

CREATING AND LINKING CUSTOMIZED SCL OBJECTS

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

INTRODUCTION

Object-oriented programming can be both spooky and threatening. Actions at one spot in a program produce changes in another spot, 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 between), 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 techniques that work in other object-oriented 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 that communicate with widgets by sending events. This allows different parts of the application to coordinate with each other while allowing each part to be implemented independently.

MAKING CHANGE EASY

Object-oriented programming evolved from structured programming, so let's start there. The SAS System introduced SCL methods (similar to subroutines in other languages) with release 6.07.

Structured programming divides large complex programs

into smaller simpler modules. This achieves several goals:

1. The code internal to a module can be changed without having to change other related modules.
2. Individual modules are easier to understand, and integration of modules into a completed system is easier. You can also start with simple stubs and elaborate modules in the future.
3. The smaller modules focus more specifically on a single task, and are therefore more likely to be generally useful and reusable in the future.

Structured program seeks modules which are highly cohesive internally yet loosely coupled with each other. A good partitioning of a complex system will place elements that are likely to change together close together (in the same module), and elements which are unlikely to change together in different modules. This reduces the number of modules which need to be changed to implement a particular revision, ideally to one.

But more is needed than simply grouping related bits of code together and factoring out unrelated bits of code. Structured programming also isolates one module from another, so they can be changed independently. Subroutine calls invoke a module and pass parameters in and out. The internals of the module remain hidden to programs which use the module — in other words, they are free to change. The internals include both the code which implements the subroutine and the local variables used by the subroutine. The modules are protected in the sense that the code of one subroutine cannot interfere with that of another subroutine (in real life there are unfortunate exceptions to this when dealing with shared resources) and the local variable namespace of one subroutine is independent of other namespaces.

This partitioning is helpful in debugging, because given a certain set of parameter values, the subroutine will always behave the same way (again, there are real-life exceptions). So the subroutine can be tested in isolation, or the calling program can be tested in isolation. Determining the location of a program defect comes down to:

1. Was the subroutine passed the right parameters? If not, the defect lies in the calling program.

2. Did the subroutine return the right parameters? If not, the defect lies in the subroutine.

This helps quickly locate ordinary program defects. Defects which involve the interaction of modules are much more difficult to track down because they do not follow these rules (side-effects such as concurrency violations in database access, memory leaks, or modifications to global variables).

This partitioning also helps when making changes to the modules, because you only need to study and retest the modules which are being changed rather than the entire application. Provided that the interface (that is, the syntax and semantics of the subroutine call) remains constant, changes to one module will not affect client modules.

OBJECT-ORIENTED PROGRAMMING

What does object-oriented programming bring to the picture? Object-oriented programming begins with structured programming, and pursues its goals even further. Object-oriented programming could thus be termed a "better" structured programming. It therefore makes no sense to talk about when to use structured programming and when to use object-oriented, because object-oriented programming supersedes and subsumes structured programming.

Although object-oriented programming evolved from structured programming, the end-results are revolutionary and rather disorienting (and the more experienced and skilled you are with classical structured programming, the more difficult your transition is likely to be). Structured programming is organized around functional decomposition of code. Database design is organized around relational normalization, the decomposition of data. Object-oriented programming locates code together with its related data.

So the overall shape or architecture of an object-oriented system looks very different from a structured system. Structured programming encourages hierarchical pyramids of code, with data being acquired by one module, returned to its calling module, and from there passed into other modules.

Object-oriented systems have no top. Because the objects contain their own data, they are self-sufficient (they may need help from other associated objects, but they typically need no assistance from object which invoked the method.) This means that any client object which can determine the address of a server object can make use of those services, as opposed to structured programs where the client object must provide data to the server.

This frees up the architecture of systems considerably.

Object-oriented systems have interconnected clusters of objects sending messages back and forth, a network rather than a hierarchy. In fact, a better comparison would be a relational database rather than a network, as the connections between objects are not hard-wired but dynamically determined through data values. So object-oriented programming gives you flexibility in the same way that relational databases give you more flexibility than hierarchical or network databases.

IT'S THE INTERFACE, STUPID!

During Bill Clinton's 1992 presidential campaign, there was a sign on the wall to always remind the campaign workers of the critical issue: "It's the Economy, Stupid!". In object-oriented programming, the critical issue is the interface between modules.

In object-oriented programming, objects have an inside and an outside. From the outside, all that is visible is the interface.

The interface is more than the syntax of valid method names and parameter positions and types, it also includes the semantics of what the parameters mean and what the parameters do. But it does not include how these methods are implemented.

So it is entirely possible to have several classes which do the same thing in different ways. For example, a collection could be sorted using one of several algorithms, yet achieve the same result.

It is also possible for two or more classes to satisfy the contract of a given interface yet to differ in other ways unspecified by and irrelevant to that contract. For example, we may have an interface specifying that up to 10 numbers may be entered and the sum retrieved. This does not restrict us from keeping an audit trail on changes to those numbers, or providing a method yielding the product of the numbers.

So within the constraints of the interface we have a great deal of freedom. We can change the properties of an existing class, or swap one class for another, provided that the interface remains satisfied. This lets us extend or modify an existing system without needing to modify the existing code and avoiding ripple effects.

We have interfaces between one object and another (the "uses" relationship) and between a class and its superclass (the "is a" relationship). In both cases we can modify either side of the relationship freely without the constraints of the applicable interface contracts.

Interfaces determine what can be changed without causing

ripple effects; anything that can be changed without modifying the interface has no impact on the calling application.

Interfaces also determine interchangeability. If an application requires a given interface, any class that supports that interface will do.

Interfaces also determine units of understanding. A programmer depending upon an interface of a given class need not be concerned with how that interface is implemented; he or she need only understand what classes supporting that interface can do for an application.

One class sending a message to another (collaboration)

Users can create their own classes in SAS, derived from base classes provided by the SAS system. These can be substituted for the base classes, so long as you avoid the pitfall of method disabling (which SAS allows).

SAS makes no differentiation between system-provided classes and those developed by users, in fact some of the system-provided classes were in fact developed by SAS Institute using the same languages and tools as user-developed classes. This makes SAS extendible to satisfy the idiosyncratic needs of various users and to become progressively suited over time to a particular environment.

The ability of SAS users to create their own classes 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 applications builders only need to know what a class does, not how it works internally.

One of the special properties that an object might have is the ability to respond to changes made in other widgets. For example, we might have a Graphics widget that behaves just like any other Graphics widget and can be substituted for one of the system widgets. But it has been extended to have an additional property -- it listens to settings from a Graphics Controller dialog box, and if a change is made in the Graphics Controller, the Graphics widget will change its properties to match.

Similarly, we might have a text-entry field which acts like any other text entry field except that it causes these special Graphics widgets to change the color of the bars in bar charts.

All this can be done behind the scenes without the user needing to do any special programming. Widgets can communicate with each other without needing the user to write any FRAME code or define linkages.

Resource Entries

Resource entries are used to control and document which widget classes are available to a frame. So to make user-specified classes accessible, they need to be added to the resource entry controlling the frame. In SAS, classes are defined by physical location (LIBNAME.MEMNAME.ENTRYNAME.CLASS).

Frames are linked to classes through resource entries, rather than directly. This eases the process of changing the location of a class (if necessary) since only the resource entry needs to be updated, not every frame. The location of the resource entry remains constant. This is an example of maintaining a constant interface and changing information (the location of a class) in only one place.

There is, however, one aspect that is expressed in purely physical terms, and that is the link between the resource entry and the frame. There is no direct way to find out which frames are using a given resource entry or even the resource entry attached to a given frame. There is also no direct way to change a resource entry definition for a frame. But it is possible.

Determining the resource of a frame

Enter BUILD followed by the catalog name on the command line. Place an '*' selection-list command next to the frame of interest. The attributes of the frame will be dumped to the message window, with the resource name in the item named "RESNAME".

It is also possible to determine the resource attached to a frame while the frame is running. On the `_SELF_` list of the frame, there is a list called `_APPL_` (this can be obtained using the `_GET_PROPERTIES_` method). On the `_APPL_` list, there is an entry "RESNAME" with the name of the resource entry. There is also a list `RESOURCE` containing the actual loaded contents of the resource list.

Changing the resource attached to a frame

Basically, resource entry names should be chosen with care, as they are difficult to change after the frame has been created. There is a circuitous way to change the resource of a frame, or more precisely creating a new frame like the old one but with a different resource entry.

- 1) Create a new frame entry attached to the new resource.
- 2) Copy the old frame entry into the new frame entry.

It is convenient to have a project or other unit of work use one or a small set of resource entries. Depending on the default resource name "BUILD" is risky, because if you develop your own entry labeled BUILD you block access

to `SASHELP.FSP.BUILD.RESOURCE`, the original.

Resource entries can be used to standardize (that is, limit) the resources used on a given project or within a given company. For example, you can control the default colors or behavior of widgets by customizing them. This is not foolproof unless you also disable the associated modification methods so as to keep a property fixed.

NON-VISUAL OBJECTS

Many people fail to realize that widgets are not the only kind of objects. SAS also supports non-visual objects, descended from the `SASHELP.FSP.OBJECT` class. Like other objects, non-visual objects contain instance variables and respond to messages. The only difference is that they are not associated with a frame and cannot draw themselves onto a screen.

Non-visual objects play a pivotal role in object-oriented programming. It is inadvisable to embed the business rule logic in the user interface, because the user interface is vulnerable to change. It is better to restrict the user interface code to visual representation and manipulation of the application, while the nonvisual component (customarily called the "Model") provides a stable center of control.

The use of a nonvisual Model fundamentally changes the architecture of a SAS/AF® system. Instead of passing information from one frame to another, with the focus of the system on the current frame, the frames become merely an interface to the core of the system (the Model). It becomes easy to make changes such as adding or deleting frames or moving responsibilities from one frame to another. For that matter, you can provide alternative interfaces to the same system.

Perhaps surprisingly, one of the most difficult aspects of using non-visual objects can be finding their location. For widgets on the current frame, the `CALL NOTIFY` statement will automatically resolve the name of the widget into its object ID at runtime. For non-visual objects, the object ID needs to be determined at runtime, as it changes from one execution of the application to the next.

One way to acquire the object ID is to pass it as a parameter from one frame to another. Another way, described by Kevin Brown (1995), is to define the object as a singleton. A singleton class has the property that only one instance of the class can exist at a time. The class object responds to the `GET_INSTANCE` method by

- creating an instance if one does not already exist, or
- giving you a pointer to the current instance if it does exist.

Class objects themselves are singletons which can be easily accessed from anywhere, by using the `LOADCLASS` function together with the class name.

Composites

Objects can be used as libraries or containers of other objects. So you can take several smaller objects and combine them into a larger more complex object. This lets you achieve greater complexity without having the mental load of dealing with each object individually. It also provides you with a degree of flexibility, because as long as the interface between the larger composite objects remains constant and the internal structure is kept private, you can make whatever internal changes you want. (SAS provides a Composite Widget class in release 6.11 which allows you to group widgets together and define the relationships between them. There is currently no nonvisual Composite class, but this can be handled programmatically.)

So an interface can take the form of one composite connecting to another composite. Users then don't need to connect up many different individual elements to each other; this can be arranged by the composites. For example, a nonvisual model may be made up of many different components, and a frame connected to the model may have many different widgets. When the frame is connected to the model, the widgets can connect to the components of the model, based on an identifier (which "attribute" of the model we are dealing with.)

Events

So far, we have been discussing interfaces in terms of method calls:

```
call send (object, method, parms);
```

This assumes that the calling object knows the method and object to use. This model makes sense if the object is providing a service to the calling program.

But suppose what we really are doing is notifying other objects of an event of interest. Conceptually, this is a quite different matter. For the current object, it is of no concern whether no other objects are interested in the event, or one, or many. Similarly, it is of no interest to the current object what other objects do in response to the event.

SAS supports this with the Event Handler. The current object need only specify that a certain event is to be sent:

```
call send (_self_, '_send_event_', event_name,
           parms);
```

There need not be concern with who has subscribed to the event and what they will do with it.

Objects that wish to receive notifications of an event in a certain object (or any object) may register their interest with the Event Handler. In this registration, they also

specify what action they would like to take place when the event occurs, such as running a given method.

Event handling can be thought of as sending a message "To Whom It May Concern." It simplifies the interface between objects and allows for changes behind a constant interface. It shifts responsibility for designating the target object and method to the target object rather than from the sending object.

Model-View-Controller

Separation of user interface (the View/Controller) from business rules (the Model) and connection of these two components through events are fundamental concepts of 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).

Several views may be connected at the same time to a given model, displaying the current state of the model and allowing the user to make changes. The model itself need not be concerned with these views, only posting an event to notify its subscribers when a change occurs. 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 refresh itself.

This separation of concerns simplifies the code and encourages reuse. Models are only responsible for responding to change requests, maintaining and reporting their current state, and notifying subscribers when a change has occurred. Views are responsible for displaying data, and refreshing this display when notified of a change. Controllers are responsible for responding to user actions by requesting a change in the Model or View.

POLYMORPHISM

If different classes of objects support the same interface, they can be used interchangeably in contexts requiring that interface.

Delegation

The target of a message need not be fixed. It can be a

variable, which might be set in a loop such as:

```
do i = 1 to listlen (object_1);
  object = getitemn (object_1, i);
  call send (object, 'print');
end;
```

or passed in as an argument:

```
PRINT:
  method
    in_print_device 8
  ;
  call send (in_print_device, 'print_object',
    _self_);
endmethod;
```

or created from an automatic instance variable specifying which class to use:

```
ADD_ITEM:
  method
    in_label $;
  call send (av_item_class, 'new_', item_o,
    0, in_label);
  dummy = insertn (av_item_1, item_o, -1);
endmethod;
```

All of these examples require that the dynamically specified object be able to accommodate the interface required by the calling program. If the different objects which could be used in this position are members of the same class, then they will behave in the same way. If they are members of different classes which support the same interface, then the same message may be processed differently when received by different objects. It is therefore crucial that different implementations of the same interface do the same thing, even though it may be done in different ways.

SAS does not require that dynamically specified objects all be of the same class. In contrast to C++, it is entirely possible to have classes from different hierarchies which support the same interface. But there is no enforcement of this, and if the programmer makes a mistake a message may be unrecognized (bringing execution to a halt) or be misinterpreted. So this simplicity and flexibility is obtained at the cost of increased risk in having an untyped language.

Using Resource Entries

Resource entries let applications avoid hard-coding which class entry to use when creating objects. This allows applications to be extended by defining new classes which support the same responsibilities as existing objects, plus more.

SAS resource entries are primarily designed for use with widget classes, but can also be used for nonvisual classes. Resource entries are essentially lists of classes assigned to aliases. Applications can refer to the classes by alias, rather than directly. So if you replace the class assigned to a given alias with another, the application will begin to create objects with the new class instead of the old one.

For example, you might create an instance of a Model class with:

```
project_res = loadres ('project');
alias_1 = getnitem1 (project_res, 'alias');
model_c = getnitem1 (alias_1, 'model.object');
call send (model_c, '_new_', model_o);
```

Exactly which class is used to create MODEL_O depends on the definition of the MODEL.OBJECT alias in the PROJECT.RESOURCE catalog entry. The resource entry thus becomes a plugboard upon which different "expansion board" classes can be swapped.

The process is a little trickier with widgets, because existing frames are stored on catalogs together with the instance variable information. When a frame is opened (for editing or execution), the widget objects are created by combining the class information of the resource entry with the stored instance variable data. So you must be certain that if you switch widget classes that the new widget class can be instantiated using the old data. In general there is no guarantee of this (because instance variable data is a private implementation matter), but if the new class is a subclass of the old class you will be all right so long as any additional instance variables are assigned default values.

Object Marketplace

There may be several objects which can provide the desired service. In some applications, we have addressed this issue by developing "marketplace" objects which can receive a request and forward it to a suitable server. The marketplace supports three methods: CAN_PROCESS, ESTIMATE, and EXECUTE. Each object 'bids' on the current request, the successful bidder executes the request.

This framework provides further flexibility by avoiding the hard-coding of server objects into client programs. The target object for a given method is dynamically determined through an algorithm which chooses the best target for a that message.

INHERITANCE

Finally we come to the type of interface most distinctively associated with object oriented programming: Inheritance. Inheritance lets us extend 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 as we do not need access to the internals of the parent class we avoid the risk of damaging what is already working.

Inheritance allows developers of a class to incorporate the specifications of another class by reference, without actually making a copy of the parent class's code. This is advantageous because subclasses maintain a dynamic link to their parent classes. When a method is added to a

parent class, it is added by implication to all subclasses of that parent. This insures that subclasses continue to support all responsibilities of the parent class and thus can be used wherever the parent class can be used.

This substitutability feature makes "frameworks" possible. A framework is a class which calls methods that have not yet been defined, or objects that have not yet been specified. When the framework is used in an application, the programmer adds in the additional methods or objects, much like adding your own egg to a cake mix. The framework can be reused for different situations by customizing it in different ways (to continue the metaphor, you might choose between margarine and of butter, interchangeable at one level of abstraction). The reason frameworks are possible is because the method calls coded into the framework remain constant as different objects, classes, and method implementations are used to construct a concrete application.

Subclasses are written using the interface of the parent class (actually, they can also make direct use of the parent class instance variables, but this violates encapsulation and limits future changes to the parent class). Developers can make changes to parent classes (maintaining the interface 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.

Similarly, subclasses can be defined without any modification of the superclass. So subclasses can evolve independently of their parent classes (and vice versa) in the same way that classes can evolve independently of applications (and vice versa), provided that the interfaces are kept constant.

CONCLUSION

Keep your eye on those interfaces! The fundamental principle of object-oriented programming is that a constant interface can be used to communicate with different implementation. This added degree of flexibility will help you attain the flexibility, ease of maintenance, control of complexity, and reusability that graphical user interfaces and other demanding applications require.

REFERENCES

Brown, Kevin Tyler (1996) "The Singleton Class in Screen Control Language," in *Proceedings of the Twenty-First Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides ((1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Publishing Company.

Lewis, Simon (1995), *The Art and Science of Smalltalk: An Introduction to Object-Oriented Programming Using VisualWorks*, London, UK: Prentice-Hall.

Norton, Andrew A. (1995), "Designing Complex Systems using SCL Data Objects," in *Proceedings of the Twentieth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Norton, Andrew A. (1996), "Creating and Linking Customized SCL Objects," in *Proceedings of the Twenty-First Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

ACKNOWLEDGEMENTS

Thanks to Mark Olson for reviewing a draft of this paper.

NOTES

An earlier version of this paper was presented as a Hands-On Workshop at the SAS Users Group International Conference (SUGI 21) in March, 1996 (Norton, 1996).

A revised version of this paper will be presented at the SAS Users Group International conference (SUGI 22) in March, 1997.

SAS and SAS/AF are registered trademarks of SAS Institute, Inc. ® indicates USA registration.

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

The author may be contacted at:

Andrew A. Norton
Object Technology Group
Trilogy Consulting Corporation
5278 Lovers Lane
Kalamazoo, MI 49002
(616) 344-4100 voice
(616) 344-6849 fax
Internet: 76350.1604@compuserve.com