SAS/AF® Composite Class:
A Detailed Example

Richard A. DeVenezia, Remsen, NY

ABSTRACT

The SAS/AF Composite class is the starting point for implementing useful multi-component objects. These objects can be deployed for use by SAS/AF Frame developers. For the entrepreneur, catalogs of unique and timesaving composites can be a marketable and saleable commodity. For the project manager or system developer, using composites can increase productivity and promote GUI consistency.

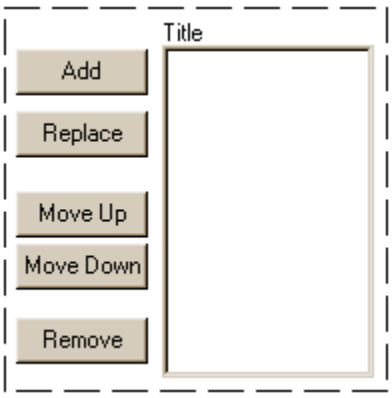
A class designer must consider event handling and the composite's intra-object and extra-object behavior. The class user must understand how the class will behave, and how to work with the class in ways the designer did not anticipate.

INTRODUCTION

This paper will concentrate on a variety of techniques available to both the class designer and class user in regard to getting the most out of a composite class. The design and use of a sample composite class, ***ListOfValues***, will be used to illustrate these techniques.

The reader should be comfortable with or cognizant of:

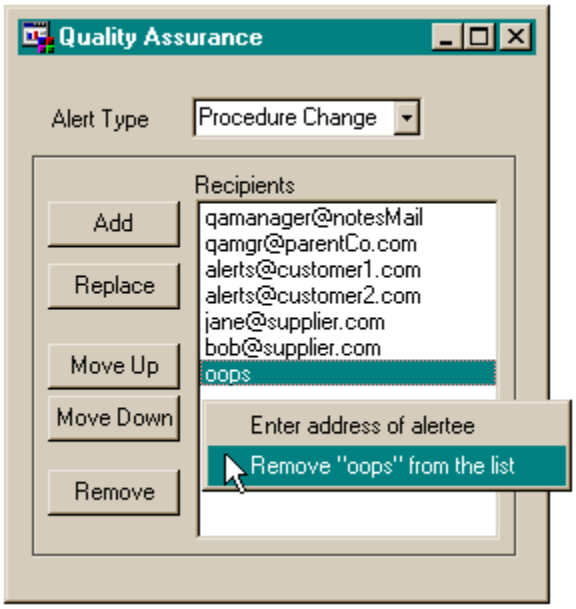
- Version 8 components
- Properties window
- Class editor
- Overriding attributes and methods
- Writing custom access methods
- Model / Viewer technology
- Events and event handlers
- Writing methods



ListOfValues composite class

SAMPLE SCENARIO

A developer is implementing a frame for use by a Quality Assurance Department. Each Quality Technician is required to maintain his own list of e-mail addresses to notify when certain quality related issues arise. Some other program generates a report and sends it to each address in the list. The developer designs and implements the ***ListOfValues*** class for use in frames where individual lists of values need to be maintained.



Sample Frame using ListOfValues

LEARNING ABOUT THE COMPOSITE CLASS

There are two online sources of documentation available to each SAS installation.

SAS SYSTEM HELP (F1)

Documentation on the operation of the composite class can be found in Version 8 SAS System Help, Contents tab, Help on SAS Software Products book, SAS/AF Component Reference sub-book, Legacy Classes (from Version 6) sub-book, Composite Class document.

Searches on keyword *composite* will return help entries that are pertinent to only SAS Version 6. Any documents that mention the *Composite Attributes Window*, *Edit Definition*, *Object Attributes*, *Region Attributes* or *Attachments* are for Version 6 and are not applicable to Version 8.

In Version 8, the *Class Editor* and *Properties Editor* are used to interactively design the implementation of a composite sub-class.

SAS ONLINE DOC VERSION 8 (CD)

The first release of the Online Docs CD has no information on the composite class.

STEP BACK - DEVELOPING CLASSES

There are several ways to develop a SAS/AF class. In all cases, code within a CLASS / ENDCLASS must be compiled using the SAVECLASS command, and code in a USECLASS / ENDUSECLASS block must be compiled using the COMPILE command. At this time, there is no automatic make feature such as “compile this object and all its changed dependencies.”

Here are three typical scenarios:

CASE 1. A SINGLE .SCL ENTRY

Contains all declarations of attributes, methods, events, event handlers and interfaces and their implementations within a CLASS / ENDCLASS block.
The .class entry is created using the SAVECLASS command. This scenario is used when there are a few short methods.

CASE 2. TWO (OR MORE) .SCL ENTRIES.

- The first contains declarations of attributes, methods, events, event handlers and interfaces within a CLASS / ENDCLASS block. The .class entry is created using the SAVECLASS command.
- The second (and others) contain implementations. Each SCL entry must be compiled. SAS recommends using a naming convention that indicates the relatedness of the SCL entries; for example:
  - *classname.scl* contains declarations
  - *classnameCode.scl* contains the implementations.
  - When the implementation code is within a USECLASS / ENDUSECLASS block, all the attributes, methods, events, event handlers and interfaces of the class are early-bound at compile time and *implicitly* available. Early binding promulgates the best goals of object orientated programming.
  - When the implementation code is not within a USECLASS / ENDUSECLASS block, none of the attributes, methods, events, event handlers and interfaces of the class are known at compile time. The code must reference the class explicitly using the single implicit variable _SELF_ and dot notation (or call send()). All the features of the class are late bound at run-time.
    Late-binding provides a developer a way to have a method that is used by classes in different class hierarchies.

CASE 3. A .CLASS ENTRY AND A .SCL ENTRY

- The CLASS entry is edited using the Class Editor. The entry is automatically recreated when the editor is exited and changes are saved. The .scl source for a .class entry can be obtained by using the SAVEAS command while in the ClassEditor and entering a .scl name in the save dialog.
- The Class Editor allows you to create or override methods, and by default suggests the implementations reside in *classname.scl*. The SCL may or may not contain a USECLASS / ENDUSECLASS block. See above for the discussion of early and late binding.
  Note: The ClassEditor is very similar to the PropertyEditor used when developing FRAMEs. The default entry for overrides suggested by the PropertyEditor is *framenameMethods.scl.* If you use the SAVECLASS command, do not overwrite your implementation code.
  This is the development scenario used for the ListOfValues class, with an additional SCL entry for component event handling.

COMPOSITE CLASS IMPORTANT FEATURES

Component layout and component scope are two important features of the Composite Class a developer must understand.

COMPONENT LAYOUT

The layout of the components that comprise a composite class are stored in the **componentDefinition** attribute in the System category. The layout tool is invoked when this attribute is edited. Select the attribute's **InitialValue** metadata column. Select the ellipsis to enter the **CompositeDefinition** window. This window allows you to manipulate components in the same manner you do when building SAS/AF Frame entries (drag and drop, model / viewer, linking and attachments.)

The componentDefinition attribute is **Scope Protected**.

Upon exiting the CompositeDefinition window, the class will automatically receive one attribute of Type Object for each component comprising the composite. These are also Scope Protected.

SCOPE RULES

Attributes with Scope Protected are only available to SCL code implementing a class method within a USECLASS / ENDUSECLASS block.

When instantiated in a frame, the Protected properties (attributes and methods) of a class are not available.

A composite class design will require some manipulation of the composite components. Often one or more components need to show some changes that reflect changes to a new Public attributes in your class. The setcam... methods, of these new attributes, are implemented by you in your class SCL and apply the desired changes to your composite.
[Public attributes are editable when instantiated in a frame. setcam... methods are Protected and unavailable at the frame level (except through assigning a value to a public attribute).]

The available feature set of a class is the sum total of the Public attributes and methods and their implementations.

To allow unfettered access to the composite components, you would have to create new Public non-Editable Type Object attributes with a name similar to those of the components. A getcam... method would have to be written for each of these to return the object id of the component. In most cases, unfettered access to the composite components is not desired. Such access can bypass checks and balances coded into the class SCL, and may lead to incorrect or non-functional uses of your class.

LISTOFVALUES CLASS DESIGN

LAYOUT

Visual controls must be available to display the list of values, add a value to the list, remove a value from the list, change a value in the list and change a value's position in the list.

PERSISTENCE

The list of values is desired to be available in any future SAS session or invocation of frame. This means the list of values must be stored somewhere where it can be easily retrieved later, such as a data set, a data file, an SLIST catalog entry, a LIST catalog entry... The catalog SLIST entry was chosen for persistent storage due to ease of use when used with component *SLIST Entry List Model.*

INTERNAL BEHAVIORS (PROTECTED)

Changing the list box when an add, edit, move, or remove push button is clicked. Displaying a list box context menu to allow adding or removing a value. The design precludes an instantiated use that can intercept or otherwise alter these actions.

EXTERNAL ASPECTS (PUBLIC)

Some visual aspects of the class are customizable for instantiated use. Properties such as list box title, persistent storage location and user prompts to present when adding an item are part of the public feature set.

COMPONENTS

The components in the composite are:
List Box Control, SLIST Entry List Model, and Push Button Controls - one for each action: add, replace, up, down, remove.

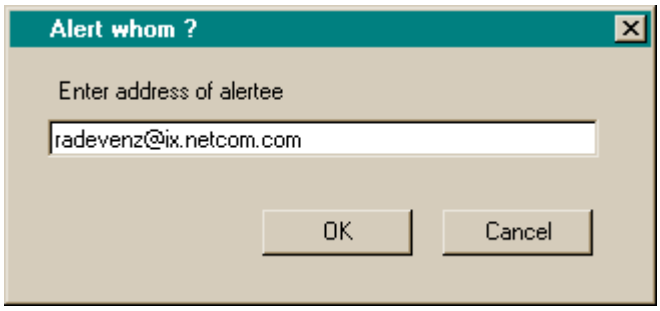
The *List Box* model/viewer interface requires a static SCL list. The *SLIST Entry List Model* model/viewer interface supports a static SCL list. Thus, these two components can be simply connected by setting the value of the *model* attribute of the *List Box* to the name of the *SLIST Entry List Model* object (see implementation section).

The *SLIST Entry List Model* reads the list of values from a catalog entry and delivers that list to the *List Box Control* where it is visualized.

An override of the *_term()* method will be where the list of values shown in the list box will be written to a catalog entry using the *savelist()* function.

INPUT A VALUE DIALOG - ENTERVALUE.FRAME

A generic helper frame used to conserve screen real estate. This frame is called when the user needs to type in a value. The value is passed back to the class.



If this dialog was not used, an additional text entry component would have to be added to the composite. The designer decided the conciseness of the composite required not having an input field as part of it.

These kinds of decisions impact the reusability and deployment of composites. It is also a small decision that may affect how end-users 'feel' about an application. While some users may not like having an extra window open just to enter a value, others may prefer having a 'wizard' like input frame.

LISTOFVALUES CLASS IMPLEMENTATION

The mode of development chosen for the sample class was scenario 3 - A .class entry and a .scl entry. Additionally, examples of event handling in a *classname*_methods SCL entry are shown, along with an example of custom events and handling.

Assume these are given:
Library `EXAMPLE` exists.
Catalog `EXAMPLE.SUGI26` exists.

CREATING THE SUB-CLASS

Use Explorer to open catalog `EXAMPLE.SUGI26`.
Right click and select catalog **`New...`** from the menu.
Choose the **`Class`** icon and click **`OK`**.
This will open the Class Editor. Enter these values:
Description: **`User Maintained List of Values`**
Parent Class: **`sashelp.fsp.composit`**
The Class Editor will update itself with the parent class attribtes.
Save your work so far as a class entry. Use the menu **`File / Save`** (or **`Save`** icon in the toolbar), name the class **`ListOfValues`**.

ADD NEW PUBLIC ATTRIBUTES

Add these character attributes:

```
valuesAddDialogTitle
valuesAddDialogPrompt
valuesAddDialogMaskCharacter
valuesListBoxTitle
  Set CAM=setcamValuesListBoxTitle
valuesSLISTEntryName
  Set CAM= setcamValuesSLISTEntryName
```

The setcam... methods will affect changes in the appropriate composite components before the attribute value is assigned.

OVERRIDE METHODS

Override this method:

```
_term
  Source Entry= Example.Sugi26.Listofvalues.Scl
  Source Label=term
```

ADD NEW EVENT

Add this event:

```
valuesListBox ContextMenuRequested
  Send=Manual
```

ADD NEW EVENT HANDLER

Add this event handler:

```
Event Name=valuesListBox ContextMenuRequested
Method Name=onValuesListBoxContextMenuRequested
```

NEW METHODS

At this point there should be three new methods.

```
onValuesListBoxContextMenuRequested
setcamValuesListBoxTitle
setcamValuesSLISTEntryName
```

All the methods will be implemented within a USECLASS / ENDUSECLASS block.

EVENT HANDLING SCHEMES

Two techniques are going to be implemented. All involve overriding _on*Event* methods of each the composite components. All the override methods will reside in an SCL entry named *classname*_methods.

The first technique is coding all the nuts and bolts of the event handling in methods in the *classname*_methods SCL entry. All objects in the composite can be determined using _self_.ownerId._getWidgets().

The second technique is to use the event handling method in *classname*_methods to generate a custom event against _self_.ownerId. The custom event is then processed by its event handler.

LISTOFVALUES COMPONENTS

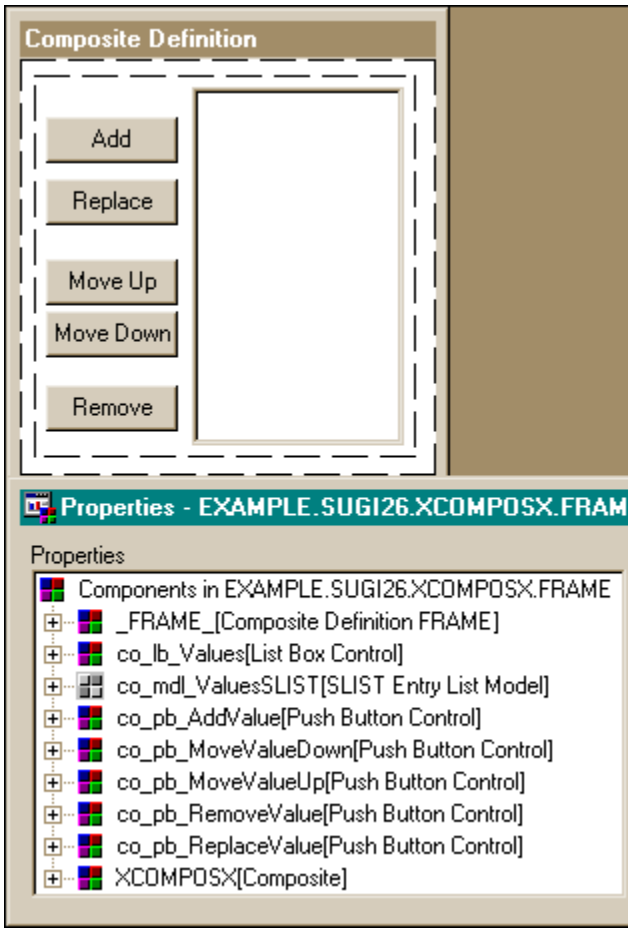
INITIALIZING COMPONENTDEFINITION

Locate the componentDefinition attribute in the System category. Edit the attribute's current Initial Value by clicking on the elipsis (...); this will open the Composite Definition window. A screenshot of this operation is located at the end of the paper

LAYING OUT THE COMPONENTS

Place the following components (**name** [*type*]) in the Composite - Definition window inside the dash outlined rectangle (an automatic component **XCOMPOSX** [*Composite*])
co_lb_Values [*List Box Control*]
co_mdl_ValuesSLIST [*SLIST Entry List Model*]
co_pb_AddValue [*Push Button Control*]
co_pb_ReplaceValue [*Push Button Control*]
co_pb_MoveValueUp [*Push Button Control*]
co_pb_MoveValueDown [*Push Button Control*]
co_pb_RemoveValue [*Push Button Control*]

Using conventions for object names makes reading implementation code much easier.
Note the naming convention:
co_ implies an object within a composite
pb_ implies a push button control
lb_ implies a list box control
mdl_ implies a model

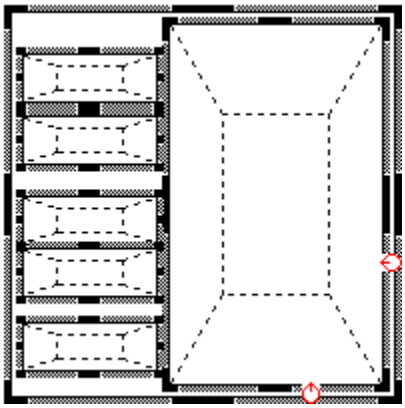


ListOfValues Components in Composite Definition Window

COMPONENT ATTACHMENTS

If the composite is to be reuseable, it must be smart enough to resize itself appropriately when an instance of itself is placed in a frame. (The instance itself may be attached to other component's in a frame, which would cause a redraw of the composite to occur)

The attachments of the composite's components are defined by selecting the **XCOMPOSX** [*Composite*] component and using command `RM ATTACH` (or menu choice `Layout / Attach / Define Attachment`)

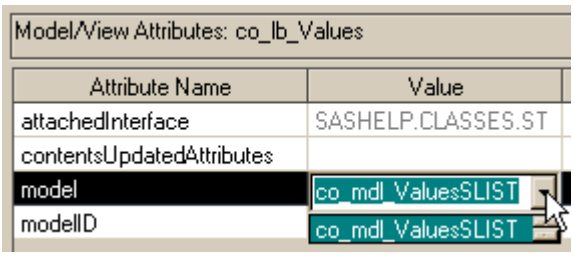


Composite's Component Attachments

The list box component is attached to the right side and bottom side of the composite using single directional attachment types. Hence, when an instance of the composite is made larger, the list box will enlarge to fit in the new size.

COMPONENT MODEL/VIEWER LINKAGES

The **co_lb_Values** *model* attribute is set to **co_mdl_ValuesSLIST**. When the model object is changed an event is fired that is handled by all objects using that model. In the ListOfValues sample, when the **co_mdl_ValuesSLIST** *entryName* attribute (the four level SAS catalog SLIST entry) is changed, the entry is read and the model fires an event. The **co_lb_Values** handles the event by setting its *items* attribute to that of **co_mdl_ValuesSLIST's**, which in turn causes the list shown to be redrawn.



Setting the Listbox's model attribute

The above screenshot shows a portion of the Properties window which was brought up when in the CompositeDefinition window. The dropdown list will only show objects in the composite that support the interfaces required by the object being edited.

OVERRIDING PUSH BUTTON _ONCLICK() METHOD

When a push button is clicked, a *click* event is generated. The push button event handler invokes method _onClick(). The **ListOfValues** class will respond to its internal push button clicks only if the push buttons in the composite specify an _onClick() method override. Where you want the override methods to reside impacts how you code the overrides. Here are two feasible possibilities:

- The override methods are coded in the same SCL entry as the composite class (*classname.scl*.) This SCL can **not** have a USECLASS / ENDUSECLASS block for two reasons:
  - The _onClick() method override name has not been added to the composite class
  - Even if the _onClick() override method name is added to the composite class, when the push button click event fires and the override method is run, the component doing the firing is not the same as the component listed in the USECLASS statement. The AF executor system recognizes this and will halt the frame.

  If this scenario is used the composite components can obtained using `_self_._getWidget(widgetName, widgetId)`
  The *widgetId* can be assigned to a variable declared as an object and that variable can be used for dot notation.
- The override methods are coded in a single SCL entry named similar to the composite class (*classname*_methods.scl.) There are two considerations:
  - If USECLASS sashelp.classes.pushbutton_c is desired, all overrides have to point to a _onClick() method, and the method implementation would have to dispatch to other methods based on the *name* attribute.
    This is a wasteful redundant implementation. Perform the proper dispatching by overriding each _onClick() to a different method.
  - Implementation is more clear if each component overrides the _onClick()method to one named _onClick*ComponentName()*, and have all these methods reside in a single SCL entry. However, this precludes a USECLASS statement since the component named in the USECLASS statement will not have the _onClick*ComponentName()* method already declared.

Component methods and attributes are referenced through the automatic _self_ variable.

COMPONENT METHOD OVERRIDES

For all the overrides, the Source Entry metadata will be "Example.Sugi26.Listofvalues_methods.scl".

```
co_lb_Values
   _onPopup(), Source Label=onPopupValues
co_pb_AddValue
   _onClick(), Source Label=onClickAdd
co_pb_ReplaceValue
   _onClick(), Source Label=onClickReplace
co_pb_MoveValueUp
   _onClick(), Source Label=onClickMoveUp
co_pb_MoveValueDown
   _onClick(), Source Label=onClickMoveDown
co_pb_RemoveValue
   _onClick(), Source Label=onClickRemove
```

EXAMPLE.SUGI26.LISTOFVALUES_METHODS.SCL

This is the onClickAdd method demonstrating how _self_ is used to perform explicit introspection:

```
onClickAdd: protected method return=num;

  dcl char aValue;
  dcl num rc;

  dcl example.sugi26.listofvalues lov
  = _self_.ownerId;

  dcl object co_lb_values;
  lov._getWidget('CO_LB_VALUES',co_lb_values);

  call display (
    'Example.Sugi26.EnterValue.frame'
  , aValue, rc
  , lov.valuesAddDialogTitle
  , lov.valuesAddDialogPrompt, '',''
  , lov.valuesAddDialogMaskCharacter );

  * insert value if not already in the list;
  if rc = 0 and aValue ne '' then do;
    if not searchc (co_lb_values.items, aValue)
    then do;
      rc = insertc (co_lb_values.items,
                                  aValue, -1);
      co_lb_values._refresh();
    end;
  end;
endmethod;
```

The other onClick*ComponentName* methods are similarly coded; _self_ is used to get the ownerId (which is **ListOfValues**), and _getWidget is used to get the objectId of the listbox.

This is the onPopupValues() method generating a custom event:

```
onPopupValues: protected method return=num;

  dcl example.sugi26.listofvalues lov
  = _self_.ownerId;

  lov._sendEvent
  ('valuesListBox ContextMenuRequested');
endmethod;
```

EXAMPLE.SUGI26.LISTOFVALUES.SCL

This is the *valuesListBox ContextMenuRequested* event handler method (defined within a USECLASS Example.Sugi26.Listofvalues block.) The **ListOfValues** class attributes have been bolded for emphasis.

```
onValuesListBoxContextMenuRqstd:
public method;

  dcl list _menu = {};
  dcl num rc choice;

  rc = insertc (_menu,valuesAddDialogPrompt);
  if co_lb_values.selectedIndex > 0 then
    rc = insertc (_menu, 'Remove ' ||
quote(co_lb_values.selectedItem) ||' from the
list', 2);

  choice = popmenu (_menu);
  rc = dellist (_menu);

  select (choice);
    when (1)
co_pb_addValue._sendEvent('click');
    when (2)
co_pb_RemoveValue._sendEvent('click');
    otherwise;
  end;
endmethod;
```

CLASS SETCAM... METHODS

When the class attribute ValuesListBoxTitle is assigned a value, its setcam... method will run. The method simply assigns the list box component's title attribute to be the same value. The setcamTitle method of the listbox will visually render the new title.

```
setcamValuesListBoxTitle:
protected method
  attributeValue:update:char
  return=num;

  co_lb_values.title = attributeValue;

  return 0;
endmethod;
```

When the class attribute ValuesSLISTEntryName is assigned a value, its setcam... method will run. The method performs checks to ensure the value is a valid catalog entry name, and that the SLIST catalog entry can be created if necessary. Finally the SLIST entry is read from the catalog.

```
setcamValuesSLISTEntryName:
protected method
  attributeValue:update:char
  return=num;

  dcl list _null;
  dcl char libname memname objname objtype;

  * When the SLIST entry name is cleared,
  * clear the list box;

  if attributeValue = '' then do;
    co_mdl_valuesSLIST.entryName
    = attributeValue;
    co_lb_values._refresh();
    return 0;
  end;

  * Ensure the attribute value is a four
level SLIST (that can be created if neces-
sary);
```

```
  libname = scan (attributeValue,1,'.');
  memname = scan (attributeValue,2,'.');
  objname = scan (attributeValue,3,'.');
  objtype = scan (attributeValue,4,'.');

  if objtype = '' then objtype = 'SLIST';

  if libname = '' or memname = '' or
     objname = '' or upcase(objtype)^='SLIST'
  then do;
    put attributeValue
       'is not a four level .SLIST';
    return 1;
  end;

  attributeValue = libname || '.' || memname
|| '.' || objname || '.SLIST';

if not cexist (attributeValue) then do;
    _null = makelist ();
    if 0 ne savelist ('CATALOG',
                      attributeValue, _null)
then do;
      put 'WARNING: Can not save a list to'
          attributeValue;
end;
    _null = dellist (_null);
  end;

if cexist (attributeValue) then do;
    co_mdl_valuesSLIST.entryName
    = attributeValue;
    co_lb_values._refresh();
    return 0;
  end;
  else
    return 1;
endmethod;
```

Some very important hidden processing is going on here. When the model's (co_mdl_valuesSLIST) entryName attribute is assigned, the model's setcamEntryName() runs, loading the SLIST from disk to memory. Additionally, since the model is connected to a viewer (the list box), when the model changes itself, the list box is notified and the list box visually renders the current list of values in the model as items in the listbox.

OVERRIDDEN _TERM METHOD

Persistence of the list of values is the end goal of the **ListOfValues** class, hence at some point the values have to be written to disk. The most appropriate place for doing so is when the class is being terminated as the frame is being shutdown.

```
TERM: PUBLIC method;
  dcl num rc;

  if co_mdl_valuesSLIST.entryName ne '' then
    rc = savelist ('CATALOG'
          , co_mdl_valuesSLIST.entryName
          , co_lb_values.items);

  _super ();
endmethod;
```

CONCLUSION

The composite class is a very useful asset to the AF developer. A deep understanding of the new technology in SAS Version 8 is needed to properly assess when and how often to use this class.

RECOMMENDED READING

SAS Institute Inc., *SAS® Guide to Applications Development, First Edition (55888),* Cary NC, SAS Institute Inc., 1999.

Bud Whitmeyer, Observations Volume 5, Number 1: *Using Region Attachments Effectively in a FRAME Application*, Cary NC, SAS Institute Inc.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1999.

ACKNOWLEDGEMENTS

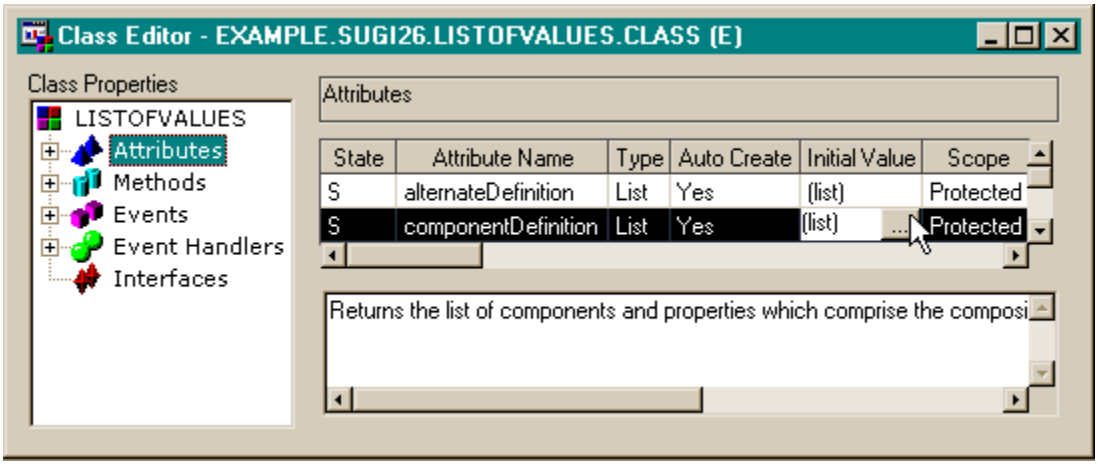
SAS and SAS/AF are registered trademarks or trademarks of SAS Institute Inc, in the USA and other countries. ® indicates USA registration.

AUTHOR INFORMATION

Richard A. DeVenezia
9949 East Steuben Road
Remsen, NY 13438

radevenz@ix.netcom.com
http://www.devenezia.com

A sample application will be available at the author's website.



How the Composite Definition window is invoked