Using OOP Techniques to Make FRAME Applications More Intuitive & User Friendly.

Mark Bodt, SUNKEN TREASURE SOFTWARE SYSTEMS LIMITED, New Zealand

Written for SUGI 22 Applications Development.

Abstract

Subclassing of SAS®/AF Frame objects allows developers to tailor their own classes to best suit the needs of the application. Issues that affect how intuitive and user friendly an application is will be discussed, along with some examples of subclassing that promote the usability of an application.

Introduction
The classes that are provided with SAS/AF can achieve a wide range of tasks. Using these classes as they are is often satisfactory, however often an application has particular requirements. These may be achieved with the supplied classes, but may require additional code. The nature of OOP allows the creation of subclasses which incorporates this additional functionality. These subclasses can be added to the 'toolbox' of classes that are used to build Frame applications.

Scope
In this presentation, it is my intention is to demonstrate some real life applications where subclasses that I have developed have been used to achieve tasks specific to the client's application. Some of the subclasses will be examined at code level and there will be some discussion on some aspects of application design.

Background
The examples in this paper are from a financial application which has data entered on a quarterly basis. The data is collected from offices around the world. This is currently in hard copy format, but later may be done electronically. The data entry involves some quite complex validation. And the subsequent processing of the financial data follows a series of very complex rules. From the results reports are generated and data interfaced electronically with a General Ledger system. In terms of value, the application handles transactions totalling in excess of $100 million per annum. [1]

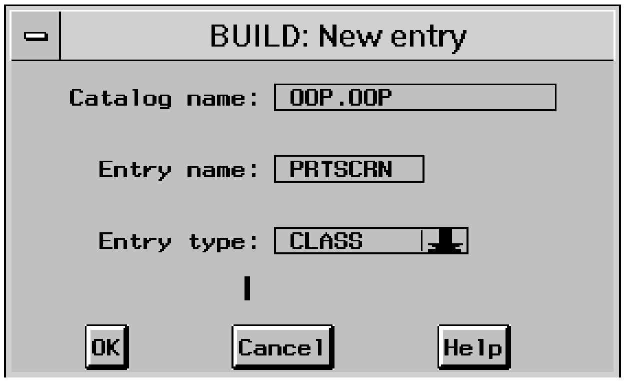
The Screen Print Button Class.

The application has many data entry screens, as well as browse screens for viewing the results of processing. The client required print screen functionality on many of the screens. In Windows, Screen Print functionality can be achieved with the Display Manager Command DLGPRT. It is an easy task to code this operation to work with a Push Button Widget in a Frame. As the Screen Print push button was used on many screens throughout the application, it made an ideal candidate to subclass and add to the application development 'toolbox'. Creating a Screen Print class means that a screen print button can be placed on any frame, providing screen print functionality without the need to add any code whatsoever to the Frame's SCL entry.

Creating a Screen Print class sounds like a simple task, but it involved some trickery as it is not possible to issue a Display Manager window command from a subclass run method as a display manager command can only be issued from an SCL entry that has a window (ie an associated Frame).

Creating the Screen Print Button Class.

The Screen Print Button Class is a subclass of the Push Button class. In the catalog where all the application subclasses are located, create a new class. In this example, the catalog OOP.OOP has been used and the class has a name of Prtscrn.class



BUILD: New entry

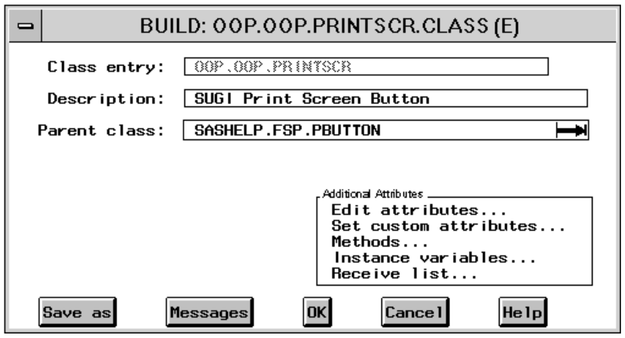
Catalog name: OOP.OOP

Entry name: PRTSCRN

Entry type: CLASS

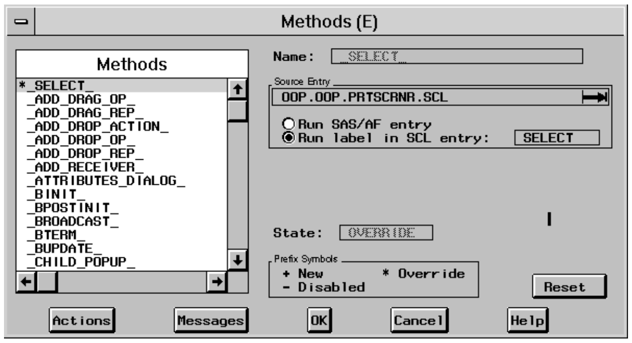
The description is changed to 'SUGI Print Screen Button' and the Parent Class set to SASHELP.FSP.PBUTTON

Clicking on Methods displays the Methods dialog. The _select_ method is overridden, the source entry set to OOP.OOP.PRTSCRNR.SCL and the label set to SELECT. Double clicking on the source entry opens the SCL entry containing the run method.


BUILD: OOP.OOP.PRINTSCR.CLASS (E)

Class entry: OOP.OOP.PRINTSCR
Description: SUGI Print Screen Button
Parent class: SASHELP.FSP.PBUTTON

Additional Attributes
Edit attributes...
Set custom attributes...
Methods...
Instance variables...
Receive list...


Methods (E)

Methods
```
*_SELECT_
 _ADD_DRAG_OP_
 _ADD_DRAG_REP_
 _ADD_DROP_ACTION_
 _ADD_DROP_OP_
 _ADD_DROP_REP_
 _ADD_RECEIVER_
 _ATTRIBUTES_DIALOG_
 _BINIT_
 _BPOSTINIT_
 _BROADCAST_
 _BTERM_
 _BUPDATE_
 _CHILD_POPUP_
```

Name: _SELECT_
Source Entry: OOP.OOP.PRTSCRNR.SCL
Run SAS/AF entry [✓ Run label in SCL entry:] SELECT
State: OVERRIDE
Prefix Symbols: + New * Override - Disabled

The code for the select method is very simple:

```
/* Run methods for the SUGI Print Screen Class
    Written by Mark Bodt 25 Sept 1996 */

*avoid compilation warnings;
_self_=_self_;


select:
  method;
  /*The display manager command to print the screen
    as a bitmap cannot be issued from a run method.
    for this reason a frame is called. */

    call display('prtscrn.frame');

    call super(_self_,'_select_');
endmethod;
```

Because the Display Manager Command DLGPRT does not work from a method, a frame is called that will issue the command.

The frame that is called from the _select_ run method is now built. The frame does not contain any widgets and the SCL for the frame is as follows:

```
*********************************************************;
* Program to: Print the screen as a bit map.            *;
*             Part of subclass prtscrn                  *;
*                                                       *;
* Written by: M.R.Bodt  28 June 1995                    *;
*                                                       *;
*********************************************************;

/*The print screen subclass calls the select method
  prtscrnr.sclwhich calls this frame because you cannot
  call a frame as a select method. You have to use a
  frame to use the DLGPRT function.
  The frame has been made into a dialog box so that it
  does not zoom max at run time. If it was a standard
  window then the frame would zoom max if the calling
  program was zoomed, causing the calling window to be
  blocked out with this frame and a blank screen being
  printed.

  This frame assumes that the printer driver and
  destination parameters are present in the global
  environment list.
*/
```

```
  index=nameditem(envlist('g'),'Screen Print Driver');
  if index=0 then do;
      *error handling;
       call display('error.frame',
         'The Screen Print Driver parameter could not',
         ' be found in the global environment list.',
         'The screen will not be printed.');
      end;
    else do;
     /*check that the Screen Print Dest is present in
       the G env list */
      index=nameditem(envlist('g'),
            'Screen Print Destination');
      if index=0 then do;
      *error handling;
         call display('error.frame',
           'The Screen Print Destination parameter ',
           'could not be found in the global ',
           'environment list.The screen will not',
           'be printed.');
        end;
      else do;
        *get printer settings from environment list;
         prtdrv =getnitemc(envlist('g'),
                  'Screen Print Driver');
         prtdest=getnitemc(envlist('g'),
                   'Screen Print Destination');
        /*get current option setting and save for
          resetting option later*/
         current=optgetc('sysprint');

        *build option;
         option='"'||prtdest||"' '"||prtdrv||"'";
         rc=optsetc('sysprint',option);
         call execcmdi('DLGPRT NOSOURCE SCREENBITMAP');

        *reset printer setting;
         rc=optsetc('sysprint',current);

       end;
    end;
   *terminate screen;
    call execcmd('end');
return;

main:

return;


term:
   rc=rc;
return;
```

The only problem with this is that the frame that is called will also appear in the screen print. The trick is to make the frame as small as possible. The General attributes for this frame are set to:

- Dialog Box
- Command=None
- Window Size: 1,1,1,1

When the _select_ method runs, this frame will still display over the top of the frame that we want to screen print, but it will appear only as a tiny box at the top left hand corner.

To complete the subclassing, the new subclass is entered in the resource entry.

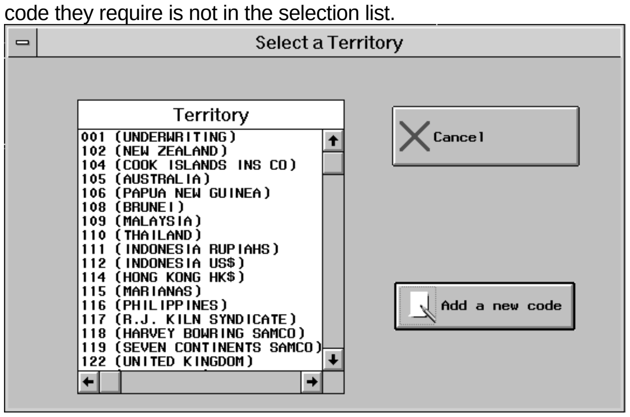
When using this subclass, remember that the parameters that are used in the _select_ method: Screen Print Driver & Screen Print Destination must be in the Global Environment list.

Using Parameters
Typically, the parameters that are used in the applications that I build are held in an SLIST catalog entry and these are loaded during the start up of the application into the environment list.

Using parameters in this way, rather than hard coding the values into programs, means that it is a simple task to change the operation of the application. For example, the user may decide to print their screen prints to a different printer, or the printer may be upgraded to a different type. This particular application was originally developed by another company in the days of release 6.04. Most of the reports had the company name hard coded. Since then the company has changed their name. This meant that each of the reports needed modification to print the correct name. Since then parameters such as: The company name & address, company logo, interface file locations etc have been used. This has paid off as since incorporating the use of parameters the company name has changed two more times as it expanded it's operations.

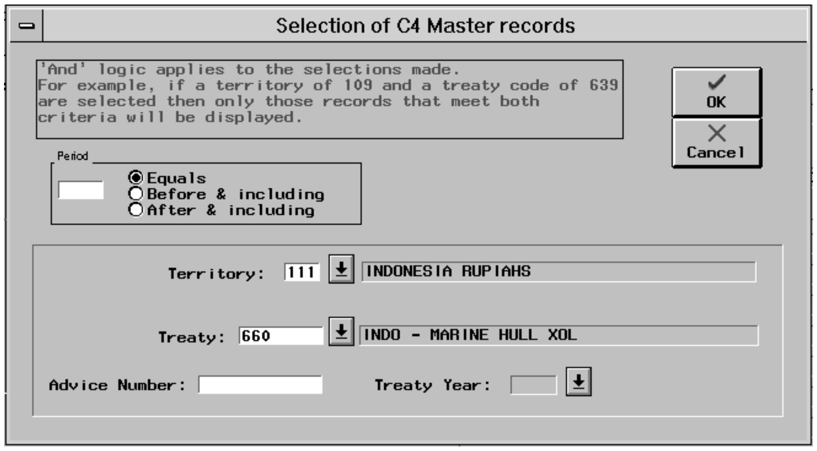
Selection of codes

In the interests of saving space, data is often stored as codes. These codes may become familiar for the every day user, however codes that are not used often may not be known to the user. For this reason, it is good practice to include both the code and the description in selection lists. An example of this is shown below.
The reference table data in the list is stored in a dataset. It is sometimes necessary to add a new code. This functionality is built into the selection screen, so that the user does not have to exit out to a completely different part of the application if the code they require is not in the selection list.



```
001 (UNDERWRITING)
102 (NEW ZEALAND)
104 (COOK ISLANDS INS CO)
105 (AUSTRALIA)
106 (PAPUA NEW GUINEA)
108 (BRUNEI)
109 (MALAYSIA)
110 (THAILAND)
111 (INDONESIA RUPIAHS)
112 (INDONESIA US$)
114 (HONG KONG HK$)
115 (MARIANAS)
116 (PHILIPPINES)
117 (R.J. KILN SYNDICATE)
118 (HARVEY BOWRING SAMCO)
119 (SEVEN CONTINENTS SAMCO)
122 (UNITED KINGDOM)
```

In the case of 'reference tables' (tables which contain unique codes & descriptions), there are typically only two variables being the code & the description. In this application there are many reference tables. Operations on these tables such as selection lists, code entry validation will be similar for all the reference tables, and this property suggests the use of OOP techniques. Methods and subclasses have been written for the selection of a code from a list box, a popmenu as well as data entry.


Selection of C4 Master records

'And' logic applies to the selections made.
For example, if a territory of 109 and a treaty code of 639 are selected then only those records that meet both criteria will be displayed.

Period: [✓ Equals] Before & including / After & including

Territory: 111 INDONESIA RUPIAHS

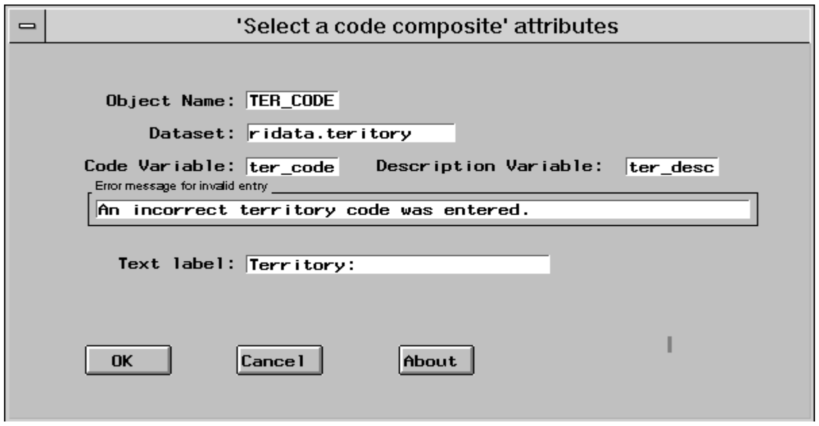
Treaty: 660 INDO - MARINE HULL XOL

Advice Number:

Treaty Year:

The above screen shows some examples of some composite subclasses that I have created. Both the territory & treaty code entry widgets are composites consisting of four widgets, the label, the code, the dropdown widget and the description. Using these composite widgets when developing a frame is very straight forward. It is just a matter of placing them in the desired position, specifying the reference table dataset, code variable and description variable. The subclass offers the following functionality: validation of the entered code, required attribute, display of the description and when the dropdown is clicked, a selection list displays both the code & description in a popmenu, or a selection dialog. Messages are sent to the message line if there are errors such as a required code not being entered, or an invalid code being entered. These messages can be customised.

The attributes screen for the composite are shown to the left


'Select a code composite' attributes

Object Name: TER_CODE

Dataset: ridata.teritory

Code Variable: ter_code

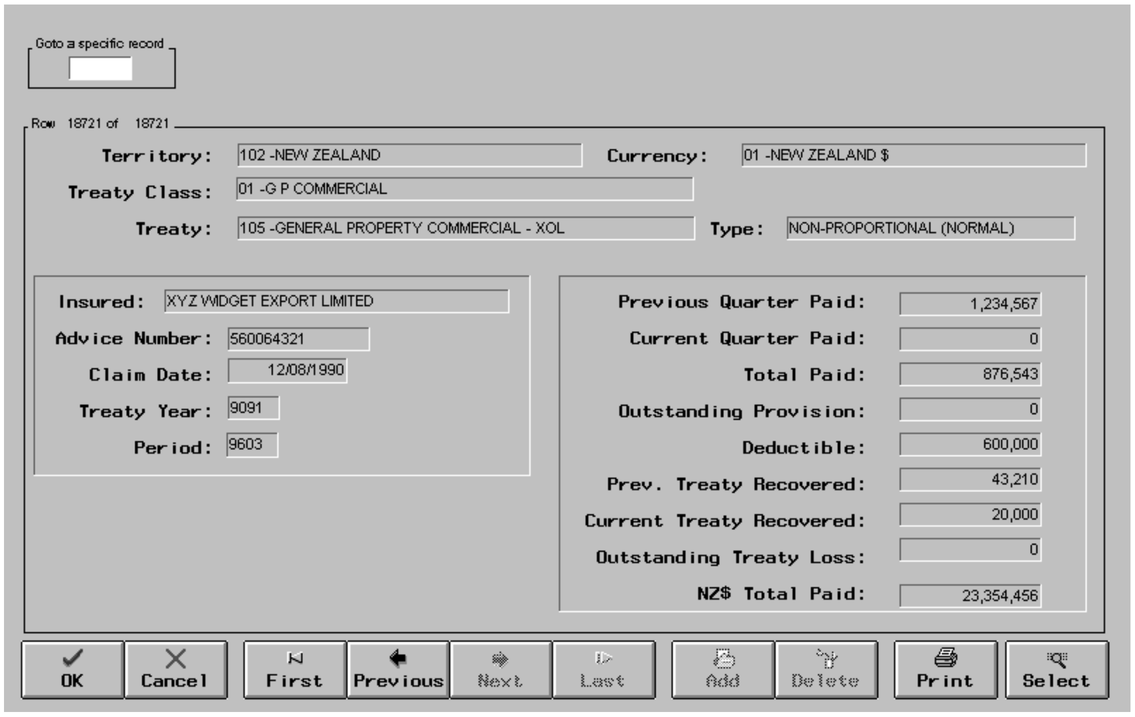
Description Variable: ter_desc

Error message for invalid entry: An incorrect territory code was entered.

Text label: Territory:

Subclassing the dataform class

In this application there are numerous reference tables that need editing from time to time. There are also data files that hold the results of the complex financial processing that occurs each quarter. I have found the new dataform class an excellent


Goto a specific record

Row 18721 of 18721

Territory: 102 -NEW ZEALAND

Currency: 01 -NEW ZEALAND $

Treaty Class: 01 -G P COMMERCIAL

Treaty: 105 -GENERAL PROPERTY COMMERCIAL - XOL

Type: NON-PROPORTIONAL (NORMAL)

Insured: XYZ WIDGET EXPORT LIMITED

Advice Number: 560064321

Claim Date: 12/08/1990

Treaty Year: 9091

Period: 9603

Previous Quarter Paid: 1,234,567

Current Quarter Paid: 0

Total Paid: 876,543

Outstanding Provision: 0

Deductible: 600,000

Prev. Treaty Recovered: 43,210

Current Treaty Recovered: 20,000

Outstanding Treaty Loss: 0

NZ$ Total Paid: 23,354,456

addition to the toolbox of widgets that are available with SAS/AF software. Several subclass have been created for working with the dataform, adding extensive functionality. The result is a data entry or browse screen that is user friendly and has all the functionality of a data entry screen, but without any code at all in the Frame's associated SCL entry. These screens are very quick to assemble and because all the subclasses that are used have already been thoroughly tested, there is little in the way of testing to be carried out when a new screen is built.
The above screen is a browse screen, but it contains the same widgets as for an edit screen. Each screen has up to three widgets being a toolbar subclass, a dataform subclass and a goto composite subclass. Event handlers are used to communicate between each of the widgets which means that each widget is completely independent and errors do not occur if one of the widgets is not present. For example if the Goto composite is not necessary and is not placed in the frame, then the other two widgets will still function normally. All functionality is contained in the subclasses. When this screen is developed there is no SCL to code at all.

Event handling

The widgets communicate with each other using the event handlers. The toolbar subclass for example sends various events such as:

```
Events sent:     Delete record
                 goto first record
                 goto previous record
                 goto next record
                 goto last record
                 add a record
                 select dialog
```

When the user clicks on Previous on the toolbar, then the toolbar sends an event 'goto next record'. The event is not sent to a specific widget, but if a widget is listening for the event then it will do something with it and in this case the dataform widget will be listening for this type of event and will action the request by going to the next record.
Similarly, the toolbar widget receives events. It listens out for events sent by other widgets and actions particular events. In the case of the tool bar it listens for the events:

```
Events received: New record (record x of n)
                 browse mode
                 no selection
                 where clause (Y/N)
```

The dataform widget sends the New Record event whenever the record (observation) number displayed and the record number and the total number of records is also sent with the event. The tool bar picks up on this and will change it's appearance as appropriate for the record number ie if (as is the case in the screen above) the record is the last record in the dataset, then the toolbar will grey the 'Next' and 'Last' buttons. Similarly if the record displayed by the dataform is the first in the dataset then the 'First' & 'Previous' buttons are greyed. If an event was received of 'Browse mode' this means that the dataform is in browse only mode and the 'Add' & 'Delete' buttons are greyed.

```
objlabl:
 method;
  *_object_label_ run method;
  call super(_self_,_method_);
  /*this code, being after the call super, runs after the frame
    scl label processing. */

  *get index of toolbar button pressed;
   call send(_self_,'_get_last_sel_',tbindex,issel);

     select(tbindex);
      *OK;
       when(1)
        ;
      *Cancel;
       when(2)
       ;
      *Goto beginning of dataset;
       when(3)
         call send(_self_,'_send_event_','goto first record');

      *Goto previous record;
       when(4)
         call send(_self_,'_send_event_','goto previous record');
     .......... some similar code snipped .........
otherwise
       put 'Error detected in DFTOOLBR run methods. Invalid tbindex'
           tbindex;
   end;
endmethod;
```

Sending Events

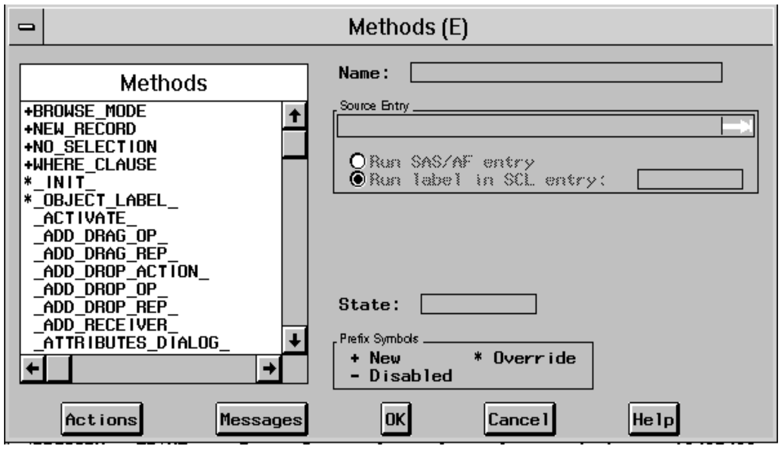
Sending & receiving events is quite a straight forward process. To send an event the method _send_event_ is used. Below is the run method which overrides the _object_label _ method for the toolbar class.

If the user clicks on the 'First' button (button number 3) then the 'goto first record' event is issued. This is all that is required to send an event. As you can see the toolbar subclass is very simple. The actual functionality of going to the first record is done in the dataform subclass.

Receiving Events.

There are a few more steps in setting up a widget to receive events. The widget needs to know which events to listen out for and what action to take if it receives that event. In the overridden _init_ method for the toolbar class below, the event handlers are set using the _set_event_handler_ method. The first parameter * specifies that the event can come from any widget. The second parameter is the event to listen for eg 'browse_mode' and the third parameter is the method to run when the event is received. In the examples below, the methods are new methods that have been added to the subclass.
These new methods are defined for the class as can be seen in the toolbar subclasses methods window. An example of the run method for the 'browse mode' event is as follows:

```
browse:
 method;
   call send(_self_,'_gray_','add');
   call send(_self_,'_gray_','delete');
endmethod;
```



The method simply greys the add and delete buttons as these are not applicable when the dataform is in browse mode.

```
init:
 method;
  *_init_ run method;

   call super(_self_,_method_);

  *set event handlers;
   call send(_self_,'_set_event_handler_','*',
            'new record','new_record',_self_);
   call send(_self_,'_set_event_handler_','*',
            'browse mode','browse_mode',_self_);
   call send(_self_,'_set_event_handler_','*',
            'where clause','where_clause',_self_);
   call send(_self_,'_set_event_handler_','*',
            'no selection','no_selection',_self_);
   where_clause='N';

endmethod;
```

Event handling in the dataform subclass

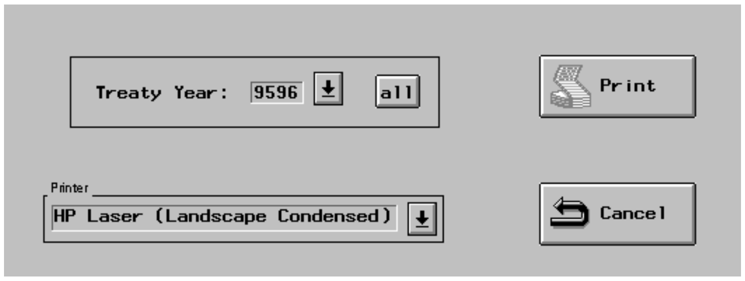
The dataform sends & receives events as follows:

```
/*event handling:
  Events received: Delete record
                   Add record
                   goto first record
                   goto previous record
                   goto next record
                   goto last record
                   goto record
                   add a record
                   delete record
                   selection dialog

  Events sent:     New record (record x of n)
                   browse mode
                   no selection
                   where clause (Y/N)
*/
```

The run methods for the events are quite straight forward. Below is an example of the run method for the next event:

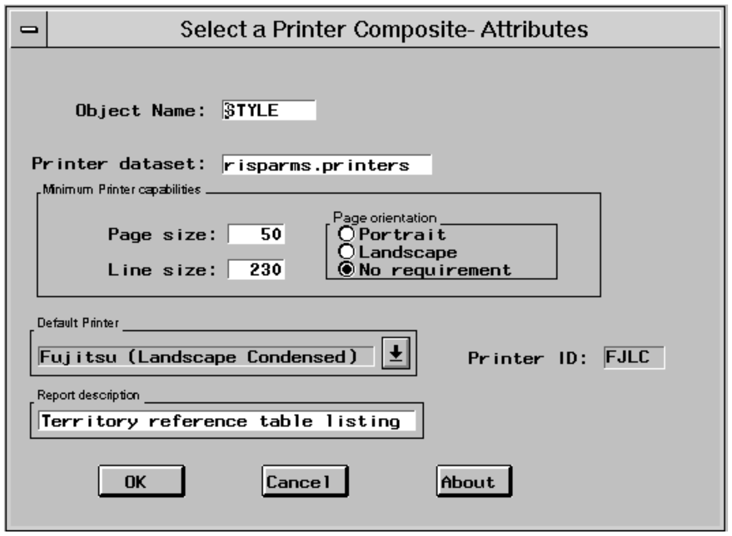
```
next:
 method;
 *goto next record method;

  link update;*to overcome bug;

  call send( _self_, '_vscroll_','row',1);
  link settitle;
endmethod;

update:
 /*save before going to another record. Refer to bug in useage notes
   Using _goto_absolute_row_ may copy changes from the current row.
   This is not applicable to dataforms that are in browse mode*/
  proplist=makelist();
  call send(_self_,'_get_properties_',proplist);
  model_info=getniteml(proplist,'model_info');
  browse_status=getnitemc(model_info,'dsmode');
  if listlen(proplist)>=0 then rc=dellist(proplist);
  if browse_status ne 'BROWSE' then do;
   call send(_self_,'_update_row_');
   call send(_self_,'_unlock_row_');
  end;
return;
```

The Printer Subclass


Treaty Year: 9596

The application generates many reports. It is a requirement that the application was very easy to use, and this included the printing of reports. The user had to be able to select which printer the report would print on and this selection had to be easy to do. At the same time, the reports differed in size. Some reports were to print on A4 paper, while others were to print on line flow. The range of linesizes ranged from 60 through to 220 characters. It was necessary to control the printer orientation, destination (driver & port), font and point size. *SAS* provides the ability to do this using the printer set up dialog, but this was too involved. The user also needed to view the reports prior to printing.

I came up with a couple of frames and a subclass that would address all these printing requirements. When the user selected a particular report, The report selection screen would display. In this screen they could make any selections that would restrict the data in the report, in the above example, the user can select the treaty year. The user can also select the printer. This composite widget is the subclass that was created to handle the printer selection and set up. Information about each of the printers that is used by the application is stored in a dataset, and a meaningful description is allotted .


Select a Printer Composite- Attributes

Object Name: STYLE

Printer dataset: risparms.printers

Minimum Printer capabilities

Page size: 50

Line size: 230

Page orientation: Portrait [✓ No requirement] Landscape

Default Printer: Fujitsu (Landscape Condensed)

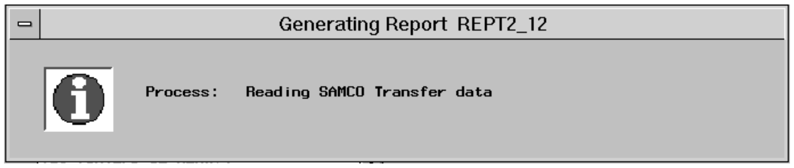
Printer ID: FJLC

Report description: Territory reference table listing

The attributes dialog for the widget allows the developer to specify the minimum page & line size requirements for the report and if there is an orientation requirement. The default printer can also be selected and the report description.

When the report is run, the default report is set on initialisation of the printer selection widget. If the user clicks on the select dropdown, then only those printers that meet the minimum page & linesize requirements are displayed. This way the user cannot select a printer that can only print 80 characters wide for a report that needs a width of 230.

Typically, when the user runs the report, by clicking on the Print button, the frame calls another frame. the frame contains the submit blocks that process the data & print the report (well at least to the output window). Rather than submitting the report in one submit block, the processing is divided up into different submit blocks and between each submit block , the ‘Process’ message is updated. The process message does not have to be that meaningful, but indicates to the user that the report is running. If instead, the standard hourglass ‘busy’ pointer displayed with no changes to the screen for 5 minutes, or however long it takes to run the report, then the user may think that the system has crashed.


Generating Report REPT2_12

Process: Reading SAMCO Transfer data

```
 *update status indicator;
     process='Adding General Ledger suffix codes';
     refresh;

     submit continue;
.... submitted SAS code ....
 endsubmit;

  *update status indicator;
     process='Adding General Ledger total codes to report.';
     refresh;
     submit continue;
     ... More SAS Code....
       endsubmit;
  *update status indicator;
     process='Reading SAMCO Transfer data';
     refresh;
     submit continue;
... More SAS Code....  etc etc
```

Example of the code.

Then the report has finished, the output has been listed in the output window, however rather than viewing the results in the output window, the option -autopop off was set to prevent the output window from displaying. Instead, the process window closes and the Report Selection screen closes. The printer selection widget has the _term_ method overridden. When the widget's _term_ method runs, if the frame status was E ie not cancel, then the output window contents is saved using the woutput function to a output type catalog member. The _term_ method then calls a frame that allows the user to view the catalog member and from that window they can print, delete, print & delete etc the report. On closing the viewer frame, the remaining code in the _term_ method runs. This has added significant functionality to the generating of reports. Another frame has been developed which allows the user to maintain the stored reports.

```
term:
  method;
  *on termination of widget, save the contents of the
output window;

    *get frame status as we only want to do this for
_status_ of E;
     call send(_frame_,'_get_status_',frame_status);

    *save output;
     if frame_status='E' then do;
        *build cat member name;
            *get month part;
             Monthn=month(today());
             month=substr('ABCDEFGHIJKL',monthn,1);
            *get day part;
             day=put(day(today()),z2.);;
            *get time part;
             time=put(time(),z5.);
            *build catalog name;
             catname=month||day||time;
            *build catalog description;
             catdesc=put(id,$5.)||report_desc;
```

```
rc=woutput('save',

compress('reports.reports.'||catname||'.
output'),
              catdesc);
       *clear output window;
        rc=woutput('clear');
       *view generated report;
        call
display('risys.risys.vrept.frame',catnam
e);

       end;
  call super(_self_,_method_);
endmethod;
```

Conclusion:.

By using Object Oriented Programming, it is possible to radically reduce the time required to develop applications. By subclassing, the default widgets that are provided with SAS/AF can be tailored to the specific requirements of the application. This paper demonstrated some examples of what can be achieved with OOP and SAS/AF Frames.

Contact details

Sunken Treasure Software Systems Limited
Specialising in SAS Software Consultancy for the Asia - Pacific Region

73 PINE STREET MT EDEN AUCKLAND NEW ZEALAND
PH 025 725 386 FAX +64-9-620-9079
INTERNET MARKBODT@STSS.CO.NZ

[1] *SAS* Insight, The Newsletter for *SAS* Institute (NZ) Ltd Issue No. 1