Paper 4

SAS® Component Object Model (SCOM) in Version 7 of SAS/AF® Software

Tammy Gagliano, SAS Institute Inc., Chicago, IL
Sue Her, SAS Institute Inc. Cary, NC

ABSTRACT

What does it mean to design a new class in SAS/AF software so that it fully exploits the new SAS Component Object Model (SCOM) architecture? What benefits do the new features bring? How can you incorporate these new features into existing applications? Is it worth your time?

This paper will answer these questions and more while describing the new SCOM architecture in detail. Examples will be shown to illustrate how it can be used to reduce the time it takes to develop applications as well as improve your application's performance.

INTRODUCTION

SCOM is an object-oriented programming model that provides a flexible framework for SAS/AF component developers. A component in SCOM is a self-contained, reusable object with specific properties, including

- A set of attributes that define certain characteristics about the component such as its description, color, size or any other data that is stored with the component
- A set of methods which are the operations that can be executed on the component
- A set of events that the component sends
- A set of event handlers that the component executes in response to various types of events
- A set of supported or required interfaces which indicate its ability to establish model/view relationships with other components

With SCOM, you can develop plug-and-play components that adhere to simple communication rules, which in turn make it easy to share information between components. You can design components that communicate with each other using any of the following processes:

Attribute linking	Enabling a component to change one of its attributes when the value of another attribute is changed.
Model/view Communication	Enabling a viewer (typically a visual control) to communicate with a model based on a set of common methods that are defined in an interface.
Drag and drop Operations	Enabling information to be transferred from one component to another by defining "drag" attributes on one component and "drop" attributes on another.
Event Handling	Enabling a component to send an event that another component can respond to by using an associated event handler.

The remainder of this paper is written from a component developer's view, illustrating various aspects of the SCOM architecture and how to exploit those features in the components you design. This paper assumes you have had some previous experience creating new classes (i.e. subclassing) in SAS/AF software or in another object-oriented development tool.

MANAGING PROPERTIES AND THEIR METADATA

Each property on a class whether it is an attribute, method, event, event handler or registered interface has additional characteristics that are stored with that property. These characteristics are referred to as its *metadata*. Metadata further defines the behavior or functionality that a specific property has to offer.

Since so much of the SCOM functionality and performance improvements made to Version 7 revolve around the new metadata, a lot of effort was put into improving the user-interface (UI) of the build environment tools to manage this data. Specifically, you can use

- the Class Editor to add, manage and delete properties at the class-level
- the Properties window to manage per-instance properties on an object in a frame

At the same time, the SAS Component Language (formerly referred to as Screen Control Language) was enhanced to match the functionality of the UI. One of the biggest enhancements made to the language for Version 7 is the ability to create and save a class entry in batch using class syntax. Using the new CLASS/ENDCLASS statement block, you can create a block of statements that completely define and implement a class. For example,

```
class sasuser.sample.my.class
     extends sashelp.fsp.object
     / ( Description="My Sample Class");
  public  char description
     / (state="O");
  public  list employees;
  countEmployees: method return=num
     / (scl='sasuser.sample.my.scl');
endclass;
```

To create a class from an SCL entry containing class syntax, you simply use the SAVECLASS command or choose File → Save as Class from the SCL entry's menu .

The above will create `sasuser.sample.my.class` which is a subclass of `sashelp.fsp.object.class`. It contains a new attribute named `employees` which is of type list. The class also has a new method named `countEmployees`. Class syntax allows for the method implementation to be included within the CLASS/ENDCLASS block as well; however, in this example, the method code for the new method is stored in a separate SCL entry named `sasuser.sample.my.scl`.

You can generate class syntax for an existing class from within the Class Editor using File → Save as... from the menu and storing the class out to an SCL entry. Or, you can programmatically generate the class syntax using the new CREATESCL function.

```
rc = createSCL('lib.cat.className.class',
         lib.cat.yourSCL.scl, 'description');
```

The above creates an SCL entry named `lib.cat.yourSCL` that contains the class syntax for the CLASS entry named `lib.cat.className`.

Most of the examples discussed throughout the remainder of the paper illustrate the Class Editor as the primary tool for managing class properties. For more details on class syntax, refer to the on-line help.

ATTRIBUTES

Unlike previous releases of SAS/AF software where methods were the key properties that controlled an object's behavior, attributes are the focal point for much of the internal workings of SCOM. It's primarily through the use of attributes that you can enable your component to communicate with other components through attribute linking, drag and drop and model/view.

When you define an attribute on your class, you can define the following metadata:

Attribute Metadata	
Item	**Description**
name	the name of the attribute
description	a short description for the attribute which appears as help information in the Class Editor and Properties windows
type	specifies the type of data stored which can be character, numeric, list, generic object, specific class name object, or an array
state	specifies whether the attribute is new (N), inherited (I), overridden (O) or a system (S) attribute
category	specifies a logical grouping for the attribute used by the Class Editor and Properties window to group related attributes
initialValue	specifies the initial value of the attribute
validValues	returns a list of valid values for the attribute. Available only for character type attributes. Valid values can be specified as a string of values separated by commas or blanks, the name of an SLIST entry, or the name of an SCL entry. If an SCL or SLIST entry is specified, it must be proceeded by a backslash character (e.g. validvalues = '`\lib.cat.entry.scl`').
editor	specifies a FRAME, PROGRAM or SCL entry designed by the component developer as a dialog to assist the user in selecting a value for the attribute. Available only for character, numeric, and list type attributes.
autoCreate	specifies if the attribute is automatically created and deleted by SAS/AF. Valid for list and specific class name object types only
scope	controls permission level for accessing the attribute: public, protected or private
editable	indicates whether the attribute value can be modified or just queried
linkable	specifies whether the attribute can receive its value through an attribute link
sendEvent	specifies whether the attribute automatically sends an event when modified (i.e., "*attributeName* changed")
textCompletion	specifies whether user-supplied values for the attribute are matched against items in the validValues metadata for text completion
honorCase	specifies whether user-supplied values must match the case of items in the validValues list
setCAM	specifies the name of the custom access method to be invoked when the attribute value changes
getCAM	specifies the name of the custom access method to be invoked when the attribute value is queried

Using the Properties Window to Change Attribute Values

In earlier releases of SAS/AF software, each object in a frame had its own Object Attribute window. At build-time, you'd use these separate windows to set certain characteristics on each object.

In Version 7, a new Properties window is available for use by all objects. It offers many advantages over defining individual object attribute windows.

- All components in a frame, including non-visual components, can be managed through the Properties window.
- The end-user sees a consistent user-interface when interacting with components in a frame.
- The component developer no longer has to create and maintain individual object attribute windows.
- As new properties are added to the class, these properties automatically display in the Properties window with no extra work on the part of the component developer.
- The Properties window enables users to interact with all properties of a component (attributes, methods, events and event handlers).
- The same attribute across multiple components can be changed in a single step using the multiple-selection feature.
- Per-instance properties can be added using the Properties window as well.

The user can change the value for an attribute using the Properties window by typing directly in the *Value* cell of the Attributes table. Depending on the other metadata defined for the attribute, they might also have the choice of several selectors within the cell to assist them. Specifically, if the component developer has specified either of the following metadata items, the Properties window will use this information as follows:

VALIDVALUES

If supplied, is used by the Properties window (when editing the *Value*) and the Class Editor (when editing the *Initial Value*). SAS/AF automatically displays a combo box for the user to choose a value. The valid values list is used to populate the list of items displayed in the combo box. The value chosen from the list is returned to the field and stored on the attribute.

EDITOR

If supplied, is used in the Attributes table by the Properties window (when editing the *Value*) and the Class Editor (when editing the *Initial Value*). SAS/AF displays an ellipses button. When the user clicks on the button, the editor is automatically launched. The value specified in the editor is then returned and stored on the attribute.

Since both of these metadata items are used internally by SAS/AF, there are rules that must be followed by the component developer when designing them.

Example: You have two attributes on your component: `table` and `column`. For the `table` attribute, you want the user to specify the name of a SAS data set. For the `column` attribute, you want them to specify the name of a column that exists in the chosen `table`.

First, let's create the EDITOR that will be used for the `table` attribute. You create an editor the same way you'd create any FRAME or SCL entry. It behaves exactly the same, except that you must include an ENTRY statement using the exact SCL variable names as defined below.

```
entry optional= objectId:object
   classId:object
   environment:char(2)
   frameId:object
   attributeName:char(32)
```

```
attributeType:char(83)
value:char(41)
mode:char(1);
```

Where...	*Represents...*
objectID	the object identifier of the current object whose attribute is being edited when invoked from the Properties window. When invoked from the Class Editor, the value will be zero since there is no instance of the object.
classID	the class identifier of the class used to create the current object when invoked from the Properties window. In the Class Editor, it contains the identifier of the class that's currently loaded and displayed in the Class Editor.
environment	the environment from which the editor is being invoked: "CE" for Class Editor, "PW" for Properties window
frameID	the object identifier of the active frame where object exists when invoked from the Properties window. From the Class Editor, this value will be zero since there is no frame currently being built.
attributeName	the name of the attribute that is being edited
attributeType	the type of the attribute, which can be Numeric, Character, or List
value	the current value of the attribute. The type of this argument may be either a string or number depending on the attributeType of the attribute
mode	whether the attribute is editable: "E" indicates it is editable

The new syntax used on the ENTRY statement to declare the types of each incoming variable is new to Version 7. Refer to the on-line help for more details.

When designing your editor, you may or may not use all of the above variables that get passed in on the ENTRY statement. Remember, an editor can potentially get invoked from the following places:

- The *Value* cell in the Attributes table in the Properties window as well as from the same field in the dialog that displays when using *Add..., Edit...,* or *Override...* from the table's popup menu
- The *Initial Value* cell in the Attributes table in the Class Editor as well as from the same field in the dialog that displays when using *Add..., Edit...,* or *Override...* from the table's popup menu

You might not necessarily perform the same tasks if being invoked from the Class Editor versus Properties window. The `environment` variable tells you where your editor is being used. Knowing that information, you might conditionally query back to the frame to get additional information from the frame (using `frameID`) or from the object whose attribute is being edited (using `objectID`).

In our example, the core of the editor implementation is simple and would function properly regardless of where it's being used:

```
init:
  value = openSASFileDialog('DATA VIEW');
return;
```

We take `value`, which is passed into the editor and contains the current value of the `table` attribute at the time. By setting it as the return value for the function, it gets passed into the selector window, which uses that value to initialize itself. When the selector window is closed, the 2-level name of the selected table gets placed back into the `value` variable. This value in turn gets returned to the Properties window and stored on the object.

Now, let's create the VALIDVALUES list for the attribute. The list of values is dynamic. Meaning, we cannot store them on the class because we have no idea what table the user will specify. Instead of entering the values directly, we'll write an SCL program that dynamically creates the list of column names at run-time.

Similar to the editor, there is a required ENTRY statement that must be used when designing SCL entries that are assigned as the VALIDVALUES metadata.

```
Entry list:list
      optional = objectID:object
                 attributeName:char(32)
                 environment:char(2);
```

Where...	*Represents...*
list	The identifier of the list that contains the valid values. All values on this list must be character values
objectID	Represents the object identifier of the current object whose value is being edited when invoked from the Properties window. When invoked from the Class Editor, it contains the identifier of the class that's currently loaded and displayed in the Class Editor.
attributeName	the name of the attribute that is being edited
environment	the environment from which the VALIDVALUES entry is being invoked: "CE" for Class Editor, "PW" for Properties window

Similar to the editor, the VALIDVALUES metadata for an attribute gets used internally by SAS/AF in several ways:

- To display the items in a combo box control from the same places the editor is used when setting the value or initial value for an attribute (refer to list provided earlier on exact locations)
- Additionally, the _setAttributeValue method uses the VALIDVALUES metadata item. Refer To the section labeled *Behind-the-Scenes With Attributes* later in this paper for more details on when _setAttributeValue executes.

In our example, we need to define an SCL program that generates the list of column names from a specific table. Along with the required ENTRY statement (see above), the remainder of the implementation is as follows:

```
❶Import
    sashelp.classes.variablelist_c.class;
init:
  ❷dcl char(41) dsname, num rc;
  ❸if environment = 'CE' then do;
    ❹dcl list attrlist = makelist();
    ❺objectId._getAttribute('table',
                attrlist);
    dsname = getnitemc(attrlist,
              'initialValue',1,1,'');
    rc = dellist(attrlist, 'Y');
  end;
  ❻else dsname = objectID.table;
  rc = clearlist(list);
  if dsname ne '' then do;
    ❼dcl variableList_c modelID = _new_
          variablelist_c(),
    ❽modelID.dataset = dsname;
    ❾list = copylist(modelID.items,'',list);
    ❿modelID._term();
  end;
return;
```

❶ The IMPORT statement specifies a search path for

CLASS entry references in an SCL program. Throughout the remainder of the program, `sashelp.classes.variablelist_c.class` can be referred to in short hand by simply referencing the class name, which is `variableList_c`. See step 7.

❷ The DCL statement (short for DECLARE) also new to Version 7 is used to declare variables and specify their data types. It should be used in the future instead of the LENGTH statement as it offers more functionality. For more information on how to use this statement, refer to the online help.

❸ We're checking the value of the `environment` variable to see where the entry is being used.

❹ The DCL statement can be used anywhere in an SCL program. Here, we're declaring the variable `attrlist` as an SCL list. In the same step, we're also creating the list using the MAKELIST function. The new list is used as input to the method call on step 5.

❺ The syntax shown here, `objectID._getAttribute` , is referred to as dot notation. We're using dot notation to invoke the `_getAttribute` method on `objectID`, which contains the object identifier of the loaded class. This method returns a list of the `table` attribute's metadata, including its INITIALVALUE if one was specified. This value gets stored in the local variable `dsname` and used later in step 8.

❻ Otherwise, the entry is being called on the instance of the object. Using dot notation again, we are retrieving the current value of its `table` attribute and storing the value in the local SCL variable DSNAME.

❼ The _new_ operator is used to create an instance of the variableList_c class and store its object identifier in a local SCL variable `modelID`. VariableList_c is a new, non-visual component. It has an attribute named `dataSet` which when specified, generates the list of variable names that exist on that data set and stores that list in an attribute named `items`. These attributes are used later in steps 8 and 9.

❽ Use `modelID.dataset` to set the value of the `dataset` attribute on the `modelID` object to the current value stored in the `dsname` variable.

❾ Use `modelID.items` to retrieve the list of variable names. The list retrieved is then copied into the `list` variable. Because the `list` variable is on the ENTRY statement, its value gets returned to the calling program. SAS/AF will then use it as the contents of the drop-down list in the Properties window or Class Editor or for validation purposes whenever a _setAttributeValue occurs on the attribute.

❿ Because we have instantiated the non-visual model programmatically in our SCL, we also need to perform proper clean up by invoking the `_term` method on the object. Dot notation was used here again to invoke the method on the model.

What Is Dot Notation?

In the previous SCL example, there were several places where dot notation was used to change the value of an attribute on an object, retrieve its value, or invoke a method. What exactly is dot notation and what can it do for you?

In previous releases of SAS/AF software, to query an object for information about its current state or attribute settings, you'd find yourself writing a mixture of SCL statements -- maybe some method calls, maybe some SCL list function calls etc. For example:

```
list = makelist();
call notify('obj1','_get_region_', list);
title = getnitemc(list, 'border_title');
style = getnitemc(list, 'border_style');
color = getnitemc(list, 'border_color');
rc =  dellist(list);
put title= style= color=;
```

In Version 7, the above code would look like:

```
put graphOutput1.borderTitle=
    graphOutput1.borderStyle=
    graphOutput1.borderColor=;
```

Notice the number of lines have been reduced. More importantly, there is a significant difference in the coding style. This example clearly illustrates the user-friendliness of dot notation. The approach to both setting and querying an object for information is consistent and straightforward. To set the value of an attribute, you'd follow similar conventions:

```
graphOutput1.title = 'Sample title';
graphOutput1.borderStyle = 'embossed';
graphOutput1.borderColor = 'black';
graphOutput1.graph='lib.cat.entry.grseg';
```

The syntax is exactly the same across all components. Also note the object name 'graphOutput1' is longer than eight characters and is used as the first level qualifier in the dot notation.

By default, when you add a new Version 7 component to a frame, the object name automatically references the object's ID, not its value. These components are automatically enabled for use with dot notation programming style within frame SCL.

This programming style can be used with legacy objects as well; however, you must either

- specify "ID" as the value for the object's `objectNameUsage` attribute
- or in your SCL program, declare the variable that contains the object identifier of the legacy object as an object

Otherwise, the object name represents the value of the object itself, not the object identifier. This is for compatibility purposes. Also keep in mind that legacy classes were not designed under the SCOM architecture. So while you may be able to use dot notation to invoke methods on a legacy object, there may be other enhancements needed in order for your legacy class to totally function as an SCOM component (i.e. embrace attribute linking and model/view as communication mechanisms for example). Refer to the SUGI23 paper entitled *Dressing Up Your Version 6 Objects to be Version 7 Components* for more details on this subject.

To declare a variable as an object within your SCL program so that dot notation can be used, you use the DCL statement. There are two types of object declarations. You can declare a variable as a generic object or as an object of a specific class name. The example below illustrates both:

```
dcl object dataTableID,
    sashelp.classes.colorList_c.class
colorID;
```

The `colorID` variable is declared as an object of a specific type. Specifically, you are stating that it will be an instance of the `sashelp.classes.colorList_c.class`. Instead of a keyword (like OBJECT, CHAR, LIST or NUM) you use the four-level name of the class to define its type.

You should always declare your objects using this approach when possible. Doing so improves performance and enables the compiler to validate all tasks performed on that object. For example, if you are trying to access an attribute that does not exist on the object or if the method arguments used are incorrect, the compiler can catch these errors. Previously, you'd have to do your debugging at run-time, which is slower. See the section labeled *Method Signatures* for more discussion on invoking methods using dot notation.

There are situations, however, where you may not be able to declare an object by its specific type. This is what is referred to as declaring a generic object. The `datatableID` variable in the above code illustrates this feature. You'd use this in situations where you do not know the object type. Another example of when you'd need to use this feature is when the object uses method delegation or has been designed with per-instance properties. Per-instance and delegated methods, while legal to invoke on the component, will not be recognized by the compiler because they do not exist in the class definition itself. The compiler will post an error when it encounters one of these methods. An example of a component that uses delegation is the Data Table, which is a legacy class. Therefore, if you have a variable in your program that references an instance of the Data Table class, you must declare it as a generic object in order to use dot notation to invoke methods on the object.

Using Attribute Linking

Aside from setting the value of an attribute using dot notation, attribute linking is yet another way for an attribute to obtain its value. Attribute links enable components to interact without the need for any SCL code. Instead, interactions are specified with the Properties window.

Attribute linking involves setting the *Link To* value of a component's attributes. For example, you have two controls on a frame: a list box control and a graph output control. The list box displays a list of 4-level GRSEG entry names. At run-time, when the end-user selects an item in the list box, you want the graph to automatically display in the graph output control. To set this behavior at build-time, simply go to the *Link To* cell for the graph control's `graph` attribute and link it to the `selectedItem` attribute of the list box control. The Properties window provides you with a selection dialog to quickly establish these links. And that's it!

You get the rest of the behavior for free as long as the class developer has specified the correct attribute metadata settings on the class.

- To enable an attribute to receive its value via an attribute link, specify "Yes" for the attribute's LINKABLE metadata item. In this case, the graph output control has a `graph` attribute that has LINKABLE="Yes". If the value for this metadata item is set to "No", no attribute links can be set and the Properties window automatically disables the *Link To* cell for this attribute.
- To enable an attribute to function as a source attribute (that is, the attribute to which you link to) the attribute's SENDEVENT metadata item should be set to "Yes". In this case, the `selectedItem` attribute on the list box control is the source attribute. Its metadata is set to SENDEVENT="Yes".

Refer to the section later in this paper labeled *Events and Event Handlers* for more detail on attribute links and how they work behind-the-scenes.

Behind-the-Scenes With Attributes

It is important once you begin using attributes, that you understand there is additional processing that takes place automatically by SAS/AF software whenever an attribute's value is queried or set. Basically, when you use dot notation, the statement gets translated internally to a _getAttributeValue or _setAttributeValue method call (depending on whether you are getting or setting the value).

These methods are inherited from Object class and control a lot of the SCOM functionality that you get for free in a component. These methods are the backbone for much of the behavior attributes provide.

For example, the _setAttributeValue method executes whenever dot notation is used to change the value of an attribute. The method

- verifies the attribute exists
- checks to make sure the attribute is not being accessed outside of its defined scope
- verifies the value being set matches the actual type of the attribute
- validates the value being set against the VALIDVALUES list (if one exists in the attribute's metadata)
- invokes the setCAM method (again, if one has been defined in the attribute's metadata).

If none of the above conditions produce an error, _setAttributeValue continues processing and

- stores the value of the attribute
- sends an '*attributeName* changed' event if the attribute's SENDEVENT metadata is set to "Yes" (this step is crucial in order for attribute linking to function properly)
- sends a 'contents Updated' event if the attribute exists on the object's contentsUpdatedAttributes attribute (this event is the key behind model/view communication and will be discussed in more detail later in this paper)

Similarly, the _getAttributeValue method executes when dot notation is used to query the value of an attribute. The method

- verifies the attribute exists
- checks to make sure the attribute is not being accessed outside of its defined scope
- verifies the type of the attribute matches the type of the variable that will contain the retrieved value
- invokes the getCAM method (if one has been defined in the attribute's metadata)

If none of the above conditions produce an error, the method then retrieves and returns the value of the attribute.

METHODS

With so much emphasis on attributes, you might think that methods are no longer important properties on a class. That is not true. Methods are used when you need to

- perform additional processing when an attribute's value is changed or queried by adding a new method and assigning it as the attribute's setCAM or getCAM metadata
- add completely new behavior to a class, which represents an action on a component versus something that can be specified through an attribute setting. For example,

```
rc = obj1.printYourself();
```

And, just like attributes, methods have their own metadata as well.

Method Metadata	
Item	**Description**
name	the name of the method
description	a short description for the method
state	specifies whether the method is new (N), inherited (I), or overridden (O)
entry	is the name of the SAS/AF catalog entry that contains the method implementation. Typically this item is an SCL entry
label	is the name of the SCL labeled section where the method is implemented
signature	is a list of sublists that correspond to the arguments in the method signature
scope	Controls permission level for accessing the method: public, protected or private
enabled	determines whether a method can be executed

New Naming Conventions

The first two things you might notice about method names when you edit a class in Version 7 using the Class editor are

- fewer underscores
- mixed casing

In Version 6, SAS/AF used underscores to separate words in method names (e.g. `_set_backround_color_`). In Version 7, the new convention is to use a lowercase letter for the first letter and subsequent uppercasing of any joined word (such as `_setBackgroundColor`).

The embedded underscores have been removed to promote readability. However, for compatibility purposes, `_setBackgroundColor` is equivalent to `_set_background_color_`. All existing Version 6 code that uses the old style naming conventions along with CALL SEND and NOTIFY routines will still function with no modification. The new naming convention, however, must be used with dot notation.

While it is possible in Version 7 for you to name new methods using a leading underscore, you should use caution when doing so. Your method names may conflict with future releases of SAS/AF software as new SAS-supplied methods are added to the parent classes.

Using Dot Notation To Invoke Methods

Along with the new style of method names which improves the readability of your SCL program, using dot notation to invoke methods will also reduce your programming effort. For example, previously, you would invoke a method as follows

```
call send(obj1,'_set_text_color_','red');
```

In Version 7, you can invoke the method as follows:

```
object._setTextColor('red');
```

The above is easier to type with fewer quotes and underscores. This will hopefully lead to fewer typing mistakes.

Method Signatures

Using dot notation alone does not improve your application's run-time performance. You must declare your objects as specific class name types (discussed earlier in the section labeled *What is Dot Notation?*) and you must also define method signatures for all of the methods.

A method signature is a set of parameters that uniquely identifies it to the SCL compiler. A method signature in SAS/AF software is usually represented by a shorthand notation called a *sigstring*. This sigstring is displayed in the Signature column of the Methods table in the Class Editor. The `_setTextColor` method above would have a sigstring of (C)V, for example. If the method took two arguments, the first being a numeric argument and the second being a character argument, the sigstring would be (NC)V.

The parentheses group the arguments and indicate the type of each argument, which can be numeric, character, list, object, the four-level name of a specific class, or an array. The value outside of the parentheses represents the return argument.

What's a return argument you ask? In Version 7, methods can support return arguments. For example, the method used to retrieve the color might be invoked as follows:

```
dcl char(15) color;
color = objectID._getTextColor();
```

The sigstring for the `_getTextColor` method would be ()C. There are no arguments passed in to the method when it is invoked (i.e., no arguments listed inside the parentheses). The method instead, returns a character string (represented by the value outside of the parentheses, which indicates it returns a character value). When a sigstring shows the V character (for "void"), that indicates that no return argument is used.

When compiling a program, SAS/AF software uses the SIGNATURE metadata and displays compile errors in situations where

- the method you are trying to invoke does not exist on the class
- the arguments passed in are too many, too few or of the wrong type

It's important to note that the compile-time validation and run-time performance gains adhere to the following rules:

- These features are only supported through dot notation, not through the old-style CALL SEND or NOTIFY routines. And *only* if the compiler knows the type of the object (i.e., declaration of the object as a specific class name type or within a USECLASS/ENDUSECLASS statement block, which is illustrated later when implementing a setCAM method).
- These improvements are also not recognized for methods that display a signature of "(None)", which is supported primarily for compatibility purposes since classes prior to Version 7 contained no signature metadata. This does not mean that the method has no arguments but rather the signature for the method has not been defined in the class metadata.

Overriding versus Overloading Methods

When you override a method, you are not able to change the signature of an inherited method. While in previous releases, SAS/AF did not stop you from doing this; we have tried to put in some protection against this in Version 7. It is generally bad programming to change the signature of an already existing method. This would obviously affect users of your component and cause their applications to stop working.

Instead, you should overload a method when the situation arises where you need a different method signature for the same method. This is supported in Version 7 and is referred to as method overloading. A component is allowed to have more than one method by the same name as long as their arguments differ in number, order and/or type. When you invoke an overloaded method (using dot notation), SAS/AF checks the method arguments, scans the signatures for a match, and executes the appropriate code.

For example, if you had an overloaded setColor method on your class, it would be possible for you to do the following in an SCL program using those methods:

```
dcl char color,
    num r g b;
myobj.setColor(color);
myobj.setColor(r,g,b);
```

In the example above, the first method's sigstring would be (C)V whereas the second method's sigstring would be (NNN)V. Both methods change the object's color. The first method takes a color name as input and the second method takes numerical RGB values. The advantage of overloaded methods is that they require programmers to remember only one method instead of several different methods that perform the same function but with different data.

There are three rules to remember when overloading methods:

1. A method with a signature of "(None)" cannot be overloaded.

2. Each signature in an overloaded method can have a different return argument type, but the arguments inside the parentheses must be different for each signature. For example, the following signatures would be allowed for an overloaded method:

   sigstring1 = (NN)V
   sigstring2 = (CC)N

   But the method could not be overloaded as follows:

   sigstring1 = (NC)V
   sigstring2 = (NC)N

   In the latter example, the signatures differ only by their return type. If someone tried to invoke the method as `object.method(3,'red'),` the compiler would have no way of knowing which method implementation to invoke.
3. Use dot notation when invoking an overloaded method. Because CALL SEND and NOTIFY routines perform no signature look-up, these routines invoke the first method found on the class -- which may or may not correspond to the first method you see listed in the Class Editor Methods table and may or may not be the correct implementation.

For more detail on overloading methods and signature rules, see *SAS Guide to Applications Development, First Edition*.

Custom Access Methods (CAMs)

Throughout the paper, there has been mention of setCAM and getCAM methods. Up to this point, we know that they are metadata assigned to an attribute. We also know that they get invoked automatically by SAS/AF software whenever an attribute's value is changed or queried (through _setAttributeValue and _getAttributeValue calls). But, when would you really need to use them? And are there specific rules to follow when implementing custom access methods for your component's attributes?

Let's go back to our earlier example where we had a class with the following attributes: `table` and `column`. When the user enters the name of a SAS data set for the `table` attribute, do we do any kind of validation to ensure that the data set actually exists? By implementing a setCAM for the `table` attribute, we can do this. SAS/AF will automatically invoke this method when the value of the attribute is changed. The setCAM implementation can perform additional validation that might be necessary before allowing the change. For example:

```
❶useclass sasuser.test.myclass.class;
❷setcamTable: ❸protected method
   ❹attrval:update:char(41) return=num;
   dcl num rc;
   ❺rc = exist(attrval);
   if rc = 0 then do;
      ❻errorMessage = 'Data set does
         not exist';
      ❼return (1);
   end;
   ❽else return (0);
endmethod;
❶enduseclass;
```

❶ The USECLASS/ENDUSECLASS statement block is used to implement methods for a class and bind them to the class at compile time. The use of these statements improves code readability, maintainability, run-time performance and compile-time detection of many types of errors. The above statement binds the methods implemented to the `sasuser.test.myclass.class`. If throughout the SCL, any methods or attributes are incorrectly referenced or used, the compiler will flag the error.

❷ The method implementation begins the same as in previous releases with the exception that method labels can now be up to 32 characters. A good rule to use when writing method code is to use the method name as the label in your SCL program. Also, when naming setCAM or getCAM methods, we recommend using the name of the attribute as part of the method name, prefixed by 'setcam' or 'getcam' to indicate that it is a custom access method. We're using `setcamTable` because we're implementing a setCAM for the `table` attribute.

❸ Note the `protected` keyword used on the method statement. This indicates the scope of the method. Another good rule of thumb is to implement all getCAM and setCAM methods as protected methods. By default all methods are public. Protected means only instances of the parent class or any of its subclasses can invoke this method. This is to discourage users from invoking these methods directly when setting or getting the value of an attribute. You want them to reference the attribute, which triggers the entire sequence of steps performed by the _getAttributeValue and _setAttributeValue methods. If they were to invoke the CAMs directly, much functionality would be lost and they might get unexpected results.

❹ When writing a setCAM, the sigstring must be (*attributeType)*N where *attributeType* matches the type of the attribute that is being changed. In our example, `table` is a character attribute, so the sigstring for our setCAM method would be (C)N indicating that the argument passed into the method is a character value. A numeric return argument is also a requirement when writing a setCAM. The method statement above shows `return=num` which defines the method as having a numeric return argument.

❺Check for the existence of the SAS data set, which is passed into the method and stored in the `attrval` variable.

❻If the `exist` function fails, we want to return immediately from the setCAM and stop processing. We first need to set the `errorMessage` attribute so that an error message will be displayed to the user indicating that the setting of the attribute value failed. The `errorMessage` attribute exists on all objects and is used by _setAttributeValue upon return from the setCAM to display the error. Because we're inside the USECLASS block, we do not have to reference the attribute with the `_self_` qualifier (i.e., `_self_.errorMessage`). Within USECLASS, the compiler first validates all variables in the program against the class attribute list. It will recognize this as a legal attribute. The same compile time checking occurs for methods inside USECLASS.

❼ The return statement ends processing of the method and returns the value 1, which indicates to _setAttributeValue that an error has occurred. Any return value from the setCAM greater than zero indicates an error. _setAttributeValue will post the error and stop its processing. The attribute value will not be replaced.

❽ When your method supports a return argument, you must make sure there is a return statement from all possible exits in the code. Here, we're returning 0 to indicate the method was successful if it reaches this point. _setAttributeValue will then continue its processing and the attribute's value will successfully be changed.

In the same example we've been working on, we'd also want to add a setCAM for the `column` attribute. The reason for this is that when an attribute has been assigned VALIDVALUES, _setAttributeValue will post an error if the user tries to set the value for that attribute to be anything other than one of the items in the list of valid values. In our example, if the user tries to blank out the column name, they will get an error because of this validation. A blank value should be allowed.

We can implement our method in the same SCL entry and inside the same USECLASS/ENDUSECLASS statement block as our setcamTable method. Only one USECLASS is allowed per SCL entry. The setCAM needs to accept blank values but return a bad return code in all other cases where the value is not on the VALIDVALUES list. The method implementation is as follows:

```
setcamColumn: protected method
   attrval:update:char(32) return=num;
   if errorMessage ne '' and
      attrval ne '' then return (4);
   else return (0);
endmethod;
```

It is possible for `errorMessage` to be non-blank when entering a setCAM because of the validation that takes place by _setAttributeValue before it invokes the method. One of the steps it performs is to validate the value against the VALIDVALUES metadata list. If the value does not exist on the list, it initializes the `errorMessage` attribute with a message indicating the value is invalid at this point. However, it continues processing and invokes the setCAM.

This provides the setCAM developer the chance to perform additional validation. The return code on the setCAM method controls whether _setAttributeValue continues successfully or stops execution upon returning from the setCAM. Zero indicates success. Any value greater than zero indicates failure. There is a published set of return codes that _setAttributeValue uses. A value of four has been reserved to indicate validation failed against the VALIDVALUES list. The text stored in the errorMessage attribute will reflect this. We could have chosen to specify our own error message by changing the value of the `errorMessage` attribute before returning. Instead, we're allowing SAS/AF to use the default error message. This message will get displayed in the Properties window as well as at run-time whenever a bad value is set for the attribute.

We've shown two examples of when you'd use a setCAM, but what about getCAMs? In the event you want additional processing to occur when a user retrieves the value of an attribute, the getCAM method is where this processing can be placed.

When implementing a getCAM, the signature is the same as it would be for the setCAM. The argument passed into the method should match the same type as the attribute being queried. The return value is numeric and provides the developer of the getCAM the opportunity to stop processing of the _getAttributeValue method and not return the attribute value.

EVENTS AND EVENT HANDLERS

Events alert applications when there is a change of state. Events occur when a user action takes place (such as a mouse click), when an attribute value is changed, or when some user-defined action occurs. Event handlers provide a response to the change.

Attribute linking is a good example of how events and event handlers can be utilized as a way to communicate between components. Let's take a closer look at a simple attribute linking example. A frame contains a text entry field and a graph output control. An attribute link has been defined on the **graph** attribute of the graph output control to receive its value from (or be linked to) the **text** attribute of the text entry control. As a result, when the user enters the name of a GRSEG entry into the text field, the graph control will automatically display the graph. How does this work exactly?

Attribute Linking Behind-the-Scenes

1. When a source attribute is modified, it sends an "*attributeName* changed" event if its metadata states SENDEVENT="Yes". The event passes an object as its data. The object is an instance of the sashelp.classes.attributeChangedEvent.class, which contains attributes that provide more information about the event being sent. In our example, the "text changed" event is sent.
2. The target component has an event handler method that's listening for the above event. The event handler method is the _onAttributeChanged method. The default behavior of the _onAttributeChanged method is to set the new value on the attribute that has the link defined. In our example, the `graph` attribute's value changes to whatever was specified for the `text` attribute on the text entry control.

All of this happens internally and automatically through the attribute metadata that is defined (SENDEVENT and LINKABLE). The _setAttributeValue method is where the event actually gets sent. The event handler is established on the receiving object at the moment when the attribute link is defined. This entire behavior you get for free with SCOM.

INTERFACES

Interfaces are completely new to SAS/AF software. They are collections of abstract method definitions that define how and whether model/view communication can take place between two components. Interfaces are stored in SAS catalog entries of type INTRFACE. You create new INTRFACE entries using the Interface Editor. You then use these entries by registering them on a component in the Class Editor indicating whether the component *supports* or *requires* a specific interface. If a component has no interface metadata defined, that means the component is not enabled for model/view communication.

What is model/view communication? Model/view is another way you can implement communication between components. Components are often separated into two categories: those that display data (viewers) and those that provide access to data (models). Typically, models are non-visual entities. Viewers are typically visual things in a frame. Several components in the SAS/AF class hierarchy have built-in model/view communication. For example,

You can use any of these models...	*with any of these viewers...*
Catalog Entry List	Combo Box Control
Catalog List	List Box Control
Color List	Radio Box Control
Data Set List	Spin Box Control
External File List	
Library List	
LIST Entry List	
SLIST Entry List	
SAS File List	
Variable List	

Model/View between any of the above models and viewers enables the application developer to

1. drag a viewer from the Components window on to a frame
2. drag a model on to the viewer in frame
3. automatically see the viewer updated by data from the model as changes are made to the model.

Absolutely no code is necessary from the application developer's viewpoint. They simply drag and drop models on top of the viewers. SAS/AF software does all the rest!

As a component developer, implementing model/view can possibly require that you add significant code. To develop a component so that it can behave as a model or viewer, you must first understand what happens behind the scenes during the model/view process.

Behind-the-Scenes with Model/View

The `model` attribute exists on every object. When the `model` attribute is set (either through drag and drop or by setting the value directly in the Properties window),

- SAS/AF verifies whether the two components know how to communicate with each other by comparing the interface metadata stored on each component. If the

model has an interface stored as a *supported* interface and the viewer has the same interface stored as its *required* interface, it allows the model/view attachment to continue.

- At the time an attachment is made, an event handler is automatically defined on the viewer component to run its _onContentsUpdated method when it receives the 'contents updated' event from the model.

Both of the above steps take place in the `model` attribute's setCAM method as part of the default behavior the _setcamModel method provides. At attachment time, the viewer also needs to query back to the model to retrieve the information it needs (using methods defined in the interface) and perform whatever setup is needed. It is the responsibility of the component developer to override this method and perform whatever logic is needed here to accomplish this.

While a model/view relationship is in progress, the model sends the 'contents updated' event whenever the value for one of its key attributes change. Key attributes are defined in the `contentsUpdatedAttributes` attribute on the model. The event gets sent automatically as part of the built-in functionality of the _setAttributeValue method. When the value of an attribute changes, it looks to see if the attribute is listed on the object's `contentsUpdatedAttributes` attribute and if so, sends the 'contents updated' event. Along with the event, the name of the attribute that is changed is also sent.

The _onContentsUpdated method (the event handler) then executes on the viewer to perform any necessary actions based on the changes made to the model. The functionality here is probably similar to what's needed in the _setcamModel method at attachment time. If possible, this code should be implemented in a way that promotes code reuse by both methods.

Guidelines for Designing Model/View Components

The first thing to evaluate is whether there is an existing interface that provides the design you need. If not, create one.

When designing model components,

- Add the interface as the *supported* interface, which indicates that the model has implemented all of the methods defined in the interface
- Implement all of the methods defined in the interface.
- Create and/or identify the key attributes on the model that when modified, the viewer needs to be notified. Specify these attributes on the `contentsUpdatedAttributes` attribute.

When designing viewer components,

- Add the interface as the *required* interface, which indicates that it can use any of the methods defined in the interface and invoke them directly on the model successfully.
- Override both the _setcamModel and _onContentsUpdated methods and implement the behavior needed to query back to the model and retrieve the appropriate information it needs. In situations where the component may have more than one interface defined, you may need to query the object's `attachedInterface` attribute, which will indicate which interface is currently being used for the attached model.

For an example, refer to *SAS Guide to Applications Development, First Edition.*

ENHANCING EXISTING APPLICATIONS

All existing applications will continue to run under Version 7 with no changes, however, with minimal work your legacy classes can be enhanced to exploit many of the new features discussed throughout this paper. Taking the time to enhance these classes will not only ease your maintenance burden but it can also improve your application's performance. Specifically, you should consider making the following changes to your existing class library for performance purposes:

- Set the `objectNameUsage` attribute to "ID".
- Implement methods using USECLASS notation.
- Add method signatures to all of your methods.
- In your method implementation, declare all objects as specific class name types.
- Convert all CALL SEND and CALL NOTIFY method invocations to dot notation.

Additionally, the following changes would further exploit the SCOM architecture:

- Examine the list of instance variables your legacy class supports and implement them as attributes. This is a bigger undertaking in that you may have to implement setCAM or getCAM methods to make the implementation complete.
- After completing the above step, use the Class Settings window to set the "Use Properties window" option, which displays the object's properties through the Properties window like all other components.

For more details on enhancing your legacy class to become a true SCOM class, refer to the SUGI23 paper *Dressing Up Your Version 6 Objects to be Version 7 Components*.

CONCLUSION

As you can see, there are many new features added in Version 7. The intent of this paper was to introduce you to the new technology the SCOM architecture provides so that you can begin working these features into the designs of your new components.

The examples used in this paper were written and tested under Version 8; however, the key concepts and features described are available in Version 7 as well. If you run into any problems trying to use any of the technology described in this paper, please feel free to contact the authors.

REFERENCES

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

Version 7 SAS/AF Software - The New Component Technology written by Glen Walker and Tammy Gagliano for SUGI23.

Dressing Up Your Version 6 Objects to be Version 7 Components, written by Glen Walker and Tammy Gagliano for SUGI23.

CONTACT INFORMATION

Please feel free to contact us at:

Tammy L. Gagliano
SAS Institute Inc.
2 Prudential Plaza
Chicago, IL 60601
Work Phone: (630) 724-9496
Fax: (312) 819-6849
Email: sastlg@wnt.sas.com

Sue Her
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Work Phone: (919) 677-8000
Fax: (919) 677-4444
Email: sasshh@wnt.sas.com