The Mediator Pattern Can Organize an Object-Oriented Application

Kevin Tyler Brown, Keane, Inc., Grand Rapids, MI

Abstract

The Mediator Pattern[1] (Gamma, Helm, Johnson, Vlissides, 1995) allows programs to organize the relationships and interactions between widgets, frames, non-visual objects and events in a controller class specifically tailored for an application. This idea can be used in any object-oriented programming language such as SAS/AF®, Java® or C++. This paper will discuss creating an abstract mediator class that will handle the mundane chores such as: connecting to the application framework, setting up dynamic delegation, and initializing the application as well as an object-oriented pmenu system[2] (Fuller, 1997). The subclass implementation of a concrete mediator will illustrate specific logic to 'mediate' the interactions between an application class, a frame class with several widgets, a data wrapper class as well as several other non-visual component classes. To close, this paper will discuss the larger application design, the benefits of a mediator as well as some inherent problems that can occur if your mediator has become a 'god' class[3] (Riel, 1996).

Introduction

Connecting and coordinating the relationships of object hierarchies is the basis of object-oriented programming. These software design patterns can organize object relationships between both visual and non-visual objects. One pattern that is extremely useful is the Mediator Pattern. The mediator decouples objects from making direct references to each other and lets them interact and grow independently. It concentrates the explicit references and logical dependencies together into one object. This protects objects from becoming too inter-dependent and promotes reusability.

Application Background Information

Before getting into the class definitions of the mediator, some background application information is necessary. This program has an *application object* to handle several issues: single-point access to information, operating system interfacing and application-level information. The *application object* needs to be carefully monitored that it doesn't become a catch-all for all functionality that was lost in the original design or that it becomes too powerful, handling everything for everybody. When this happens, the whole application design needs to be re-visited.

The *application object* is derived from a Singleton Class[4] (Brown, 1996). This allows access from anywhere in the application to a single object instance. The *application object* will have the following functionality: a *dynamic delegation* method, connection to a data object reference, a stacked mediator reference.

The *dynamic delegation* method allows delegation to be done at run-time. It permits an object to create a dynamic relationship that propagates its methods down a chain. This will be necessary to allow widgets to connect to a mediator as well as having the mediator connect to the application model.

This application also delegates to a *data object.* A *data object* is a wrapper to a dataset. It encapsulates information about a dataset such as the current where clause conditions as well as the need for refreshing.

A stacked mediator reference means that the *application object* can support multiple mediators, setting the one on top as the active mediator. In this example, there is only one mediator.

Although the *application object* and the *data object* are quite interesting, their scope size does not allow further discussion in this paper. Their general idea is all that is required at this time.

Creating the Abstract Mediator

Creating an abstract mediator doesn't seem to make a lot of sense, since by definition a mediator contains mainly application specific information and a abstract class is usually created for generalized functionality. This is true, but there were tasks general enough that all mediators could use them. A parent abstract class was created to reuse as much of the mediator class as possible. These tasks included:

- Connecting to the application object
- Setup of the pmenu system
- Connecting to a frame entry
- Connecting of all widgets
- Setup of the refresh flag
- Creating a state machine
- Dynamic delegation between the mediator and the widgets
- Handling events

The code for the abstract mediator class follows:

```
/* _NEW_ method Creation of the mediator class */
_NEW_:
   method
      in_frame_o     8
      optional=
      pmenu_entry    $
      command_entry  $;
   call super (_self_, _method_);
   /* Singleton Application object */
   app_c = loadclass('app.system.app.class');
   call send (app_c,'_new_', app_o);
   /* Push ourselves on the stack */
   call send ( app_o,'SET_CURRENT_MEDIATOR',_self_);
   call send (app_o, '_TERM_');

   /* Set up if frame has pmenus */
   if pmenu_entry ne '' then do;
      pmenu_c = loadclass ('tools.pmenu.main.class');
      if command_entry ne '' then
         call send (pmenu_c,'_NEW_',pmenu_o,.,
                                    pmenu_entry,
                                    in_frame_o,
                              command_entry,_msg_);
      else
         call send (pmenu_c,'_NEW_',pmenu_o,.,
                                    pmenu_entry,
                                    in_frame_o);

      call send (_self_,'SET_PMENU',pmenu_o);
      call send (pmenu_o,'ATTACH',in_frame_o);
   end;

   /* Add frame to delegate to the mediator as well */
   call send ( _SELF_, 'ADD_DELEGATE', in_frame_o,
               _self_);

   /* Keep the frame object reference that mediator
      links to */
   call send ( _self_, 'SET_FRAME',in_frame_o);

   /* Set all widgets to delagate to the mediator */
   call send (_self_,'SET_WIDGET',in_frame_o);

   /* This initalizes the mediator */
   call send (_self_,'INITIALIZE_STATE');
endmethod;
```

In the NEW method, the abstract mediator sets up the active mediator in the *application object*, the object-oriented pmenu system, delegation from the frame to the mediator, delegation from the widgets to the mediator and initializing the state machine, internal to the mediator.

```
_TERM_:      /* _TERM_ method */
   method;
  call send (_self_,'GET_PMENU',pmenu_o);
  if listlen(pmenu_o) ne -1  then
     call send (pmenu_o,'_term_');

  call send (_SELF_,'REMOVE_MEDIATOR',_self_);

  /* Clean up the state machine */
  call send (_self_, 'get_state', state_o);
  if (0 LT listlen (state_o)) then do;
     call send (state_o, '_term_');
  end;

  call super (_self_, '_term_');
endmethod;
/* Register widgets for use in the meditator */
SETWID:
   method
     in_frame_o 8;
/* Get the entry from the class this object is of */
   call send( self_,'_GET_METHOD_','_OBJECT_LABEL_'
                    ,info_l);
   entry = getnitemc(info_l,'ENTRY');

   widgets = makelist();
   call send (in_frame_o,'_GET_WIDGETS_',widgets);
   do i = 1 to listlen(widgets);
      name = nameitem(widgets);
      widget = popl(widgets);

      rc = insertn(_self_,widget,-1,name);
/* Dynamic delegation from widgets to the mediator */
      call send ( _SELF_, 'ADD_DELEGATE',widget,
                  _self_);
/* If the widget has a widget refresh method */
/* then setup a WREFRESH event to call the WREFRESH
   method */
      call send ( widget, '_HAS_METHOD_','WREFRESH',
                  status);
      if status then
         call send (_self_,'_SET_EVENT_HANDLER_',
                    _self_,'WREFRESH','WREFRESH',
                    widget);
/* Override the _Object_Label_ to add a change event */
      call send ( widget, '_SET_INSTANCE_METHOD_',
                          '_OBJECT_LABEL_', entry,
                          'OBJLBL','AFTER');
/* Here is the change event response */
      call send (_self_,'_SET_EVENT_HANDLER_',widget,
                        'WIDGET_CHANGED','WIDGET_CHANGED',
                        _self_);
   end;
   rc = dellist(widgets,'Y');
   rc = dellist(info_l,'Y');
endmethod;
```

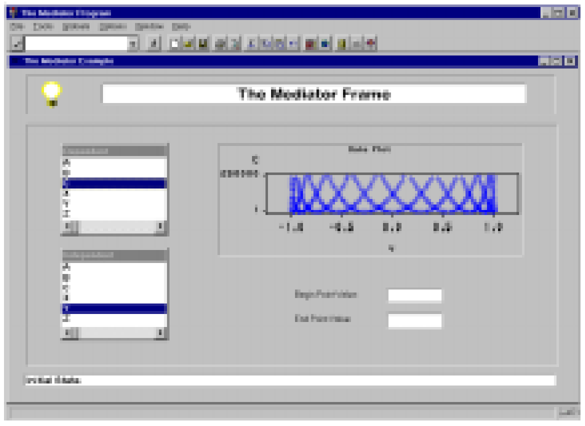
SET_WID method plays a key role in the mediators communication interactions. The use of several methods are used to enable communication and functionality. *Dynamic delegation* is first used to set up a link between the widgets and the mediator. Next, a widget refresh event allows the mediator a route back to the widgets. Lastly, _Set_Instance_Method_ is used to override the widgets _Object_Label_. This is done so that the widgets don’t need to be subclassed in order to let the mediator know when something has changed. The new _Object_Label_ will send a WIDGET_CHANGED event, which triggers the mediator into action.

These methods set up the basic tools that will allow the concrete mediator to perform its object navigation as well as to control application specific interactions.

The Pmenu System is an object-oriented approach to handle commands from the command line as well as from the pmenu. They set up the use of a command catalog, which is an object-oriented paradigm that permits added functionality to the application without changing a bit of code. The set up is here, but is not discussed further.

Creating a Frame Interface

Currently, the application has a *data object*, an *application object*, and an abstract mediator. Before a concrete mediator can be created, some objects, which need mediation, need to be created. A frame is constructed with two list boxes, a graphics object and two label widgets. The screen will look as follows:



All other widgets are incidental. The important widgets are the named ‘GRAPH’, the two list boxes, named ‘LST_D’, and ‘LST_I’, respectfully, and the two text labels below the graph, named ‘TXT_B’ and ’TXT_E’.

The main function of this example will be to display a dataset, graphically, allowing the user to select the dependent and independent numeric variables from the list boxes. The user will also have the ability to select two points on the graph, which will be displayed in the text boxes, and allow the graph to zoom in on that area.

The list boxes and text boxes are straight widgets that have not been subclassed. The graph has to incorporate several enhancements. These enhancements include: process a sub-menu that displays the state of the graph, keep track of the two selected points, refresh using information available from the mediator. The display of the graph states are ‘Zoom Start’, ‘Zoom End’ and ‘Cancel Zoom’. The choices can be selected by clicking the right mouse button and selecting from a menu. The code for the graph is as follows:

```
 _INIT_:
   method;
   call super (_self_,_method_);
   call send (_self_,'SET_GRAPH_STATE',"A");
endmethod;

WREFRESH:
   method
   optional=
   settings_l 8;
      call send (_SELF_,'GET_DATASET_ID',datadsid);
      if datadsid ne 0 and datadsid ne _BLANK_ then do;
         call send (_SELF_,'GET_IVAR',indep_var);
         call send (_SELF_,'GET_DVAR',dep_var);

         /* Find the mean to set up reference lines */
         ignored = varstat (datadsid, indep_var, 'mean', mean);
         ref_l = makelist ();
         dummy = insertn (ref_l, mean, -1);
         call send (_self_, '_set_ref_lines_', 'vaxis', ref_l);
         call send (_self_, '_set_color_', 'vrefline', 'magenta',
1);
         call send (_self_, '_set_style_', 'vrefline', 1,
'dashed');
         ignored = dellist (ref_l, 'y');

         /* Setup the graph */
         call send (_self_, '_set_dsid_',     datadsid);
         call send (_self_, '_set_indep_var_',  indep_var);
         call send (_self_, '_set_dep_var_',    dep_var);
         call send (_self_, '_set_title_', 1,'Data Plot');
         call send (_self_, '_need_refresh_');
     end;
     call send (_self_,'_UPDATE_');
endmethod;
```

The WREFRESH method makes use of the *dynamic delegation* to the mediator. The calls are indicated by the ‘_SELF_’ in upper case. They demonstrate the call back to the mediator to acquire the appropriate information.

```
_LABEL_:
   method;
   if (_event_ in ('P')) then do;
      /* Find the currently select values */
      call send (_self_, '_get_value_', value_l,status);

      /* Check to see if there is a point selected */
      if (listlen (value_l) GT 0) then do;
         call send (_self_, 'GET_GRAPH_STATE',state);
         call send (_self_, 'SET_' || state, pop_l);
         call send (_self_, '_popup_', pop_l, select);
         call send (_self_, 'ACTION_' || state, state, select );
         /* Set the new state */
         call send (_self_,'SET_GRAPH_STATE',state);
         call send (_self_,'SELECT_' || compress(put(select,
best9.)) );
         pop_l = clearlist (pop_l);
      end;
   end;
   call super(_self_,_method_);
endmethod;
```

The label method handles the capture of the right mouse click. Some method mashing is done to create a method name that includes the current state. This is a bit easier than subclasses three graphs and then delegating to them.

```
/* The following methods handle the state of the graph.
  This graph has three states.
  A - All available
  B - Zoom started, looking for zoom end or cancel zoom
  C - Zoom mode, looking for zoom cancel */
ACTION_:
   method
   in_state $
   list_str $;
endmethod;

ACTION_A:
   method
   in_state $
   list_x 8 ;
   select (list_x);
      when (1)         in_state = "B";   /* "Zoom Start" */
      otherwise;
   end;
endmethod;

ACTION_B:
   method
   in_state $
   list_x 8 ;
   select (list_x);
      when (2)         in_state = "C";  /* "Zoom End" */
      when (3)         in_state = "A";  /* "Cancel Zoom" */
      otherwise;
   end;
endmethod;

ACTION_C:
   method
   in_state $
   list_x 8 ;
   select (list_x);
      when (3)         in_state = "A";  /* "Cancel Zoom" */
      otherwise;
   end;
endmethod;

GET_LIST:
   method
      out_menu_l 8;
   if listlen(out_menu_l) = -1 then
      out_menu_l = makelist();
   else
      ignore = clearlist(out_menu_l,'Y');

   dummy = insertc (out_menu_l,"Zoom Start",-1);  /* 1 */
   dummy = insertc (out_menu_l,"Zoom End",-1);    /* 2 */
   dummy = insertc (out_menu_l,"Cancel Zoom",-1); /* 3 */
endmethod;

SET_A:
   method
      out_menu_l 8;
   call send (_self_,"GET_LIST", out_menu_l);
   rc = setlattr( out_menu_l,'ACTIVE',1);
   rc = setlattr( out_menu_l,'INACTIVE',2);
   rc = setlattr( out_menu_l,'INACTIVE',3);
endmethod;

SET_B:
   method
      out_menu_l 8 ;
   call send (_self_,"GET_LIST", out_menu_l);
   rc = setlattr( out_menu_l,'INACTIVE',1);
   rc = setlattr( out_menu_l,'ACTIVE',2);
   rc = setlattr( out_menu_l,'ACTIVE',3);
endmethod;

SET_C:
   method
      out_menu_l 8;
   call send (_self_,"GET_LIST", out_menu_l);
   rc = setlattr( out_menu_l,'INACTIVE',1);
   rc = setlattr( out_menu_l,'INACTIVE',2);
   rc = setlattr( out_menu_l,'ACTIVE',3);
endmethod;

SELECT_0:
   method;
endmethod;

SELECT_1:   /* Start Zoom */
   method ;
   call send (_self_,'GET_INDEP_VAL',val);
   call send (_self_,'SET_BEGIN_POINT',val);
endmethod;

SELECT_2:   /* End Zoom */
   method   ;
   call send (_self_,'GET_INDEP_VAL',val);
   call send (_self_,'SET_END_POINT',val);
endmethod;

SELECT_3:  /* Cancel Zoom */
   method ;
   call send (_self_,'SET_BEGIN_POINT',' ');
   call send (_self_,'SET_END_POINT',' ');
endmethod;
```

The SELECT_, SET_ and ACTION_ are all driven by the state of the graph. They perform; the display of the menu, the setting of the zoom points as well as the logic between them.

Creating the Concrete Mediator

Now that a frame entry has been created and populated with widgets, a concrete mediator can be set up. The concrete mediator will handle the following: the zoom and unzoom requests, the selection of the independent and dependent variables, and all of the interactions between widgets, the data object and the application.

The code follows:

```
REFRESH:
   method;
   call send (_self_,'GET_REFRESH_STATE',need_refresh);
   if need_refresh = 1 then do;
      call send (_self_,'GET_STATE',state_o);
      call send (state_o,'DISPLAY');
      call send (_self_,'SETUP_WIDGETS');
      call send (_self_,'BUILD_WHERE');
      call send ( _SELF_, 'PROCESS_WHERE');
      call send ( _self_, '_SEND_EVENT_','WREFRESH',
                  settings_l);
   end;
   call send (_self_,'NEED_REFRESH',0);
   call super (_self_, _method_);
endmethod;
```

REFRESH gets the current state of the mediator, updates the pmenus, re-builds the where clause, informs the data object to process the where clauses if necessary, coordinates the widgets and send out a widget refresh event.

```
I_STATE:
   method;
   call super(_self_,_method_);
   call send (_self_,'NEXT_STATE','INITIAL');
   call send(_SELF_,'GET_NUMERIC_VARS',var_l);
   call send (_self_,'GET_FRAME',frame_o);
   call send (frame_o,'_GET_WIDGET_','LST_I',wid1_o);
   call send (frame_o,'_GET_WIDGET_','LST_D',wid2_o);
   do i = 1 to listlen(var_l);
      call send(wid1_o,'_ADD_',getitemc(var_l,i),-1);
      call send(wid2_o,'_ADD_',getitemc(var_l,i),-1);
   end;
   call send(_SELF_,'GET_DVAR',dvar);
   call send(wid1_o,'_select_text_',dvar);
   call send(_SELF_,'GET_IVAR',ivar);
   call send(wid2_o,'_select_text_',ivar);
endmethod;

N_STATE:
   method
   in_statename $;
   call send (_self_, 'get_state_catalog', state_cat);
   if (state_cat NE ' ') then do;
      state_entry = state_cat || '.' || in_statename || '.class';
      if cexist (state_entry) then do;
         state_c = loadclass (state_entry);
         call send (state_c, '_new_', state_o,.,_self_);
         call send (state_o, 'PREVALIDATE',preval_rc,msg);
         /* Perform the actions */
         if (preval_rc) then do;
            call send (state_o, 'ACTIONS', act_rc, msg);
         end;
         /* Set the new state */
         if (preval_rc) and (act_rc) then do;
            call send (_self_,'SET_STATE',state_o);
         end;
         else do;
            call send (state_o, '_term_');
            call send (_self_, 'set_state', .);
         end;
      end;
      call send (_self_,'GET_STATE',state_o);
      if (0 < listlen(state_o)) then do;
         call send (state_o,'DISPLAY');
         call send (_self_,'SET_STATUS_MSG',msg);
      end;
   end;
endmethod;
```

I_STATE and N_STATE handle the changing of the mediator state, which we don't use in this example.

```
BLD_WHR:
   method;
   call send (_self_,'GET_FRAME',frame_o);
   call send (frame_o,'_GET_WIDGET_',"GRAPH",wid_o);
   call send (wid_o,'GET_GRAPH_STATE',g_state);
   select (g_state);
   when ('A') do; /* Waiting for a zoom to happen */
         av_zoom_begin_o = listlen(av_zoom_begin_o);
         av_zoom_end_o = listlen(av_zoom_end_o);
         call send (_SELF_,'CLEAR_WHERE');
      end;
   when ('B') do; /* Waiting for a end zoom to happen */
         av_zoom_begin_o = listlen(av_zoom_begin_o);
         av_zoom_end_o = listlen(av_zoom_end_o);
         call send (_SELF_,'CLEAR_WHERE');
      end;
   when ('C') do;
   /* Zoom has occurred and now waiting for a cancel zoom */
      call send (_self_,'GET_IVAR',ivarname);
      call send (wid_o,'GET_BEGIN_POINT',val);
      call send (_SELF_,'SET_COMPONENT',
               ivarname || " ge " || val ,
               "AND",av_zoom_begin_o);
      call send (wid_o,'GET_END_POINT',val);
      call send (_SELF_,'SET_COMPONENT',
                 ivarname || " le " || val ,
                 "AND",av_zoom_end_o);
      call send (_SELF_,'SET_DATA_REFRESH');
      end;
   end;
endmethod;
```

The BLD_WHR method queries the graph for its current state and sets up the *data object* where clause depending on that state. State 'C' indicates that the graph has acquired two points and is now in zoom mode.

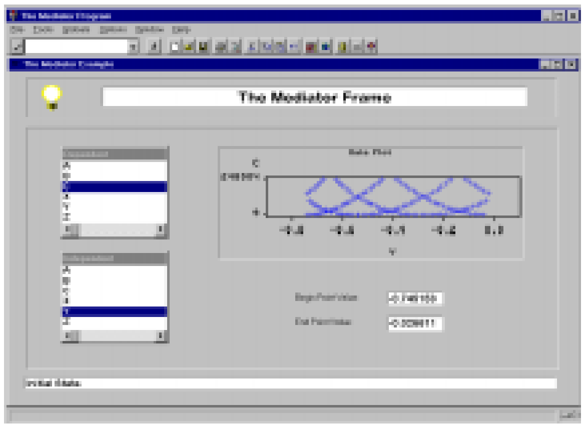
```
W_CHANGE:
   method
   in_who 8;
   wid_name = getnitemc(in_who,'NAME');
   call send (_self_,'GET_FRAME',frame_o);
   call send (frame_o,'_GET_WIDGET_',wid_name,wid_o);

   select (wid_name);
      when('LST_I') do;
         call send(wid_o,'_get_last_sel_',row,issel,text);
         if issel then do;
            call send(_self_,'SET_IVAR',text);
         end;
      end;
      when('LST_D') do;
         call send(wid_o,'_get_last_sel_',row,issel,text);
         if issel then
            call send(_self_,'SET_DVAR',text);
      end;
      when('GRAPH') do;
         call send (wid_o,'GET_BEGIN_POINT',val);
         call send (frame_o,'_GET_WIDGET_','TXT_B',wid2_o);
         call send (wid2_o,'_SET_TEXT_',val);
         call send (wid_o,'GET_END_POINT',val);
         call send (frame_o,'_GET_WIDGET_','TXT_E',wid2_o);
         call send (wid2_o,'_SET_TEXT_',val);
      end;
      otherwise;
   end;
   call send(_self_,'NEED_REFRESH');
endmethod;
```

The W_CHANGE method handles updating of the widgets. It is the method called when objects have changed value or state and need to inform the mediator.

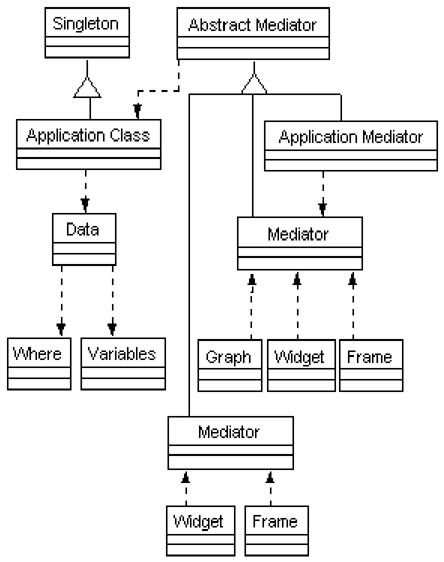
```
OBJLBL:
   method
   ;
   call send(_self_,'_SEND_EVENT_','WIDGET_CHANGED',_self_);
endmethod;
```

This is the over-written _OBJECT_LABEL_ for the widgets. It sends a 'WIDGET_CHANGED' event that the mediator responds too.

Here is the application screen in zoom mode:



The System Architecture



Conclusions

In this example, the interactions were minimal and only between a couple widgets. A mediator can minimize class dependencies as well as isolating the program logic. The specific object references are limited to within the mediator. A collaborator can grow independently of its role in the application without ties that would trap its reuse.

In the architecture diagram design above, there is an Application Mediator that coordinates the frame entries. In the example presented in this paper, the *application object* performed these responsibilities. It may be observed that the mediator tends to take some of the power away from the inherent abilities of the frame. This is true. In SAS/AF the frame entry takes on a lot of responsibilities. The mediator shoulders that responsibility freeing the frame entry to evolve independently of the application logic.

There have been several problems that have plagued the idea of implementing a mediator pattern in SAS/AF. These problems involve: the inability to subclass the widget class, by-passing the frame entry's control over its widgets, and the controlling and setting of events in the event handler. The mediator in this example used several methods to control and communicate to its collaborators. These methods encompass the following: dynamic delegation, the use of the _SET_INSTANCE_METHOD_ to override the _OBJECT_LABEL_, and events. These methods allowed the mediator to accomplish its tasks.

The advantages of a mediator are quite clear, but pitfalls are present. One of the largest, the 'god' class heuristic, can easily happen to a mediator, and any other classes. This can occur when one class does the majority of the work, leaving the rest of the classes out to dry, assigned only menial chores. Care must be used to restrain from changing the original intent of a mediator and falling prey to the 'god' class syndrome. This can be done by monitoring the interactions within the mediator, making sure that it is only collaborating the objects and not becoming part of them. Keep as much of the information in the widgets or non-visual objects as possible, only querying them if needed. This will separate the mediator, allow it to make decisions, and letting other objects do their work.

REFERENCES

[1]Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, (1995), *Design Patterns, Elements of Reusable Object-Oriented Software*, Reading MA: Addison Wesley Publishing Company, 273-282.

[2]Fuller, Jack E. (1997), "Using Design Patterns to Implement Object-Oriented Menus in a SAS/AF Application", *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*, Cary, NC:SAS Institute Inc.

[3]Riel, Arthur J., (1996), *Object-Oriented Design Heuristics*, Reading MA: Addison Wesley Publishing Company, 32-36

[4]Brown, Kevin T. (1996), "The Singleton Class in Screen Control Language", *Proceedings of the Twenty-First Annual SAS Users Group International Conference*, Cary, NC:SAS Institute Inc.

Acknowledgments

I would like to thank Jack Fuller, Andrew Norton and John Ellis for the lengthy discussions surrounding Design Patterns and their implementation.

Author Contact

Kevin T. Brown
Project Manager
Keane, Inc.
5989 Tahoe Drive S.E.
Grand Rapids, MI 49546
E-mail: brown@net-link.net