Paper 006-31

Modular Programming using AF/SCL

Kevin Graham, Montura, San Francisco, California

ABSTRACT

Build modular SAS Frame applications using advanced programming techniques and features available in SAS SCL. Easily separate critical logic and business rules from your code base without creating tangled source code. Isolate and integrate decision-logic from task-logic without separating your brain from your sanity.

INTRODUCTION

Object programming is the science of program structure and organization. The goal in object programming is to produce modular and reusable code. The problem is that applications can be object-oriented but not modular. Rules are added, removed, or modified frequently and some changes can affect multiple sections of the application. The implementation of rules is what usually kills program modularity.

This tutorial shows how to make a modular business rule.

WHAT IS OBJECT PROGRAMMING?

Objects and SAS/Macro actually have something in common – they both serve as a structure for the source code that is a SAS program. Thinking of the object, and macro, as simple structures used to encapsulate a SAS program will make this tutorial easier to understand.

The two most important aspects in object programming are “separation of concerns” and “integration of concerns”. During the separation phase, the logic programmers call “decision logic” and “task logic” are physically separated into different programs. During the integration phase, business rule programs are dynamically attached to the other modules. This allows a rule to perform complex operations on the application without adding complexity.

STRUCTURES

A structure is composed of statements that mark the beginning and ending of a program. In Base/SAS, a macro is encapsulated with **%macro** and **%mend**. Object-oriented programs are encapsulated with **class** and **endclass**. This tutorial shows how legacy Base/SAS code can move from macro structure into object structure.

The code fragment below shows an SQL procedure transitioning into an object structure -- without any alteration.

Macro

```
%macro standardStructure;

    proc sql;
        create table work.temp as
        select distinct company
        from main.warehouse;
    quit;

%mend;
%standardStructure;
```

Object

```
class standardStructure;
      step1: method;

         proc sql;
             create table work.temp as
             select distinct company
             from main.warehouse;
         quit;

      endmethod;

      step2: method;
      endmethod;
endclass;
```

Think of the **method** tags as replacements for %macro and %mend. These tags create a substructure used to separate logical steps within a program. Each method must be called individually to execute and methods do not execute in parallel. The following command invokes one method.

call send(_self_, ‘step1’);

Most programs have a lot of steps, so I use an important coding standard at this point; which is coding method names into a LIST at the top of the program. The list named **activeMethods** contains methods considered to be part of the standard execution process. Not all methods will be executed during the standard process.

runInterface is a naming convention that indicates the “main driver”. Every object-oriented program in the application is required to have this method. A do-loop iterates over each method in the specified LIST.

```
class standardStructure;
    public list activeMethods / (initialValue={
                                              'step1',
                                              'step2'
                                             });

    runInterface: method;
        dcl num i;

        do i=1 to listlen(activeMethods);
            call send(_self_, getitemc(activeMethods, i));
        end;
    endmethod;

    * cut ;
endclass;
```

FOUR SIMPLE RULES

A standard program template is used on all object programs, including Frame widgets. These rules become critical during the development and debugging phases.

1. Use a do-loop to iterate through method execution.
2. Declare new variables at the top of the program.
3. Code and execute methods beginning at the top and ending at the bottom of the program.
4. Always insert numeric and character variables from a SAS table into a LIST, replicating the data vector.

OBJECT-ORIENTED DATA VECTOR

Base/SAS
The Data Step data vector, implemented by the SET statement, makes every data value in one row of a SAS dataset accessible by name.

Object Oriented/SAS
The object-oriented data vector, implemented through a LIST, makes every element in the LIST available to every program in the application. In the next section, I'll show why objects are inserted into a LIST.

The figure below shows SCL code moving data into a list.

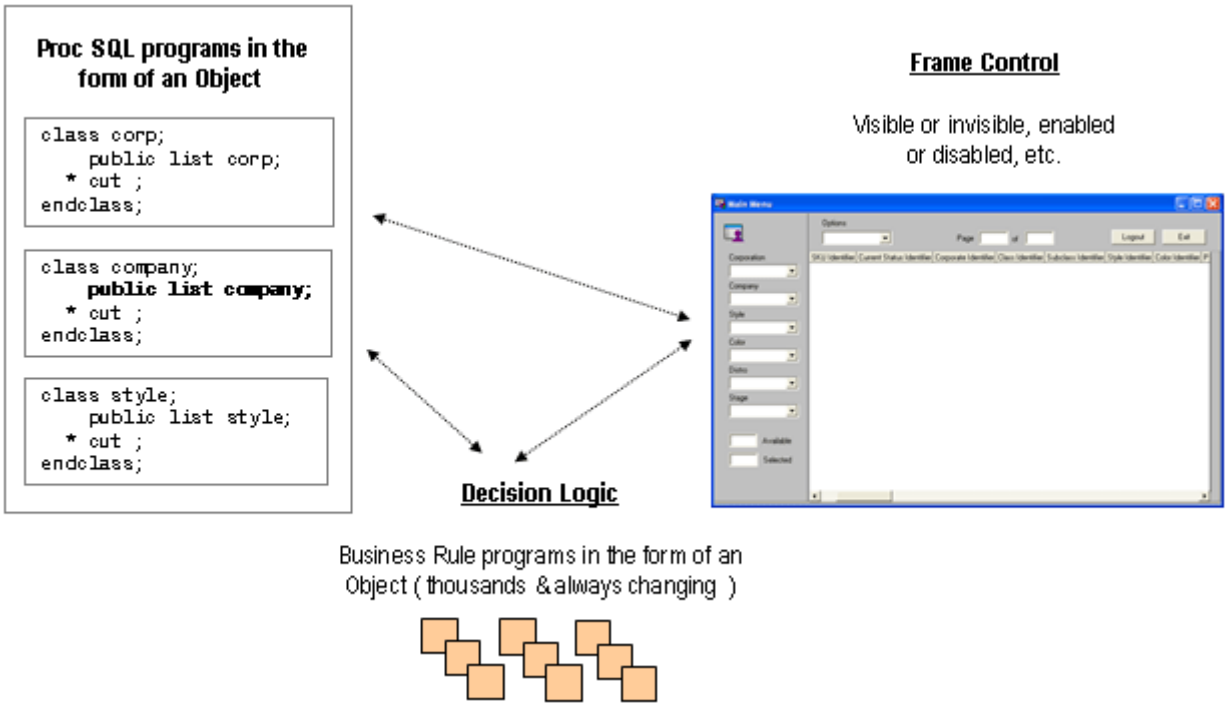
```
class standardStructure;
    public list dataVector / (sendEvent='N');

    step1: method;
    endmethod;

    step2: method;
        dcl num rc dset;
        dset=open('work.temp', 'i');
        do while (fetch(dset)=0);
            rc=insertc(dataVector, getvarc(dset, varnum(dset, 'company') , -1);
        end;
        rc=close(dset);
    endmethod;
endclass;
```

INTEGRATING BUSINESS RULES

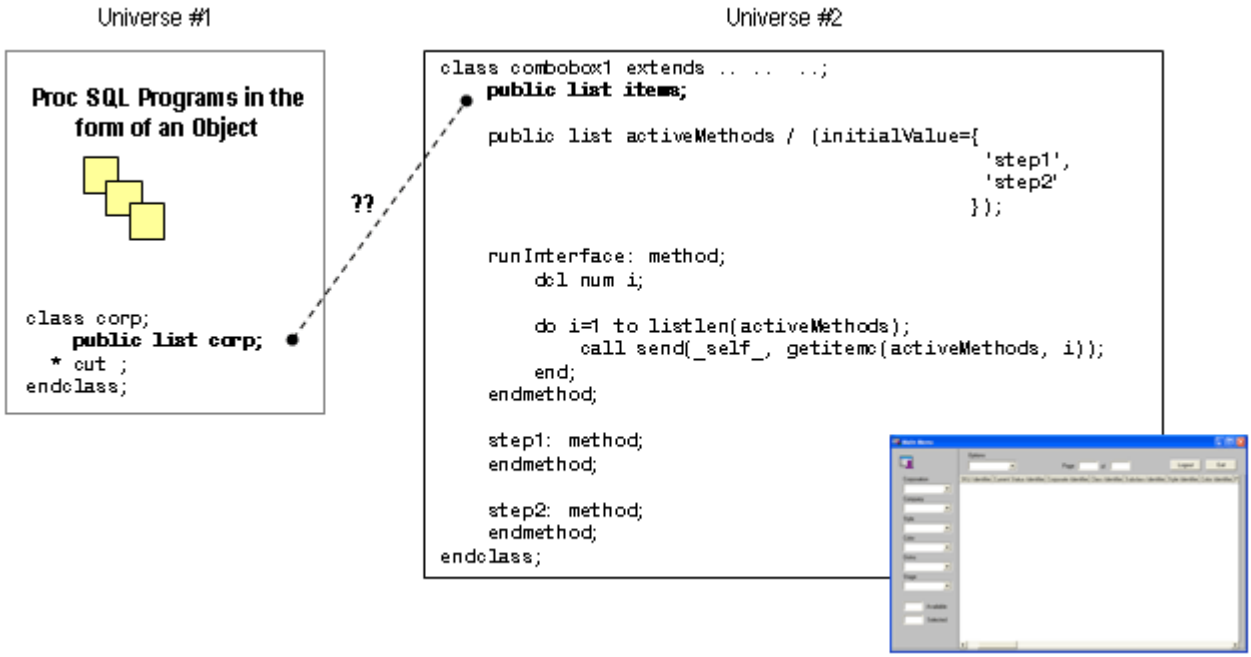
Integrating business rules into any application can create two types of challenges.

1. The planning challenge is to account for a large number of business rules. The number of rules can be expected to increase or decrease, at any time.
2. The technical challenge is to account for the planning challenge without going insane.

Batch-mode applications are far easier to design and code than interactive applications, so normally, the presence of a GUI like SAS/Frame would add complication. The following figure shows a three-way communications path. In this example, business rules modules may need data stored in non-visual program and the status of several Frame widgets at the same time.



A VERY OLD CHALLENGE
OBJECT PROGRAMS DO NOT SHARE



CORP and COMBOBOX1 were both coded to perform a single function. Corp obtains a list of unique values from one column in a relational table and stores the data in a list. COMBOBOX1 simply takes data from CORP for display on the Frame.

CORP and COMBOBOX1 compile and execute with no problem whatsoever. If you attempt to move data from CORP into COMBOBOX then we have a major problem. This is where I found that two object programs in the same

application are unable to communicate. There is no way to transfer data between CORP and COMBOBOX1, so they may as well exist in different galaxies.

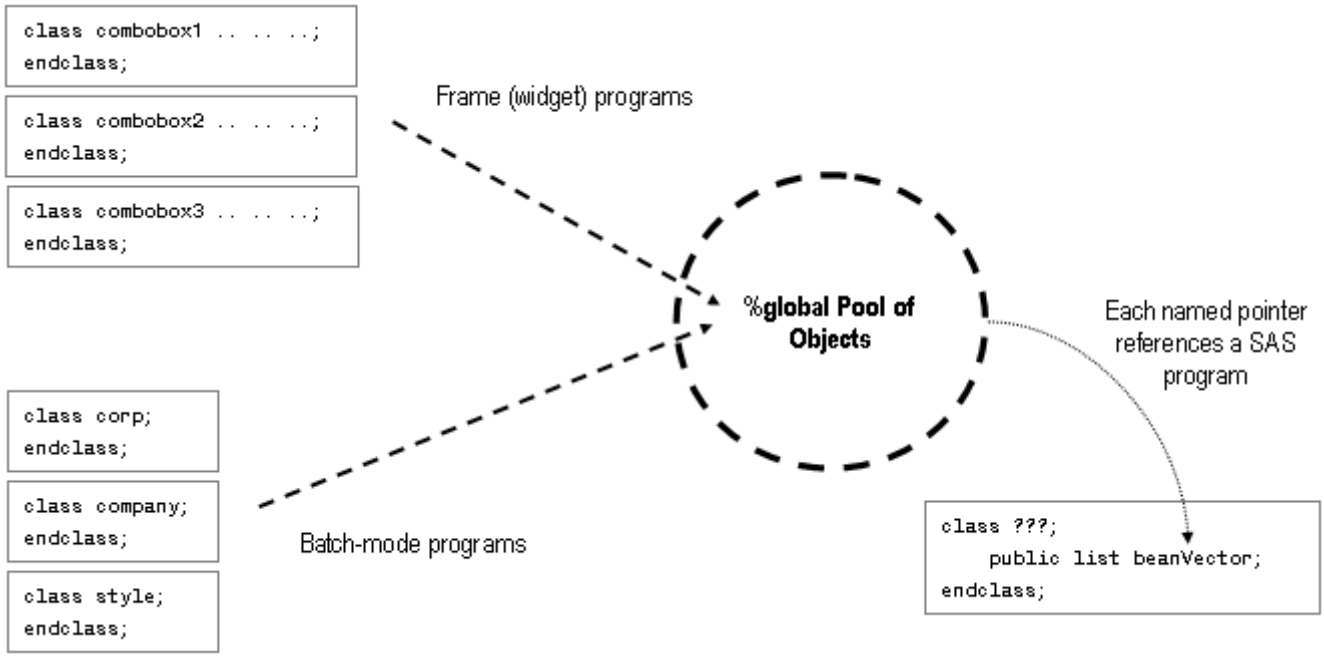
I have never seen documentation that says objects in the same application can't share data because they are unable to "see" each other. Someone forgot about that one. The following code fragment was coded into COMBOBOX1 and shows my first attempt to obtain data located in the CORP program. Of course, it didn't work.

```
step3: method;
    items=corp.corp;
endmethod;
```

%GLOBAL FOR OBJECTS

I began looking for a quick-fix or workaround. To keep my programs and business rules modular, I had to have a simple way to share data between business rule and Frame programs. The %global statement gave me a bright idea -- I needed a globally accessible container that could be filled with object programs.

The Figure below illustrates my thought.



From a logical standpoint, I need to see two things happen.

1. Every program containing data or functionality needed elsewhere must be able to contribute a "pointer" to a global pool - easily. The pointer must point back only to the contributing program.

2. Any program must access to the global pool.

From a financial standpoint, cost was no object, so long as the solution was free. And whatever the cost, the global pool solution had to work exactly the same with Frame widgets and batch-mode programs. It was a major surprise to find that SAS already supported the use of pointers.

WHAT IS A POINTER IN SAS?

A pointer is a location identifier in the form of a numeric variable. The SAS System maps this variable to the physical location of the instance (object) in the computer's memory. The pointer is an automatic variable named **_self_**.

In SAS source code, a pointer is located in a variable declared as an OBJECT. The variable **anyName** does not contain a program. It can only contain information that points to a program.

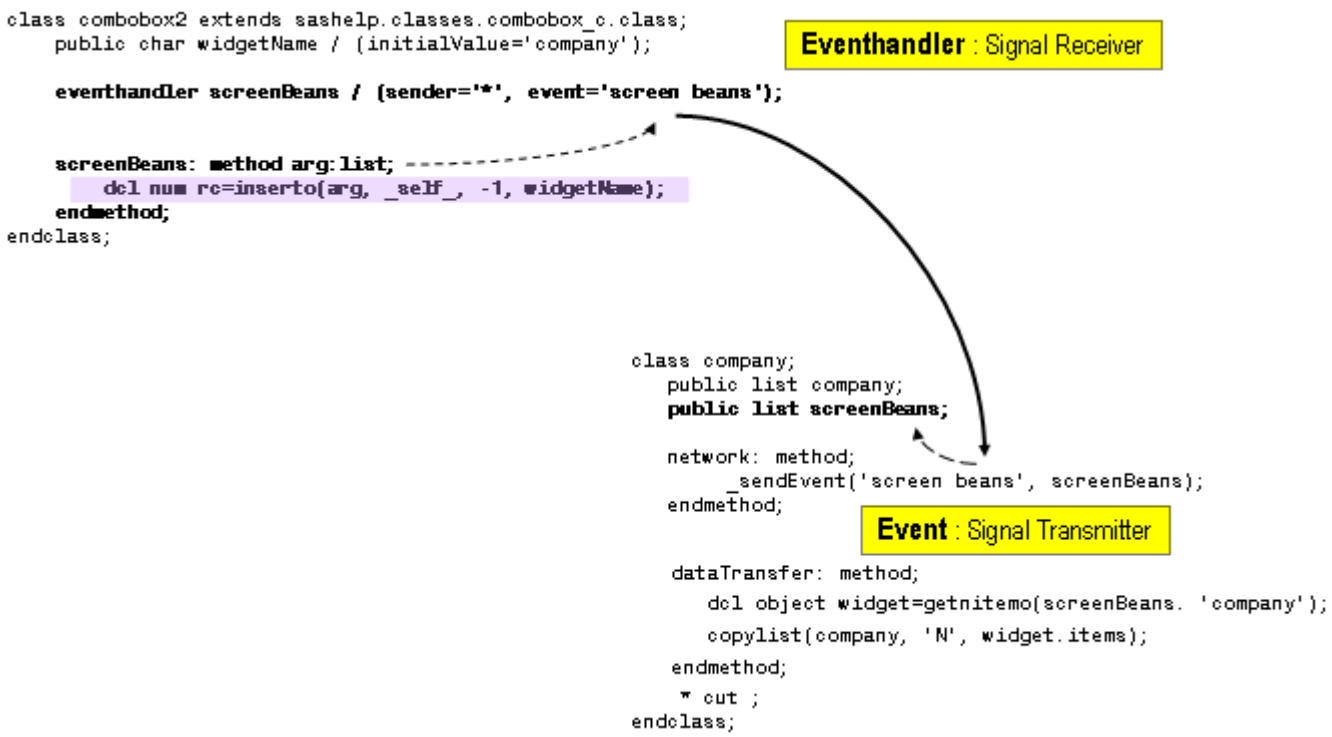
```
dcl object anyName;
```

THE %GLOBAL POOL

My global pool idea would only work if the container could hold an unlimited number of pointers. The SCL LIST fits the bill here. LIST works like the Data Step array, but there's no need to predetermine the number of elements. A list can expand until it takes up all of your system memory.

The process begins with the event named "**screen beans**". The event queries each program in the application looking for eventhandlers of the same name. Only programs with an eventhandler of the same name can respond to this event. Responding programs insert **_self_** into the event parameter. The pointer is immediately stored in the LIST located in the calling program. Event directives act as wireless connectors between SAS programs.

The following figure demonstrates the return path of a pointer, and the subsequent storage in a list.



Any number of eventhandlers can respond to a single event and the calling program does not know how many programs will respond. This is why we need a list to contain an unknown number of elements.

Presto! A single SAS command (or two) performs a lot of work, as usual. The most powerful functionality in object programming is the combination of EVENT and EVENTHANDLER. The SAS System extends this feature into the realm of batch-mode programs.

BUSINESS RULES

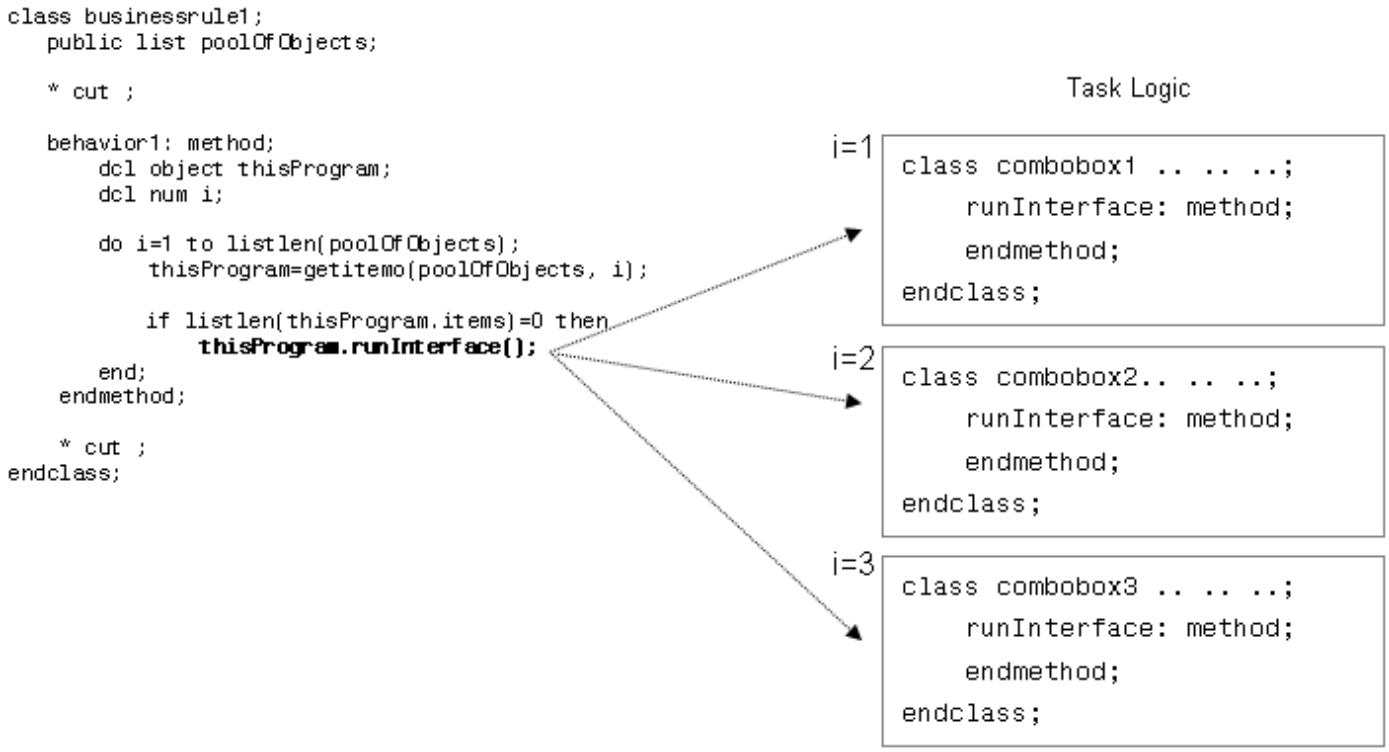
The most important aspect in modular programming is the ability to **engage** and **disengage** a business rule.

The goal is to code each rule as standalone module that can be added, removed, and modified without affecting any other program in the application. We really want the ability to change the functional response of several different parts of the application at the same time, without reprogramming.

The solution is to pool object references of specific programs into a business rule module. The standard process of the entire application can be changed dramatically through rule programs. Rule modules provide a mechanism to have a standard process with many options.

- Include extra modules based on values in the data stream
- Enable or disable functions in multiple programs at the same time.
- Deviate from the standard process with new functions inserted into old code at a specific point.
- Execute optional steps.
- Alter data values in any program that control standard process execution, on a per-observation basis

The figure below shows a business rule that uses pointers to operate on several Frame widgets. In this case, decision logic runs in batch-mode while task logic runs in interactive-mode.



This example was taken from a production program that contains several hundred business rules.

CONCLUSION

The transition into object-oriented programming begins with solid Base/SAS programming skills.

Object programming is easier than it looks.

REFERENCES

Repository Relationship Programming, www.uspto.gov

AUTHOR CONTACT INFORMATION

Montura Consulting
Kevin Graham
(510) 798-8367
Kevin@montura.com