Dressing Up Your Version 6 Objects to be Version 7 Components

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

ABSTRACT

Version 7 SAS/AF® software exploits the SAS Component Object Model (SCOM) Architecture to provide application developers with a rapid application development tool for object-oriented 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 fully exploit the new features. In this paper we will discuss the few simple steps you can take to make your existing classes work as SCOM components and what advantages this will bring.

INTRODUCTION

While the BUILD environment is new and improved, much effort was put into Version 7 to ensure that your existing classes (referred to as legacy classes) operate as expected with no additional work on your part. You can still view the same object and region attribute windows when you create an instance of your class at design time. Any custom attribute windows you created will also work. And while the Class Editor has been significantly enhanced, both from a user-interface perspective as well as its feature set, you can still modify and maintain your legacy classes using it.

There are some major differences in the internals of the class entry itself, however. Most of the changes will be transparent. But in order to move your application forward and take advantage of the new features, there are some basic concepts that you will need to learn such as

- What is dot syntax?
- How are attributes different from instance variables?
- What is attribute linking and how can you exploit it in your application design?
- Why should you add method signatures to your methods?
- How does method scoping affect your component design?

All of these issues are described in more detail throughout the remainder of the paper. But to give you a sneak preview as to why you'd want to incorporate any of these changes, let's take a quick look at what a difference Version 7 will make.

In Version 6, if you wanted to change some of the characteristics of a graph output object, at run-time you'd find yourself writing the following SCL program:

```
call notify('obj1','_set_border_title_','title');
call notify('obj1','_set_border_style_','embossed');
call notify('obj1','_set_border_color_','red');
call notify('obj1','_set_graph_','sasuser.sugi.af.grseg');
put obj1=;
            /* obj1 = 'sasuser.sugi.af.grseg'; */
```

Note the PUT statement which references the object by its name. The value printed would be the name of the graph displayed which is the value of the object itself.

In Version 7, the code would look like:

```
GraphOutput1.borderTitle = 'title';
GraphOutput1.borderStyle = 'embossed';
GraphOutput1.borderColor = 'red';
GraphOutput1.graph = 'sasuser.sugi.af.grseg';
```

About the same number of SCL statements are used but notice the consistency in the Version 7 coding style. This is referred to as dot syntax. The syntax is exactly the same across all objects. No need to have to memorize the many different method invocations, just to set or get values on an object.

Also note that the object name 'GraphOutput1' is longer than 8 characters and is used as the first level qualifier in the dot notation. Using dot syntax in Version 7, the object name is used to reference the object's ID, not its value.

Expanding our example to include querying an object for specific values. Sample Version 6 code:

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

Retrieving attributes, Version 7 style:

```
put GraphOutput1.borderTitle=
    GraphOutput1.BorderStyle=
    GraphOutput1.borderColor=
    GraphOutput1.graph=;
```

The number of lines has been reduced. And again, there is a significant difference in the coding style. This example clearly illustrates the user-friendliness of dot syntax over having to create and manage lists using SCL list functions to perform the same functionality.

Before we discuss more Version 7 (V7) enhancements, it would probably be useful to briefly review how things worked in Version 6 (V6) with respect to class structures – specifically what functionality instance variables provided.

How Things Worked in Version 6

In V6, information about a class was stored in instance variables (IVs). IVs had a name, type, initial value and whether they were automatically initialized when an SCL method executed (automatic IVs). All objects created from the same class had the same set of IVs.

In class method code, IVs were directly accessed via SCL list functions. For example, if you have a legacy class with an IV called *color*, you would access it using

```
rc = setnitemc(_self_, 'red', 'color');
value = getnitemc(_self_, 'color', 1,1,'');
```

Since IVs are considered private class information and should only be modified directly from within class code, methods (such as getColor and setColor) would need to be implemented for the class to surface this same information for users. Often the ability to set IV values were surfaced through the object attribute window as well. The combination of object attribute windows and methods provided users of the class a way to set these values both at build-time and programmatically at run-time.

How Things Work in Version 7

In V7, the focus has shifted from the use methods to control an object's behavior to using attributes.

Keeping with the same example used above, in V7 you can implement *color* as an attribute. You, as well as users of your class, can then use the same dot syntax to query and set its value. For example,

```
objectName.color = 'red';
value = objectName.color;
```

The component does not need the traditional setColor or getColor methods which were required in V6 to give the users access to the underlying IVs. As a result, application development and maintenance costs will be dramatically reduced using this approach since the class can be leaner with fewer methods to implement. The dot syntax is clean and consistent – whether you're accessing it as the component developer or a user of the component in a frame.

Another advantage of using attributes is that they eliminate the need to design and maintain individual object attribute windows for each class. Attributes are displayed automatically in the new Properties window providing a consistent and easy-to-use interface across all components in a frame.

New features such as the support for multiple selections also makes it easy for you to change the value for the same attribute on several components at one time. For example, if I have three container box controls in my frame and I want all three to use the same font for their **borderTitleFont** attribute, I can select all three controls and in the Attributes table, change the font once. The change gets applied to all currently selected components.

Overall, attributes are smarter than IVs. They contain much more information about themselves than name, type and initial value. The information stored for an attribute is commonly referred to as its metadata.

Item	**Description**
name	the name of the attribute
description	a short description for the attribute which appears as help information on the Class Editor and Properties windows
type	specifies the type of data stored which can be character, numeric, list, object or 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	specifies a set of valid values for the attribute
editor	specifies a FRAME, PROGRAM or SCL entry that allows a user to enter a value for the attribute. If supplied, the editor is automatically launched by the Properties window when a user clicks the ellipses button in the value cell
autoCreate	specifies if the attribute is automatically created (valid for list and object types only)
scope	controls permission level for accessing the attribute
editable	indicates whether the attribute can be modified or just queried
linkable	specifies whether the attribute is linkable
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

Table 1: Attribute Metadata

Through the use of attributes in your component design, you can take advantage of

- dot syntax to set or query data items that were previously available to you via the V6 style object attribute or region attribute windows.
- attribute linking which enables rapid application development by allowing you to define and then connect attributes of different components
- additional ways of establishing communication between objects in your frame applications such as model/view and drag and drop which can also be easily accomplished through the use of attributes
- use of the new Properties window which eliminates the need for you to have to develop and maintain individual object attribute windows.

Creating Attributes from Instance Variables

The Class Editor provides you with an easy way to change your component so that you can begin using attributes instead of IVs. By default, an attribute list is created and stored on the legacy class for you. For every IV on your class, an attribute is created with the same name and type and is displayed in the attributes table.

For all practical purposes, IVs have been replaced by attributes. They remain a part of the class structure for compatibility purposes only. They still exist; however, if you choose to continue using IVs in your component design, you need to be aware that the following limitations exist:

- You cannot use SCL dot syntax to access IVs. To create, set or retrieve values stored as an IV, you must use the appropriate SCL list function.
- the BUILD environment, specifically the Class Editor, does not provide a mechanism for you to add an IV to a class. Adding an IV can only be done programmatically using SCL.

Steps for Enhancing a Version 6 Class to Become a Version 7 Component

Follow these simple steps:

1st. Decide whether you're going to modify an existing class in place or make a copy of the class to create a completely new component. We refer to it as a 'component' to indicate that its design takes advantage of the SCOM architecture.

In our example, we're making a copy of the SAS/GRAPH® Output class and naming it the Graph Output Control. We will not have to rewrite any of the underlying class implementation. Rather we will add our enhancements on top of what is there, taking advantage of what's already in place.

2nd. Review the list of IVs used by the class that you want to surface as attributes and define the appropriate metadata for each attribute. This process is where the bulk of the changes are made and is described in more detail below.

3rd. All legacy classes when edited in the Class Editor will have **objectNameUsage** as an attribute. For V6 compatibility purposes, the value for this attribute is *value*. As mentioned earlier, this is because in V6 when you reference the object by its name in your SCL, you are actually setting or querying the value of the object. For example, if you have a text entry control (OBJ1) in a frame, and you want to set the text to display, you can use

```
OBJ1 = 'This is my string';
```

In V7, in order for you to use dot syntax, you must change the value for the **objectNameUsage** attribute to be *ID*. This indicates that when the object name is referenced, it represents the object ID for the object, not its value. So you can do things like

```
OBJ1.text = 'This is my string';
```

where **text** is the attribute on OBJ1 that you are changing.

Or, if you leave the value for this attribute as the default, SCL programmers can still program using dot syntax in their SCL by using the new DCL statement to locally declare the variable as an object.

For example:

```
DCL sashelp.fsp.efield.class textobj;
INIT:
    _frame_._getWidget('obj1', textobj);
   textobj.text = 'This is my string';
return;
```

4th. From the Class Editor's Class Settings window, specify that you want to use the new Properties window to display the properties of your component. You will no longer need to support individual object attribute windows for your components.

Example: Graph Output Control

The V6, SAS/GRAPH Output Class functions primarily as a graph viewer. It enables you to build frames to display SAS/GRAPH output images (or GRSEG entries). Our Graph Output Control will offer this same functionality but through the use of attributes instead of methods.

Step two involves examining the list of IVs the legacy class uses. From this list along with what features were surfaced previously via the object's attribute window, we need to decide which ones should be surfaced as attributes in our new component.

When editing the class from the Class Editor, SAS/AF software makes this process of examining the IVs simpler by automatically creating an attribute list on the class for you. In the Attributes table, you will see that an attribute has been created with the same name as each IV on your legacy class. Since the IV list and attribute list are stored separately on the class, behind-the-scenes a link is established between the attribute and the IV. This allows existing code to function as is since it's written to access IVs. But more importantly, the link enables you to move forward and begin using the new dot syntax in your SCL. It ensures that when you use dot syntax to change or query an attribute, the value is actually stored and retrieved from the IV.

The features we want to surface as attributes in our component are:

- the name of the graph to display
- how the graph is resized and scaled
- whether horizontal or vertical scrollbars display

These are the ones you want to make PUBLIC attributes. If an attribute is PUBLIC, it displays in the Properties window and can be set (if EDITABLE = 'Yes') or queried from any SCL code. If an attribute is PRIVATE, only the class instance can access it via its method code. For PROTECTED attributes, only the class and any subclasses can access it from method code. By default, all new attributes are added as PUBLIC attributes, including the ones added based on the IV list.

It is possible that the initial attribute list may need to be cleaned up since it was created from the IV list and many of the IVs were added for internal use only. You can do this one of two ways:

- change the scope to PRIVATE which will leave them as attributes but limit their accessibility to the class instance only
- delete them from the Attributes table. Remember, this will not automatically delete the IV that it is linked to since IVs are stored separately on the class. You have to specifically tell the Class Editor to delete the associated IV. It will prompt you when deleting the attribute.

So for our component, we're left with the following attributes. The first thing we'll do is rename them to something more meaningful and assign their metadata as follows:

Attribute = graph	**(IV=GRSEG)**
editor	sashelp.classes.grsegEntryEditor.scl
setCAM	setcamGraph
description	Returns or sets the name of the GRSEG entry to be displayed

Attribute = magnify	**(IV=MAGNIFY)**
initialValue	100
setCAM	setcamMagnify
description	Returns or sets the percentage to scale the graph based on the graph's natural size. Only valid when the graph is not resized to fit the containing region

Attribute = resizeToFit	**(IV=CONTORT)**
initialValue	No

validValues	Yes No
setCAM	setcamResizeToFit
description	Returns or sets the state that determines whether to resize the graph output both horizontally and vertically to fit the shape of the containing region

Attribute = scrollbars	
initialValue	No
validValues	Yes No
setCAM	setcamScrollbars
description	Returns or sets the state that determines whether vertical and horizontal scrollbars are displayed

Explanations for some of the metadata settings are described below:

initialValue
indicates what value to use by default if the user does not specify a different value when creating an instance of the component.

validValues
are useful when the attribute has a discrete list of values that are valid. The Properties window uses this information to display a combo box control in the table cell when the user clicks in the value cell to change the attribute's value. In the Class Editor, you can specify

- a list of values separated by blanks or by a '/' (forward slash) for items with embedded blanks
- the name of an SCL or SLIST entry preceded with '\' (back slash). In this case, the list is obtained from the SLIST entry or by executing the SCL entry at run-time allowing you to create a dynamic list of values

This information is also used for validation purposes when dot syntax is used to change the attribute's value.

textCompletion
only valid when validValues are present. Used in our class because we want _setAttributeValue (which is discussed later in more detail) to perform text completion to find a match on the value.

For example, if the user specifies
graphoutput1.scrollbars = 'Y'
we want the value to actually complete and store as 'Yes'.

honorCase
again only valid when validValues are present. Used in our class because we don't care what case the user types the value in, we want the textCompletion to perform ignoring the case. For example, if the user specifies
graphoutput1.scrollbars = ' y'
we want the value to actually complete and store as 'Yes'

editor
assigned for the **graph** attribute to display a catalog entry selector which only lets the user choose GRSEG entry types. In this example, the editor is an SCL entry that has the following code

```
entry optional= objectID:object classID:object
environment:string(2)
          frameID:object attributeName:string(32)
          attributeType:string(83)  value:string(83);
INIT:
   value = catlist('*', 'GRSEG', 1, 'Y');
return;
```

Another example of an editor would be a FRAME entry that's designed to assist the user in selecting a color.

Specific rules must be adhered to when designing your editors. They must use the above ENTRY statement as those are the variables that SAS/AF software guarantees will be passed in and can be used by the editor code.

- objectID, is the name of the object that is currently being edited
- classID, is the ID of the object or class currently being edited
- environment, indicates whether the editor is being invoked from the Class Editor ('CE') initialValue cell or from the Properties window ('PW') value cell
- attributeName, is the name of the attribute being edited
- attributeType, is the type of the attribute
- value, is the current value of the attribute and when the editor is closed, is the value that gets passed back and set in the table cell

All of the new attributes in our graph output component should also have the following metadata defined:

scope=public
which offers the least restrictive access and will automatically display in the Properties window when an instance of the object is created in a frame

editable=yes
because we want users to be able to change the value on the instance

linkable=yes
because we want to be able to let users set attribute links for this attribute to obtain its value from another attribute. For example, the application may have a text entry control (textentry1) on the frame along with the graph output control. You want the value for the **graph** attribute to be set to whatever the user types in the **text** attribute for the text entry control.

You can set up an attribute link via the Properties window such that the **graph** attribute has a link of *textentry1.text*. At run-time, as the user types the 4-level name of a GRSEG entry into the text entry control, the graph control will automatically display the graph due to this attribute link having been set.

For more detail on attribute linking and how it works, refer to a separate paper titled, *Version 7 SAS/AF Software - The New Component Technology* by the same authors.

sendEvent=yes
to automatically send an 'attributeName changed' event when the attribute value changes.

This event is the key behind attribute linking. All linked attributes have handlers automatically set up to be listening for these 'changed' events and update themselves appropriately. In the above example, by setting an attribute link for the **graph** attribute to be *textentry1.text*, an event handler was automatically set for the graph output control to be listening for the '*text changed*' event. The graph's event handler then gets the current value of **text** and sets it on the **graph** attribute. No SCL programming is required to establish this kind of communication between objects. It's part of the SCOM architecture that all components inherit for free.

category='Data'
which is the category used by all SAS supplied components to indicate they are key attributes on the object (versus other categories like Appearance, Drag and Drop, Size and Location which are typically inherited attributes from Widget or Object class) and can easily be

accessed under this category in the Properties window or Class Editor navigation tree.

setCAMs and getCAMs
If some kind of additional processing needs to take place when the attribute value is either set or queried, you can assign a method to an attribute as its setCAM or getCAM. While it might be tempting to access the attribute by invoking these methods directly, which was the V6 way of doing things, this is discouraged. Instead, you should

- use dot syntax as described earlier
- use _setAttributeValue or _getAttributeValue method calls

To get a clearer picture on when and how the CAMs are utilized, it might help to explain the flow of control when an attribute's value is queried or changed.

Basically, when you use dot syntax, it gets translated internally to _setAttributeValue or _getAttributeValue method calls. 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, some of what the _setAttributeValue method does is that it

- 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 validValues list if one exists
- 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 might not perform as expected since functionality like attribute linking and keeping the IV and attribute value in sync would be lost.

Here is an example of what the setCAM code would look like for our Graph Output Control:

```
useclass sashelp.classes.graphoutput_c.class;

/* _setcamGraph: setcam for graph attribute */
setcamGraph: method attrvalue:update:char(83)
return=num;
   _setGraph(attrvalue);
   return (0);
endmethod;

/* _setcamMagnification: secam for magnification
attribute */
setcamMagnification: method attrnvalue:update:num
return=num;
   _setMagnification(attrnvalue);
   return (0);
endmethod;

/* setCAM for resizeToFit attribute */
setcamResizetofit: method attrvalue:update:char return=num;
   if errorMessage ne '' then return (4);
   _setContort(attrvalue);
   return (0);
endmethod;

/* setCAM for scrollbars attribute */
setcamscrollbars: method attrvalue:update:char(3)
return=num;
   if errorMessage ne '' then return (4);
   if upcase(attrvalue) = 'YES' then _scrollbarsOn();
   else _scrollbarsOff();
   return (0);
endmethod;

enduseclass;
```

As you can see, the method code is simple and basically turns around and invokes the appropriate V6 method so as to alter the object's behavior as expected.

The other thing to point out about the setCAM for the **scrollbars** and **resizeToFit** attributes is the first line

```
if errorMessage ne '' then return (4);
```

This line is used because the attributes have validValues defined as part of their metadata. Part of what you get for free when you assign a validValues list is that when the user changes the value of the attribute and _setAttributeValue gets called, it performs validation against the validValues list and will set the value of **errorMessage** to indicate that the value being entered does not exist on the validValues list. This may or may not be acceptable as there might be situations where you want to support other options that aren't on the validValues list.

By checking the value of **errorMessage**, you can choose to return from your setCAM immediately or continue. In both of our CAMS, if the value is not on the validValues list (i.e. 'Yes' or 'No'), we do not want the CAM to successful complete so we set the return code and it exits immediately.

When writing setCAMs or getCAMs, there are method signature requirements that your CAMs must adhere to since these methods get invoked internally by SAS/AF when the attribute value is queried or changed. For example, the return argument on your setCAM gets propagated down as the return argument for the _setAttributeValue call. This is then used by the Properties window to know whether or not to display an error condition for the attribute being changed. The following rules apply:

rc = 0, that indicates the _setAttributeValue call was successful

rc > 0, that indicates the _setAttributeValue call was not successful and displays an error dialog containing the message stored in the object's **errorMessage** attribute

rc < 0, indicates the _setAttributeValue call was   successful but there is a NOTE or WARNING message display. The Properties window again displays the message stored in **errorMessage**.

As the component developer, if you do not like the default error message that is provided, you can also change the value of **errorMessage** attribute in your CAM code and it will be displayed instead.

Are Methods Needed Anymore?

With the exception of possibly implementing setCAM or getCAM methods, there has been little discussion about methods. Methods were the primary means for controlling the behavior of an object in Version 6. How do methods play a part in this new component architecture?

Methods have not disappeared in Version 7, by any means. But as discussed above, if a component is designed correctly, most of the basic manipulations of the component can be controlled by getting and setting attribute values. Methods are still needed to request actions from a component, for example

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

Methods are also used to trap events as event handlers. (As an aside, events and event handlers can now be defined on your class using the Class Editor. Per-instance methods, events and handlers can also be set via the Properties window at build-time.)

When writing new methods or even maintaining existing class methods, there are some significant enhancements you should consider taking advantage such as

- follow the new method name conventions when creating new methods
- use dot syntax when invoking methods
- utilize the method metadata that is now available for each method such as specifying a method's signature, its scope as well as a description for documentation purposes

New Naming Conventions
In V6, SAS/AF used underscores to separate words in method names (e.g., _set_text_color_). In V7, the embedded and trailing underscores have been removed from all SAS/AF methods, making them easier to type. For example, _setTextColor is valid and equivalent to _set_text_color_.

In the Class Editor and Properties window, all inherited method names are displayed using the new naming convention; however, existing code which uses the old style will still function with no modification.

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

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

In V7, you can invoke the method using dot syntax as follows:

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

The above is easier to type with fewer quotes and underscores. This will hopefully lead to fewer typos. This new syntax also allows for methods to have return arguments which were not previously supported in V6. For example, the method used to retrieve the color could defined as follows:

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

The DCL statement is also new in Version 7. It is similar to the LENGTH statement in that in the above example, it lets you define a local SCL variable type as character with a length of 15. The DCL statement has many other features which you will want to explore.

Method Signatures
Using dot syntax to invoke your methods brings you much more benefit than just improved program readability. Combined with registering method signatures for your methods, it also brings you compile time syntax checking as well as run-time performance gains. Method signatures are part of the metadata that can be defined along with a method description and its scope.

In the Class Editor, method signatures can easily be added to even existing methods using the Method Signature Window. When defining a method signature, you specify information about the arguments passed in on the method invocation as well as whether the method supports a return argument.

SAS/AF uses this information at compile time and will show compile errors in situations where the method you are trying to invoke does not exist on the object or if the arguments you're passing in are too many or too few or of the incorrect type. As you can see, this could greatly reduce the amount of time you spend at run-time catching these errors.

The following chart describes the metadata defined for each argument which can be entered in the Method Signature window.

name	name of the argument primarily used for documentation purposes and for sample method code generation that the Class Editor provides for cut and paste into your SCL entry
type	argument type which can be character, numeric, list, object or array
inout	whether the argument is to be used as an INPUT, OUTPUT or UPDATE value
description	argument description for usage and documentation purposes only

Table 2: Method Signature Argument Metadata Descriptions

Method Scope
Method scope information is also verified at compile time. If a method is marked PRIVATE, for example, but a user of your class tries to invoke it from their FRAME SCL, the error will be caught at compile time. PRIVATE indicates that only an instance of the class can invoke the method within its class method code. PROTECTED means the class or its subclasses can invoke the method. PUBLIC means the method can be invoked from anywhere. In V6, basically all methods were 'public' methods.

How Else Can You Enhance Your Component?

There are other attributes that your component inherits from the Object and Widget classes that you should explore:

- **dragInfo** and **dropInfo**
- **model**
- **defaultAttribute**

These are discussed in more detail in a separate paper titled *Version 7 SAS/AF Software - The New Component Technology,* by the same authors. This paper focuses on component technology with respect to the different ways you can establish communication between objects in your application. The internals of attribute linking, drag and drop and model/view through the use of interfaces all are discussed in more detail. Check it out!

CONCLUSION

The primary goal behind our development efforts for Version 7 SAS/AF software was to add value and increase the return on the investment you have already made in your application development software. Along with that goal, we also wanted to ensure the integrity of your existing applications such that they would run as is in Version 7 with no additional effort on your part.

Keeping those goals in mind, it was clear that we needed to provide a migration path that would enable you to easily bring your components and applications forward. Hopefully, this paper illustrates how that can be done. And as you begin using these new features, you will discover that the majority of the product enhancements point to designing smarter components which will result in less programming and maintenance efforts in the future.

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