INTRODUCTION

While the initial costs of learning object oriented programming (OOP) techniques can be high, the payoff can be significant: applications are developed more quickly, have a more consistent interface, have fewer bugs, and are easier to maintain. This paper considers object oriented programming as it applies to SAS/AF applications development. Special emphasis is placed on the creation of customized classes.

It is important to note that this paper limits its discussion of subclassing to graphical objects. That is, it focuses on the widgets used to build frame entries. OOP principles can also be applied to data structures. For more information on this, you are referred to an excellent SUGI 19 paper, *The Benefits of Object-Oriented Application Development Using the SAS System* by Randy Pierce.

This paper presumes experience with SAS/AF Frame entries but does not presume knowledge of OOP. It is suited to programmers who are just getting started with OOP features of Frame.

OOP: CLASSES AND OBJECTS

While this paper will not dwell on OOP concepts (for this you are referred to the books listed in the reference section) it is important to begin by clearly defining the terms object and class.

A class is a template for software objects that has inherent attributes and behaviour. Each class has clearly defined characteristics. For example, through its attributes a class ‘knows’ it can have a color, a length, a name, etc. As well, it ‘knows’ all actions it can perform (such as validating input, becoming invisible under certain conditions, etc.). Think of a class as a template from which objects are made.

An object is simply an instance of a class that incorporates all the attributes and behaviour of its class. Objects are the software modules you work with, and you can customize them to your current need (e.g., change color, size, etc.). However, you cannot add new attributes or behaviour — all instances of a class share the same attributes and behaviour. Think of objects as software modules belonging to an “object family” which has well-defined characteristics.

You can enhance the attributes or behaviour of a class through a process called subclassing. A subclass starts as a simple copy, or child, of an existing class. As with objects, subclasses inherit the attributes and behaviour of their parent. However, in your new class you can add or modify attributes and behaviour. Very importantly, if the parent class itself had a parent, then your new class indirectly inherits its characteristics as well. Thus, inheritance results in a hierarchy (see figure 1) and provides a key benefit of OOP: new classes must define only the new behaviour or attributes you wish to add. This can dramatically reduce the work required to create new software objects.

\[
\begin{array}{|c|c|}
\hline
\text{Class A} & \text{attribute} = \text{color} \\
\text{behaviour} = \text{change color upon selection} \\
\hline
\text{Class B (subclass of A, inherits from A)} & \text{attributes} = \textcolor{italics}{\text{color}} \ast \\
& \text{zoom size} \\
& \text{behaviour} = \text{change color upon selection} \\
& \text{change size upon selection} \\
\hline
\text{Class C (subclass of B, inherits from A and B)} & \text{attributes} = \text{color} \\
& \text{zoom size} \\
& \text{name of graph to display} \\
& \text{behaviour} = \text{change color upon selection} \\
& \text{change size upon selection} \\
& \text{display selection list of graphs upon selection} \\
\hline
\end{array}
\]

* italics indicates an inherited value

Figure 1 -- Illustration of a class hierarchy
SUBCLASING: OVERVIEW

SAS/AF frame entries are composed of one or more objects. In general, these objects are created from the library of classes provided with SAS/AF. For example, SAS/AF provides a text entry class, a push button class, a class for display of SAS/Graph output, etc. In creating a SAS/AF frame entry screen you select one or more of these classes from a resource list (i.e., the list of classes that appears when you press the right mouse button). Each time you make a selection from the resource list an object of the selected class is placed on your screen. You can then modify the attributes of this object to meet your current need.

As you gain experience creating frame entries you may find a pattern to your work -- that is, you may discover that you repeatedly copy an object and modify it in the same way. When this occurs, you can save yourself work by saving your modified object as a new class. Once you add this class to a custom resource list it becomes available just like the SAS-supplied classes.

The process of modifying an instance of a class and saving it as a new class is called subclassing. When creating your own customized classes from existing classes you will take advantage of the OOP behaviour called inheritance. Through inheritance, subclasses retain all attributes and behaviour of their parent class except for those you modify. Modifying existing behaviour is done through a process called overriding.

Each object maintains a list of its attributes and behaviour. Since each attribute is simply a named item in this list, you can modify an attribute by modifying the value of its named item. For example, if an object can have a color, then its list could have an item named COLOR, and that item may have a value of RED. To change the object's color to blue, you simply change the value of the item COLOR to BLUE. These named items are called instance variables, and it is through instance variables that you modify the attributes of an object.

In the case of SAS/AF, instance variables are stored as named items in an SCL list (SCL lists are documented in Technical Report P-216). You can modify instance variable values through SAS/AF attribute windows, through the class editor, or programmatically through SCL list functions and methods. Very importantly, the values of all instance variables for an object are available to the programs that control the object's behaviour. This is a powerful benefit of OOP, reducing the amount of information you must pass to these programs.

A preceding paragraph stated that the behaviour of an object is stored in its SCL list. This is not quite accurate. Actually, what is stored in the list is the location of the program which creates the behaviour. In OOP, these programs are called methods. In the case of SAS/AF, methods implement behaviour through SCL code. For each method, the SCL list contains the location of the SCL entry (its libname, catalog, etc.) containing the method.

<table>
<thead>
<tr>
<th>class</th>
<th>a template from which objects are made. Has inherent attributes and behaviour.</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>an instance of a class which incorporates all of the attributes and behaviour inherent in the class; the software modules you work with.</td>
</tr>
<tr>
<td>subclass</td>
<td>a class derived from another class. Inherits all attributes and behaviour of its parent class except for those you modify. Modifying existing behaviour is done through a process called overriding.</td>
</tr>
<tr>
<td>instance variable</td>
<td>for each attribute of an object, an instance variable stores the attribute value (e.g., color, size, etc.)</td>
</tr>
<tr>
<td>method</td>
<td>a procedure associated with an object. Through methods, an object acquires behaviour — a set of actions it can perform.</td>
</tr>
<tr>
<td>overriding</td>
<td>in subclassing, substituting new behaviour for inherited behaviour.</td>
</tr>
</tbody>
</table>

Table 1: Summary table of OOP terms

Just as attributes are inherited, so are methods. When you create a new class, you can modify these methods, disable them, or even add entirely new methods (i.e., new behaviour). You can do this through the methods attribute window of the class editor, or programmatically through SCL functions.

The remainder of this paper will consider the process of subclassing in detail. Table 1 summarizes some of the terms used in this section.
SUBCLASSING: PROCESS DETAIL

For this section we will use a very simple example to illustrate some basics of subclassing. In our example we will create a custom push button that allows the user to toggle the value of a macro variable, and which provides feedback on the current status of that variable. Please note that this simple example barely touches upon the OOP features in SAS/AF, but it serves well as an introduction to the subject given the length of this paper.

Next we need to “attach” our desired behaviour to the QTOGGLE class. You can do this by adding a method or overriding an existing (i.e., inherited) method. We will use the latter approach, writing two methods and then overriding the object’s inherited _INIT_ and _SELECT_ methods. Before we begin, though, it’s necessary to consider methods in greater detail, particularly the special status of certain methods like _SELECT_.

METHODS

A method is a unit of screen control language code stored in an SCL entry. A method is like a subroutine, often accepting information through parameters, performing an action, and then returning information through parameters. Methods are documented in Technical Reference P-216 and especially in the FRAME manual. In particular, see Guidelines for Writing Methods on page 115 of the FRAME manual.

Through methods, you can define behaviour for an object — have it validate input, have it set initial values, etc. Remember that each class inherits methods from its parent. Thus, if you create a subclass like QTOGGLE from the push button class, it retains all of the original behaviour (i.e., methods) of its parent class, the push button class.

It is important to note that some methods execute automatically (see table 2). For example, when an object is selected its _SELECT_method always executes. Thus, if you want to perform custom processing when an object is selected you can use the _SELECT_method to trigger your processing. We will do this in our QTOGGLE object.

When you modify methods that execute automatically you must use caution. Some methods, _INIT_, for example, perform vital
processing. On occasion you may override a method and receive an error message like ERROR: unable to send _INIT_ method. In this case, there is processing performed in the overridden method which should be retained. For example, the _INIT_ method is required to “set up” the object’s environment.

To retain the original processing and add your own method to be triggered at the same time (e.g., at initialization) you can first override the inherited method, substituting your own method, and then use the CALL SUPER function to invoke the inherited method. Using CALL SUPER you can force execution of both methods, and even affect the order in which they execute. In our QTOGGLE example we will do this for both the _INIT_ and _SELECT_ methods (see figure 6 and note 0). In the case of _INIT_, this allows us to enhance the behaviour of the object at initialization time yet also retain all inherited initialization time behaviour.

In a moment we will create an SCL entry with our custom methods, but first we will override the _INIT_ and _SELECT_ methods for QTOGGLE.

To override methods, use the class editor to open an edit session on QTOGGLE.CLASS (you can use the E line command in the directory window). Next, from the class editor (refer to figure 3), choose Methods.... When you choose Methods... a window will display which lists all methods of this class (Fig 4). Now scroll down until you find the _INIT_ method, then select it.

For the _INIT_ method, you should override the inherited method by entering values in the fields to the right of the methods list. These fields will contain the location of your method. The SCL code for your method is stored in an SCL entry, so you must provide the libname and catalog in which your SCL entry resides. In addition, you must provide the name of the labelled section in your SCL entry which contains the SCL code for your method.

To override _INIT_ enter SASUSER, CLASDEMO, FORDEMO, and MY_INIT as shown in figure 4. To override _SELECT_, enter SASUSER, CLASDEMO, FORDEMO, and TOGGLEIT.

Next we will add an instance variable to our class. Then we will create an SCL entry to hold the SCL code for our custom methods.

**INSTANCE VARIABLES**

In the Methods section of this paper it was mentioned that the _INIT_ method “sets up” the environment of an object. This process consists primarily of building an SCL list which contains information on the attributes of the object (e.g., its color, size, location of methods, etc.). It is through the values of this SCL list that the object keeps track of its inherent behaviour and the data it uses (i.e., instance variables values).

For our example, we will add one instance variable. To create your instance variable, select Instance Variables... from the class editor (refer to fig. 3). From the Instance Variables window, choose the Actions command, then Add mode on, and then enter the values as shown in figure 5 (name is QSTATUS, automatic is Yes). Then select OK, OK to return to the directory window.
SCL Entries for Methods

At this point we need to finish creating our custom methods. As noted above, the SCL code for your methods is stored in an SCL entry, and in overriding the _INIT_ and _SELECT_ methods we gave the location of our custom methods as an entry named FORDEMO. Now we must create the SCL code for our methods.

To do this, create a new SCL entry named FORDEMO (i.e., from the directory window execute the command EDIT FORDEMO.SCL) and then type in the code shown in figure 6. When you have finished, remember to compile the entry.

```
/*-----------------------------*/
METHODS FOR THE QTOGGLE CLASS
QTOGGLE toggles the status of a subsetting WHERE clause. Toggling is done by changing the value of macro variable QONOFF, which is used in SUBMIT blocks to conditionally include or exclude a subsetting WHERE clause.

length qstatus $3;
MY_INIT: /* The user can toggle the status of QONOFF in any frame, so */
    method; /* when the QTOGGLE object initializes it must get the current */
    /* QONOFF value and set the pushbutton text accordingly ON/OFF */
1    call super(_self_,'_INIT_');
2    qstatus = symget('qonoff');
3    if qstatus = _blank_ then qstatus = 'OFF'; /* default is OFF */
4    call send(_self_,'_set_label_','Query subsetting ' || qstatus);
endmethod;
RETURN;

TOGGLEIT: /* If they select this object then toggle its value and set _MSG_ */
    method;
1    call super(_self_,'_SELECT_');
2    if qstatus = 'ON' then qstatus = 'OFF';
3    else qstatus = 'ON';
4    call symput('qonoff',qstatus);
5    call send(_self_,'_set_label_','Query subsetting ' || qstatus);
6    if qstatus = 'OFF' then call send(_frame_,'_set_msg_','NOTE: Query subsetting OFF. All patients now being used');
7    else call send(_frame_,'_set_msg_','NOTE: Query subsetting ON. Only ' || 'patients of the active query will now be used');
endmethod;
RETURN;
```

Figure 6 -- Code used to override _INIT_ and _SELECT_ methods

1. In general, when you override inherited methods you should still execute the original, inherited method through the CALL SUPER function. For the method you pass to CALL SUPER as a parameter (in this case, the _INIT_ method) CALL SUPER determines the location of the inherited version of that method (i.e., the method of the parent class, or superclass). In some cases, you must retain the processing of the inherited method. For details, see Methods section above, see Table 2, and see Frame manual, especially pages 116 and 142.

2. We can use a simple assignment statement for our instance variable QSTATUS because it is an automatic instance variable. If it were not automatic, then you would need to use SCL list functions to set the value. See FRAME p. 136.

3. _SELF_ is a special variable. It refers to the object that has called the currently executing method. Its use allows the creation of generic methods and can result in more efficient code. See FRAME manual, especially page 114.

4. _FRAME_ is a special variable (like _SELF_). It refers to the calling frame, and is used to set FRAME attributes. See FRAME manual pages 213, 239.
**RESOURCE LISTS**

You cannot use your new class until you add it to a resource list. Resource lists are the lists which appear when you press the right mouse button. They are used to store information on the available classes for a frame. Resource lists are well documented in SAS publications, so a cursory review of this topic is presented here.

The next section guides you through creation of a resource list to allow you access to your QTOGGLE class.

To create a custom resource list just copy the default list and edit it. Copy the default list by entering the enter the command:

```
COPY SASHELP.FSP.BUILD.RESOURCE
```

from the directory window. This will create an entry BUILD.RESOURCE in the CLASDEMO catalog. From this point on, any frame entry you create in this catalog will use this resource list.

Next, open an edit session on the resource entry (e.g., EDIT BUILD.RESOURCE). From the resource entry window (see figure 7) select the Actions pmenu and then select Add from the pop-up list. Finally, from the Select window, choose SASUSER, CLASDEMO, QTOGGLE, and then select OK. Your class is now added to your custom resource list. When you next open a frame entry in this catalog and press the right mouse button you will see the class QTOGGLE at the bottom of the pop-up list of available classes.

One warning on resource lists: if you edit an existing class (e.g., add an instance variable) you must update its instances by using the SYNCHRONIZE option in the resource entry window. If you do not, then existing instances of your class will not reflect your modifications.

**TESTING YOUR NEW CLASS**

To test your class, edit a frame and create a QTOGGLE object. Compile the frame and then just TESTAF it. When you click on the QTOGGLE pushbutton, the text value on the button toggles between ON and OFF. At the same time, a macro variable value is being updated. In the application in which this class is used, this macro variable controls the inclusion/exclusion of subsetting statements in SUBMIT blocks in all modules of the system.

**CONCLUSION**

Creating custom classes can be beneficial if you work frequently in SAS/AF, particularly for teams of developers working on large projects. While it is vital that careful planning be used in creating class hierarchies, proper application of OOP principles to SAS/AF application development can provide significant benefits.

For further information, feel free to contact the author at daniel.gronell@roche.com

**REFERENCES**


Taylor, David, Object-Oriented Information Systems: Planning and Implementation, 1992, John Wiley & Sons


SAS/AF Software: FRAME Entry, Usage and Reference, Version 6, SAS Institute, Cary, N.C.


Cartier, Jeff, et al., Object-Oriented Graphical User Interface (OOGUI) Using FRAME Entries in SAS/AF Software, Parts I-IV, SUGI 18 Proceedings

Pierce, Randy, The Benefits of Object-Oriented Application Development Using the SAS System, SUGI 19 Proceedings

186