

Dynamically Instantiating Widgets on SAS® Frames – Why, How, and When A Sample

David Ward, InterNext, Inc., Somerset, NJ

ABSTRACT

SAS/AF Frames can be a quick and powerful way to build SAS-based application in your organization. There may be times when you need your users to be able to build frames dynamically, for example, to build data entry screens. It is possible to use the work area class from version 6 to build a run-time frame "editor" by which you can allow users to add/remove/move widgets and set their properties. A frame can then be written which will dynamically instantiate widgets the user has built at design-time. The parent frame can even add and change methods and properties of those widgets as needed. Presented in this paper is a data entry application that allows users to build screens and uses a SAS/AF Frame to draw the screens at run-time. This technique can provide a wealth of rich functionality to your SAS/AF applications, especially if you need your users to design part of your application.

INTRODUCTION

The ability to dynamically instantiate widgets on your SAS/AF frames comes from two SCL operators: `_new_` and `_neo_`. According to the SAS OnlineDoc version 8, `_new_` "creates an object and runs an associated class constructor". The class constructor is also referred to in AF-speak as the `_init` method. This method is called automatically, just like the `_term` method when an object is deleted. Again, according to OnlineDoc, "the `_NEO_` operator provides a faster and more direct way to create an object. It combines the actions of loading a class with `LOADCLASS` and initializing the object with the `_new` method, which invokes the object's `_init` method".

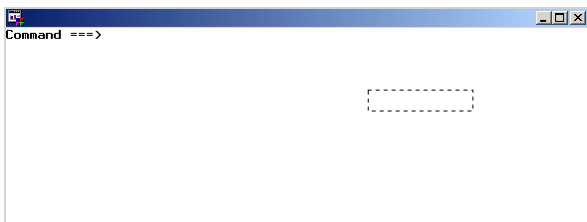
YOUR FIRST EXPERIMENT

To get an idea of what creating widgets at run time looks like, create a test frame named `sasuser.widgets.test1.frame`. Do not create any widgets; simply go directly to the source code for the frame. Enter the following code:

```
Import sashelp.classes;
Dcl textentry_c text1 = _new_ textentry_c();
init:
return;
```

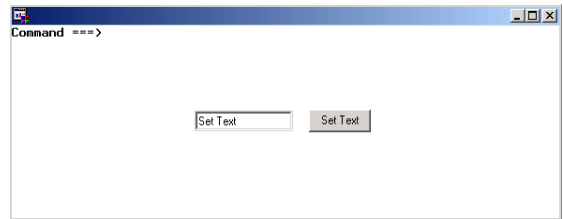
The code above creates a new object named `text1` when the frame is executed. You may have seen this syntax when creating non-visual objects such as the logical, library, or catalog classes in the catalog `sashelp.fsp`. In version 6, SCL used the `instance()` and `loadclass()` functions. The `_new_` operator provides a more intuitive syntax to do the same thing. In this example, however, we are creating a new instance of the text entry class, the same class used to create text boxes on frames.

Next, run the frame. Immediately you will see the outline of a text entry box that follows your mouse pointer around the frame. Clicking anywhere on the frame displays your text entry in the location where you clicked. You can now reference the text entry box in the same way you always have.



We can add a button to the frame at design time that sets the text property of our text entry box. Once the widget is created, clicking on the button will display the text.

```
button1:
text1.text='Set Text';
return;
```



SETTING THE LOCATION OF THE WIDGET

The previous example would prove frustrating to end-users. Each time they ran the frame they would be forced to decide where to place the text entry box. As a developer, you would probably not want to give that kind of control to your users either. The solution is to pass the `_neo_` operator a list of region attributes so that the text box will know how to "draw" itself. The frame SCL should now be:

```
Import sashelp.classes;
Dcl textentry_c text1;
init:
startcol=10; startrow=10;
AttrList={};
RegionList={};
rc = setniteml(AttrList, RegionList,
'_region_');
rc = setnitemn(AttrList, -1, 'num');
rc = setnitemn(RegionList, startcol,
'ulx');
rc = setnitemn(RegionList, startrow,
'uly');
text1=_neo_ textentry_c(AttrList);
return;
button1:
text1.text='Set Text';
return;
```

We have added code to create an attribute list with two parameters, `ULX` and `ULY`. These correspond to the column/row position that the text entry control should be created at.



Once your widget has been created and is placed on the screen you can set other desired properties. Any editable property (see the OnlineDoc or the properties window of the frame builder) can be accessed. If you wanted to change the color of the text entry box you would write:

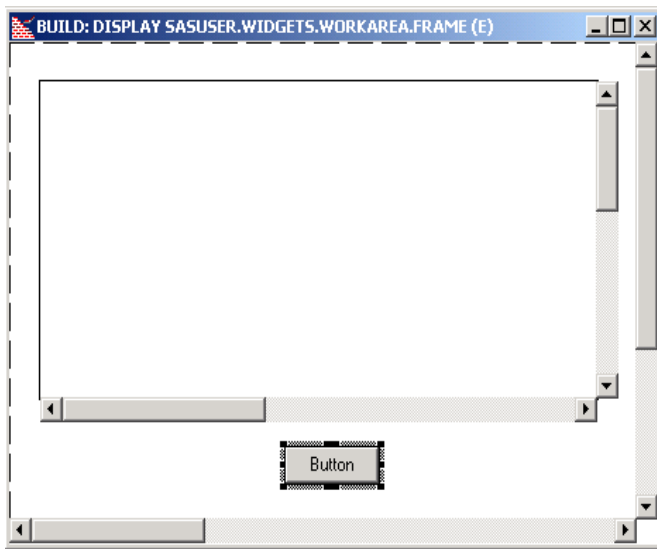
```
text1.backgroundColor='red';
```

Of course, make sure you place this code after the `_neo_` statement.

SAVING WIDGET PROPERTIES

Typically, developers would use this kind of functionality to allow users to build or customize part of an application themselves. For example, you could allow your users to drag and drop and set properties for a welcome frame or main menu. This way, they could customize images, widget sizes, etc. But how can we save the information each user creates about where they would like their widgets placed and what properties they would like to set? The answer is the work area control from version 6. It allows users to create and manipulate widgets at run time and to save the entire area out as an SCL list.

Create a new frame called `sasuser.widgets.workarea.frame`. Place a simple workarea in the frame and add a push button anywhere. Your frame should now look something like this:



Enter the following SCL code for the frame:

```
pushbutton1:
  widgets=makelist();
  call notify ('obj1','_get_widgets_',widgets);
  rc=putlist(widgets,'',0);
  rc=dellist(widgets,'Y');
  return;
```

After running the frame, you will notice that you can right-click anywhere inside the workarea control and choose the option "add item". Doing so shows you a list of all built-in SAS widgets (or controls). Choosing the text entry control will display an outlined control much like in our first example. It is up to you to decide where to place it. Once you have created a widget, right-clicking on the widget and choosing "properties" will allow you to set design-time properties of the widget. Once you have created some widgets and set their properties click on the push button. In your log you should see a rather large list containing one sublist per widget. The list contains all information about each widget, even the ULX and ULY values we looked at earlier. You can use the `_region_` sublist in conjunction with the `_neo_` operator to create exact copies of the designed widgets on a frame.

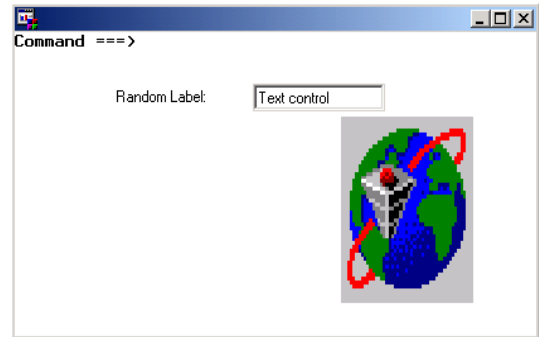
USING THE WORKAREA LISTS

After saving the list you viewed in our last example, you need to use it in a practical way to generate widgets. Simply using the workarea again is not practical because users can make changes and you cannot easily manipulate widgets inside programmatically. The answer is to build a new frame that reads the widget list and creates a new widget for each in the list. Here is some code to get you started:

In this case we need to use the "old-fashioned" `instance()` and `loadclass()` functions because the type of widget is not known at

compile time. Notice that each time we used `_neo_` we had to hard-code the widget type (textentry_c above).

Here is a screen shot of our simple example and the SCL code used to create it:



And the SCL source code:

```
Dcl object widget;
Dcl char classname;
init:
  widgets={};
  rc=filllist('catalog',
    'sasuser.widgets.workarea.slist',
    widgets);
  do i = 1 to listlen(widgets)-2;
    * LAST 2 ARE NOT WIDGETS *;
    sublist=getitem1(widgets,i);
    region=getnitem1(sublist,'_REGION_');
    ** REMOVE BORDER ATTRIBUTE **;
    rc=delnitem(region,'border_style');
    classname=getnitemc(sublist,
      '_CLASSNAME_');
    classname='sashelp.classes.'||
      scan(classname,1,'.');
    widget=instance(loadclass(classname),
      sublist);
  end;
  return;
```

This code exactly reproduces what the user designed in the workarea.

NEXT STEPS

In order to use your instantiated widgets effectively, you will need to store the object identifiers in SCL variables or in an SCL list. If you have an undetermined number of widgets you should use an SCL list with the `inserto()`, `getitemo()` functions. This way, you can programmatically set properties of the widgets. Additionally, you can use the `_addEvent`, `_addAttribute`, etc. functions to trap events, override methods, or do anything else that you would normally do at design time. This way you can execute code when a user uses these new widgets.

In the presentation of this paper a data entry screen builder will be demonstrated that uses these techniques to allow the design of potentially complex screens.

CONCLUSION

In this paper you have been exposed to a technique to dynamically instantiate widgets on AF frames. This is a powerful way to let your users customize or design part of your application and can be used for such things as building data entry screens.

ACKNOWLEDGMENTS

SAS and SAS product names are registered trademarks of the SAS Institute. Other trademarks are the properties of their respective owners.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David L. Ward
InterNext, Inc.
254 Resnik Ct.
Somerset, NJ 08873
Work Phone: (732) 745-9823
Email (preferred): dward@sashelp.com