

SAS/AF® Application Development Using Templates

Carl R. Haske, Ph.D., STATPROBE, Inc.

ABSTRACT

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

Standard templates 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. Templates also provide default behavior in an application. An application can completely override default behavior for any special application needs or perform tailored behavior and branch to the default behavior.

Templates are organized in SAS® catalogs. An application is developed quickly in SAS by assembling template entries that are stored in template catalog files. This paper describes different types of templates and methods for integrating generic templates and customized elements into a complete software package.

Introduction

The relatively new object-oriented features in the SAS/AF system provide a good framework for the development of applications in the SAS system. Version 6.11 provided additional classes that are very useful in database applications. This paper describes the concepts of object-oriented programming and classes in the SAS system. The first section illustrates the fundamental concepts of object-oriented programming and describes how these concepts are implemented in the SAS system. The second section shows how to derive a class and shows how to use the class in a simple application. The third section shows how to design an application development framework to allow quick prototyping and enable rapid development of the application front end. The final section shows how to logically organize templates that can be reused in multiple SAS/AF applications.

Object-Oriented Programming

In object-oriented programming, the emphasis is on objects, which are programming constructs that consist of data and actions performed on the data. Objects in a program or system are actually created as instances of classes. Classes provide the template for objects in an application. Therefore, object-oriented programming involves techniques that enhance the development of applications.

A SAS/AF application consists of a series of frames that interact with the user and perform actions based on user input. A frame consists of a set of objects that the user manipulates graphically in order to perform the application tasks. In an application, it is very common to use similar objects on different frames to perform actions specific to the frame. The example in the next section is an object that allows the user to select items from a list with the intent of performing an action on the selected items. For

example, an application component may provide the ability to select specific data sets from a list for printing, browsing, copying, or some other action. During the design phase of an application, objects and their purposes are defined. Objects of similar purpose should be organized into classes and act as the basis for a class library.

Classes in SAS/AF

The notion of a classes is a fundamental concept in object-oriented programming. A classes is a programming construct that embodies both data and methods. The data contain information about the class and the methods are procedures to access the data, modify the data, and send messages to other objects. SAS/AF includes several classes, including buttons, icons, list boxes, radio boxes, and graphic displays, with which to build a user interface for applications. The data for these classes are referred to as attributes and are stored in instance variables. Methods are the specific actions that occur on all the instance variables in the class. Subclasses are derived from a parent class and they inherit all of the parent's data and methods. The methods of the parent class can be overridden, replaced, or supplemented for use by the sub-class. In addition, new attributes and methods can be defined for sub-classes.

The steps for developing a sub-class in SAS/AF are as follows.

- Derive the sub-class
- Identify new instance variables
- Identify new methods and methods to be overridden
- Program SCL entries for new methods and overridden methods
- Design a custom attribute window to initialize instance variables
- Add the new class to the SAS/AF resource list

The focus of this paper does not involve the derivation of a sub-class. Taking Advantage of Inheritance in SAS/AF® Applications in the SUGI 21 proceedings by the author details the process of deriving a sub-class. Instead we discuss an example of a sub-class and how it is used in an application.

Example: Tiny Chooser Class

Figure 1 shows the graphic display of the *Tiny chooser* class. This class is used in many applications developed at STATPROBE. It is a child of the composite class that contains six objects.

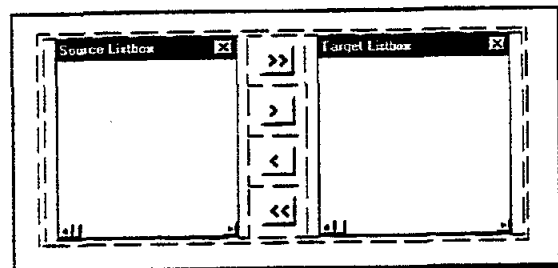


Figure 1. Tiny Chooser Class

Two of the objects are list boxes and four of the objects are controls. The class is referred to as *tiny* because there are other chooser classes in the class library that contain more control objects. This particular class contains a minimal number of

controls to move items between two list boxes. The function of the object is to provide the user with a list of items for selection. The tiny chooser is typically used in a frame by allowing the user to select certain items and then perform some type of action on the items.

Instance Variable	Position	Description
SOURCE	List box on the left	Contains items to select from
TARGET	List box on the right	Contains selected items
ADDALL	Top control	Moves all items from SOURCE to TARGET
ADD	Second control	Moves selected items from SOURCE to TARGET
REM	Third control	Moves selected items from TARGET to SOURCE
REALL	Bottom control	Moves all items from TARGET to SOURCE

Table 1. Tiny Chooser instance variables

Table 1 lists the new instance variables defined in the tiny chooser class. Each graphical object in the tiny chooser class has a corresponding instance variable. These instance variables comprise the data for a tiny chooser object.

When designing the tiny chooser class, it was necessary to consider the methods needed. The primary methods involve moving items between the SOURCE and TARGET object when one of the four control objects are selected. These methods should execute when any of the four controls ADDALL, ADD, REM, and REALL are selected. Also, if the mouse is double clicked on an item in a list, that item should move to the opposing list.

Method	Status	Description
MOVE_ALL	New	Moves all items from one list box to another list box
MOVE_SELECT	New	Moves selected items from one list box to another list box
REPOP_SOURCE	New	Forwards _REPOPULATE_ to SOURCE
REPOP_TARGET	New	Forwards _REPOPULATE_ to TARGET
SET_SOURCE_TITLE	New	Forwards _SET_TITLE_ to SOURCE
SET_TARGET_TITLE	New	Forwards _SET_TITLE_ to TARGET
_GET_TEXT_	Override	Forwards _GET_TEXT_ to TARGET
_OBJECT_LABEL_	Override	Performs specific actions when various objects are selected

Table 2. Tiny Chooser methods

The four controls of the tiny chooser class can be categorized to two types of controls performing different actions on a pair of lists. The two different actions are

- Move all items between lists
- Move selected items between the lists

Therefore, two primary methods are needed. These methods will accept two lists as parameters. The order that the lists are passed to the methods determines which direction to move list items. Additional methods are needed in order to repopulate list boxes and set titles. Table 2 shows the complete list of all new and over-ridden methods used in the tiny chooser class.

```

*** MOVE_ALL ***;
MOVEALL:
  Method src tar 8;
  call send(src, '_GET_MAXROW_', nrow);
  Do i=1 to nrow;
    call send(src,
              '_GET_TEXT_', i, itemtext);
    call send(tar, '_ADD_', itemtext, -1);
  End;
  call send(src, '_DELETE_ALL_');
Endmethod;

*** MOVE_SELECTIONS ***;
MOVESELS:
  Method src tar 8;
  call send(src, '_GET_NSELECT_', nsel);
  Do i=1 to nsel;
    call send(src, '_SELECTED_', i, row);
    call send(src,
              '_GET_TEXT_', row, itemtext);
    call send(tar, '_ADD_', itemtext, -1);
  End;
  Do i=1 to nsel;
    call send(src, '_DELETE_', row);
  End;
Endmethod;

```

Listing 1. MOVE_ALL and MOVE_SELECTION methods

The code for the MOVE_ALL and MOVE_SELECTIONS methods is displayed in listing 1. The numeric parameters of the methods represent object ids for two list boxes. The parameter src represents the list box in which items are selected to move to the list box represented by the parameter tar.

The method MOVE_ALL performs three basic steps. The step determines how many items are in the src list box by sending the _GET_MAXROW_ method. The second step executes a loop over the items in src and gets the text of each item and adds the text to the tar list box. The third step deletes all the items in src. The method MOVE_SELECTED performs the same three steps, except only the selected items are acted on.

Note that these two methods are usable by other chooser classes or in any setting where items are moved between list boxes. In other words, the actual functioning of the methods is independent of the overall intent of the class. This is a good example of object-oriented programming resulting in reusable code.

Listing 2 shows the code behind the _OBJECT_LABEL_ method. This method executes whenever any object contained in a tiny chooser object is selected. The first step of the method determines which object was selected by sending the _GET_CURRENT_WIDGET_ method. A select block then branches to the appropriate code based on the selected objects.

```

*** _OBJECT_LABEL_ ***;
OBJLABEL:
Method;
    call send(_self_,
              _GET_CURRENT_WIDGET_,
              obid);
    Select (obid);
    When (source)
        If _EVENT_='D' then
            call send(_self_,
                      'MOVE_SELECTIONS',
                      source,target);
    When (target)
        If _EVENT_='D' then
            call send(_self_,
                      'MOVE_SELECTIONS',
                      target,source);
    When (addall)
        call send(_self_,
                  'MOVE_ALL',
                  source,target);
    When (add)
        call send(_self_,
                  'MOVE_SELECTIONS',
                  source,target);
    When (rem)
        call send(_self_,
                  'MOVE_SELECTIONS',
                  target,source);
    When (remall)
        call send(_self_,
                  'MOVE_ALL',
                  target,source);
    Otherwise;
    End;
    call super(_self_,_METHOD_);
Endmethod;

```

Listing 2. _OBJECT_LABEL_ method

For both the SOURCE and TARGET objects, if a mouse double click is detected, then all selected items are moved from the list box to the other list using the MOVE_SELECTIONS method.

The remaining code blocks handle the cases where one of the controls was clicked with the mouse. In each case, the appropriate method, either MOVE_SELECTIONS or MOVE_ALL, is sent with SOURCE and TARGET passed according to the direction of item movement.

The final step executes the _OBJECT_LABEL_ method of the parent composite class.

Figure 2 shows a sample application that uses a tiny chooser. This application consists of a frame entry called *Chooser.frame* with a SCL entry called *Chooser.scl*. Table 3 shows the objects in *chooser.frame*. This application has no practical operation, it is used here to depict how the tiny chooser might be used in a real application. When the application starts, both list boxes are populated with text items. At any point while the application is running, when the "Get Selections" button is clicked, the items displayed in the "Target Items" list box are displayed in the "Selected Items" list box.

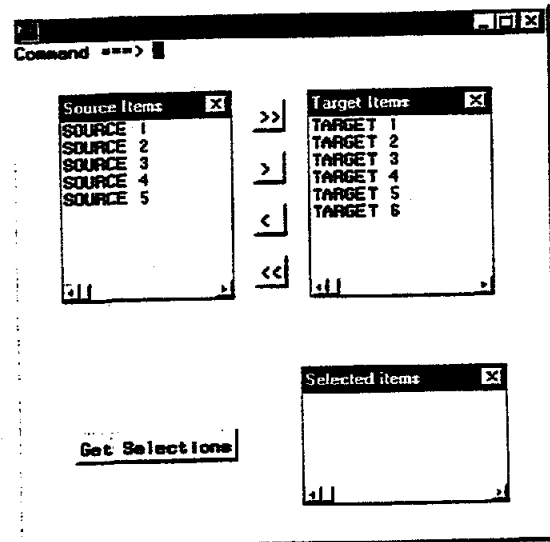


Figure 2. Sample application using the tiny chooser

Figure 3 shows the application after items in the lists have been moved and the "Get Selections" button has been clicked.

Object Name	Class	Description
Chooser	chooser	Tiny chooser
Getsels	pbutton	Push Button
Display	listbox	List box

Table 3. Objects in Chooser.frame

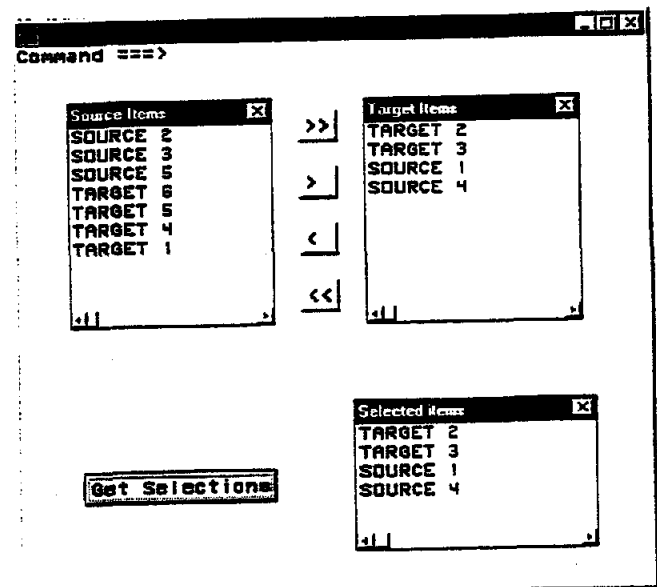


Figure 3. Running the sample application

Listing 3 shows the code behind the frame. The code is very simple and shows how easy it is to implement an object from the tiny chooser class.

The INIT section of the code initializes the items in both the SOURCE and TARGET list boxes. The GETSELS push button gets all the items in the TARGET list box and displays the items in the DISPLAY list box. In an actual application, the GETSELS block of code would be used to perform an action on all selected items. For example, an application may print or copy a set of selected data sets.

```
Length text $40;
```

```
INIT:
```

```
source=makelist();
rc=insertc(source,'SOURCE 1',-1);
rc=insertc(source,'SOURCE 2',-1);
rc=insertc(source,'SOURCE 3',-1);
rc=insertc(source,'SOURCE 4',-1);
rc=insertc(source,'SOURCE 5',-1);
```

```
target=makelist();
rc=insertc(target,'TARGET 1',-1);
rc=insertc(target,'TARGET 2',-1);
rc=insertc(target,'TARGET 3',-1);
rc=insertc(target,'TARGET 4',-1);
rc=insertc(target,'TARGET 5',-1);
rc=insertc(target,'TARGET 6',-1);
call notify('chooser',
            'SET_SOURCE_TITLE',
            'Source Items');
call notify('chooser',
            'SET_TARGET_TITLE',
            'Target Items');
```

```
Return;
```

```
GETSELS:
```

```
call notify('display','_DELETE_ALL_');
call notify('chooser',
            '_GET_MAXSEL_',nsl);
Do i=1 to nsl;
  call notify('chooser',
              '_GET_TEXT_',i,text);
  call notify('display',
              '_ADD_',text,-1);
```

```
End;
```

```
Return;
```

```
TERM:
```

```
rc=dellist(source);
rc=dellist(target);
```

```
Return;
```

Listing 3. Chooser.scl

Application Framework

An application framework consists of a set of tasks. Tasks are organized in a hierarchical manner. Higher level tasks can branch to additional sub-tasks. For example, an application task might be "Database Administration", however this task 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 sub-tasks or complete the execution of the task.

The natural structure for this task management is a tree (see diagram 1). 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 branches to as many as four levels in the task hierarchy.

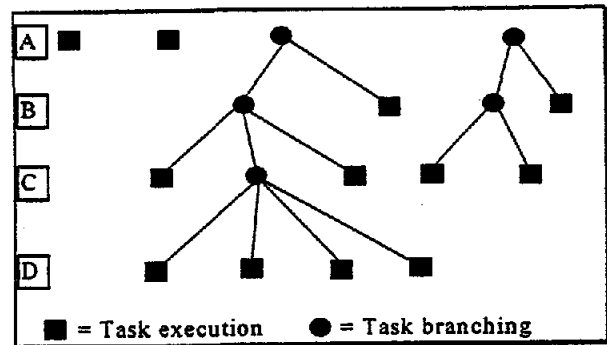


Diagram 1. Application Schematic

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. STATPROBE has developed a general system that models this process with a user interface commonly referred to as a "switchboard". Successive switchboards are represented by parameters in the system.

To realize this application structure in SAS/AF, STATPROBE developed a sub-class of the Image Icon class. The sub-class is referred 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. To complete an application, it is only necessary to program the frames for 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 4.

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

Table 4. Menu Image Icon Class instance variables

Designing an application front-end has been reduced to a matter of minutes. The developer can produced several prototypes and meet with the users to discuss the direction of the application before investing too much development time.

Another advantage of this framework arises during the application development process. There are many cases where the specific frame behind a Menu Image Icon object is general enough that the frame can be save as a template and reused in other applications.

Other generic functions are independent of application specifics. In our framework, all independent functions include

- Login system
- Change password
- Application node navigation
- Error log
- Quit to SAS
- Exit

Listing 4 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 application development. Main.scl has four primary sections consisting of

- Initialize DBFS
- Initialize system environment
- Login user
- Execute application

```
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 4. Main.scl

The standard library DBFS is included in all applications. The acronym DBFS stands for "Database File System". The DBFS consists of the user table, project table, assignments table, and any other specific files necessary to track data for an application.

The DBFS files typically reside in a sub-directory DBFS of the application directory. 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.

The method *environ* is invoked to set up the application environment. Listing 5 shows the environ method for a basic employee candidate survey application used by STAPROBE 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 5. 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 in 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 then environ method is invoked. For example, the application title is passed to the login frame to display the application name (see figure 7 in the next section). The login system requires a standard user table with fields representing user id, user name, and password.

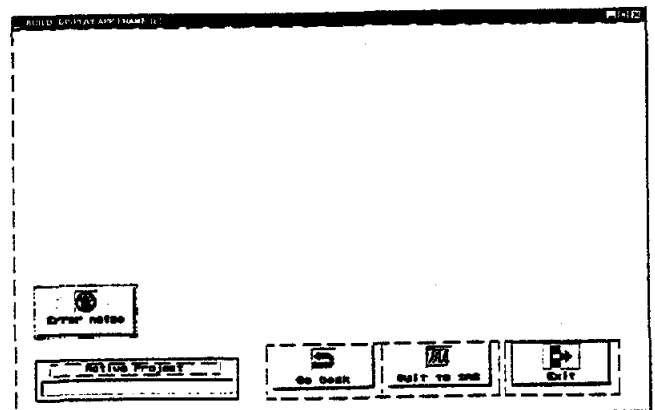


Figure 4. App.frame

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. This frame is a generic application node navigator. Figure 4 displays app.frame. There are basic controls for viewing error notes, going back along the node tree, quitting SAS, and exiting the application. All these controls are standard to every application. There is a large blank area above the control that is reserved to display switchboards.

An additional level of standard functions are typically included in STATPROBE's applications. These include

- User administration
- Project administration
- Project selection
- Active project display

Figure 5 shows the primary switchboard of an application.

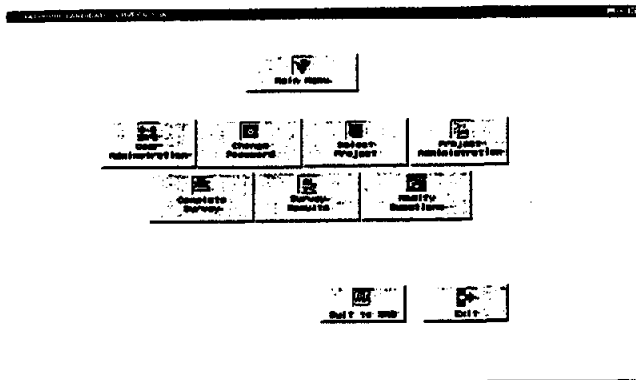


Figure 5. Primary switchboard

Figure 6 shows a second level switchboard. This switchboard displays after the "Survey Results" menu image icon object on the primary switchboard is selected.

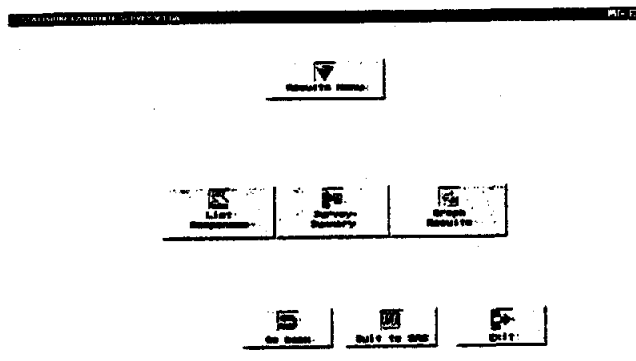


Figure 6. Second level switchboard

Reusable Templates

This section describes some of the template frames that are usable in all applications. The previous section described the login procedure. The login frame is displayed in figure 7. The code for the INIT section of the frame is displayed in listing 5. Notice that the application name is a parameter in the code and is used to set the title of the login frame. If the variable appname is blank, then the title of the login frame will simply be "Login". Each application uses a standard structure for the system user table. This allows the login frame to be used by all applications. Also, if the login frame is every 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 since it is usable in all applications. If the login frame fails to open the user table, the error handler is invoked. The error handler writes information to a list entry in the *errorlog* catalog for an application. It allows the developer to track user problems and debug the system.

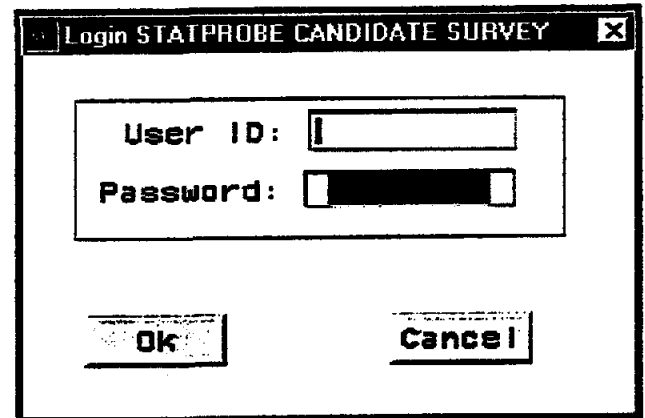


Figure 7. Login screen with title

```
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 7. INIT section of login.frame

Figure 8 shows the SELECTOR template. This template is typically called from a form editor to quickly locate a record and "pop" that record into the form. However, the selector can be used by procedure to select a data set record and perform an action on that record.

The data set that is being edited in the form is passed as a parameter to the selector frame. The selector frame displays the data set in a data table. When the user clicks on the desired record, the record number of the selected record is returned to the calling procedure. The calling procedure receives this

information and performs the action on the record. Listing 8 shows the code for the selector template.

GASUSER CLASS Record Selection					
	NAME	SEX	AGE	HEIGHT	WEIGHT
1	Alice	F	13	56.5	84
2	Becka	F	13	65.3	98
3	Gail	F	14	64.3	90
4	Karen	F	12	56.3	77
5	Kathy	F	12	59.8	84.5
6	Mary	F	15	66.5	112
7	Sandy	F	11	51.3	50.5
8	Sharon	F	15	62.5	112.5
9	Tammy	F	14	62.8	102.5
10	Alfred	M	14	69	112.5
11	Duke	M	14	63.5	102.5
12	Guido	M	15	67	133
13	James	M	12	57.3	83

Figure 8. Selector template

The INIT section of selector.scl establishes the data set for the data table. After any action occurs on the frame, the MAIN section gets the current row then halts immediately. This row number is returned to the calling procedure.

```
INIT:
  call send(_frame_,'_GET_WIDGET_',
    'tbl',tblid);
  call send(tblid,'_SET_DATASET_',
    dataset);
  If keyname^=_BLANK_ then do;
    vals=makelist();
    rc=insertc(vals,keyval,-1,keyname);
    call send(tblid,'_SET_KEY_',
      rc,keyname,'EQ','SCROLL',vals);
    rc=dellist(vals);
  End;
  If row^=. then
    call send(tblid,
      '_GOTO_ABSOLUTE_ROW_',row);

  call send(_frame_,
    '_SET_TITLE_',dataset||
    ' Record Selection');
Return;

MAIN:
  call send(tblid,
    '_GET_CURRENT_ROW_NUMBER_',row);
  _STATUS_='H';
Return;
```

Listing 8. Code for selector template

Conclusion

This paper presented the approach to object-oriented programming in SAS/AF. We discussed the use of classes, an application framework, and two templates that have been developed at STATPROBE. These techniques are powerful and provide a great deal of streamlining to the application development process. A typical application can be prototyped in

a couple of days and user can be involved in the software design phase to a greater degree. A basic application can be fully developed and test in less than one week. More complex applications typically take one month to two months to fully compose.

There are many templates that have been by STATPROBE, too numerous for the scope of this paper. In addition to the templates presented here, we have also developed

- Data entry form
- Relational entry form
- Informational dialog
- Question dialog
- Chooser
- Filter
- Browser
- Application builder

This library of templates took two months of development time. As STATPROBE develops more applications, our library of templates continues to grow.

REFERENCES

- 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.
- Haske, Carl R. (1995), *Using SAS/AF and Frame Entry to Access Data*, SUGI 20 Proceedings, Cary, NC SAS Institute Inc.
- Haske, Carl R. (1995), *Developing SAS/AF® Applications for Reviewing Clinical Data*, Proceedings of MWSUG '95, 5-9, Cleveland, OH.
- Haske, Carl R. (1995), *Taking Advantage of inheritance in Developing SAS/AF® Applications*, Proceedings of MWSUG '95, 32-37, Cleveland, OH.
- Haske, Carl R. (1996), *Taking Advantage of inheritance in Developing SAS/AF® Applications*, SUGI 21 Proceedings, Cary, NC SAS Institute Inc.
- Haske, Carl R. (1996), *A Clinical Data Management System in SAS®*, SUGI 21 Proceedings, Cary, NC SAS Institute Inc.
- Haske, Carl R. (1996), *Developing SAS/AF® Data Management Applications*, Proceedings of PharmaSUG '96, San Diego, CA.

ACKNOWLEDGMENTS

SAS and SAS/AF are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

AUTHORS ADDRESS

Carl R. Haske, Ph.D.
 STATPROBE, Inc.
 3885 Research Park Drive
 Ann Arbor, MI 48108
 (313) 769-5000 x115
 E-Mail:
 • 72700,1075@compuserve.com
 • statprob@oeonline.com