Modular Programming using AF/SCL

Kevin Graham, Montura, San Francisco, California

ABSTRACT

How to build modular SAS/Frame applications, separating decision-logic from task-logic without separating your brain from your sanity. Effectively accommodate changing program specifications.

INTRODUCTION

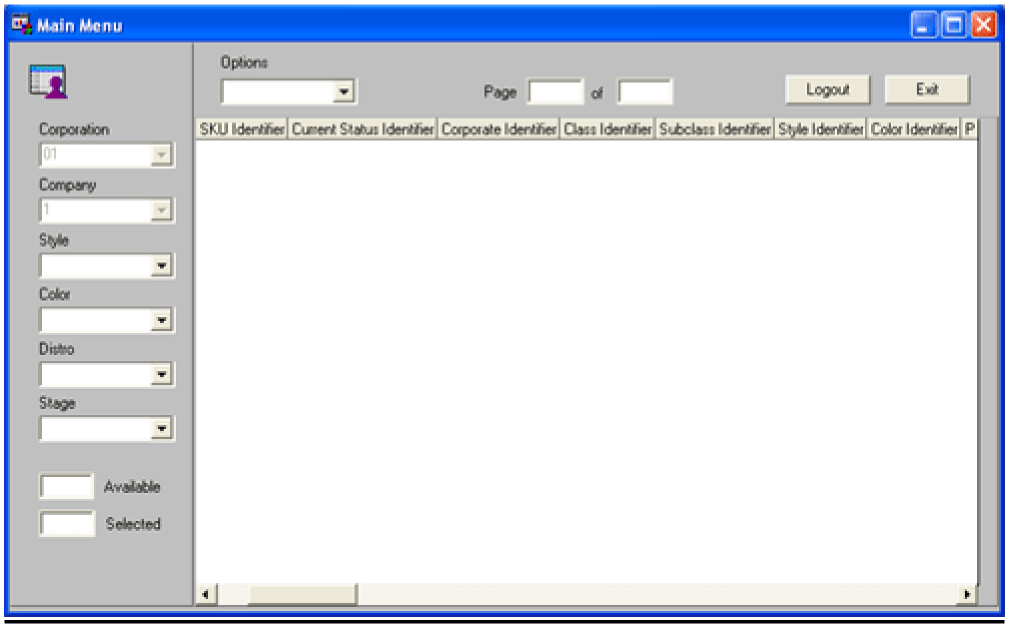
This paper provides an introduction to object-oriented programming with SAS/Frame. The traditional problems of coordinating behaviors and data content between a large number of SAS/Frame widgets, such the combobox and SCL List Model, have been swept away.

Program structure

- Widgets (Frame, combobox, etc.) exist in a catalog separated from all other programming.
- Non-visual programs are organized into different catalogs, according to function or behavior.
- Every SAS program in the non-visual layer can be connected to every Frame widget in the visual layer, and vice versa.

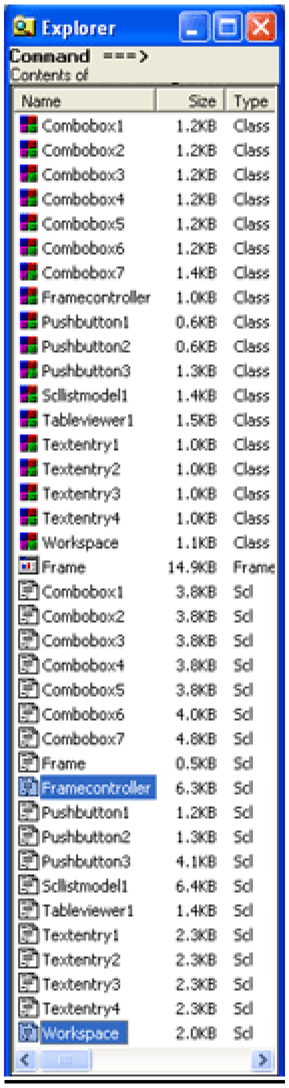
EXAMPLE APPLICATION

Widgets include seven combobox, four text field, two pushbutton, one scl list model, one table viewer are visible. Non-visual programs include SAS and SQL programs running in the background.



Corporation: 01

Company: 1



Name	Size	Type
Combobox1	1.2KB	Class
Combobox2	1.2KB	Class
Combobox3	1.2KB	Class
Combobox4	1.2KB	Class
Combobox5	1.2KB	Class
Combobox6	1.2KB	Class
Combobox7	1.4KB	Class
Framecontroller	1.0KB	Class
Pushbutton1	0.6KB	Class
Pushbutton2	0.6KB	Class
Pushbutton3	1.3KB	Class
Sclistmodel1	1.4KB	Class
Tableviewer1	1.5KB	Class
Textentry1	1.0KB	Class
Textentry2	1.0KB	Class
Textentry3	1.0KB	Class
Textentry4	1.0KB	Class
Workspace	1.1KB	Class
Frame	14.9KB	Frame
Combobox1	3.8KB	Scl
Combobox2	3.8KB	Scl
Combobox3	3.8KB	Scl
Combobox4	3.8KB	Scl
Combobox5	3.8KB	Scl
Combobox6	4.0KB	Scl
Combobox7	4.8KB	Scl
Frame	0.5KB	Scl
Framecontroller	6.3KB	Scl
Pushbutton1	1.2KB	Scl
Pushbutton2	1.3KB	Scl
Pushbutton3	4.1KB	Scl
Sclistmodel1	6.4KB	Scl
Tableviewer1	1.4KB	Scl
Textentry1	2.3KB	Scl
Textentry2	2.3KB	Scl
Textentry3	2.3KB	Scl
Textentry4	2.3KB	Scl
Workspace	2.0KB	Scl

PROGRAMMING THE FRAME

Frame.scl is usually the program where most of the SCL program methods are located.

Using this programming approach ***frame.scl*** contains the following two statements.

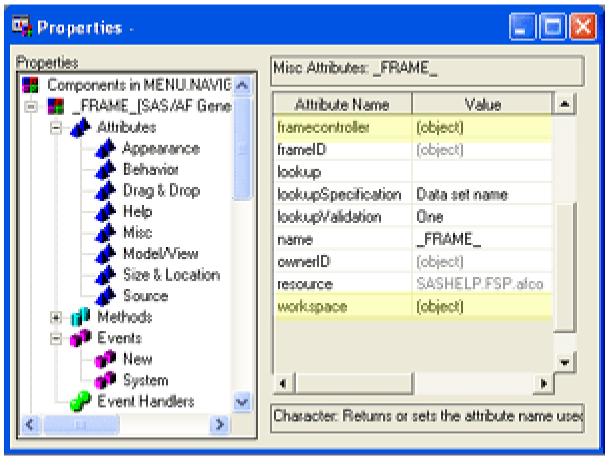
```
init:
return;
```

Where's the Code? All SAS and SQL statements used to drive this application can be found within SCL programs that are compiled as objects.

Add Two Attributes

Two classes [a.k.a. programs] need to be attached directly to the Frame.

The program named **framecontroller** manages things like the object-oriented "include statements", while **workspace** is a container for macro-level data used by multiple programs.



Misc Attributes: _FRAME_

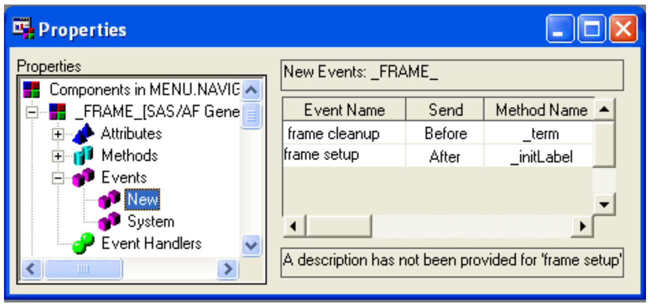
Attribute Name	Value
framecontroller	(object)
frameID	(object)
lookup	
lookupSpecification	Data set name
lookupValidation	One
name	_FRAME_
ownerID	(object)
resource	SASHELP.FSP.afco
workspace	(object)

Character: Returns or sets the attribute name used

Add Two Events

The first event identifies the point **after** the frame has appeared on-screen. This is used to trigger initialization routines.

The second event identifies the point **before** the frame disappears from the screen. This is used to trigger termination routines.



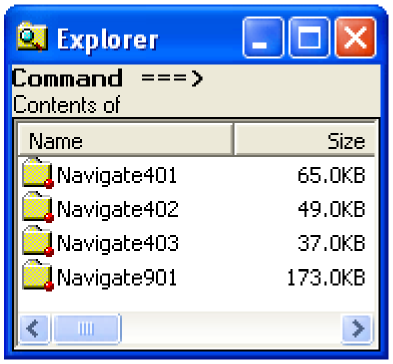
New Events: _FRAME_

Event Name	Send	Method Name
frame cleanup	Before	_term
frame setup	After	_initLabel

A description has not been provided for 'frame setup'

APPLICATION ASSEMBLY & MAIN DRIVER

Framecontroller acts as the applications main driver (for a limited timeframe) right after the frame appears on-screen. In the previous step **framecontroller** and **workspace** attached to the Frame as object attributes.



Name	Size
Navigate401	65.0KB
Navigate402	49.0KB
Navigate403	37.0KB
Navigate901	173.0KB

Object attributes are automatically instantiated during Frame startup, which is a very attractive undocumented feature. The **main driver** has three functions.

1.) Accept the FRAME SETUP event from the Frame and "%include" our object oriented programs. In this example, programs are found in catalogs named NAVIGATE501 and NAVIGATE401. See method **runInterface()**.

2.) Accept the FRAME CLEANUP event from the Frame and terminate our object oriented programs when the application shuts down. See method **cleanup()**.

3.) Kick start the application. Indicate the sequence of program execution that results in the initial data presentation for the user. See method **task()**.

APPLICATION OVERVIEW

The **framecontroller** program is intended to emulate the autocall facility, commonly used in SAS/Macro programming.

Repository400 series [SOURCE]

Contains SAS programs that pull data into the application, using PROC SQL and SCL functions.

This paper includes source code from only two programs in this repository.

Repository900 series [UI]

Contains Frame and visual widget programming.

This paper includes only program fragments from this repository.

```
class framecontroller;
    public list component   / (sendEvent='N');
    public list objectName  / (sendEvent='N');

    eventhandler runInterface / (sender='*', event='frame setup');
    eventhandler cleanup      / (sender='*', event='frame cleanup');

    runInterface: method;
        call send(_self_, 'loadService1');
        call send(_self_, 'loadService2');
        call send(_self_, 'loadService3');
        call send(_self_, 'task');
    endmethod;

    loadService1: method;
        submit continue sql;
        create table work.temp as
            select * from sashelp.vcatalg
            where libname='MENU' and memname in('NAVIGATE901','NAVIGATE401') objtype='CLASS';
        endsubmit;
    endmethod;

    loadService2: method;
        dcl num rc dset;
        dcl char arg1 arg2 arg3 arg4;
        dset=open('work.temp', 'i');
        do while (fetch(dset)=0);
            arg1=getvarc(dset, varnum(dset, 'libname'));
            arg2=getvarc(dset, varnum(dset, 'memname'));
            arg3=getvarc(dset, varnum(dset, 'objname'));
            arg4=getvarc(dset, varnum(dset, 'objtype'));

            rc=insertc(objectName, compress(arg1||'.'||arg2||'.'||arg3||'.'||arg4), -1);
        end;
        rc=close(dset);
        rc=delete('work.temp');
    endmethod;

    loadService3: method;
        dcl num rc i;
        dcl object thisProgram;
        do i=1 to listlen(objectName);
            thisProgram=instance(loadclass(getitemc(objectName, i)));
            rc=inserto(component, thisProgram, -1);
        end;
        rc=clearlist(objectName);
    endmethod;

    task: method;
        _sendEvent('network');
        _sendEvent('source primary');
    endmethod;

    cleanup: method;
        dcl num i;
        dcl object thisProgram;

        do i=1 to listlen(component);
            thisProgram=getitemo(component, i);
            thisProgram._term();
        end;

        _term();
    endmethod;
endclass;
```

The Communications Model

MC Hammer says "Java … can't touch this"

Requesting the Global Program Pool

Use a LIST attribute as the event parameter, which doubles as a container to be filled with the object identifiers of other programs. The object identifier provides complete access to data and functions for the related program instance. Works just like a remote control for its related TV.

```
class standardselector;
    public list beans         / (sendEvent='N');

    eventhandler network      / (sender='*', event='network');

    network: method;
        _sendEvent('screen beans', beans);
    endmethod;
endclass;

class scllistmodel1 extends sashelp.classes.scllistmodel_c.class;
    public char widgetName     / (initialValue='list model');
    public list beans         / (sendEvent='N');

    eventhandler network      / (sender='*', event='network');

    network: method;
        _sendEvent('screen beans', beans);
    endmethod;
endclass;
```

Filling the Global Program Pool

Program(s) responding to event "screen bean" inserts its <u>own</u> object identifier into the event parameter. Repeat for each widget and non-visual program, if and when needed.

```
class combobox1 extends sashelp.classes.combobox_c.class;
    public char widgetName     / (initialValue='corp');
    public num  cursorPosition / (state='o', initialValue=1);

    eventhandler screenBeans   / (sender='*', event='screen beans');

    screenBeans: method arg:list;
        dcl num rc=inserto(arg, _self_, -1, widgetName);
    endmethod;
endclass;

class combobox4 extends sashelp.classes.combobox_c.class;
    public char widgetName     / (initialValue='color');
    public num  cursorPosition / (state='o', initialValue=1);

    eventhandler screenBeans   / (sender='*', event='screen beans');

    screenBeans: method arg:list;
        dcl num rc=inserto(arg, _self_, -1, widgetName);
    endmethod;
endclass;
```

The Modular Programming Model

The Need for… Organization

A programmer needs to be able to locate a specific section of source code -- without hunting through thousands of lines of source code.

Properly organized code removes the need for hunting.

Picture in your mind a Klingon battle cruiser attacking your server. His phasers knock a bug into one of your SAS programs on a Friday, around 4:45 pm. It should be so obvious that you need to find and fix that program in a 15 minute timeframe.

Making it Happen by 5:00 pm

A LOT of program organization is required to make a fixit happen in 15 minutes. In fact, programs will need to be so organized they actually tell the programmer where they are located, what they are supposed to do, not to mention – how to fix the error. Yeah!

Separation of Concerns

The most important aspect of organized programming is the ability to separate logic by category, sequence, and dependency.

Make sure that each logical step is a physically separate program. To make this happen, SAS catalogs are named to indicate a general category of logic, and sometimes a series of similar logic.

SCL program names indicate specific logic.

Programs that pull data into the application from another source would be located in a catalog named REPOSITORY401. If a sql JOIN's or data step MERGE's are needed to subset information from multiple tables, additional programs would be located in a catalog named REPOSITORY402.

The concept to keep in mind that we need to **visually** indicate multiple steps are necessary to carry out a multiple-step operation.

Extending this concept to include every category of logic possible in SAS programming, it becomes easy to create a physical program layout that can be read like an automobile dashboard.

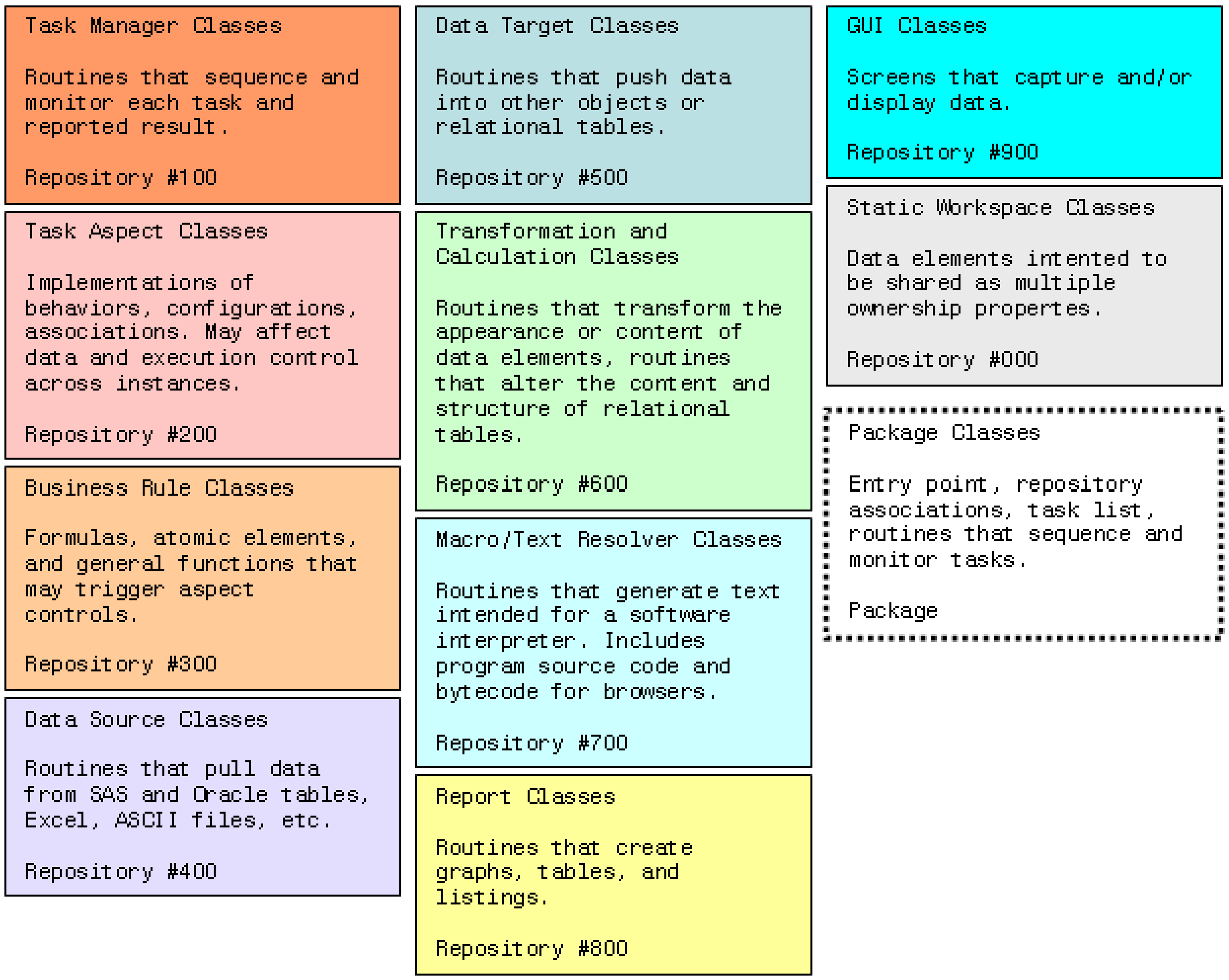
I really want to be able to read my source code dashboard like Captain Kirk reads gauges and dials on the bridge of Star Trek Enterprise. Everything should be self-indicating, as to purpose and status, at a single glace.

SDLC Documentation

Who on this planet has time to read hardcopy documentation that is likely to be out of date, or is simply flat wrong? Especially when you're in a hurry to get something done – preserving your job is usually job #1, right?

Logical Program Layout

The Programmers Dashboard



Task Manager Classes

Routines that sequence and monitor each task and reported result.

Repository #100

Task Aspect Classes

Implementations of behaviors, configurations, associations. May affect data and execution control across instances.

Repository #200

Business Rule Classes

Formulas, atomic elements, and general functions that may trigger aspect controls.

Repository #300

Data Source Classes

Routines that pull data from SAS and Oracle tables, Excel, ASCII files, etc.

Repository #400

Data Target Classes

Routines that push data into other objects or relational tables.

Repository #500

Transformation and Calculation Classes

Routines that transform the appearance or content of data elements, routines that alter the content and structure of relational tables.

Repository #600

Macro/Text Resolver Classes

Routines that generate text intended for a software interpreter. Includes program source code and bytecode for browsers.

Repository #700

Report Classes

Routines that create graphs, tables, and listings.

Repository #800

GUI Classes

Screens that capture and/or display data.

Repository #900

Static Workspace Classes

Data elements intented to be shared as multiple ownership propertes.

Repository #000

Package Classes

Entry point, repository associations, task list, routines that sequence and monitor tasks.

Package

This sample dashboard is intended to provide a starting point for source code organization.

Physical Program Layout

Physical program layout is all about organization. The idea is to factor source code into small task oriented steps, where each category of task may be located in a different SAS catalog.

Program assembly across SAS catalogs and implementing the communication model across so many programs takes about 5 nanoseconds, so we can feel free to get our programs ***very*** organized.

Repository Task Program [SOURCE]

For each iteration, within the SAS program named ***standardselector*** (following page), SQL is used to obtain distinct values from within one column of a relational table.

The distinct values are pulled from a SAS table named work.temp and pushed into a SAS/Frame combobox.

runInterface is "main driver" and the program has a several internal functions, called methods, which are executed in sequence.

The specific sequence of method execution can be found hard-coded in the attribute named **activeMethods**.

Critical Point #1

This non-visual program controls the data content and multiple widgets. The ability to perform this trick is in the method named **network**. The attribute named **beans** contains the object identifiers of many programs after the _sendEvent() operation.

Critical Point #2

The data "load" operation is performed in method named **loadWidget(),** which is programmed to operate on a variable number of widgets. The names of each widget can be found in the attribute named **widgetName**.

```
class standardselector;
    public list messages      / (sendEvent='N');
    public list dataVector    / (sendEvent='N');
    public list beans         / (sendEvent='N');
    public list activeMethods / (initialValue={
                                              'distinctValues',
                                              'getData',
                                              'loadWidget',
                                              'iterationCleanup'
                                             });

    public list widgetName    / (sendEvent='N', initialValue={
                                                             'corp',
                                                             'style',
                                                             'color',
                                                             'distro'
                                                            });

    eventhandler network      / (sender='*', event='network');
    eventhandler runInterface / (sender='*', event='source primary');

    network: method;
        _sendEvent('screen beans', beans);
    endmethod;

    runInterface: method / (description='the main driver');
        dcl num xMethod xWidget rc;

        do xWidget=1 to listlen(widgetName) while (listlen(messages)=0);
            do xMethod=1 to listlen(activeMethods) while (listlen(messages)=0);
                call send(_self_, getitemc(activeMethods, xMethod));
            end;

            rc=rotlist(widgetName);
        end;
    endmethod;

    distinctValues: method;
        dcl num rc;
        dcl char columnName=getitemc(widgetName);

        submit continue sql;
        create table work.temp as
            select distinct &columnName
            from sysdata.ipl
            where &columnName is not null;
        quit;
        endsubmit;

        if symgetn('sqlrc') then
            rc=insertc(messages, 'Unable access central database', -1);

        if symgetn('sqlobs')=0 then
            rc=insertc(messages, 'Unexpected absent values in database', -1);
    endmethod;

    getData: method;
        dcl num dset rc;

        dset=open('work.temp', 'i');
        do while (fetch(dset)=0);
            rc=insertc(dataVector, getvarc(dset, varnum(dset, getitemc(widgetName))), -1);
        end;
        rc=close(dset);
    endmethod;

    loadWidget: method;
        dcl object thisProgram=getnitemo(beans, getitemc(widgetName));
        thisProgram.runInterface(dataVector);
    endmethod;

    iterationCleanup: method;
        dcl num rc;

        rc=clearlist(dataVector);
        rc=delete('work.temp');
    endmethod;
endclass;
```

Frame widgets and non-visual programs respond to the **screen beans** event using the method named **screenBeans()**. This operation removes the need for traditional parent-child relationships between programs, which is created during program instantiation.

There is no equivalent for this operation in Base/SAS. It permits every program in the application to recursively call every other program in the application, run functions, examine widget status, change widget status, etc. etc.

```
class combobox3 extends sashelp.classes.combobox_c.class;
    public char widgetName     / (initialValue='style');
    public char widgetType     / (initialValue='selector');
    public num  cursorPosition / (state='o', initialValue=3);

    eventhandler screenBeans   / (sender='*', event='screen beans');

    _onSelect: method arg:num / (state='o');
        _super(arg);

        _sendEvent('source criteria change');
    endmethod;

    screenBeans: method arg:list;
        dcl num rc=inserto(arg, _self_, -1, widgetName);
    endmethod;

    runInterface: method arg:list;
        dcl num rc;

        rc=clearlist(items);
        copylist(arg, 'N', items);

        if listlen(items)=1 then
            selectedItem=getitemc(items);

        if listlen(items) in (0,1) then
            enabled='No';

        _refresh();
    endmethod;
endclass;
```

BONUS SECTION
Extending the Concept: Global Program Pool

This program fragment demonstrates the concept of a non-visual program using Frame widget(s) as a source of information. It's actually the full text of a production program, with repeating sections chopped out. The intent is to build and execute a complete PROC SQL statement. The tokenizing approach used to build the PROC SQL statement across multiple steps is something that should be familiar to advanced-level SAS/Macro programmers.

The Frame combobox provide information to build a complex SQL ***where*** clause. All of combobox selections are optional. It's possible there will not be a single selected item across six combobox.

```
class viewindex;
    public list messages    / (sendEvent='N');
    public list beans       / (sendEvent='N');
    public char clauseToken / (sendEvent='N', initialValue='where');
    public char delimiter   / (sendEvent='N');

    public list activeMethods / (initialValue={'step1',
                                              -- cut --
                                               'step8'});

    eventhandler network      / (sender='*', event='network');
    eventhandler runInterface / (sender='*', event='present data segment');

    network: method;
        _sendEvent('screen beans', beans);
        _sendEvent('application beans', beans);
    endmethod;

    runInterface: method;
        dcl num i;

        do i=1 to listlen(activeMethods) while (listlen(messages)=0);
            call send(_self_, getitemc(activeMethods, i));
        end;

        call send(_self_, 'cleanup');
    endmethod;

    step1: method;
        submit;
        proc sql undo_policy=none;
        create table work.selectedindex as
            select key_id
            from sysdata.ipl
        endsubmit;
    endmethod;

    step2: method;
        dcl object thisProgram;
        dcl char distinctValue;

        thisProgram=getnitemo(beans, 'corp');
        if thisProgram.selectedIndex=0 then return;

        distinctValue=getitemc(thisProgram.items, thisProgram.selectedIndex);

        submit;
        &clauseToken &delimiter corp = '&distinctValue'
        endsubmit;

        delimiter='and';
        clauseToken='';
    endmethod;

    * repeat the concept of STEP2 for each combobox providing data for the WHERE clause ;

    step8: method;
        dcl num rc;
        submit continue;
        ;
        quit;
        endsubmit;

        if symgetn('sqlrc') then
            rc=insertc(messages, 'Unable access CC information', -1);

        if symgetn('sqlobs')=0 then
            rc=insertc(messages, 'Zero observations selected', -1);
    endmethod;

    cleanup: method;
        dcl num rc;

        delimiter='';
        clauseToken='where';

        rc=clearlist(messages);
        rc=delete('work.temp');
    endmethod;
endclass;
```

CONCLUSION

A physical program layout is used to benefit the human programmers, during program development and during production support. Programs that are EASY to trace and flowchart are EASY to fix, and that provides a major benefit the corporate budget during secondary development efforts and when trouble-shooting another programmer's code during an "emergency" production-support call.

REFERENCES

Repository Relationship Programming, www.uspto.gov

AUTHOR CONTACT INFORMATION

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