Paper 31

An Object-Oriented Approach to File Management within a SAS® AF Application

Charles W. Bininger,
Trilogy Consulting,
A Division of InfoTech Services, Inc.
Kalamazoo, Michigan

ABSTRACT

As applications become larger and more complicated, the thought of writing applications one line at a time seems like an overwhelming task. Even though SAS has provided us with an object-oriented language, very few of us take advantage of the power and reusability that SAS/AF provides. This paper is intended to introduce to the programmer the benefits of object-oriented programming as well as showing a practical application of such a design. Together we will design and implement a File Management Object that can be plugged into most any SAS/AF application with very little time and effort.

INTRODUCTION

As an application developer I find myself constantly writing code that saves the state of a document, project or model. It is the exception rather than the rule that an application does not save some user activity to disk. Before I started using an object-oriented approach to programming, I would locate similar code that I had written before and cut and paste it into my current application. Of course I would have to scour the code for pieces I didn't need and make additions for specific functionality that were not present. This was very time consuming. I thought there must be a better solution.

AN OBJECT-ORIENTED APPROACH

Object-oriented programming is the concept of designing objects that contain data and related procedures that act on that data. The methodology encourages packaging objects together to create an application, rather than writing the code line by line. Since each object is autonomous, objects become tested and proven over time. Objects that perform the same function and have the same signature can be interchanged without worry on how it will affect the application.

Consider an HVAC system. In its simplest form it is nothing more than a thermostat, furnace and an A/C unit. Together these objects perform the task of heating and cooling your home. However each piece has a specific function and may be interchanged with another object that behaves the same.

This division of labor is extremely important with large projects because it allows many programmers to work on an application simultaneously. It also allows the programmer to solve small manageable problems instead of being overwhelmed by large complicated ones.

Before we get to the File Management Object, Let us get some definitions out of the way.

- **Class** - The template or instructions that defines how the object should act and what it should store.
- **Object** – An instance of a class. Many objects can be created from a class.
- **Instance Variable** - A variable that contains information about the object. It may or may not influence behavior of the object.
- **Method** - A procedure within the object that allows communication with that object.
- **Encapsulation** – Data and procedures that act on that data are self-contained within the object.
- **Inheritance** - The concept that a child class will inherit all variables and methods from its parent class.

PREREQUISITE OF THE FILE MANAGEMENT OBJECT

The File Management Object must have the following functionality.

- **New** – Load default data
- **Open** – Get data from disk and load it
- **Save** – Save data to disk using default file name
- **Save_As** – Ask the user where to save data and save it there

LET'S LOOK AT THE CODE!

```
****************************************
**      FILE CLASS (abstract class)     **
**                                      **
** Written by Chuck Bininger            **
**            Trilogy Consulting        **
**                                      **
** This class should be used as a parent **
** for your storage class.  It will     **
** inherit all these file management    **
** capabilities.                        **
**                                      **
```

```
** INSTANCE VARIABLES - FILE_NAME       **
**                      SAVED           **
**                                      **
****************************************;

length member      $8
       dir         $200
       answer      $1
     ;

rc = rc;
_self_ = _self_;


_INIT_:   /* _INIT_ (override) */
  method  /* Create a file object     */
  ;
  call super(_self_,'_init_');
ENDMETHOD;

NEW:      /* NEW (new)                  */
  method  /* Load the default file list */
  ;
  call send(_self_,'check_saved');
  retrieved_l = makelist();
  list = 'sugi.demo.default.slist';
  call send(_self_,'load_object',list);
ENDMETHOD;

OPEN:     /* OPEN (new)                 */
  method; /* Open a file and populate the */
          /* file object                */

  rc = filedialog('OPEN',file_name,
                  '','','*.sc2');
  if rc >= 0 then do;
    call send(_self_,'check_saved');

    call send(_self_,'get_directory',
                     file_name,dir);
    call send(_self_,'get_member',
                     file_name,member);

    if (libname('saver',dir)) then
      _msg_ = sysmsg();
    list = 'saver.'||member||
           '.'||member||'.slist';
    call send(_self_,'load_object',list);
    rc = setnitemc(_self_,file_name,
                   'FILE_NAME');
  end;
ENDMETHOD;

LOAD_OBJ: /* LOAD_OBJECT (new private) */
  method  /* Load the object with the  */
          /* provided list             */
    in_list  $
  ;

    retrieved_l = makelist();
    rc = fillist ('catalog',in_list,
                  retrieved_l);

    do i = 1 to listlen(retrieved_l);
      instance_var = nameitem(retrieved_l,i);
```

```
      type = itemtype(retrieved_l,i);
      if type = 'C' then do;
        c_val = getitemc(retrieved_l,i);
        rc = setnitemc(_self_,c_val,
                       instance_var);
      end;
      else if type = 'N' then do;
        n_val = getitemn(retrieved_l,i);
        rc = setnitemn(_self_,n_val,
                       instance_var);
      end;
      else if type = 'L' then do;
        l_val = getiteml(retrieved_l,i);
        rc = setniteml(_self_,l_val,
                       instance_var);
      end;
    end;

    retrieved_l = dellist(retrieved_l);
ENDMETHOD;


SAVE:     /* SAVE (new)           */
  method  /* Save the file object */
  ;

  file_name = getnitemc(_self_,'FILE_NAME');
  if file_name ne _blank_ then do;

    rc = setnitemn(_self_,1,'SAVED');
    save_l = copylist (_self_, 'y');
    do i = listlen(save_l) to 1 by -1;
      name = nameitem(save_l,i);
      if name =: '_' or name='DESC' then do;
        rc = setlattr(save_l,'DELETE',i);
        rc = delnitem(save_l,name);
      end;
    end;

    call send(_self_,'get_directory',
                      file_name,dir);
    call send(_self_,'get_member',
                      file_name,member);

    if (libname('saver',dir)) then
      _msg_ = sysmsg();
    list = 'saver.'||member||
           '.'||member||'.slist';
    rc = savelist('catalog',list,save_l);
    rc = libname('saver','');
    save_l = dellist(save_l);
  end;
  else do;
    call send(_self_,'SAVE_AS');
  end;
ENDMETHOD;

SAVE_AS:   /* SAVE_AS (new)           */
  method;  /* Save the file object as */

  file_name = getnitemc(_self_,'FILE_NAME');
  if file_name ne _blank_ then do;
    call send(_self_,'get_directory',
                      file_name,dir);
    call send(_self_,'get_member',
                      file_name,member);
  end;
  else do;
    dir = '';
    member = '';
  end;

  rc = setnitemn(_self_,1,'SAVED');
  save_l = copylist (_self_, 'y');

  if filedialog('SAVEAS',file_name,member||
           '.sc2',dir,'*.sc2') >= 0 then do;
    rc = setnitemc(_self_,file_name,
                          'FILE_NAME');
    call send(_self_,'get_directory',
                      file_name,dir);
    call send(_self_,'get_member',
                      file_name,member);
    if (libname('saver',dir)) then
      _msg_ = sysmsg();
    list = 'saver.'||member||'.'||
           member||'.slist';
    rc = savelist('catalog',list,save_l);
    rc = libname('saver','');
  end;

  save_l = dellist(save_l);
ENDMETHOD;

GET_DIR:  /* GET_DIRECTORY (new)      */
  method  /* Parse the file name for  */
          /* the directory            */
    in_file_name
    out_dir      $
  ;
  out_dir = substr(in_file_name,1,
            sum(length(in_file_name),
            -(indexc(reverse(trim
            (in_file_name)),'/\'))));
ENDMETHOD;

GET_MEM:  /* GET_MEMBER (new)      */
  method  /* Parse the file name   */
          /* for the member        */
    in_file_name
    out_member      $
  ;
  out_member = scan(substr(in_file_name,
               sum(length(in_file_name),2,
               -(indexc(reverse(trim
               (in_file_name)),'/\')))),1,'.');
ENDMETHOD;

GET_FILE: /* GET_FILE_NAME (new)  */
  method  /* Get the file name    */
    out_file    $
  ;
  out_file = getnitemc(_self_,'FILE_NAME');
ENDMETHOD;

NEEDSAVE: /* NEEDS_TO_BE_SAVED (new) */
  method  /* Tell object it needs to */
          /* be saved                */
  ;
  rc = setnitemn(_self_,0,'SAVED');
```

```
ENDMETHOD;

CHECK_S:  /* CHECK_SAVED (new private) */
  method  /* If file needs to be saved */
          /* ask user                  */
  ;
  saved = getnitemn(_self_,'SAVED');
  if not saved then do;
    msg1 = 'File not saved!';
    msg2 = 'Would you like to save it?';
    call display('tools.file.dialog.frame',
                 msg1,msg2,answer);
    if answer = 'Y' then
      call send(_self_,'save_as');
  end;
ENDMETHOD;

_TERM_:   /* _TERM_ (overide)          */
  method; /* Terminate the file object */

  call send(_self_,'check_saved');
  call super(_self_,'_term_');
ENDMETHOD;
```

Notice that none of the code is application specific. What makes this class useful is that it can now be used as an abstract class. This means that this class will never be instantiated. This class will only be used as a parent to an applications storage class. The storage class will have application specific data and methods and also inherit the attributes of the File Management class.

A SAMPLE APPLICATION

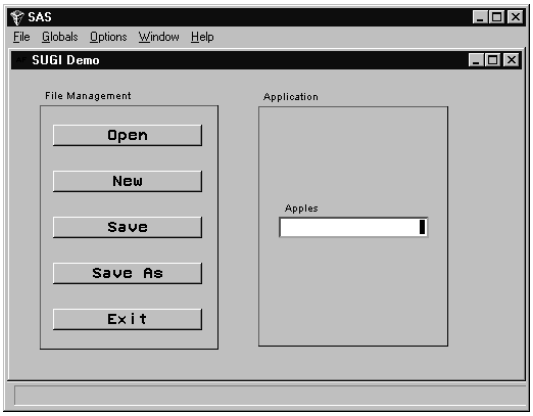
We can now create a simple application that has all the File Management functionality without ever changing proven code. Here is a simple application that does nothing more then store the number of apples in inventory.

The first step will be to create a subclass of our file management object. This subclass will have one instance variable called apples and two methods to get and set the values.

```
**************************************
**                                  **
**    Demo APP.CLASS for SUGI       **
**                                  **
**  This just hold one variable     **
**  called APPLE and allows GETTING **
**  and SETTING the value.          **
**                                  **
**  PARENT CLASS - FILE.CLASS       **
**                                  **
**  INSTANCE VARIABLE - APPLES      **
**                                  **
**************************************;

_self_ = _self_;
rc = rc;

_INIT_:   /* Method _INIT_ (overide) */
  method  /* Create the app object   */
  ;
  call super(_self_,'_init_');
ENDMETHOD;

G_APPLES: /* Method GET_APPLES        */
  method  /* Get the number of apples */
    out_value  8
  ;
  out_value = getnitemn(_self_,'APPLES');
ENDMETHOD;

S_APPLES: /* Method SET_APPLES        */
  method  /* Set the number of apples */
    in_value  8
  ;
  rc = setnitemn(_self_,in_value,'APPLES');
ENDMETHOD;

_TERM_:   /* Method _TERM_ (overide)  */
  method; /* Terminate the app object */

  call super(_self_,'_term_');
ENDMETHOD;
```

This subclass will now inherit all the functionality of the file management object.

Now for the user interface; the application itself needs only a widget (input field) and a method for trapping the user's command or instructions. This would usually be done with a menu, however for demonstration purposes I have used a series of push buttons.



```
****************************************
  **                                    **
  ** Demo Application showing the use of **
  **    the file management object      **
  **                                    **
  ****************************************;

length
  title  $80
  ;

INIT:
  ****************************************
  **    Load fruit object               **
  ****************************************;
  fruit_c =loadclass('sugi.demo.app.class');
  call send(fruit_c,'_new_',fruit_o);
RETURN;

APPLES:
  ****************************************
  ** if the apple changes then send the  **
  ** new value to the fruit object.  Now **
  ** tell the fruit object that it still **
  ** needs to be saved.                  **
  ****************************************;
  call notify('apples','_is_modified_',
                       apple_modified);
  if apple_modified then do;
    call send(fruit_o,'set_apples',apples);
    call send(fruit_o,'needs_to_be_saved');
  end;
RETURN;

MAIN:
  ****************************************
  ** Trap the command and send it       **
  ** to the fruit object                **
  ****************************************;
  command = compress(curfld());
  if command
    in('OPEN','NEW','SAVE','SAVE_AS')
    then do;
    call send(fruit_o,command);
    link UPDT_SCR;
  end;
  _msg_ = command;
  command = ' ';
RETURN;

UPDT_SCR:
  ****************************************
  **   Get the value of apple and       **
  **   file_name from the fruit object.  **
  ****************************************;
  call send(fruit_o,'get_apples', apples);
  call send(fruit_o,'get_file_name',title);
  if title = ' ' then
    title = 'SUGI Demo (New)';
  call notify('.','_set_title_',title);
RETURN;

EXIT:
  call execcmd('end');
RETURN;
```

```
TERM:
  ****************************************
  **  Terminate the fruit object        **
  ****************************************;
  call send(fruit_o,'_term_');
RETURN;
```

Notice the code does not deal with any file management issues other than sending a general instruction to the storage object. The storage object does not need to concern itself with how to save, just that it needs to be saved. This is equivalent to a thermostat sending a message to the furnace to start heating the house. The thermostat does not care how the furnace is heating the house.

Keep in mind that the File management Object was stripped down to bare essentials for the purposes of this paper. I tried to keep it simple to show the power of object-oriented design without dealing with details.

CONCLUSION

The File Management Class can be a very useful tool in creating applications. Once written to your specifications it can be used over and over again. This is just one reason applications should be written using an object-oriented approach. By using this paper as a guide many more abstract classes and tools can be built. This will allow developers to quickly create new applications and be confident that their code has been tested and proven over time.

REFERENCES

SAS Institute, Inc., *Object-Oriented Programming Using SAS/AF Software, Course Notes* , Cary, NC: SAS Institute Inc., 1996

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

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

Taylor, David A., Ph.D., *Object-Oriented Technology: A Manager's Guide*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1990

ABOUT THE AUTHOR

Chuck Bininger is a Senior Applications Developer with Trilogy Consulting. He has over 5 years experience in developing SAS/AF applications and has been using SAS for nearly 8 years.

ACKOWLEDGEMENTS

Special thanks to …

Jack Fuller Trilogy Consulting

The author may be contacted at:

Charles W. Bininger
Senior Applications Developer

Trilogy Consulting Corporation
5278 Lovers Lane
Kalamazoo, MI 49002
(616) 344-4100 voice
(616) 344-6849 fax
Internet: cwbining@trilogyusa.com