# Using Design Patterns to Implement Object-Oriented Menus in a SAS/AF® Application

Jack E. Fuller, Trilogy Consulting Corporation, Kalamazoo, MI

## ABSTRACT

One of the more frustrating aspects of designing a Graphical User Interface (GUI) for SAS/AF applications lies in the basic difficulty of reconciling object-oriented programming techniques with SAS® software's *pmenus*. Fundamental object-oriented concepts such as encapsulation and polymorphism are not built into *pmenus*.

This paper introduces design-patterns to construct a system that brings *pmenus* into the realm of object-oriented programming. The key concept is that while a frame still requires a traditional *pmenu* for menuing, all programming interactions are now routed through a corresponding non-visual object structure which mirrors the *pmenu* and provides the encapsulation and polymorphism which were previously missing from *pmenus*.

## INTRODUCTION

I have been asked the question, "Why spend the effort to build an object-oriented *pmenu*? It seems to be such a trivial thing." For some minor *pmenus* (*'OK'*, *'Cancel'*, *'Help'*), this is true; however, even simple *pmenus* can contain surprising levels of sophistication: e.g. until a file is actually opened, '*File* ➠ *Close'* should be grayed. This may seem a minor point, but since menus tend to be one of the primary means by which new users learn a system (i.e. a pedagogic vector (Cooper, p. 280)), a client will inevitably select it , crash the system, and cry 'BUGGGG!' As a programmer, one has a very limited pool of good will upon which to draw; don't spend it on these types of easily foreseen problems.

By incorporating a method of implementing *pmenus* at the very beginning of the application life cycle, one is forced *a priori* to determine the different states of an application's execution. The challenge lies in seamlessly integrating this insight with the rest of an application from the first keystroke. The menuing system presented here encompasses three major functions: first, a method for building a non-visual *pmenu* corollary; second, a method for linking a *pmenu* selection to the code which will be executed; third, a method for easily extending this system to the unique requirements of different applications.

## BUILDING THE MIRROR

### *A Box Inside a Box Inside a Box Inside…*

From a structural perspective, a *pmenu* can be thought of as simply a tree. For example, a partially constructed tree of SAS software's *build* window *pmenu* would look like as follows:

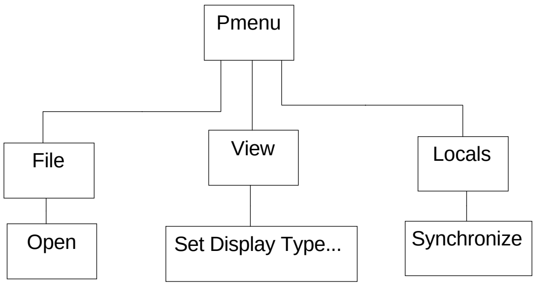


**Figure 1**

This structure, consisting of containers within containers, is an example of a generic design-pattern known as *Whole-Part* (Buschmann, et. al., pp. 225-242). The main thrust of *Whole-Part* is that an aggregation of parts can be treated together as one unit; from the outside, anyone looking at the *pmenu* structure represented in **figure 1** would see only the *Pmenu* component.

Additionally, this structure can be further refined to a special case of *Whole-Part* known as *Composite* (Gamma, et. al., pp. 163-173). The principle idea behind *Composite* is that it can also be recursively defined. This allows single nodes and collections of nodes to be handled uniformly. Therefore, class definitions used by the top level of a *pmenu* would also be used by sub-menus. The actual implementation for managing these different parts of a *pmenu* is encapsulated in the *Part_Mgr* class (see **Figure 2** for the system class diagram). The following *Add_Part* and *_Term_* methods give an idea of the flavor of this class.

**PART_MGR Class**

```
length _method_ name $200;

_TERM_:     /* _TERM_ method */
method
;
   part_l = getniteml (_self_, 'part_l');
   do index=1 to listlen (part_l);
      part_o = getitemn (part_l, index);
      call send (part_o, '_term_');
   end;

   call super (_self_, _method_);
endmethod;

ADD_PART:   /* ADD_PART method */
method
   in_id_o 8
;
   part_l = getniteml (_self_, 'part_l');
   dummy = insertn (part_l, in_id_o, -1,
      'part_o');
endmethod;
```

*Constructi`on Functions*

Now that we know how an previously constructed *pmenu* mirror should look, the problem becomes one of building the mirror. This process can be broken into two phases: first, creating an interface for specifying the *pmenu*; second, translating those specifications into our mirror.

Well, it turns out that SAS Institute has already provided a graphical tool for constructing a *pmenu* (run 'sashelp.aftools.pmbmain.frame'). Be sure both to create the *pmenu* and to export the specifications to an *slist*. This *slist* will be used to seed our construction process (as an aside), be sure to give them both the same name and to place them in the same catalog).

The construction process itself is handled by the *build_objects* method which is defined in the *Pmenu* class, but actually implemented by the, *Item_One*, and *Item_Two* classes. The reason behind splitting this operation between two separate classes is that the *slist* built by the *pmenu* builder is not recursively defined; this forces us to read the same data points in different formats depending upon our level in the *pmenu*. Due to space constraints, I am unable to include the code for actually building the mirror; however, the basic idea is simply to cycle through the *slist* and generate the structure as one chugs along.

While *Item_One* and *Item_Two* exist solely to help with building the mirror, *Main* has two additional responsibilities: first, it initiates the call to *build_objects* which translates the *slist*

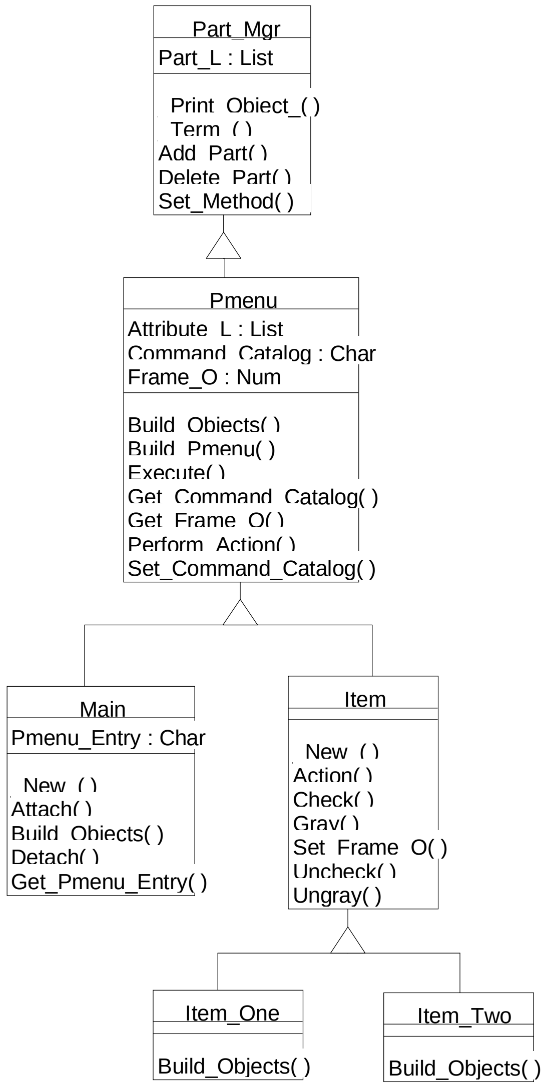


**Figure 2**

into an object structure; second, it links our non-visual *pmenu* mirror to a frame. Even though we have covered some of the class interactions included in the creation process (which can be quite involved), it is important to remember that from a client object's perspective, the creation process consists solely of instantiating an object of the *Main* class:

**A User's Perspective: Creation**

```
main_c = loadclass ('main.class');
call send (main_c, '_new_',
   pmenu_o, ., 'app.somefram.test');
```

EXECUTING THE CODE

*The Case For a Separate Attach Process*

Once a *pmenu* object has been instantiated, the next order of business is to implement a process whereby the *pmenu* can attached and detached from the frame. There are two major reasons for having mechanisms to attach and to detach: first, multiple *pmenu* objects can be instantiated and then swapped in and out, thus dynamically changing the frame's *pmenu*; second, the attach and detach methods allow us to treat the *pmenu* mirror as a component and simply hook it into the frame without having to write custom code for the individual frame.

**A User's Perspective: Attaching**

```
call send (pmenu_o, 'attach', _frame_)
```

**MAIN Class**

```
length _method_ command_catalog name text
   $200;

_NEW_:      /* _NEW_ method */
method
   in_pmenu_entry      $
         /* The THREE-Level entry */
      optional=
   in_frame_o          8
   in_command_catalog  $
   out_msg             $
   ;
   call super (_self_, _method_);

   /* Set the instance variables */
   dummy = setnitemc (_self_,
      in_pmenu_entry, 'pmenu_entry');
   dummy = setnitemn (_self_, in_frame_o,
      'frame_o');
   call send (_self_,
      'set_command_catalog',
       in_command_catalog);

   /* Retrieve the list */
   pmenu_l = makelist ();
   sysrc = fillist ('catalog',
      in_pmenu_entry || '.slist',
      pmenu_l);
   if (sysrc NE 0) then do;
      out_msg = sysmsg ();
      RETURN;
   end;

   /* Build the non-visual pmenu */
   call send (_self_, 'build_objects',
      pmenu_l);

   /* Initialize the object instance
      variables */
   call send (_self_, 'set_method',
      'set_command_catalog',
      in_command_catalog);
   call send (_self_, 'set_method',
      'set_frame_o', in_frame_o);

   /* Clean up */
   pmenu_l = dellist (pmenu_l, 'y');
endmethod;

ATTACH:     /* ATTACH method */
method
   in_frame_o          8
;
   /* First DETACH */
   call send (_self_, 'detach');

   /* Update the new FRAME_O */
   dummy = setnitemn (_self_, in_frame_o,
      'frame_o');
   call send (_self_, 'set_method',
      'set_frame_o', in_frame_o);

   /* Attach the pmenu to the frame */
   call send (_self_, 'get_pmenu_entry',
      text);
   if (text NE ' ') then do;
      call send (in_frame_o,
         '_set_pmenu_', text ||
         '.pmenu');
   end;

   /* Attach the methods to the frame */
   call send (in_frame_o,
      '_set_instance_method_',
      '_main_label_', searchpath
      ('frammeth.scl'), '_mainlb_',
      'before');
   call send (in_frame_o,
      '_set_instance_method_',
      'get_pmenu_o', searchpath
      ('frammeth.scl'), 'getpmnuo',
      'override');
   call send (in_frame_o,
      '_set_instance_method_',
      'set_pmenu_o', searchpath
      ('frammeth.scl'),
      'setpmnuo','override');

   /* Set the frame pmenu object */
   call send (in_frame_o, 'set_pmenu_o',
      _self_);
endmethod;

DETACH:     /* DETACH method */
method
;
   /* Find the current frame object */
   call send (_self_, 'get_frame_o',
      frame_o);
   if nmiss (frame_o) then do;
      RETURN;
   end;

   /* Detach the pmenu from the frame */
   call send (frame_o, '_set_pmenu_',
      ' ');

   /* Detach the methods from the
      frame */
   call send (frame_o, '_has_method_',
      'get_pmenu_o', has_method);
   if (has_method) then do;
      call send (frame_o,
         '_delete_instance_method_',
         '_main_label_', searchpath
         ('frammeth.scl'), '_mainlb_',
         'before');
      call send (frame_o,
         '_delete_instance_method_',
         'get_pmenu_o');
      call send (frame_o,
         '_delete_instance_method_',
         'set_pmenu_o');
   end;

   /* Remove FRAME_O from the pmenu
      object */
   dummy = setnitemn (_self_, .,
      'frame_o');
   call send (_self_, 'set_method',
      'set_frame_o', .);
endmethod;
```

```
GETPMNUE:    /* GET_PMENU_ENTRY method */
method
   out_text            $
;
   out_text = getnitemc (_self_,
      'pmenu_entry');
endmethod;
```

*Treating a Pmenu Mirror as a Component*

We are able to treat the mirror as a component which can in essence be dropped onto the frame because of the *_Set_Instance_Method_*s specified in the *Attach* and *Detach* methods. Since we do not want to interfere with any command trapping which may be performed by the frame itself, we will require that all *pmenu* commands be prefixed with an XXPMXX. Thus, a typical command might be XXPMXX NEW DO_WHATEVER, where XXPMXX is the trapping prefix, NEW is the caption of the component that will execute the command, and DO_WHATEVER is the command to execute (technical note: underscores are removed from the item name before passing it on to the *pmenu* object so that a match will be found for those items with embedded spaces in the item caption).

**FrameMeth.SCL**
**(used in _Set_Instance_Method_'s)**

```
length status switch item command word
   $200;

_self_ = _self_;
dummy  = dummy;

_MAINLB_:     /* _MAIN_LABEL_ (BEFORE)
   for the attaching frame */
method        /* -- Do NOT call super */
;
   call send (_self_, '_get_status_',
      status);
   if (status EQ 'K') then do;
      switch = word (1, 'u');
      /* Should this command be
         trapped */
      if (upcase (switch) EQ 'XXPMXX')
            then do;
         /* Parse the command */
         item = translate (word (2, 'u'),
            ' ', '_');
         do until (word EQ ' ');
            word = word (3, 'u');
            command = command || ' ' ||
               word;
            call nextword ();
         end;
         /* Send out the command */
         if (item NE ' ') and (command NE
               ' ') then do;
            call send (_self_,
               'get_pmenu_o', pmenu_o);
            if NOT nmiss (pmenu_o) then
                  do;
               call send (pmenu_o,
               'execute', item, command);
            end;
         end;
         call nextcmd ();
```

```
         end;
      end;
endmethod;

GETPMNUO:     /* GET_PMENU_O method */
method
   out_o       8
;
   out_o = getnitemn (_self_,
      'xx_pmenu_o_xx', 1, 1, .);
endmethod;

SETPMNUO:     /* SET_PMENU_O method */
method
   in_o        8
;
   dummy = setnitemn (_self_, in_o,
      'xx_pmenu_o_xx');
endmethod;
```

*Executing the Trapped Commands*

Commands can be executed in one of two ways: first, by specifying a command which is defined in the *Item* class ( *check*, *gray*, *uncheck*, and *ungray* are currently defined); second, by specifying a class in the command catalog which was optionally passed in to *Main*'s *_New_* method when the *pmenu* object was first created.

Since the first type of command is relatively straight forward, let's focus instead on the command catalog which implements the *Command* pattern (Gamma, et. al., pp. 233-242). The central idea behind this pattern is to encapsulate a request, thereby decoupling the sender of the command from the receiver of the command. The key to implementing this pattern is to implement the command class polymorphically by giving all of the command objects' *Execute* methods the same signature (usually by descending from a common parent class). Thus, any object which calls another object's *Execute* method does not have to know that object's concrete class: all *Execute* methods appear the same by exposing the same signature.

**A Typical Abstract Command Class**

```
EXECUTE:
 method
   in_pmenu_o        8
;

endmethod;
```

The instantiation, the execution, and the termination of these command objects takes place in the *Action* method of the *Item* class. This method is called (indirectly) when the *Execute* method of the *Pmenu* class determines that a request pertains to this node.

**Pmenu Class**

```
EXECUTE:      /* EXECUTE method */
method
   in_text       $
   in_command    $
      optional=
   io_done_f     8
;
   io_done_f = 0;

   /* Find the current text */
   text = getnitemc (getniteml (_self_,
      'attribute_l'), 'text', 1, 1, ' ');

   if (text NE ' ') then do;
      if (upcase (in_text) EQ upcase
            (text)) then do;
         call send (_self_,
            'perform_action',
            in_command);
         io_done_f = 1;
      end;
   end;

   /* Pass it on */
   if NOT (io_done_f) then do;;
      part_l = getniteml (_self_,
         'part_l');
      do index=1 to listlen (part_l)
            while (NOT io_done_f);
         part_o = getitemn (part_l,
            index);
         call send (part_o, _method_,
            in_text, in_command,
            io_done_f);
      end;
   end;
endmethod;

PERFORM:      /* PERFORM_ACTION method */
method
   in_command    $
;
   /* Parse the action */
   action = scan (in_command, 1, ' ');
   if (action NE ' ') then do;
      /* Parse the parameters */
      if (scan (in_command, 2, ' ') NE
            ' ') then do;
         len = length (action);
         start = verify (substr
            (in_command, len + 1), ' ');
         parms = substr (in_command, len
            + start);
      end;
      /* Perform the action */
      call send (_self_, 'action',
         action, parms);
   end;
endmethod;
```

**Item Class**

```
ACTION:     /* ACTION method */
method
   in_action      $
   in_parms       $
;
   /* Execute any defined methods */
   call send (_self_, '_has_method_',
      in_action, has_method);
   if (has_method) then do;
      call send (_self_, in_action,
         _parms);
      RETURN;
   end;

   /* Execute everything else */
   call send (_self_,
      'get_command_catalog',
      command_catalog);
   if (command_catalog NE ' ') then do;
      class_entry = command_catalog ||
         '.' || in_action || '.class';
      if cexist (class_entry) then do;
         command_c = loadclass
            (class_entry);
         call send (command_c, '_new_',
            command_o);
         call send (command_o, 'execute',
            in_parms);
         call send (command_o, '_term_');
      end;
   end;
endmethod;
```

## EXTENSIBILITY

### *Extending to a Specific Application*

Another application developer can use this tool as it is by very generating a *pmenu slist*, instantiating a *Main* object, and then attaching to a frame. At this point, the basic commands are available for use. The real power, however, becomes apparent when design patterns are used to their fullest potential (e.g. using the command catalog feature).

Moreover, there are other design-patterns which work quite well with this tool. For instance, it turns out that most frames will wind up having a limited number of *pmenu* graying/checking combinations which can occur. This is a perfect opportunity to implement a *Mediator* (Gamma, et. al., pp. 272-282) to control the interactions of the different *States* (Gamma, et. al., pp. 305-313) which can occur. In fact, one will often discover additional non-*pmenu* functionality which should be included in these *State* classes.

### *And For an Encore…*

Simple code enhancements which we have considered for this tool include the following: first, defining an *Execute* method in the *Pmenu* class which will work on all items (e.g. graying); second, defining an *Execute* method in the *Pmenu* class which checks for a match using some different criteria which includes the entire *pmenu* path (e.g. View→Zoom→Graph instead of just Graph); third, deriving a scheme to handle a *pmenu* caption which includes an ampersand (&).

More ambitious plans have centered around developing our own *pmenu* builder. This could then be used to change aspects of the *slist* used to seed the creation process. For instance, the generated *slist* could be designed in a truly recursive manner which would remove the need for separate *Item_One* and *Item_Two* classes since *build_objects* could then be migrated to the common parent *Item.*

## CONCLUSION

Remember that while the design of this *pmenu* system may at first glance seem to be quite complex, it derived from the experience of others. That is the beauty of a design pattern.

Remember that while the code which implements this *pmenu* can be rather obtuse, it is already written and only needs extension, NOT modification. That is the beauty of a reusable component.

## REFERENCES


Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal (1996) *Pattern-Oriented Software Architecture: A System of Patterns,* Chichester, West Sussex, England: John Wiley & Sons Ltd.

Cooper, Alan (1995), *About Face: The Essentials of User Interface Design,* Foster City, CA: IDG Books Worldwide, Inc..

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley.

SAS Institute Inc (1995*), SAS/AF Software: FRAME Class Dictionary, 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.

SAS Institute Inc (1990), *SAS Procedures Guide, Version 6, Third Edition*, Cary, NC: SAS Institute Inc.


## ACKNOWLEDGEMENTS


Thanks to Chuck Bininger, Kevin Brown, and John Ellis for contributing their insights to this paper.


For complete source code, please send an e-mail request to the author.



The author may be contacted at:


Jack E. Fuller
Lead Software Engineer
Trilogy Consulting Corporation
5278 Lovers Lane, Kalamazoo, MI 49002
jefuller@sprynet.com