Version 7 SAS/AF® Software - The New Component Technology

Glen R. Walker, SAS® Institute Inc, Cary, NC
Tammy L. Gagliano, SAS Institute Inc, Chicago, IL

ABSTRACT

The Version 7 class library has expanded to include many new components that are already enabled to perform complex tasks such as

- attribute linking
- drag and drop
- model/view

all with no programming effort on the part of the user. They are virtually plug-and-play! This is possible because they were designed to fully exploit the SAS Component Object Model (SCOM) Architecture. SCOM offers a flexible application framework that improves component development and communication. In this paper we will define what a component is and how you can build applications which utilize the new techniques for establishing communication between components.

INTRODUCTION

In Version 6 SAS/AF® software, we made significant advances in our support for object-oriented development through support for subclassing, methods, inheritance, delegation and more. We also expanded the class library to include many new and exciting classes like the Data Table, Data Form, and Process Flow Diagram just to name a few.

In Version 7, we've expanded this list to include a new breed of classes. These classes are referred to as components and are even smarter than what we've offered before! In this paper, we'll briefly summarize these components and what they bring to your desktop.

We'll also define and discuss implementation details with respect to

- What makes a class a component?
- Is there an easy way to establish communication between objects without having to resort to programming?
- Do I still have to write code in order to develop a drag and drop application?
- As a component developer, how can I write reusable code that hides complex communication issues between components?

You will find in Version 7 that you now have more choices for handling communication issues between your components – from simple problems to more complicated scenarios. All of the solutions offered eliminate much of the complex programming that might have previously been required to accomplish the same task. That's the beauty of the SCOM Architecture, it provides much of the groundwork for you. All you need to do is learn how to make it work for you!

What is a Component?

Before we begin discussing topics like attribute linking and model/view, it is important to first define some new terms that Version 7 is introducing.

Component
A component is a self-contained, reusable object with specific properties, which include

- attributes
- methods
- events
- event handlers
- a set of supported or required interfaces

Attributes
Properties that specify the data associated with a component are referred to as attributes such as its description, color, size, or any other data that is stored with the component. Attributes are similar to Version 6 instance variables with that respect. They differ in that attributes have more data associated with them than just name, value and type. The information stored for an attribute is commonly referred to as its metadata and contains

- name, type, initial value and description
- state indicating whether it is new, overridden, inherited or a system attribute
- a list of valid values and validation options such as text completion or honor case which are used by SAS/AF to automatically perform validation when the attribute's value is changed
- editor used at build-time to assist the user in setting the attribute value
- setCAM and getCAM methods that automatically get invoked to perform additional processing when the attribute value is accessed
- as well as autocreate, scope, linkable, sendevent, and editable

A more detailed explanation on each of these metadata items and how they're used can be found in a related paper titled, *Dressing Up Your Version 6 Objects to be Version 7 Components* by the same authors.

Methods
Operations that can be executed by any component you create from that class are defined as methods. Methods now contain metadata themselves which define information such as

- name and description
- entry and label where the implementation code resides
- method signature information which can bring major compile-time and run-time savings if utilized
- scope which controls accessibility

In Version 6, the primary means for communicating between objects was through the use of an object's methods. Although much of the focus in Version 7 has shifted from methods to the use of attributes for establishing communication between components, methods still play a critical role especially through interfaces in establishing the ground rules for model/view relationships.

Events
These properties alert applications when a resource or state changes. Events are key properties with respect to the mechanics of attribute linking.

Event Handlers
These properties provide the corresponding response when a resource or state changes. In other words, an event handler executes when a specific event occurs.

Interfaces
These properties indicate whether the component is enabled for model/view communication. While models and viewers are typically used together, they are still independent components. There must be a published set of ground rules where one component (typically the model) agrees to *support* and the other component (typically the viewer)

requires the same interface. An interface match has to occur between these two components before a model/view relationship can be established.

Visual component
In SCOM, visual components are called controls. Controls define objects that can be placed on the display such as icons, push buttons or radio boxes.

In Version 7, the following visual components have been added to the class library and are true native controls. This means they offer a native platform look-and-feel. For example, a combo box control under Windows will look and act like the native Window's platform combo box because that's what it is! These controls are pixel-based and have system font support which also will improve the look and feel of the applications you build using them. The new native visual controls that you will find are:

- Check Box Control
- Combo Box Control
- Desktop Icon Control
- List Box Control
- List View Control (experimental)
- Push Button Control
- Radio Box Control
- Scrollbar Control
- Spin Box Control
- Text Entry Control
- Text Label Control
- Text Pad Control
- Tree View Control (experimental)

Non-visual component
In SCOM, non-visual components are called models. Models provide attributes and methods for querying and modifying underlying data abstractions, such as objects that read or manipulate SAS data sets or catalogs. New models that have been added are:

- Catalog Entry List Model
- Catalog List Model
- Color List Model
- Data Set List Model
- External File List Model
- Library List Model
- LIST Entry List Model
- Range Model
- SAS File List Model
- SLIST Entry List Model
- Variable List Model

New to Version 7 is the Component Window, which allows you to quickly create any of the above components in a frame. From the Component window, you can

- double-click on the component which creates the object on the frame in a default location
- drag a component from the Component window and drop it on the frame where you want the object created.

Previously, the only way to create non-visual components in a frame was programmatically so this is yet another new feature you have to look forward to!

Attribute Linking

What exactly is attribute linking? Attribute linking enables one component to change the value of its attribute when the value of another component's attribute is changed.

For example, a frame contains two controls: text entry control and a graph output control. When the user types the name of a 4-level GRSEG entry name in the text entry control, you want the graph output control to display the graph in its containing region.

Previously, in Version 6, you would write FRAME SCL to establish this communication between these objects. The code would look something like:

```
TEXTOBJ:  /* text entry label */
    call notify('graphobj', '_set_graph_', textobj);
return;
```

While the above code might not seem that bad, typical frame applications involve many more objects so this code can quickly add up. Wouldn't it be nice if there were some way to just 'tell' the graph object that you want it to watch out for the text entry's value to change, grab that value and use it? Why not eliminate the above code altogether?

That's what attribute linking basically does for you. As part of the metadata for an attribute, you can specify whether or not you want it to automatically send an event when its value changes. By specifying sendEvent='Yes', an 'attributeName changed' event automatically gets sent by the object when the attribute value changes. In our example, the text entry control has an attribute called **text**. We want this attribute to send the event when its value changes.

On the other side, the graph output control has an attribute called **graph** and we want to set a link for this attribute. You can control whether an attribute can obtain its value via a link or not using attribute metadata as well. By default, all attributes have linkable='Yes'. Setting this metadata item enables the object to obtain a value for the attribute dynamically at run-time from the value of another attribute as it changes. So, as the application user types into the text entry control, we want the graph control to display what is typed in the field. We can accomplish that by specifying an *attribute link* on the **graph** attribute.

This can easily be done through the new Properties window. The Properties window enables you to view, edit, or add component properties on the instance. You can assign a specific value for an attribute or you can set up an attribute link.

To specify a link you can either type directly into the *Link To* cell in the table or you can select the ellipses button which appears in the cell and will open the Link To window which will assist you in making your selections. As a matter of fact, this window utilizes another nice feature of the SCOM design. When you select the name of the object that you want to establish a link to, the attribute name of the object's defaultAttribute will automatically be selected for you.

DefaultAttribute is discussed in more detail below but nine times out of ten, this is the attribute you will want to use when establishing a link. In our case, the text entry's defaultAttribute value is its **text** attribute. We do in fact want the graph output control's **graph** attribute to be linked to the text entry control's **text** attribute. So with no typing what-so-ever, our link has been established.

If you were to testaf your application right now, you could type the graph name in the text entry control and immediately see the graph output displayed in the frame.

Behind-the-scenes,

1. When the link is defined in the Properties window, an event handler is established on the object. The event handler is listening for the '*attributeName* changed' event, which in our example would be the 'text changed' event coming from the text entry control.
2. The text entry control, because of its attribute metadata definition of sendEvent='Yes', automatically sends an event when its value is changed.
3. The graph output control has an event handler that is listening for that specific event and its _onAttributeChanged method executes.
4. An event object is passed along with the event providing information about the name of the attribute that has been changed, its type, the object that owns the attribute and the attribute value. The implementation for the _onAttributeChanged method basically turns around and gets the information from the event

object and invokes _setAttributeValue on the attribute that has the link established.

5. The _setAttributeValue method is the one responsible for actually changing the value of the **graph** attribute on itself.

Again, all of this is accomplished through attribute metadata definitions and setting attribute links via the Properties window. There is no programming required. You get attribute linking for free when using an SCOM component because all components inherit this behavior. The _onAttributeChanged and _setAttributeValue methods already have this functionality built in as part of their implementation.

Two issues were mentioned above that might need further clarification as they are used in various places throughout this paper and in the SCOM architecture itself: the _setAttributeValue method and the defaultAttribute attribute.

What does the _setAttributeValue method do and when is it used?

Up until this point, we've been focusing on the SCOM architecture from the component side. However, there is also a language side which is equally powerful called the SAS Component Language (or SCL). SCL has been enhanced to support a new programming style referred to as dot syntax. Dot syntax goes hand in hand with the new component architecture because your SCL code is simpler and can be consistent across all components.

For example, in Version 6, to programmatically change the name of the graph displayed in a SAS/GRAPH® Output object, I could use either statement below:

```
call notify('graphobj', '_set_graph_',
        'sashelp.eisgrph.litebulb.grseg');
```

or

```
graphobj = 'sashelp.eisgrph.litebulb.gresg';
```

The first approach required you to know which method to invoke as well as the syntax for its parameter list. In Version 6, when you referenced the object by its name in your SCL, you were actually setting or querying the value of the object. That is what the second approach is doing. And while it provided a nice short cut, it was also somewhat inconsistent because the object value type differed from object to object and it was not always obvious what characteristic the object name was actually referencing.

In V7, there is a common language approach to getting and setting attribute values on your components which is dot syntax. Using the same example, the following would set the value for the **graph** attribute:

```
graphobj.graph = 'sashelp.eisgrph.litebulb.grseg';
```

In order for you to use dot syntax, the object name must be declared as an object so that it references the object id and not the value as is the default for Version 6 legacy classes. New Version 7 components use this approach by default. When using legacy classes, you can simply change the value of the object's **objectNameUsage** attribute from *value* to *ID*. This will enable you to use dot syntax even for instances of legacy classes.

Once you begin using dot syntax, it is also important to understand how that statement gets translated internally by SAS/AF software. Basically, when you use dot syntax, it translates internally to either a _setAttributeValue or _getAttributeValue method call (depending on whether you are changing or querying the attribute value).

These methods are inherited from the Object class and contain a lot of functionality. They are the backbone to much of the behavior attributes provide.

For example, the _setAttributeValue method

- verifies the attribute exists
- verifies the type of the attribute matches the type of the value being set or queried
- on a set call, validates the value against the valid values list if one exists in the attribute metadata
- invokes the setCAM or getCAM appropriately if they exist
- on set calls, if none of the above conditions have produced an error condition, the method then
    - stores the value either on the attribute list or on the IV list if it is linked to an IV
    - sends the 'attributeName changed' event if the attribute has sendEvent='yes'

If you were to invoke the setCAM or getCAM method directly, all of the above functionality would be lost. Your application with respect to attribute linking for example, would not perform as expected since _setAttributeValue is the one that is responsible for sending out the 'attributeName changed' event. So _setAttributeValue is a very important piece of the underlying architecture.

What does an object's defaultAttribute buy you?

Briefly described earlier, this attribute lets you specify the name of another attribute on the object. Typically, you would assign the attribute that has the most meaning for the component – the single piece of information that if someone queried your object, it would be the value of the defaultAttribute attribute that would be useful information to return. In the examples we've been discussing, the graph output control would set **defaultAttribute** to ***graph***. The text entry control has **text** as its **defaultAttribute.**

Setting this attribute on all of your components offers multiple advantages. From the user-interface standpoint, there are several places in the Class Editor and in the Properties window that ask you for the name of an attribute on an object. By default, these windows will initialize the value to be what is specified for the object's **defaultAttribute**. Our attribute linking example above already mentioned that when establishing an attribute link, the Link To window fills in with the name of the defaultAttribute automatically.

This information is used in other places as well such as when defining the **dragInfo** and **dropInfo** attributes which control what information is passed between objects during the drag and drop process. This saves typing since in most of these situations, you will find that the defaultAttribute is the one you want selected.

From a programming viewpoint, this attribute also has much value. If you are familiar with the SCL SET function. Using SET can significantly reduce the coding required for accessing variable values in a data set. After a CALL SET, whenever a read is performed from the SAS data set (i.e., FETCHOBS call), the values of the corresponding SCL variables are set to the values of the matching SAS data set variables.

To think of a realistic example, suppose you have a frame that contains two text entry objects and the names of these objects are 'firstName' and 'lastName'. The purpose of this frame is to read observations from a SAS data set that has variables by the same name as the text entry controls on your frame. Performing a CALL SET and then a FETCHOBS will automatically cause the text entry fields to display the values for firstName and lastName from the observation.

This will work automatically because behind-the-scenes, a _setAttributeValue call is made on each object and it uses the setting of each object's **defaultAttribute** attribute to determine which attribute to set the value for. In the case of a text entry control, its **defaultAttribute** setting is its ***text*** attribute which makes sense since that is the single most important information a text entry control owns most likely. Thus, the value from the firstName variable in the data set will automatically

get set as the **text** attribute on the firstName object. The same occurs for the lastName object and for all objects in the frame whenever SAS/AF finds an object name that matches the name of a variable in the open data set.

Drag and Drop

In Version 6, in order for you to build any application that utilized drag and drop functionality, you had to override one or more of the drag and drop methods. A simple example would be a frame that has a list box that displays the 4-level GRSEG entry names. The frame also has a SAS/GRAPH Output object on it. You want the user to be able to drag an item from the list box and have the graph automatically display in the graph output control.

To do this in V6, you'd need the following FRAME SCL code:

```
length lib cat entry type $ 8 fullname $43;
INIT:
    /* create an instance of the SAS catalog class */
    catclass= loadclass('sashelp.fsp.catalog.class');
    catobj = instance(catclass);
    entryinfo = makelist();
    /* tell it which catalog you want to retrieve entry names
from */
    call send(catobj, '_setup_', 'sashelp.eisgrph');
    /* retrieves a list of sublists with catalog entry information
for GRSEG
       entries only */
    call send(catobj, '_get_members_', entryinfo, 'libname
catname
                objname objtype', "objtype='grseg'");
    /* entries is the name of the variable specified in the list
box's object
       attribute window as the one that will contain the SCL list
to fill the list
       box */
    entries = makelist();
    /* loop through the entryinfo list to build the fullname string
to insert into
       the list that displays in the list box */
    do j = 1 to lislen(entryinfo);
        sublist = getiteml(entryinfo, j);
        lib = getnitemc(sublist, 'LIBNAME',1,1,'');
        cat = getnitemc(sublist, 'CATALOG',1,1,'');
        entry = getnitemc(sublist, 'OBJNAME',1,1,'');
        type = getnitemc(sublist, 'OBJTYPE',1,1,'');
        fullname =
trim(lib)||'.'||trim(cat)||'.'||trim(entry)||'.'||trim(type);
        rc = insertc(entries, fullname, -1);
    end;
    /* clean-up */
    rc = dellist(entryinfo, 'Y');
    call send(catobj, '_term');
return;

TERM:
   rc = dellist(entries);
return;
```

And, you'd need to override the _getDragData method on the drag site object (list box):

```
getdata: method rep op $ 40 data x y 8;
  if rep = '_DND_TEXT' then do;
     call send(_self_, '_get_last_sel_', row, issel, text);
     rc = setitemc(data, text, 1, 'y');

  end;
  else call super(_self_, '_get_drag_data_', rep, op, data, x, y);
endmethod;
```

And, you'd need to override the _drop method on the drop site (graph output):

```
drop: method rep op $ 40 data 8 from $ 7 x y 8;
   if rep = '_DND_TEXT' then do;
      grafname = getitemc(data, 1);
      call send(_self_, '_set_graph_', grafname);
  end;
  else call super(_self_, '_drop_', rep, op, data, from, x, y);
endmethod;
```

In comparison, to accomplish the same thing in V7, you would take the following steps:

- create the list box on your frame.
- from the new Component Window, select the Catalog Entry List Model and drop it on top of the list box (this establishes a model/view relationship which is discussed in more detail later in this paper).
- in the Properties window, set the following attributes on the Catalog Entry List Model component:
  - ⇒ **catalog =** 'sashelp.eisgrph'
  - ⇒ **typeFilter** = 'GRSEG'
- which will automatically display the list of GRSEG entries available from that catalog in the list box due to the model/view communication that occurs behind the scenes.
- in the Properties window again,
  - ⇒ for the list box, set **dragEnabled** = 'Yes'
  - ⇒ for the graph, set **dropEnabled** = 'Yes'

Then make sure your SCL code appears as follows:

```

```

"Where's the beef?", you ask. That's right! Absolutely ***no SCL*** has to be written to enable basic drag and drop applications like these. Just run your application and watch it go!

In the example above, the list box control already has defaults set for its **dragInfo** attribute to send the value of the currently **selectedItem** attribute when dragged. The graph output control also has default **dropInfo** attribute settings to take the value passed to it and set it on its **graph** attribute. Since the value of the **selectedItem** will be a 4-level GRSEG name, that is what the **graph** attribute expects and it will then display the graph.

When the drop occurs, the _drop method is passed an object that contains data which includes the name of the drag site attribute and its value. The _drop method uses this information to invoke _setAttributeValue on itself. The attribute set on the drop site comes from the **dropInfo** attribute.

And since all of the V7 components that SAS/AF software offers will already have default drag and drop attribute settings, it truly makes drag

and drop a very simple process. If you want to change the defaults, they're just attributes like everything else. Simply go to the Properties window and use the provided editors to specify something different. Again, no SCL code needed!

Model/View

In the above example, you probably noticed the amount of code that was eliminated to fill the list box with a list of GRSEG entries from a specific catalog. The reason that code is no longer needed in V7 is because we now offer model/view as way to establish communication between a model (like the Catalog Entry List Model) and a viewer (like the List Box Control).

The new Version 7 models are already enabled for model/view communication through their support for the *staticStringList* interface. All of our viewer controls that display things in lists have been enabled for model/view communication as well using the same interface.

Interfaces are new to V7. Using the Interface Editor, you can define an interface which is basically a set of rules that two components must follow in order to establish a model/view relationship. Basically, an interface contains a set of methods with specific method signatures. One component says it *supports* the interface. Another component says it *requires* the interface.

When dragging a model onto a frame and dropping it on top of a viewer, SAS/AF software goes to work and based on each component's interface information, establishes the model/view relationship for you. One indicator that a model/view relationship has been established is by looking at the viewer's **model** attribute. It should contain the name of the model. You can share the same model across multiple viewers in your frame by setting the **model** attribute directly on each viewer.

Due to the interface, the viewer has been designed to know how and when to communicate with the model to get data. In our example, the viewer knows how to get to the list of items that the model creates and then displays it.

Behind-the-scenes,

1. When the **model** attribute is set in the Properties window, an event handler is established on the viewer. The event handler is listening for the 'contents updated' event.
2. The 'contents updated' event gets sent by the model when one of its attributes specified in the **contentsUpdatedAttributes** attribute changes. The **contentsUpdatedAttributes** attribute contains the name of one or more attributes on the component. These attributes are the ones that the component has identified as being critical. They affect the contents of the model and the viewer should be notified when their values change.
3. The viewer has an event handler that is listening for this event and its _onContentsUpdated method executes. It's up to the viewer in its _onContentsUpdated to then call back to the model to retrieve updated information. It knows how to communicate with the model because of the methods defined in the interface.

In our example, the _onContentsUpdated method on the viewer has been overridden to invoke the _getItems method on the model. This method is defined in the *staticStringList* interface, thus, telling the viewer that it is implemented by the model and can be invoked.
While this approach to establishing communication between objects does require programming, it is encapsulated method code and transparent to the end users working with these components in the frame.

Model/View in its simplest form appears to mimic attribute linking behavior where you get the value for one attribute from the value of another attribute. But, often the model/view communication process is much more complex than simply accessing one or two attributes. Through the use of interfaces, components can be designed so that even the most complex relationships can be established by simply specifying a **model** value.

CONCLUSION

The sooner you begin working this functionality into your application designs, the sooner you will reap the rewards of lower development and maintenance costs. Through the use of attribute linking and drag and drop you will have less code to manage. By designing new components with the model/view methodology in mind and utilizing interface support, you should experience a significant drop in your project time to completion due to the code reuse that will quickly begin to take place.

There were many new terms and concepts introduced in this paper. Hopefully, this has shed some light on what they mean and how you will be able to take advantage of them in your application development environment.

AUTHORS

Tammy L. Gagliano
SAS Institute Inc.
Two Prudential Plaza, 52nd Floor
Chicago, IL 60601
phone: (312) 819-6824
email: sastlg@unx.sas.com

Glen R. Walker
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
phone: (919) 677-8000
email: sasgrw@unx.sas.com