Getting Into a New FRAME of Mind
Object Oriented Programming with SAS FRAME ENTRY Software
Paul Butler
Gartner Group

Abstract

The transition from traditional modular or structured program design to an object oriented approach is often a gradual one. Programmers not experienced in implementing applications using the OOP paradigm frequently continue to use structured techniques and slowly incorporate OOP strategies as they become familiar and more comfortable with them. It is likely many SAS Frame Entry programmers follow this trend.

This paper will examine three different techniques used to perform the same task (employing the SAS Frame Entry), and then make comparisons as to the ease of their implementation and the degree to which they follow the object oriented model. Each method reviewed will use an increasingly object oriented approach, and in so doing illustrate the often deliberate transition when shifting from a structured programming methodology to an object oriented one. It is assumed that the reader is familiar with object oriented programming (OOP) terminology and concepts such as that of class, encapsulation, instantiation and inheritance. It is further assumed that the reader has at least a basic knowledge and experience of the Frame Entry and it's interactive development environment.

Introduction

One of the Frame Entry’s predefined classes is the ICON class. When creating an icon within a frame, the developer can select from hundreds of existing images to insert into the icon. This is done in the ICON class’s attributes window. Even though a developer may not know how to create a class or inherit methods as one would when applying OOP techniques, SAS has designed Frame Entry software with an OOP approach, and so the Frame Entry user makes use of OOP; though be it unwittingly.

A key feature of OOP is the use of encapsulation; the binding together of data, and the methods (or subroutines in a structured world) that manipulate the data. Data is stored in variables and those variable's values are altered or created through the methods associated with it.

Perhaps encapsulation is the primary characteristic that differentiates objects from a collection of variables and subroutines in structured programming. Methods can be thought of as subroutines that determine the behavior any given object will exhibit. An object is a thing (in this case an ICON) that knows about itself (it's data) and the actions it can perform (it's methods).

A class can be thought of as a kind of template. It is the declaration in the generic sense of all the data and methods that compose an object. The object's attributes window provides an interactive way to edit the values of the variables that contain the data portion of the object's declaration. The variables that store the characteristics of an object are known as instance variables.

One of the major objectives of OOP techniques is to maximize the reuse of code. The primary mechanism that allows this to happen is called inheritance. In the Object Oriented paradigm, we don't copy code as we would in structured programming to reuse code, rather, OOP languages provide a means to define a new class based on some other existing class. The base class object's behaviors are passed on to the new class. Also, in OOP not only can we add behaviors to new classes but we can filter out unwanted characteristics as well.

The Sample Application

In order to illustrate how we move from copying chunks of code (by the way that is the technical term for that - "chunks of code") to inheritance, we will consider a small example application that generates a series of simple reports. These reports list average CPU time consumed for a series of program executions. There are seven different reports which can be produced, one for each program set executed on a variety of operating system environments (i.e. TSO, CMS, CICS). The format used for each listing is the same (across environments) in that each shows the CPU time consumed for each program, as well as the total CPU time for the series. However, the names and the number of programs in each series differs by environment as does (sometimes) the unit in which the CPU time is measured. The data needed for each report is contained in it's own SAS data set.
If we were designing a generic SAS routine to handle the production of a report for all the environments, a likely solution would be to implement a SAS macro with a list of parameters; one for each piece of varying information needed to generate each of our resource consumption reports. The macro might look like what is shown in listing 1.

```sas
%mACRO GenRPT(_CPUNIT=, _TITL=, _DSN=);
    proc report data = &_DSN;
        title &_TITL.;
        where datatype = "&_CPUNIT";
    run;
%mEND GenRPT;
```
Listing 1

Method 1 - Using the most of what you got.

The application will use two frames (see figure 1A and B). The first contains a series of menu icons indicating choices available for operating system environments. Each icon represents the report produced for that operating system. When an icon is selected, the second frame is displayed. This frame presents an extended table which displays the report results. Although objects in the second frame will be discussed, their creation and the code used to determine their behavior is not the primary topic of this paper, so details will be skipped. Our focus will be on the menu icons contained in the first frame. Throughout the examples, fundamentals of SCL lists will be briefly discussed because of their importance to Frame Entry programming.

![Figure 1A](image1.png) ![Figure 1B](image2.png)

One way to implement this application would be to have a separate frame and .SCL entry called by each icon in the first frame. We could copy the entries seven times and edit each one so that it's code would refer to the appropriate SAS dataset, report title and unit of measure. Unfortunately, this method increases the chance for error, elongates testing time, and makes maintenance cumbersome. If the same frame and .SCL entry could be called by all the icons within the first frame, development time would be reduced, and perhaps more importantly, maintenance would become much easier. To allow this to happen SAS Frame Entries have the ability to share code. To do so, simply specify the same .SCL entry name in the frame's general attributes window.

All our icons can now share the same code, but how do we let this generic code know about all the things that vary; like report title and dataset name? If we could pass the report generation frame a list of parameters that would logically function like the %macro example in listing 1, that would solve the problem. The first method we will explore does just that, but in a slightly different way than the macro example suggests.

A box labeled "additional attributes" in the standard icon attributes window lists several more object attributes the user can manipulate. The first of these is the value on selection attribute. This is a predefined value that can be retrieved by referencing the object by name during assignment statements or by SCL function call. The value entered into the box labeled value can be accessed by one of two SCL methods: _get_text_ (if the value was declared as character in the datatype radio box), or _get_value_ (if the value was declared as numeric). We have placed a string delimited by a ^ in the value on selection box. The delimited values are positional arguments that will be passed to the report generation frame as a single item. Listing 2 shows the SCL entry for the sample application's report generation icon menu frame. Rather than having named blocks of code for each of the icons in the frame, a couple of very useful frame methods are used.

```
001 INIT:
002    length iconame $ 8 /* stores name of current icon */
003    argstring $ 132; /* stores argument string */
004    Return;
005 MAIN: /* Get name of current selected object. */
006    call notify("", _get_current_name_, iconame);
007 /* Retrieve text from "return value" attribute. */
008    call notify(iconame_, _get_text_, argstring);
009 /* Activate report generation frame passing ICON's */
010 return value string as argument. */
011 if iconame ^= "QUIT" then
012    call display('library.catalog.member.frame', argstring);
013 Return;
014 TERM:
015 Return;
```
Listing 2

In SAS Frame Entry software, all predefined classes are derived from one class. SAS has termed this one class the object class and it is the ADAM and EVE of all classes; so to speak. It is unfortunate SAS has chosen the
word object as a name for it's super parent class as it tends to confuse the distinction between the terms class and object. Immediately child to the object class are the frame and widget classes. Any object inserted into a frame is ultimately derived from the widget class. The icon class is no exception, therefore all the methods available to the widget class become automatically available to the icon widget. The frame itself is an object and so all frame class methods are available to any SCL program. Among the frame class behaviors is the _get_current_name_ method. In line 006 of listing 2 we use a newer SCL function call notify to invoke _get_current_name_. Call notify is a general purpose way to call class methods. The _get_current_name_ method returns a string indicating the name of the currently selected object on the screen, loaded into a character variable provided as the third argument. It's syntax is as follows: Call notify('objectname', 'method_to_call', variable_to_store_return_value);

In many cases 'objectname' is passed as a text string. Line 006 of listing 2 uses a slightly different way to pass the object name to the call notify function. Instead of a string naming the object, a single quoted period is passed. The use of '.' as the object name tells call notify to use the name of the currently selected object. That way we don't need to know it's name and can reuse this code.

The method _get_text_ will return the string entered into the value on selection box when the object was created. The icons in the icon menu frame contain a string detailing the information needed by the report generation frame. This information can be thought of as one of the icon's characteristics. Retrieving and passing this information to a second frame is a behavior of the icon class that we are defining. As we shall see, argstring's value will be referenced in the INIT section of the report generation frame.

Any frame can optionally accept any number of arguments when it is called. The entry statement is used to declare arguments that are to be received by the frame. Listing 3 show the INIT: section of the report frame. The scan function is used to separate the argument string into it's individual components. Four SCL variables are used to store the extracted information, which is positional in nature. When users press the "Go" button, the frame can use the extracted information to construct and display the report.

Listing 3

This technique resolves the problem of communicating information to the report frame, allowing the same source code to be executed for all the icon objects. Maintenance is relatively easy at this level of complexity, but what if there were not four, but twenty bits of information we wished the icon object to know about itself? Clearly there are limitations to this method. The menu of icons store the information in one predefined instance variable edited in the value on selection box, if not stored there, then how does one encapsulate the desired characteristics directly into the object? If this is done, do behaviors need to be defined to manage and process the information? As we have mentioned, object oriented programming makes it possible to describe new characteristics and behaviors for a class by creating a new class based on another. This thing we call inheritance may provide a more flexible and elegant way to implement a specialized icon object.

METHOD 2 - In a class of it's own

Another way to implement a mechanism to produce the reports in the sample application is to create a new class of icon; adding new characteristics (in the form of instance variables) and behaviors (in the form of methods). Information the icon needs to know about itself, such as what dataset to use for the report, can be stored in new instance variables, and methods can be written to retrieve that information when needed. Frame Entry software provides an interactive way to create and edit classes by way of the class editor window.

The class editor window can be invoked by typing edit libname.catalog_classname.class on the command line. For example, to create a new class declaration for the report generation icon class, we can type edit rpticon.class. The class entry: field simply indicates the library, catalog and entry name for the new class (this will become a .class file type). We need a way to tell
SAS on which class to base the new class. This is done in the **Parent class**: entry box, where we must indicate the location of the declaration entry for the parent. We have based the new report icon on the predefined SASHELP.FSP.ICON class.

The **additional attributes** box lists five other class attributes that can be altered to add new characteristics and behaviors. Selecting the **Set custom attributes** calls a window where an alternate object attributes window can be specified. We have filled out the **custom attributes** text entry box to point to rsccat.dev.diffattr.frame which we will need to create. We have selected the radio option to replace the default attributes window and instead use the frame rsccat.dev.diffattr.frame. Keep in mind that the object attributes window is itself simply a frame, and so can be designed by the Frame Entry developer, rather than SAS.

The objectives of creating a new report icon class include providing a way for the report icon to store information about itself required to generate a report in our sample application. Also, this class needs a way to communicate that information to the report generation frame. Our strategy now will entail storing the data in it's individual parts instead of in a single string, and then providing a new method to package that information and pass it all at once to the object requiring it; in this case, the report frame. To do this we select the box option **Methods...** in the class editing window which calls a window (figure 2A) where the developer can declare new methods for the class, or override existing ones.

![Figure 2A](image1)

![Figure 2B](image2)

The **Methods** list box shows all the predefined methods that are either inherited or were created for the class. Though not written yet, at this point in the design of the new report icon class, a new method for retrieving report information from the class can be declared. Selecting the **Actions** button and choosing **Add mode ON** from the menu will display two new buttons, **new** and **reset**. These will allow the specification of an .SCL entry where the new method will be located. We have placed methods called GETREPT and SELECT in an .SCL entry. The SELECT block is going to override the action that is normally taken when an ICON is "clicked".

The labeled block of code where the method is located within the .SCL entry is indicated in the **Label**: text box. The **Name** box stores the string of text that will be used to call this method. This is used so that you can give a name to a method that is greater than eight characters long. In our example, I have given the method and the labeled block the same name. Clicking **OK** saves the class entry.

Now that new methods have been declared, new instance variables need to be created to store the new characteristics of our report icon class. One of the five additional attributes in the class edit window is **Instance variables...**. Selecting this calls up a window similar to the **methods** window as shown in figure 2B. After selecting **Add mode on**, new instance variables can be created simply by typing their name and choosing the appropriate radio button to describe the datatype of the variable. We have added six new instance variables (as shown in Table 1) that will store the information we want the report icon to know about itself.

When the radio box labeled **Automatic** is selected, the value of the instance variable is automatically made available so that method calls do not have to explicitly retrieve or set their values. However, this is only within methods for the class. Automatic instance variables are not globally available throughout all code. SAS stores all instance variables, and for that matter, everything about an object in form accessible as special kind of array called an SCL list. We will discuss SCL lists only briefly as a full description could easily consume the rest of this paper. Consult the SAS: Frame Entry Reference for further information on SCL lists.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Auto?</th>
<th>Is to Contain...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISP_APP</td>
<td>CHAR</td>
<td>YES</td>
<td>Object Attributes Window</td>
</tr>
<tr>
<td>DIFFLIST</td>
<td>NUM</td>
<td>YES</td>
<td>SCL List ID of package</td>
</tr>
<tr>
<td>ENVIR</td>
<td>CHAR</td>
<td>NO</td>
<td>Operating Environment</td>
</tr>
<tr>
<td>SCRT</td>
<td>CHAR</td>
<td>NO</td>
<td>Report Generation Frame's Environment Title</td>
</tr>
<tr>
<td>DDSN</td>
<td>CHAR</td>
<td>NO</td>
<td>Dataset source for report</td>
</tr>
<tr>
<td>RPTT</td>
<td>CHAR</td>
<td>NO</td>
<td>Title for Printed report</td>
</tr>
</tbody>
</table>
Table 1

We have completed three of the conceptual steps in creating a new class of object for the sample application, those being:

1. Allow for a way to store data about the object within the object itself. (Instance variables)

2. Allow the object to know how to use the data about itself to perform new and specific actions. (Methods)

3. Define a new object attributes window to use so the new data about itself can be queried when the object is created in a frame by a developer. (New attributes frame)

When we edited the class entry for the report icon class in the class definition window, we specified an alternate object attributes frame to call rather than the predefined one used with the parent icon class. This window is a simply a frame entry in itself and will need to be created. Figure 3 depicts the custom attributes window that was designed for the report icon class which has new text boxes used to store the four items of information needed by our single report generation frame (Table 1). The text entry box labeled SASSAF Entry is used to indicate the name of the report generation frame our icon is to invoke when selected. Several new features have been included into this new attributes frame. We will not review this frame's code closely as our focus is in class creation, not SCL coding techniques in general.

Further discussion about our sample report icon class would be difficult without introducing an essential element in it's creation; a very powerful and more recent feature of SCL. That feature is SCL Lists. An SCL list is a dynamic array that can store both character and numeric data within the same array. Further, the element locations can be named, so that elements can be referenced directly by name when their index position is not known. SCL lists can be created and resized at run time to accommodate lists of unknown or changing size.

SCL lists also have their own identifiers that act as one number that points to the whole list. These identifiers are given to the list at creation and are functionally similar to pointers in C or Pascal. By using list identifiers, one can nest a list within a list. List identifiers allow a program to pass an entire list's contents to a frame or method as a single argument. Our sample application will use SCL lists to package several pieces of information into one list and then pass that package to a frame as a single argument. Recall that in our structured solution to the sample application, we packaged the pieces of information to be passed to the report frame as a "^" delimited string. Now we will embed (encapsulate) the information into the object itself in the form of instance variables. We will then write methods for the class to perform the required list management and instance variable manipulation.

List packaging not only reduces the number of arguments passed during method or frame calls, but also increases code extensibility because the list size can be increased to any number of elements with no changes required to the code that processes those items. Table 2 briefly introduces some of the SCL functions used to create and manipulate SCL lists.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>makelist</td>
<td>create lists</td>
<td>listid = Makelist();</td>
</tr>
<tr>
<td>listen</td>
<td>get length of list</td>
<td>len=listen(listid);</td>
</tr>
<tr>
<td>setitemc</td>
<td>insert character item</td>
<td>lid=setitemc(lid,chartoadd, index#,autogrow list?);</td>
</tr>
<tr>
<td>setitemn</td>
<td>insert numeric item</td>
<td>lid=setitemn(lid,unumtoadd, index#,autogrow list?);</td>
</tr>
<tr>
<td>setiteml</td>
<td>insert list identifier</td>
<td>lid=setiteml(lid,listidtoadd, index#,autogrow list?);</td>
</tr>
<tr>
<td>getitemc</td>
<td>get char item</td>
<td>char=getitemc(listid,idx#);</td>
</tr>
<tr>
<td>getitemn</td>
<td>get numeric item</td>
<td>nval=getitemn(listid,idx#);</td>
</tr>
<tr>
<td>getiteml</td>
<td>get list identifier</td>
<td>lid=getiteml(listid,idx#);</td>
</tr>
<tr>
<td>nameditem</td>
<td>search for named item</td>
<td>pos=nameditem(listid,item name, occur,startidx);</td>
</tr>
</tbody>
</table>

The set and get functions are available in their named item form: setItemc, getItemc... These take an element name as opposed to an index value argument. This is only a partial list of all the SCL functions accessible for use in list manipulation.

Table 2
Now code can be written to add the following new behaviors our report icon class is going to need:

1) Provide a way to call the custom attributes frame when creating a report icon object. We have already told the new class the name and location of the new attributes frame in the set custom attributes window. Our attributes frame will need code to record instance variable values onto the object.

2) Provide a way to collect and package required characteristics into an SCL list to be passed to the report frame. The SCL list identifier containing the packaged data will be placed into an instance variable for the object.

3) Provide a way to retrieve the packaged data. This can be done in two ways; as a class behavior, or within any SCL code that will access the package (SCL list) - in this case the report frame.

A new attributes frame

OOP languages provide a way to change inherited characteristics and behaviors. The process of doing so is called overriding. In our new class entry we specified labeled block sections we named GETREPT and SELECT. GETREPT is a new method for the class that will un-package information contained within an SCL List. SELECT will replace what happens when the ICON is selected. We used the methods... window to call up the dialog that allows us to create new methods for the class. The state textbox below the source box will show the status of the method; either new, Inherited or override. We changed the source group box to point to rscsat.dev.difficon. It is in this catalog member we shall place the code for new SELECT and GETREPT methods.

Listing 4 shows the code for the SELECT block that overrides the default _select_ method. There is an important concept here to remember. Any method that is overridden by a new method must invoke the parent class method it is overriding; unless the developer is certain they are replacing all of the actions the original class method performed. One will almost never do the latter as it would mean that the designer has made a poor choice of parent. If you find yourself overriding many or most of the actions that parent class methods perform, it is likely a mistake has been made in selecting a parent class in the first place.

```sas
length disp_app $ 40 envir $ 25 listid $ 8;
SELECT;
method;
/* Get the list identifier of the SCL list containing the package of instance
variable values we can then pass as a single argument */
plist=getsitem(_self_,'difflist',1,1,0);
/* call the frame described in that instance variable */
if length(disp_app) > 0 then call display(disp_app, plist);
call super(_self_,'_select_');
endmethod;
```

Listing 4

The SCL function call _super_ is used to invoke the parent class method as shown in the last line of code in Listing 4. This function is utilizing an automatic global variable in Frame Entry software; _self_. SAS loads the _self_ variable with the identifier of the object that called the method. Therefore, when an icon object is selected at run time, the variable _self_ will contain the object’s own identifier. First _self_ is passed as the object identifier to the getinternal function. This will return the SCL list identifier stored in the new class instance variable called difflist. Recall that difflist is the name of the instance variable list containing the pieces of information about our icon that need to be passed to the report generation frame. Disp_app is the class instance variable we created to store a string that specifies the catalog and entry name for the report generation frame to be displayed when the new class icon is selected at run time. Finally, we must call the parent _select_ method so it can perform all the other behind the scenes activities that occur when an icon object is selected. Notice that the variable plist, which contains the list identifier for our "package" of data is passed as an argument to the frame we display with call display. The receiving frame can now un-package this list because it will receive the list identifier as an incoming argument.

Now our new class knows how to package information about itself, call another frame when selected and pass that frame an SCL list identifier to allow the called frame to get at the list of information. These are significant new features for our class that will allow users to create as many icon objects as they wish without having to copy or edit any code. The class can be altered by the class designer, and once the class entry is compiled, new behaviors will be exhibited in applications utilizing the class without so much as a recompile! Some extra effort is required to carefully design the class, but clearly gains can far outweigh the expenditure.

We must still consider two things. The first is how we package and unpackage the report information into the list difflist, and second, how we make our new class
available to users when they select the make menu action. The alternate attributes frame we will create for the new icon class will do the job of instantiating the class instance variables. One of those variables is `difflist`, our SCL list package.

The attributes frame contains one text box for each piece of information we will store; such as the data set name to use for the report and the report title. When the user is creating a new icon object, the attributes frame serves as a query tool for this information. Once the "OK" button has been pressed we want to save the information in the class's instance variables. The logical place for this to happen is the TERM: section of the attributes frame.

Listing 5 shows the TERM section of the SCL entry for our Icon class's attributes frame. If cancel was not pressed then each of the class instance variables are assigned a value using the SCL `setitem` function. Recall that SAS uses SCL lists to store all the instance variables for an object. Like all lists, that list must have an identifier as well. If we knew that identifier we could assign and retrieve values in the list. In fact we do know that list's identifier. It is the special Frame Entry “system” variable `_widget_` that contains the identifier for the list that stores the attributes for the object.

We do not call the new attributes window with call display anywhere in our code, rather SAS does this because we specified a new attributes frame in the class editor window when we created the new class. When this is the case, SAS passes in three parameters to the called frame; the list identifier that contains a copy of the widget's attributes, a string containing the entry name of the custom attributes window and the SCL list identifier for the class to which the object belongs. The ENTRY statement in listing 5 specifies these parameters as `_widget_`, `_uattr_` and `_class_`. We use `setitem` function to place the items in the list and name the location. First we set the instance variables outlined in table 1. Next, we create a new SCL list with the `makelist()` function. We then load a copy of our instance variables into the list referred to by `mylist`. The `mylist` identifier is then saved in the class instance variable `difflist` with `setitem`. We are nesting an SCL list containing a subset of the class instance variable's values within the SCL list storing the object's attributes, and so we will now be able to pass that collection as a single argument by referring to the `difflist` list identifier.

Listing 5

```
ENTRY optional=_widget_ uattr_ $ class_;
Code........
TERM:
Link Label;
if _status_ = 'c' then return;  /* do nothing if user selected "cancel" */
/* store attributes back on the _widget_ being edited */
.widget_ = setitem(_widget_, "NAME");
.widget_ = setitem(_widget_, "TEXT");
.widget_ = setitem(_widget_, "display", "disp_app");
.widget_ = setitem(_widget_, "scroll", "hor");
.widget_ = setitem(_widget_, "icone", "icon");
.widget_ = setitem(_widget_, "env", "envir");
.widget_ = setitem(_widget_, "diffism", "/data");
.widget_ = setitem(_widget_, "reptitle", "rpt");
/* save a copy of this list in list instance variable */
mylist = makelist();
/* add data that was selected in attributes window */
mylist = setitem(mylist, "display", "disp_app");
mylist = setitem(mylist, "scroll", "hor");
mylist = setitem(mylist, "icone", "icon");
mylist = setitem(mylist, "env", "envir");
mylist = setitem(mylist, "diffism", "/data");
mylist = setitem(mylist, "reptitle", "rpt");
/* save the pointer to mylist in instance variable difflist */
_difflist_ = setitem(_difflist_, _mylist_, _mylist_); Return;
```

Similar code is contained in the INIT: section of the attributes frame. The information is then loaded into the text boxes on the attributes frame so that current values for the object are displayed when a user edits an object previously created. The `difflist` list identifier is what is retrieved and passed to the report generation frame within the code we created to override the parent icon class's _select_ method. The life cycle of the data attributes of our new class should now be evident. We have implemented our design goal of providing storage and manipulation of new characteristics and behaviors for the class.

The final step in creating our new icon class is to make our new class available to other users. The class must be added to the resource list that users are presented when they have selected the `make` menu option. SASHELP.FSP has an entry called BUILD.RESOURCE. This is where SAS keeps a list of all available class entries. Editing this entry will call a window to manage the classes presented in the `make` pop-up menu. See SAS documentation for details.

Our sample application's report generation SCL entry making use of the new icon class is nearly identical to the non-class based SCL. The only difference is that now we are receiving an SCL list identifier containing the same information and extracting it with `getitem` rather than `scan`. The workings of the SCL is otherwise the same.
A new menu icon screen can now be created using the new class entry we have placed in the RESOURCE member. The icon menu frame now needs no code at all. The new attributes screen will query, store and pass on the characteristics the icon objects know about themselves. Creation of new report screens is as simple as typing in dataset names, titles and the like.

Message in a bottle...

We have come a long way in being able to develop easily maintained, extensible and reusable code in Frame Entry software. However we still had to write some code that was dependent on our knowledge of certain characteristics and behaviors of the new class we created. For instance, when we unpack the list of information we have to know what the names of the list items are, as in: envname=getnitemc(listid, 'envr');

What if we could also encapsulate the packaging and unpackaging of instance variables into the class definition? We could then simply tell an object: "propogate all this information", or "tell object X to do such and such". SAS Frame Entry software has given us the capability to do these things by implementing a popular Windows concept: messaging. Objects can communicate with each other and tell each other to do things. One object sends a message to other objects which can ignore the message, or act on it dependent on whether the object was designed to know what to do with the message. In Frame Entry software sending a message is called BROADCASTING and as expected, receiving a message is called RECEIVING.

Typically, users will need to develop class methods to process messages that objects based on the class will potentially receive. In keeping with Frame Entry design philosophy, SAS uses SCL lists to store all the messages an object can send or receive, as well as the names of all the objects that can send or receive messages from (or to) itself.

The Frame Entry object class contains a few useful instance variables, some of them SCL lists themselves, that are used to keep track of things like what messages an object can send or receive. Unfortunately, it is here the SAS terminology plays a little chicken and the egg routine and gets a bit confusing. Within the OBJECT class definition there is the _RECEIVERS_ instance variable. This contains a list of the objects that can receive messages from the object in question. Likewise, there is the _SENDERS_ instance variable that contains a list of all the objects that can send the object in question a message. But any given object can both send or receive messages at any point in time. What becomes confusing is knowing from what perspective are these instance variable lists referring?

To reduce confusion, think of _SENDERS_ and _RECEIVERS_ as talking about oneself. Me. If I am an object what are the objects I can send messages to. This is my _RECEIVERS_ list. Conversely, who can send ME messages. This is my _SENDERS_ list. Just to make it crystal clear, SAS throws in _RECEIVE_ (not plural) as an instance variable of the object class. The _RECEIVE_ instance variable contains a list of all the messages the object understands. The mechanism by which messages are broadcast is the _BROADCAST_ method: call notify(objectNAME, '_BROADCAST_', Themessage, AnyNumericARG, AnyCharARG);

Call notify is used to invoke the _BROADCAST_ method, instructing SAS to send Themessage, along with optional numeric and character arguments to all the objects in objectNAME's _SENDERS_ list. The behaviors defined for those receiving objects determine if Themessage is one they understand.

Users can invoke the Object Links Window at design time to establish the _SENDERS_ and _RECEIVERS_ list, however, we will illustrate a more dynamic run-time method of doing this.

Our sample report generation frame contains three text boxes that are loaded with descriptive information pertaining to the particular environment. Rather than needing to know the list element names, we can employ broadcasting to tell all the list boxes that we have received information they need. The message we send can invoke a method that will look for the piece of information in the list that each textbox requires. We perform one _BROADCAST_ method call, and the textboxes do the rest. We could achieve messaging type behavior with simple class method calls, but it would take more than one call and _BROADCAST_ automatically insures there is no infinite recursion when a receive method invokes another _BROADCAST_ to itself, removing the burden from our code.

When a _BROADCAST_ method is invoked, the message is sent to all of the object's _RECEIVERS_, and in turn the receiving object calls a built-in method to receive the message (the _receive_ method). We need to create a new class of textbox that will know how to
receive and process a message. We will do so in the same manner as illustrated earlier with our new icon class. Instead of overriding the _select_ method for the class, we will override the _receive_ method. This will allow us to intercept a broadcast and take action based on an appropriate message, that we will determine.

We have created a class entry called MYTEXT::CLASS that is no different than the default textbox class with the exception that the _receive_ method has been overridden and a new method, RECEIVE, has been indicated to be in rsceat.dev.mytext.scl. The _receive_ method expects four arguments, three of which are the same as the _BROADCAST_ method arguments. The first three are the message itself, a numeric argument and a character argument. These are simply passed on by _BROADCAST_. SAS also automatically passes a fourth numeric argument; the sending object's object identifier. This is included so that the _RECEIVE_ method can access the sender if necessary. Sometimes this is so that the sending object can be sent notification of the success or failure of the message's actions. In our case it is needed so we can invoke the parent _RECEIVE_ method. Recall that we must call the parent method we are overriding so that the rest of the behaviors we are NOT changing can be carried out.

Listing 8 shows the RECEIVE method we have created for the MYTEXT class. We are going the pass the list identifier we referred to as DIFFLIST in our prior version of the sample application, when we call the _BROADCAST_ method to send all the textboxes a message. The list contains the report title as well as the other information we need to know. We will pass the list identifier as the numeric argument part of the _BROADCAST_. This then is passed to the RECEIVE method we have told to override the default _receive_ method. The message we will send is called "UNLOADLIST". This message will need to be added to the receive list for the MYTEXT class. We can do this at design time in the class editor window by selecting the "Receive list..." option in the additional attributes box. When UNLOADLIST is received as a message, we pull out the string the user entered in the initial value box when creating the textbox and load it into the variable "SELITEM". Next, the SCL function nameditem is invoked to search the SCL list referenced by INLISTID for an element named whatever SELITEM contains. If found, the named item is extracted from INLISTID and loaded into the textbox. Finally, the parent _RECEIVE_ must be called with CALL SUPER, passing it all the arguments our overridden method received.

Listing 8

Recall that the invocation of the _BROADCAST_ method will pass, as the numeric argument, the list identifier we knew as our package of information; DIFFLIST. The _BROADCAST_ method passes these arguments to the _RECEIVE_ methods for all the objects being sent the message. In our case these arguments are passed to the RECEIVE method we used to override the _receive_ method for the MYTEXT class. One method can now process any message sent to a MYTEXT object and automatically load textboxes with items in the package. What we need to insure at design time is that each object created based on the MYTEXT class, contains the name of the SCL list element that will hold the string it is to display. For example, when we created a textbox called REPTITLE in a new report generation frame, we placed the string "RPTITLE" in the initial value box. When the above RECEIVE method is called, the package list will be searched for an element named "RPTITLE". When found, it will load it's value into the textbox.

As mentioned earlier, I chose to forego static creation of a _RECEIVERS_ and _SENDERS_ list using the Object Links window. Rather, we will do this at run time during the INIT section of the report generation frame. To do so we will make use of another OBJECT class method: _add_receiver_. This method allows us to add object's to an object's _RECEIVERS_ list. The reason we do this at run time is because we want the FRAME itself to broadcast a message to all the textboxes contained within it. The FRAME class is also derived from the OBJECT class, and so will also have a _SENDERS_ and _RECEIVERS_ list just as any Widget Object would.
The _add_receiver_method uses the following syntax:
call notify(ObjectName , _add_receiver_ ,
ObjectID_To_ADD);

Note that ObjectName can be a string containing the
name of the object or the numeric object identifier of that
object. When a string name is passed, it is translated into
the object identifier for the object. What we want to do is
get the identifier for the FRAME itself and the identifier
for the textbox we wish to add to the FRAME's
_RECEIVERS_list. When an object is added to the
_RECEIVERS_list, it is automatically added to the
_SENDERS_list. For all to work as expected then, we
need only add the textboxes to the FRAME's
_RECEIVERS_list to set up the communication "Lines"
as it were. Once added the _BROADCAST_ will invoke
the _RECEIVE_method outlined earlier to set the
textboxes' values.

Listing 9 contains the new code in a modified INIT
section for the sample application's report generation
frame. We retrieve the object identifier for each of the
textboxes with the Frame class method
_GET_WIDGET_. This method is passed the name of
the object whose identifier is to be loaded into a numeric
variable; passed in as the last argument. The special '!' argument is used to point to the current frame. Call Send
is used to invoke the _ADD_RECEIVER_method and
passes on the object list identifier retrieved with
_GET_WIDGET_. The special system variable _self_ is
used as the OBJECTNAME identifier argument
_ADD_RECEIVER_ is expecting. _self_ is loaded with
the object identifier of the FRAME when it is being
displayed at run-time.

entry optional-listid 8 ; /* SCL list identifier for our information package */

INIT:
call notify ('!', _get_widget_ , 'REPTITLE', rptlid);
call send (_self_ , _add_receiver_ , rptlid);
call notify ('!', _get_widget_ , 'ENVTITLE', scrtlid);
call send (_self_ , _add_receiver_ , scrtlid);
call notify ('!', _get_widget_ , 'DSNTITLE', dntlid);
call send (_self_ , _add_receiver_ , dntlid);
/* Broadcast the message to unload the package list */
call send (_self_ , _broadcast_ , 'UNLOADLIST', listid , '');
environment = getenvironment (listid, 'env'); /* this is graphic text */
link LOADDT ; /* load the datatype list */

Listing 9

As the frame is initialized, each of the identifiers for the
textbox objects are retrieved and added to the FRAME
object's _RECEIVERS_list. Once added, the
_BROADCAST_ call sends the "UNLOADLIST"
message to all of the textboxes. The overridden MYTEXT

_RECEIVE_method processes the "UNLOADLIST"
message by checking for the named element in the list
identifier passed along with the _BROADCAST_. If the
initial value box of the textbox contains a string that
matches the named item in the passed list, its value is
retrieved and loaded into the textbox.

This technique illustrates an alternate method of passing
information between frames and objects. In ideal
situations, the use of broadcasting can be a very powerful
way of increasing code extensibility and reuse. Arguably,
our sample application did not benefit all that much from
re-tooling it to use broadcasting, but has served to
illustrate concepts.

Summary - The inherent moral of the story

In the SAS Frame Entry, the amount of code the
developer needs to write is dramatically reduced,
allowing even novice programmers to design applications
with all the trimmings associated with GUI based
systems such as list boxes, scrollable tables, menu
buttons, graphics and more. However, this is not to imply
that applications development has become so simplified
with an interactive development environment such as the
Frame Entry that solid programming experience is no
longer required.

OOP tends to hide the details of what is happening in
code within class definitions. This can, especially at
first, confuse programmers a great deal. Coding work
done in a structured programming environment is shifted
into class design and documentation. Programmers need
to think in different terms than they did when coding
purely structured programs. Some may even feel that
time is not saved to the degree first thought by using an
OOP approach. This may be due to the fact that more
time is spent up-front during class design and
implementation. However, great savings in time and
effort can be realized when it comes time to change and
maintain applications. New applications can reuse large
portions of code much easier and as we have illustrated,
only adapt to new requirements nearly effortlessly.

Questions or comments can be forwarded by email to:

pbutler@gartner.com