Creating and Linking Customized SCL Objects

Andrew A. Norton, Trilogy Consulting Corporation, Kalamazoo, Michigan

AUTHOR'S NOTE

This paper was designed as a guide for a Hands-On Workshop at the SUGI 21 conference. If you were not able to participate in this workshop and would like to try the exercises, the computer files are available on diskette (for SAS® 6.11 for Microsoft Windows) from the author, free of charge.

INTRODUCTION

Object-oriented programming can be spooky and threatening. Actions at one spot in a program produce changes in another part, with no apparent connection. Program modules are not fixed in place, but can be unplugged and swapped with other modules, even at run time. A simple action may entail twenty or thirty "messages" (similar to subroutine calls). One module may repeat a process just executed someplace else. You might not be able to tell the difference between a variable and a function.

These characteristics may be disorienting from a traditional programming perspective, but they make sense according to the goals that object-oriented programmers value: strict isolation of data elements and related code within boundaries, partitioning of problems into less complex sub-problems, independence of one module from another (with controlled communications), interchangeability of one implementation with any other which supports the same functionality, continuity of real-world concepts through analysis and design to implementation.

SAS gives you the choice: you can use traditional coding techniques for your Screen Control Language programs, or mix in true object-oriented techniques to the degree you desire. The Object Technology Group at Trilogy has been adapting the techniques that work in other languages (such as Smalltalk) and we have found that these tried-and-true techniques are not only readily adaptable to SCL but solve many otherwise messy coding problems.

The key to solving many problems in SCL is to make use of nonvisual objects connected to widgets through events. This allows different parts of the application to coordinate with each other while allowing each part to be implemented independently.

The structure of this workshop

This lab gives the opportunity of experiencing on a broad conceptual level how object-oriented techniques help you control complex evolving applications.

First we will look at how to use some classes that I have already made. This will acquaint you with the end product.

Then we will look at how to take existing classes and modify them. Some pieces have already been completed for you, to give a jump-start into more advanced issues.

The final step would be to create your own classes from scratch. We will not be discussing this, as this has been already documented in the SAS Institute publications (SAS Institute 1993).

USING OBJECTS

Using User-Constructed Widget Classes

Key Points

1. Customized classes can be used as easily as those provided by SAS Institute.

2. Objects can communicate with each other without using FRAME code or linkages.

3. Resource entries can be used to control and document which widget classes are available to an application.

Exercise

1. Open the BUILD window and enter the command RESOURCE WORKSHOP.SYSTEM.BUILD

2. Open a new frame TEST.FRAME in the catalog WORKSHOP.LAB.

3. Press the Make key (Right Mouse Button) and note two additional classes:
   "Graphics (revised)"
   "Pushbutton for Control Panel"
   These classes were defined by program code in the catalog WORKSHOP.TOOLBOX. They are made available to the frame through the resource entry WORKSHOP.SYSTEM.BUILD.RESOURCE.

4. Create a "Graphics (revised)" widget. On the Attribute screen, specify Horizontal Bar Chart, data set SASUSER.HOUSES, midpoint variable STYLE, response variable PRICE, statistic SUM.

5. Create a "Pushbutton for Graphics Control Panel" widget.

6. In General Attributes for the frame, specify no SCL program.

7. Test the frame (using TESTAF). Press the "Control Panel" pushbutton. On the Graphics Control Panel, change the graph type to Vertical Bar Chart and the Curve Color (i.e., bar color) to Green. Press the OK button and note that the graph has changed.

Discussion

SAS allows users to create their own classes. This allows a
division of labor within organizations between class builders (expert in coding SCL) and application builders (expert in user interface design and the subject domain). The application builders only need to know what a class does, not how it works internally.

User-programmed widget classes are used just like system widget classes. The user-programmed classes inherit functionality from the classes provided by SAS Institute with the SAS/AF® product. One difficulty is creating Attribute screens comparable with those from SAS Institute, because the source code for system Attribute screens is not provided.

The widgets in this example communicate with a nonvisual object (the Graph Controller) rather than with each other directly, so no widget linking or Frame code is required.

The WORKSHOP.SYSTEM.BUILD.RESOURCE entry is a copy of SASUSER.FSP.BUILD.RESOURCE with the "Graphics (revised)" and "Pushbutton for Control Panel" widgets added. Frames use widget class definitions found in the associated Resource entry rather than loading from the Class entries directly. This is faster and lets organizations enforce use of a standard list of widgets. As we will see later, the Resource entry also allows classes to be swapped for existing applications, for upgrades with minimal maintenance.

Controlling Multiple Frames

**Key Points**

1. The Graphics Control Panel controls all Graphics widgets on all frames.

2. The Graphics Control Panel determines the initial attributes of newly created Graphics widgets.

**Exercise**

1. Open the WORKSHOP.EXAMPLE catalog.

2. Test FRAME1.FRAME. Press the "Next" button, which will take you to Frame 2. Pulldown the "View" menu and pick "Control Panel". Change the type of graph to "Pie Chart". Note that the graphs on Frame 2 have changed.

3. Return to Frame 1 and notice that those graphs have also changed.

4. On Frame 1, pulldown the "View" menu and pick "Control Panel". Make a change, then go back to Frame 2. Notice that the graphs on Frame 2 have changed.

**Discussion**

The Control Panel controls all graphs in the application, existing and future. It is not tied to a particular frame.

**Model-View-Controller**

**Key Points**

1. Nonvisual objects can be used to provide a stable center or control as frames and widgets change.

2. SCL programs communicate with nonvisual objects using CALL SEND, just as with widgets.

**Exercise**

1. In WORKSHOP.LAB.TEST.FRAME, go into General Attributes and specify program *.SCL.

2. In the INIT section for TEST.FRAME, write the following code:

   ```
   INIT:
   system_c = loadclass
   ('workshop.toolbox.system');
   call send (system_c, '_new_', system_o);
   call send (system_o, "get_graph_controller",
     graph_controller_o);
   call send (graph_controller_o, 'set_color', 'curve', 'green');
   return;
   ```

3. Compile the frame.

4. Test TEST.FRAME. Note the bars in the chart are now green instead of blue.

**Discussion**

Setting the color or type of graph programmatically is as effective as setting the attributes through the Graphics Control Panel. In fact, the Graphics Control Panel works by sending messages such as these to the Graph Controller.

The Graph Controller changes attributes of future graphs as well as setting attributes of existing graphs. To do this we need

1) A memory to store the settings,

2) A notification mechanism to update existing dependents.

We need an object rather than a simple data store (such as a macro variable) in order to provide the notification mechanism. The object needs to be nonvisual so that it will remain in memory regardless of which frames are active. Thus, the Graph Controller is a nonvisual object (what I used to call an "SCL data object"), with superclass SASHELP.FSP.OBJECT.

Perhaps surprisingly, one of the more difficult aspects of using nonvisual objects is finding their location. The System object is responsible for creating and terminating the Graph Controller, and knowing where to find it.

The System class is a "singleton", which means it is restricted to one instance. The implementation used here is described in Kevin Tyler Brown's SUGI 21 paper "The Singleton Class in Screen Control Languages" (Brown, 1996). The _NEW_ metaclass method is modified so that it always returns the same instance of the class. This lets us locate the System object from anywhere by using the LOADCLASS function.
This example is based upon the "Model-View-Controller" pattern which originated in Smalltalk (Lewis, 1995). The Model-View-Controller pattern has become standard for Graphical User Interface programming, sometimes in a generalized form called the Observer pattern (Gamma, et. al., 1995).

In this application, the Graph Controller non-visual object takes the Model role. The Graphics widget takes the View role, and the Graphics Control Panel frame takes the Controller role.

As each Graphics widget is created, it subscribes to the Model. Changes may be submitted through the Control Panel or a message sent by a program. When a change is made to the Model:
1. The Model posts a "Changed" event.
2. Each widget subscribing to the Model is marked by the "_NEED_REFRESH" method.
3. Before the frame is (re)displayed, the _REFRESH_ method runs for each widget needing refreshing.
4. The _REFRESH_ method queries the Model for whatever information the widget needs to update itself.

MODIFYING OBJECTS

Inheritance

Key Points

1. Inheritance allows existing code to be extended without changing the existing code.
2. Classes can be modified or swapped without modifying applications.
3. Nonvisual objects can take on an active role through methods. They are not simply a data store.

Exercise

In this exercise we will extend the Graphics Controller so that the text of the Graphics widget will be the same color as the curve. We can do this without touching the original Toolbox code or the application itself.

1. Edit a new class, WORKSHOP.LAB.GRAPHCTL. The super class is WORKSHOP.TOOLBOX.GRAPHCTL. The method GET_COLOR should use the GETCOLOR label in WORKSHOP.LAB.GRAPHCTL.SCL.

2. Use the following code for GRAPHCTL.SCL in the WORKSHOP.LAB catalog. This method returns the Curve color when asked for the Text Color, and otherwise uses the method as defined by the superclass.

```smalltalk
   _self = _self_
   GETCOLOR: /* GET_COLOR method */
   method
     in_color_name $
     out_color $ ;
     if in_color_name eq 'text' then do:
       call send (_self_, 'get_color',
                    'curve', out_color);
   endmethod;

   return;
   end;
else
  call super (_self_, 'set_color',
              in_color_name, out_color);
endmethod;
```

The CALL SUPER uses the GET_COLOR method as defined by WORKSHOP.TOOLBOX.GRAPHCTL.CLASS.

3. We also need to create a modified Graphics Widget class that will honor the text color settings in the Graphics Controller. Create a new class WORKSHOP.LAB.LAYOUT2 with superclass WORKSHOP.TOOLBOX.LAYOUT2. Override the REFRESH method to use the label REFRESH in WORKSHOP.LAB.LAYOUT2.SCL. Use the following code:

```smalltalk
length tcolor $ 200;
_self = _self_
REFRESH: /* _REFRESH_ method */
method
  call send (_self_, 'get_graph_controller',
             graph_controller_o);
  call send (graph_controller_o,
             'get_color', 'text', tcolor);
  call send (_self_, 'set_color', 'text',
             tcolor);
  call super (_self_, 'refresh_');
endmethod;
```

When the Graphics widget refreshes, it will set the Text color to the Text Color value provided by the Graph Controller. It then executes the _REFRESH_ method defined by the superclass, to complete the refreshing process.

5. Now we have defined both new classes and it is time to install them into the system. In WORKSHOP.SYSTEM.BUILD RESOURCE, select the "Graphics (revised)" class, and choose the "Set New Class" action. Change the class to WORKSHOP.LAB.LAYOUT2.

6. Also in the WORKSHOP.SYSTEM.BUILD RESOURCE entry, select the "Graph Controller" entry and set the class to WORKSHOP.LAB.GRAPHCTL.

7. Try out your existing application WORKSHOP.LAB.TEST FRAME. Without having made any changes to either the application or to the WORKSHOP.TOOLBOX catalog, you should now see the text color changing in synchronization with the bar color.

Discussion

Inheritance lets us modify the behavior of an existing class without changing that class. In general it is not necessary to retest or even inspect the existing code, and we avoid the risk of damaging what is already working.

The behavior of an object-oriented application is determined by the classes it uses and how they are connected. We can swap in new classes within an application by making a simple change in a Resource entry. We don't need to know the internals of those applications, or even how many applications are using the Resource entry. We do need to insure that when we plug in a new class that it
will support at least the same functionality as the old one.

Resource entries link applications to classes through symbols called "aliases". If we change the class associated with an alias, every widget that is linked to that alias will change the class it uses. In this example, Graphics widgets were connected to the class WORKSHOP.TOOLBOX.LAYOUT2 through the alias LAYOUT2.LAYOUT. After the alias was reassigned to WORKSHOP.LAB.LAYOUT2, the widgets of the next AF session were instances of the new class rather than the old one.

If we could not use aliases, we would have to manually change widgets throughout various applications, removing old widgets and installing new widgets.

Using Resource entries to alias nonvisual classes is less commonly seen. It was useful in this example as a means of swapping Graph Controller classes without modifying the System class (which creates the Graph Controller instance). I look forward to exploring this technique more in the future, as an alternative to hard coding class names into programs.

Independently Evolving Levels

Key Points

1. Superclasses can change without needing to maintain subclass code, just as classes can change without needing to maintain application code.

Exercise

1. Copy the entry WORKSHOP.TOOLBOX.GRAPHCT2.CLASS into (replacing) WORKSHOP.TOOLBOX.GRAPHCTL.CLASS.
2. Test the TEST frame, making a change with the Graphics Control Panel.
3. Test the TEST frame again. Notice that the settings from the last run are now remembered.

Discussion

We have seen that applications and classes can evolve independently, and that subclasses can evolve independently of their superclasses. Now we see that superclasses can evolve independently of their subclasses as well.

Subclasses maintain a dynamic link to their superclasses. We made no changes to the class WORKSHOP.LAB.GRAPHCTL, but when we switched to a new version of the superclass WORKSHOP.TOOLBOX.GRAPHCTL, the subclass inherited the new persistence feature automatically.

Developers can make changes to classes (adhering to the contractual responsibilities of the class) without disrupting those who have subclassed the classes. This is valuable not just for simplicity's sake but also because in a reuse environment it may not be feasible for developers to track down every use of a class.

SAS does not provide any aliasing feature (such as Resource entries) for use with superclasses. So to replace one version of a class with another, you can either copy one over the other, rename catalogs, or redefine libnames.

The revised Graph Controller object remembers settings from one session to the next by using techniques described in my SUGI 21 paper "Persistent Storage of SCL Data Objects" (Norton, 1996).

FURTHER EXPLORATIONS

Multiple views and controllers

We have been considering one view and one controller connected to a model. There can be many different types of views and controllers, and they can be opened and closed dynamically as an application runs. Several views and several controllers can be connected to the same model simultaneously.

Domain (business logic) models

The use of the Model-View-Controller paradigm is much broader than just system control panels. Nonvisual objects can be used to model the business rules of complex processes and structures (in fact, object-oriented programming originated with a simulation language, Simula 67). Views (graphical user interfaces) can then be layered over the Model. This is a useful partitioning because presentation and business rules are internally cohesive but loosely coupled with each other, and because presentation tends to change more readily than business rules.

REFERENCES


AUTHOR CONTACT

Andrew A. Norton
Object Technology Group
Trilogy Consulting Corporation
5148 Lovers Lane
Kalamazoo, MI 49002 USA

(616) 344-4100
76350.1604@compuserve.com

SAS and SAS/AF are registered trademarks or trademarks of SAS Institute Inc. In the USA and other countries. © indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.