a blank region in the central portion of the view port so that the switchboard menus can display dynamically. This frame is created from a subclass of the frame class by overriding the _bpostinit_, _init_label_, _main_label_, and _term_label_ methods. These methods are overridden to add standard application processing code to the frame. Additional methods are also defined for the frame subclass and are described later.

There are three standard icons at the lower right: *Exit*, *Quit to SAS*, and *Go Back*. The Exit icon will exit the SAS session. The Quit to SAS icon will exit the AF application but will leave SAS active. The Go Back icon causes the previous switchboard to display.

Since many applications support multiple processes or databases, the frame has a status bar at the lower right that displays the active project or database. Finally, the Error Notes icon displays if the application gets into a positive error state. This icon leads to a dialog

Figure 1. APP.FRAME

box in which the user can leave a note to help the developer debug a problem at a later time. This feature is discussed in more detail later.

The application node navigator is general and can be used for many applications, and it can easily be customized to individual applications. For example, objects such as graphics or informational text can be added to the main screen.

Listing 2 shows the code to override the _init_label_ method of APP.FRAME. When the frame is displayed, the catalog slist entry for the main menu and application title are passed. These parameters are retrieved by the _get_arglist_ method. The slist entry is a list of attributes for menu image icon objects that have been designed and saved by the developer in the application catalog.

```
INIT:
 Method;
  call super(_self_,_method_);

  call send(_self_,'_get_arglist_',arglist);
  mainmenu=getitemc(arglist,1);
  If listlen(arglist)>1 then
     apptitle=getitemc(arglist,2);

  If not cexist(mainmenu) then do;
   _msg_='FATAL ERROR: Unable to load main menu bridge';
   rtn=0;
   status='H';
  End;

  menu='MAINMENU';

  menulist=makelist();
  call method('apputil.scl','buildapp',mainmenu,menu,menulist);

  If apptitle=_BLANK_ then
    apptitle='Template Application';
  call send(_frame_,'_SET_TITLE_',apptitle);

------- More processing statements omitted for clarity -------

  currmenu=getniteml(menulist,menu);
  call send(_self_,'showmenu');
 Endmethod;
```

Listing 2. _INIT_LABEL_ method of APP.FRAME

3

The code for the application node navigator uses the attribute in the slist entry to instantiate the menu image icon objects that make up the switchboard. The instantiation occurs in the INIT section with a method call to BUILDAPP. This recursive method begins with the main switchboard menu and traverses down all of the switchboard nodes, instantiating all the menu image icons used in the application. This tree structure of menu image icon objects is returned in the SCL variable MENULIST.

Next the code sets the title of the application and creates other frame objects if they are need by the application. Finally, the code sets the current menu and displays the menu.

The _main_label_ method for APP.FRAME is overridden by code displayed in listing 3. First, the code checks the global environment list to determine if a project identifier is available and has a nontrivial value. If so, then the objects that show the status of the project identifier are initialized and made visible. Note the dynamic nature of the code here. If an application does not need status objects to show the current project, then the project identifier will not exist in the global environment list; thus these objects will never be visible.

Next, the code checks to see if the application is in error. If the application has a positive error state, then the error notes icon is displayed to allow the user to leave a message. Finally, the code checks the current menu and branches either back to the previous menu or forward to a new menu switchboard if necessary. The previous menu is hidden and the proper menu is made visible.

```
MAIN:
 Method:
  envlist=envlist('L');
  If nameditem(envlist,'PROJ_ID') then
   If getnitemc(envlist,'PROJ_ID')=_BLANK_ then do;
     call send(projctr, '_HIDE_');
     call send(cplabel, '_HIDE_');
     call send(proj_id, '_HIDE_');
   End;
   Else do;
     call send(proj_id,'_SET_TEXT_',
                 getnitemc(envlist,'PROJ_ID'));
     call send(projctr,'_UNHIDE_');
     call send(cplabel,'_UNHIDE_');
     call send(proj_id,'_UNHIDE_');
   End;

  If nameditem(envlist,'IN_ERROR') then
   If getnitemn(envlist,'IN_ERROR') then
     call send(errnote,'_UNHIDE_');
   Else
     call send(errnote,'_HIDE_');

  call send(_frame_,'_GET_CURRENT_NAME_',menu);

  If menu='BACK' then menu=getnitemc(currmenu,'PARENT');
  Else if nameditem(menulist,menu) then do;
   If currmenu=getniteml(menulist,menu) then
     return;
  End;
  Else return;

  call send(_self_,'hidemenu');
  currmenu=getniteml(menulist,menu);
  call send(_self_,'showmenu');
 Endmethod;
```

Listing 3. _MAIN_LABEL_ method of APP.FRAME

Listing 4 shows the code to override the _term_label_ method. This method simply cleans up and deletes the menu list that was created.

```
TERM:
 Method;
  rc=dellist(menulist,'Y');
 Endmethod;
```

Listing 4. _TERM_LABEL_ method of APP.FRAME

Code for other methods is displayed in listing 5. This includes the HIDEMENU method, SHOWMENU method, DONE method, and EXIT method. The HIDEMENU and SHOWMENU methods are used to manage the switchboards and make the proper switchboard visible to the user. The DONE method causes the AF application to terminate and leave the SAS session active. The EXIT method terminates the AF session and the SAS session, returning the user to the operating system.

```
HIDEMENU:
 Method;
  Do i=1 to listlen(getniteml(currmenu,'MENUICONS'));
    obid=getitemn(getniteml(currmenu,'MENUICONS'),i);
    call send(obid,'_HIDE_');
  End;
 Endmethod;

SHOWMENU:
 Method;
  If menu='MAINMENU' then
    call notify('BACK','_HIDE_');
  else
    call notify('BACK','_UNHIDE_');

  Do i=1 to listlen(getniteml(currmenu,'MENUICONS'));
    obid=getitemn(getniteml(currmenu,'MENUICONS'),i);
    call send(obid,'_UNHIDE_');
  End;
 Endmethod;

DONE:
 Method;
  and=1;
  call display('yesno.frame','Exit '||apptitle||'?',ans);
  If ans then
    call send(_self_,'_set_status_','H');
 Endmethod;

EXIT:
  and=1;
  call display('yesno.frame','Exit SAS?',ans);
  If ans then
    call execcmd('ENDSAS');
Return;
```

Listing 5. Other methods of APP.FRAME

Application Environment

There are two key techniques used in this system to enable the standardization:
- The system Database File System (DBFS) that stores metadata for the application.
- The Global Environment List, which is a SCL list that persists throughout the AF session and is used to store global information at run time. The previous section showed how information can be stored in the global environment list to cause the application to dynamically execute code.

The DBFS has a standard USER table that is referenced by the login facility. The DBFS is typically accessed by the standard libref logical DBFS, which points to a physical subdirectory named DBFS in the directory where the application is installed. For example, if the application is installed in c:\appdir, then there is a subdirectory, c:\appdir\dbfs, which houses the system database.

The starting point for each application is a catalog entry called MAIN.SCL. Listing 6 shows the code for MAIN.SCL. This module is a generic application driver. As an application prototype is developed, the code in MAIN.SCL may be modified to accomplish design changes at the high-level function of the application. However, it is typical for this program to remain static throughout the development of the application.

MAIN.SCL has four primary sections:
- Initialize DBFS.
- Initialize system environment.
- Login user.
- Execute application.

These four sections make up the login security layer and system database layer of an application. The standard library DBFS is included in most applications, and the first section of MAIN.SCL defines this library. The DBFS consists of the user table, project table, assignments table, and any other specific files necessary to track data for an application. If an application is very basic and does not need to track system data in a database file system, these statements are removed from MAIN.SCL.

```
INIT:
 *** Set the standard libref DBFS ***;
 If libref('DBFS') then
   rc=libname('DBFS');
 rc=libname('DBFS',
            pathname(scan(screenname(),1,'.'))||'\DBFS');
 If rc>0 then do;
   call method('ehandler', 'errlog', screenname(),_status_, event(),
            'FATAL ERROR: Unable to initialize system database.',
             rc, _self_);
   If rc>0 then do;
     _status_='H';
     return;
   End;
 End;

 *** Setup installation specific environment ***;
 call method('install.scl','environ');

 *** Login ***;
 call display('login.frame', rc, getnitemc(envlist('L'), 'APPNAME'));

 *** Check login success ***;
 If not rc then do;
   _status_='H';
   Return;
 End;

 *** Execute ***;
 call display('app.frame', scan(screenname(),1,'.')||'.'||
          scan(screenname(),2,'.')||'.'||getnitemc(envlist('L'),
          'APPMENU')||'.slist', getnitemc(envlist('L'),
          'APPNAME')||' V.'||getnitemc(envlist('L'), 'VERSION'));
Return;
```

Listing 6. MAIN.SCL

The method *environ* is invoked to set up the application environment. Listing 7 shows the environ method for a basic employee candidate survey application used for

- Completing surveys on candidates.
- Reporting, summarizing, and graphing results.
- Modifying survey questions.

```
ENVIRON:
 Method;
   envlist=envlist('L');

   *** Application name, version, and prime menu slist ***;
   appname='STATPROBE CANDIDATE SURVEY';
   version='1.0A';
   appmenu='cssprime';

   rc=insertc(envlist,appname,-1,'APPNAME');
   rc=insertc(envlist,version,-1,'VERSION');
   rc=insertc(envlist,appmenu,-1,'APPMENU');
   rc=insertn(envlist,0,-1,'IN_ERROR');
 Endmethod;
```

Listing 7. Environ method

Each application must always have the environ method modified for
- Application name.
- Version control number.
- Primary switchboard.

These parameters are stored in the applications local environment list. This technique facilitates the development of generic templates. When the code for a template is written, global application information can be referenced from the environment list. Typical applications store additional data in the environment list, such as standard file structures for projects.

Note that statements in MAIN.SCL reference the environment list after the environ method is invoked. For example, the application title is passed to the login frame to display the application name (see figure 4 in the next section). The login system requires a standard user table with fields representing user id, user name, and password.

The final statement in MAIN.SCL performs a call display to execute the application. The application primary switchboard, application name, and version are passed to the frame APP.FRAME, the generic application node navigator.

An additional level of standard functions is typically included in many applications. These functions include
- User administration.
- Project administration.
- Project selection.
- Active project display.

Figure 2 below shows the primary switchboard of an application. Figure 3 on the next page shows a second-level switchboard. This switchboard displays after the Survey Results menu image icon object on the primary switchboard is selected.
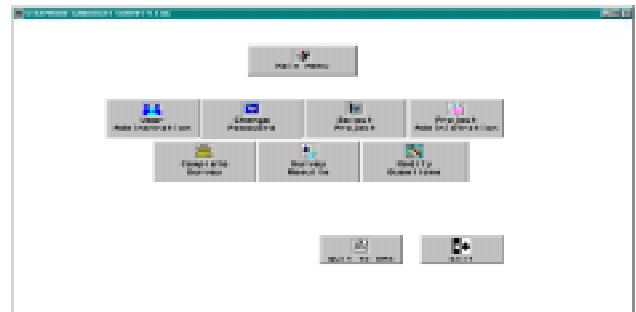
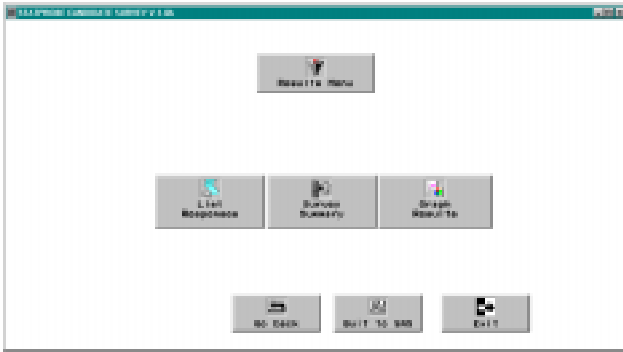**Figure 2.** Primary switchboard

Figure 3. Second-level switchboard

DEVELOPMENT ENVIRONMENT

One of the easiest ways to implement this architecture in SAS/AF is through the use of organized catalogs to classify the lower level features of an application.

Generic functions are independent of application specifics. In our framework, all generic functions include:
- Login system.
- Change password.
- Application node navigation.
- Error handler.
- General dialogs.

These functions correspond precisely to five catalogs that store the entries to support this system:
- Login.
- Password.
- Driver.
- Ehandler.
- Dialogs.

The Login catalog has only two entries, LOGIN.FRAME and LOGIN.SCL. Figure 4 shows the login screen. Note the application title in the login window. This is a parameter to the login frame SCL entry.
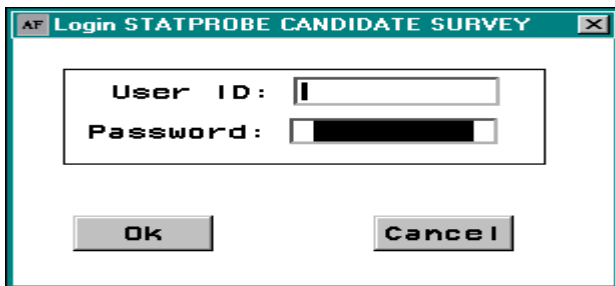

Figure 4. Login screen with title

The code for the INIT section of the frame is displayed in listing 8. As was mentioned, the application name is a parameter in the code and is used to set the title of the login frame. Each application uses a standard structure for the system user table, allowing use of the generic login frame. Also, if the login frame is ever changed or updated, it will automatically be updated for all applications. The code for the login frame also shows how the error handler works. Although the error handler is not a graphical component of applications, it consists of an SCL entry and is considered a template, because it is usable in all applications. If the login frame fails to open the user table, the error handler writes information to a slist entry in the Errorlog catalog for an application, allowing the developer to track user problems and debug the system.

```
INIT:
 *** Assume failure ***;
 rtn=0;
 *** Set application title ***;
 call send(_frame_, '_SET_TITLE_', 'Login '||appname);
 *** Make a list to hold the  user id and set key ***;
 userlst=makelist();
 *** Open USER data set ***;
 user=open('dbfs.user');
 If not user then do;
   call method('ehandler', 'errlog', screenname(),
               _status_, event(), sysmsg(), dsid, _self_);
   call notify('ok', '_GRAY_');
 End;
Return;
```

Listing 8. INIT section of LOGIN.FRAME

The Password catalog also has two entries, CHGPSWD.FRAME and CHGPSWD.SCL. Figure 5 shows how the frame appears; listing 9 is code for the "Ok" button. This code shows how to use error checking to ensure the logical correctness of users' interactions with the frame. The block of code starts with four separate verifications before writing the new password to the password data set. The code examines for nontrivial password field, correct password field, nontrivial new password field, and equivalence of new password and verification fields. When all tests are passed, the new password is updated in the password data set.

Figure 5. Frame to change password

```
OK:
 If password=_BLANK_ then do;
   _msg_="ERROR: Password is blank.";
   Return;
 End;
 If password^=getvarc(dsid,varnum(dsid,'PASSWORD')) then do;
   _msg_="ERROR: Password is incorrect.";
   Return;
 End;
 If npswd=_BLANK_ then do;
   _msg_="ERROR: New password is blank.";
   Return;
 End;
 If npswd^=vpswd then do;
   _msg_="ERROR: New password was not re-typed correctly.";
   Return;
 End;
 call putvarc(dsid,varnum(dsid,'PASSWORD'),npswd);
 rc=update(dsid);
 If rc then
   call method('ehandler','errlog',screenname(),
           _status_,event(),sysmsg(),rc,_self_);
 Call execcmd('END');
Return;
```

Listing 9. OK section of code for CHGPSWD.FRAME

The error handler is reusable in many applications as an error log, allowing the system to maintain an error log to help in debugging and validation. If a system error occurs, a record is written to the error log. The user can post a note on this record describing the circumstances of the error, helping the application developers to track errors and maintain the system. Table 2 shows the structure of the error log data. These data are written to a SAS catalog and stored by use of slist structures.

Table 2. Error Log Structure

ATTRIBUTE	DESCRIPTION
Err_date	Date of the error
Err_time	Time of the error
User_id	User identification
Err_msg	Error code
Err_src	Source location of the error
Err_info	Processing information
User_msg	User-posted message

The method code for the error handler is in listing 10. The method accepts six parameters: the name of the calling entry, the value of _status_, the value of _event_, the value of _msg_, an error code, and the identifier of the calling frame. Listing 11 shows code extracted from listing 6 that demonstrates a call to the error handler.

The error handler writes an entry to a catalog named ERRORLOG in the application directory. The handler records the user id, time and date, operating system, contents of the environment list, and all the parameters passed to the method. It writes this information to the loglist.slist entry in the ERRORLOG catalog. Furthermore, the method displays the message to the user using the standard error dialog discussed below.

```
Length user_id $8 user_msg $200;

ERRLOG:
 Method sn $ st $ ev 8 msg $ ec 8 optional= frameid 8;
   If ec>=0 then do;
     envlist=envlist('L');
     If nameditem(envlist,'USER_ID') then
       user_id=getnitemc(envlist,'USER_ID');
     errlist=makelist();
     rc=insertn(errlist,datetime(),-1,'DATETIME');
     rc=insertc(errlist,symget('sysscp'),-1,'OPSYS');
     rc=insertl(errlist,envlist,-1,'LOCAL_ENVIRONMENT');
     rc=insertc(errlist,msg,-1,'MESSAGE');
     rc=insertc(errlist,sn,-1,'SOURCE');
     rc=insertc(errlist,st,-1,'STATUS');
     rc=insertn(errlist,ev,-1,'EVENT');
     loglist=makelist();
     If cexist(scan(sn,1,'.')||'.ERRORLOG.LOGLIST.SLIST') then
       rc=fillist('CATALOG',scan(sn,1,'.')||
             '.ERRORLOG.LOGLIST.SLIST',loglist);
     rc=insertl(loglist,copylist(errlist),-1,user_id);
     rc=savelist('CATALOG',scan(sn,1,'.')||
             '.ERRORLOG.LOGLIST.SLIST',loglist);
     rc=dellist(errlist);
     rc=dellist(loglist);
     rc=setnitemn(envlist,1,'IN_ERROR');
     call display('errormsg.frame',msg);
   End;
   Else if frameid^=. then
     call send(frameid,'_SET_MSG_',msg);
 Endmethod;
```

**Listing 10.** Method code for error handler

```
rc=libname('DBFS',pathname(scan(screenname(),1,'.'))||'\DBFS');
  If rc>0 then do;
    call method('ehandler','errlog',screenname(),
         _status_,event(),
         'FATAL ERROR: Unable to initialize system database.',
          rc,_self_);
```

Listing 11. Example call to the error handler

When an error occurs during runtime, the application goes into a positive error state. The user sees an icon on the desktop, indicating that the application is in error, and can leave a note by activating the icon. Figure 6 shows the frame that allows the user to leave a note regarding an application error.
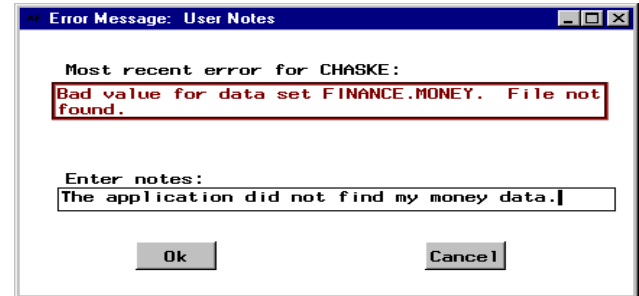

Figure 6. ERRNOTE.FRAME

The code for ERRNOTE.FRAME is in listing 12 on the following page. The code opens the error log and recalls the latest error for the user. When the frame terminates with the OK button, the code writes the content of the user note to the error log record.

The Dialogs catalog has nine entries. Eight entries are four frames and their corresponding code used to supply an error message, a warning message, an informational message, and ask a yes-no question. They are:

- Errormsg
- Message
- Warnmsg
- Yesno

The ninth catalog entry, DIALOGS.SCL, contains methods used by all of the frames. Figure 7 shows how the yesno dialog can be used.
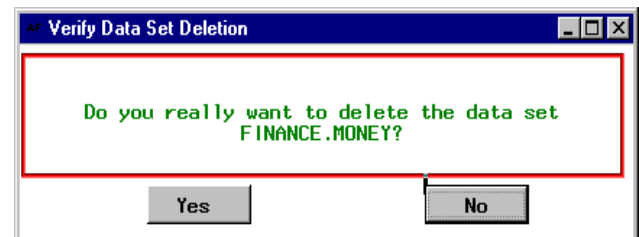
Figure 7. Yesno dialog

The code for YESNO.FRAME is in listing 13 on the following page. The frame has three parameters: a message, a default answer, and a window title. An answer value of 0 indicates a 'No' response and an answer value of 1 indicates a 'Yes' response. When the frame initializes, the value of ANS is used to activate either the Yes or No button. The text for the question is set, then the frame enters a wait state until one of the response buttons is selected.

The code uses a generic method SETMSG to center and adjust the text within the view port of the text box.

```
Length user_id $8 msg $200 note $200;

INIT:
 envlist=envlist('L');
 If nameditem(envlist,'USER_ID') then
   user_id=getnitemc(envlist,'USER_ID');
 call notify('emsg','_SET_TEXT_',
              'Most recent error for '||user_id||':');

 loglist=makelist();
 If cexist(scan(screenname(),1,'.')||
           '.ERRORLOG.LOGLIST.SLIST') then
   rc=fillist('CATALOG',scan(screenname(),1,'.')||
           '.ERRORLOG.LOGLIST.SLIST',loglist);
 erridx=nameditem(loglist,user_id,1,-1);
 If not erridx then
   Return;

 msg=getnitemc(getiteml(loglist,erridx),'MESSAGE');
 call notify('errmsg','_GET_VISCOL_',ll);
 ll=int(ll);
 line=1;
 stpos=1;
 cc=ll+1;
 Do while (length(substr(msg,stpos))>ll);
   done=0;
   Do while (not done);
     cc=cc-1;
     done=substr(msg,stpos+cc,1)=_BLANK_ or cc=0;
   End;
   If cc=0 then cc=ll;
   call notify('errmsg','_SET_LINE_',
              substr(msg,stpos,cc),line);
   line=line+1;
   stpos=stpos+cc+1;
   cc=ll+1;
 End;
 call notify('errmsg','_SET_LINE_',substr(msg,stpos),line);
 call notify('errmsg','_SNUG_FIT_');
Return;

OK:
 call notify('errnote','_GET_TEXT_',note);
 rc=setnitemc(getiteml(loglist,erridx),note,'USER_MSG');
 rc=savelist('CATALOG',scan(screenname(),1,'.')||
           '.ERRORLOG.LOGLIST.SLIST',loglist);
Return;

TERM:
 rc=setnitemn(envlist,0,'IN_ERROR');
 rc=dellist(loglist);
Return;
```

Listing 12. Code for ERRNOTE.FRAME

```
Entry msg $200 ans 8 optional= title $40;

INIT:
 If title^=_BLANK_ then
   call send(_frame_,'_SET_TITLE_',title);
 If ans=0 then
   call notify('no','_CURSOR_');
 Else
   call notify('yes','_CURSOR_');
 call send(_frame_,'_GET_WIDGET_','quest',obid);
 call method('dialogs.scl','setmsg',msg,obid);
Return;
```

```
YES:
 ans=1;
 _status_='H';
Return;

NO:
 ans=0;
 _status_='H';
Return;
```

Listing 13. Code for YESNO.FRAME

COMPILING AN APPLICATION

Step 1: Designing the Switchboards

The first step in compiling an application is to design the set of application switchboards. This step is the "pencil and paper" stage and actually occurs quite rapidly. Figure 8 shows a template switchboard that can be used as a starting point. This template for the main application switchboard has four standard icons across the top. Two icons are for administering user, project, and assignment tables. The remaining two icons are for changing password and selecting projects. Three generic tasks icons are available to attach to dialogs or branch to other switchboards.

Figure 9 shows the custom attribute frame for the Task 1 Button. Note that the Slist Entry has the value "APPTASK1.SLIST." This means that when Task 1 is selected, the application will branch to the switchboard defined by the slist APPTASK1.

The code in listing 14 on the following page compiles the switchboard and saves all necessary attributes for the switchboard icons to an slist entry.

This compiler loops through all icons on the switchboard extracting the icon attributes and inserting the attributes into an ordered SCL list that is later saved permanently as a slist.
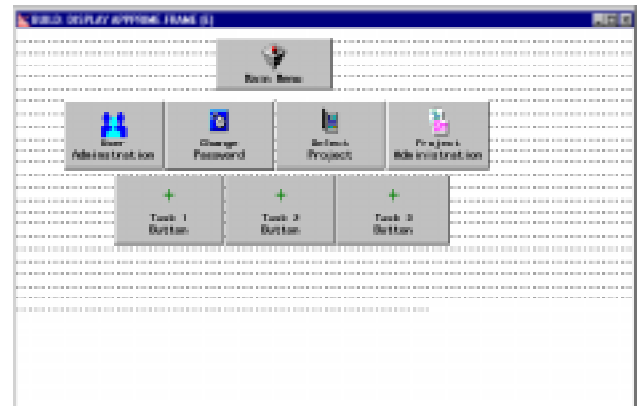
Figure 8. Template switchboard
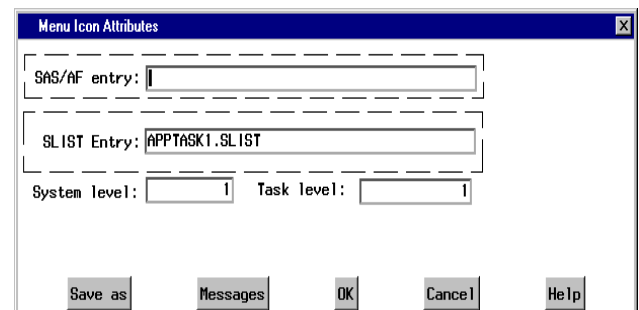
Figure 9. Custom attribute window for Task 1 button

8

```
call send(_frame_,'_GET_WIDGET_',name,obid);
 If obid then do;
   rc=clearlist(attrlist);
   rc=clearlist(savelist);
   call send(obid,'_GET_PROPERTIES_',attrlist,atype);
   call send(obid,'_GET_CLASS_',clid);
   call send(clid,'_GET_NAME_',class);

   rc=insertl(savelist,copylist(attrlist),-1,'ATTRLIST');
   rc=insertc(savelist,class,-1,'NAME');
   rc=insertl(objects,copylist(savelist),-1,'obj'||(listlen(objects)+1));
 End;
```

Listing 14. Code to save switchboard icon attributes

Step 2: Designing Custom Dialogs

All applications need custom dialogs designed, unless the application is very simple. Figure 10 shows an example of a custom dialog that is useful in database applications. The dialog allows the user to select a data set and then define criteria for subsetting the data by the data set variables. Then the dialog can browse, print, or perform another action on the data that the developer specifies. This dialog illustrates how a compact design can accomplish a significant
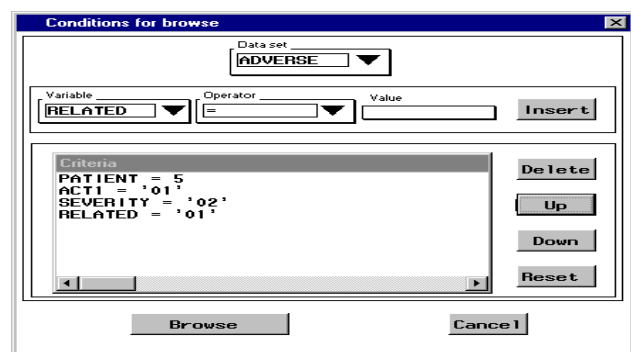
Figure 10. Example custom dialog

```
%Macro Product(system=,custom=,features=);
 %*** Specify libraries;
  Libname target "&system";

  Libname custom "&custom";

  Libname develop 'c:\develop';

  *** Compile custom specs;
  Proc build c=custom.custom batch;
    compile et=frame;

  *** Compile feature set;
  %If &features ne %then %do;
    Proc build c=develop.features batch;
      compile select=(&features);
  %End;

  *** Merge kernel, custom, and features into application  ***;
  Proc build c=target. batch;
    merge c=develop.kernel replace;
    merge c=custom.custom replace nosource noedit;
    %If &features ne %then %do;
      merge c=develop.features replace nosource noedit
          select=(&features);
    %End;
  Quit;
%Mend Product;
```

Listing 15. Example compiler

task while using comparatively little application real estate. The application contains combo boxes to select the data set, variable, and operator. The specified criteria are displayed in a scrollable list box. The action button in the lower left (labeled 'browse' this example) performs the desired action on the data reduced according to the criteria.

Step 3: Writing the Compiler

The application compiler is the SAS program that uses Proc Build to tie all the pieces together into the final application. The code in listing 15 shows an example of a compiler. This is a basic compiler that shows how to combine features from a feature set catalog, custom entries from a catalog that contains entries specific to an application, and a kernel catalog that contains common elements used by many applications.

A compile utility like this is useful for the developer to call and specify the location of the custom-developed features and pull reusable code by specifying a feature set from a shared feature library.

CONCLUSION

In this tutorial, we learned how to work in SAS/AF and develop subclasses of widgets and frames that can be integrated into a larger application development environment. These techniques provide the developer with the power necessary to meet the demands for rapid application development and swift deployment of applications. These techniques are also important for standardization. Designing and implementing an application development environment is necessary within the SAS/AF system and is useful for anyone who plans to develop a significant number of applications.

REFERENCES

Haske, Carl R. (1997), "Application Development Templates in SAS/AF®," *Proceedings of the Twenty-Second Annual SAS® Users Group International Conference,* 7-12.

Haske, Carl R. (1997), "Using SAS/AF® for Managing Clinical Data," *Proceedings of the Twenty-Second Annual SAS® Users Group International Conference,* 557-562.

SAS Institute, Inc. (1993), *SAS/AF Software: FRAME Entry, Usage and Reference, Version 6, First Edition,* Cary, NC: SAS Institute Inc.

SAS Institute, Inc. (1994), *SAS Screen Control Language: Reference, Version 6, Second Edition,* Cary, NC: SAS Institute Inc.

Stanley, Don (1994), *Beyond the Obvious with SAS® Screen Control Language*, Cary, NC: SAS Institute Inc.

Stanley, Don (1997), "Cursor Tracking in SAS/AF FRAME Applications," *Proceedings of the Twenty-Second Annual SAS® Users Group International Conference,* 148-153.

ACKNOWLEDGMENTS

AUTHOR'S ADDRESS

Carl R. Haske, Ph.D.
STATPROBE, Inc.
3885 Research Park Drive
Ann Arbor, Michigan 48108
(734) 769-5000 x115
E-mail: chaske@statprobe.com