Object Interfaces

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 somewhere 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 subproblems, 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.

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, then migrate to more elaborate modules in the future.

3. Smaller modules focus more specifically on a single task, and are therefore more likely to be generally useful and reusable in the future.

Structured programming seeks modules which are highly cohesive internally yet loosely coupled with each other. A good partitioning of a complex system will place elements close together if they are likely to change together. 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 name-space of one subroutine is independent of other name-spaces.

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.

These rules help to locate ordinary program defects quickly. 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" form of 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. 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 are composed of 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 hardwired but dynamically determined through data values. So object-oriented programming gives you more flexibility than structured programming in the same way that relational databases give you more flexibility than hierarchical or network databases.

KEEP YOUR EYE ON THE INTERFACE

This paper uses the term "interface" in a very general sense: *the set of specifications defining how one object communicates with another to achieve a desired purpose*. Interfaces are contracts defining valid expectations in a relationship between objects. Any client that makes a request following this format and within the supported constraints will get the information or action they want.

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 <u>not</u> include how these methods are implemented.

It is thus entirely possible to have <u>different</u> implementations of the same interface, 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.

This is absolutely crucial to object-oriented programming:

> **Applications use a constant interface to manipulate objects with different implementations.**

This lets the application ignore irrelevant differences between classes of objects. If an application requires a given interface, any class that supports that interface will do.

The separation of implementation and interface also prevents ripple effects (changes in one module leading to changes in other modules); anything that can be changed without modifying the interface has no impact on the calling application. You can thus modify and extend programs without altering code you've already written.

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.

In this paper, I will be examining four kinds of interfaces: Instance Of, Uses, Type Of, and Delegates. All of these interfaces allow implementations to be changed provided the contract promised to client objects is maintained.

Interface	Relationship	Benefit
Instance Of	One Class, many Instances	Change all instances at once.
Uses	One Server, many Requestors	Avoid ripple effects
Type Of	One Superclass, many Subclasses	Add on to existing code.
Delegates	One or more Delegees, many Delegators	Dynamically change method interface.

'INSTANCE OF INTERFACE

The "Instance Of" interface is a relationship between many Objects and one Class. The Class knows the methods supported by the Objects, and default values for instance variables of the Objects. The Object knows where to locate its Class, and non-default instance variable values.

This interface allows the methods or default instance variable values of every Object associated with the Class to be changed at once. Such a change can be undertaken before execution (by changing the class definition with the Class Editor) or during execution using the methods of the Class class.

Changing Widget Classes

It must be remembered that widget classes are loaded from the Resource entry of the frame rather than from the Class entry itself. So changing the properties of the Class entry alone is insufficient. You must update the Resource entries associated with the class by either opening the class through the Resource entry or synchronizing the Resource entry after changes have been made.

Example: Year 2000 maintenance

Here is an example of editing a widget class before execution. Suppose that a user-defined class named `DATE` was used for every date in the application, with a two-digit year. You could change every widget in the application with a single operation, by opening the `DATE` class definition using the Resource entry and changing the format from `MMDDYY8.` to `MMDDYY10.`

'USES' INTERFACE

The Uses interface is a relationship between many Requestors and one Server. The Server knows how to implement a request, and how to collaborate with other Servers as needed. The Clients know where to locate the Server, and how to make a request using the Uses interface of the Server. The Uses interface is so important that in many object-oriented texts this is simply termed the "interface" of the object.

A primary benefit of the Uses Interface is that it prevents "ripple effects." When changes in one object require changes in another object, this is a "ripple effect," and it can be the cause of maintenance nightmares. Ripple effects can result in unmaintainable systems, in which problems cannot be fixed without breaking something else. But you can avoid ripple effects by keeping the Uses interface of a Server constant while making a change. There will then be no need to modify the Requestor classes that make use of the server.

It is quite simple to change the Server used by a Requestor during execution of a program. Objects (including servers) are identified in SAS by numeric values. So all you need to do is change the value of the variable identifying the Server to a value pointing to a different Server, and future requests will go to the new Server.

It can sometimes be difficult to determine the numerical address of a Server object, 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. You use the `LOADCLASS` function to find the class object, and then send the `GET_INSTANCE` method to the class object to obtain the address of the Server.

Non-visual Objects

Servers are typically non-visual objects. Many people fail to realize that widgets are not the only kind of object. 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, all requesting services from a common Server (the Model.)

Object Marketplace

There may be several Servers which can provide the desired service. In some applications (such as Multidimensional databases), we have developed "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, then 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 that message. For example, the Marketplace might identify a presummarized table which can quickly provide a requested set of statistics (McNee, 1997.)

'TYPE OF INTERFACE

The Type Of interface is a relationship between many Subclasses and one Superclass. The Superclass knows how to execute methods common to all Subclasses, how to execute frameworks using Subclass details, and the location of its own Superclass. The Subclass knows the location of the Superclass, extensions to Superclass methods, and additional methods.
A primary benefit of the Type Of interface is that it lets you extend the behavior of an existing class without modifying or copying that class.

In contrast to C++, SAS does not use the superclasses for type-checking. SAS lets you interchange classes from different hierarchies which support the same interface. But if the programmer makes a mistake a message may be unrecognized (bringing execution to a halt) or be misinterpreted. So simplicity and flexibility are obtained at the cost of the increased risk of an untyped language.

The Type Of interface is also known as "Inheritance," a distinctive feature of object oriented programming. 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 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.

Example: Reporting System

A useful approach to application design is to design a generalized application which depends upon a certain Uses interface as implemented in an abstract Superclass (an abstract class is one that cannot be instantiated.) You can then develop a library of Subclasses which each support the same interface, doing the same thing in different ways. As

new needs arise, you can extend the library of Subclasses without having to retest the existing Superclass framework. You can also extend the capabilities of the Superclass being assured that the new capabilities will work for each current and future Subclass.

We have developed several systems which organize reporting and graphing programs in this way. Each type of task is defined by a class, and each requested task is defined by an object. You can store, retrieve, and modify the task specifications, execute the task, and view the results. Enhanced capabilities include cross-indexing of task output by subject, and caching of output for efficiency.

This system is similar in many ways to data-driven systems, but the tasks are defined by stored objects (Norton, 1996b) rather than rows in a table. I consider the object-oriented approach to be superior because it allows different tasks to use different instance variables and to share common functionality through inheritance.

Swapping Widget Classes

One of the most useful tricks I have found is swapping widget classes for existing applications. This lets you extend the user interface of an application in the same way that the last example let you extend reporting capabilities. You can change the behavior and appearance of an application simply by plugging in a different class.

The key is to recognize that widgets are not directly attached to their classes; if they were, you would need to modify each widget individually. Widgets are attached to classes through Resource entries, which are lookup tables of which classes to use. If you change the appropriate Resource entry, the widgets will use different class definitions.

Let's look at Resource entries in detail, then I will return to the question of how to change the class associated with a given widget.

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 "RENAME".

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.

Redefining Resource Entry Aliases

Resource entries are essentially lists of classes assigned to aliases. Applications can refer to the classes by alias, rather than directly. The alias names themselves are fixed, and derived from the entry name of the original class added to the resource entry. For example, the class `APPL.WIDGETS.SELECT.CLASS` (a Radiobox) would

be assigned the alias `SELECT.RADIOBOX`.

Aliases provide SAS/AF programmers with a way to change libnames or move class definitions from one catalog to another. But they also provide a way to assign existing widgets in an application to new classes. If you replace the class assigned to a given alias with another, the application will begin to use the new class instead of the old one. For example, the alias SELECT.RADIOBOX could be reassigned to another radio box class.

Resource entries are primarily used for Frame entries, but can also be for nonvisual objects. You might create an instance of a Model class with:

```
project_res = loadres ('project');
alias_l = getniteml (project_res, 'alias');
model_c = getniteml (alias_l,
                     '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 alias associated with each widget is translated to a class entry according to the alias definitions in the resource entry. The widget objects are created by combining the class information of the resource entry with the stored instance variable data. You can change the resource entry associated with a frame so that a given alias is mapped to a different widget class, but if you do so you must be certain that the new widget class can be instantiated using the instance variables stored by the old widget class.

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.

If you use a single Resource entry for your entire application, then changes to an alias will apply throughout that application. You can change some frames but not others by using multiple resource entries; you can change some widgets but not others by using different aliases (which may or may not point to the same class).

Redefining Aliases Using Composite Widgets

Earlier I said that aliases can only be redefined to other classes of the same type; for example, `SELECT.RADIOBOX` could not be redefined to point to a listbox class. A useful workaround is to use Composite widgets as wrappers around whatever you want to put on the screen. A Composite widget fills in a rectangle on the screen with whatever contents you define. So you could define a Composite widget class containing a radiobox, and it might be assigned the alias SELECT.COMPOSITE. Later, you could redefine the alias to point to a different Composite widget class containing a listbox. This would work well, as long as you remembered to define methods in the listbox composite class that would emulate the Uses interface of a radiobox.

Runtime Widget Specification

Using composite widgets, it is even possible for a single widget to display different contents at different times during a single execution. Suppose we have different nonvisual objects which are represented differently on the screen (perhaps some are displayed using SAS/GRAPH Output widgets while others are displayed as Catalog Entry Viewer widgets.) We can connect a composite widget to one of these items, and let the nonvisual object determine what is displayed within the composite. (We have even managed to make this work within extended tables, so some of the rows contain graphs and others contain text.)

1) Define a nonvisual class `DISPLAY`. Override the `_DRAW_` method to accept an argument `IN_CANVAS_O` (a composite widget). The `_DRAW_` method will draw items into the composite widget using the `_NEW_` metaclass method. Sample code is shown on the next page.

2) Define a Composite class `CANVAS`, with no child widgets. Define a `SET_DISPLAY` method which removes any existing widgets. Define a `_REFRESH_` method which invokes the `_DRAW_` method of the current `DISPLAY`, and add a `_REFRESH_` command to `_POSTINIT_`.

3) Now you can use `SET_DISPLAY` at will to attach instances of the `DISPLAY` class to a `CANVAS` widget, and they will draw themselves on the screen.

This approach is similar to OLE, which lends a section of the screen to another program.

```
DRAW:
   method
     in_canvas_o 8
   ;
   /* IN_CANVAS_O is parent region
      of new widget */
   region_l = makelist();
   dummy = insertn (region_l,
          in_canvas_o, -1, '_parent_');
   /* size new widget to fit
      IN_CANVAS_O */
   defaults_l = makelist();
   dummy = insertl (defaults_l, region_l,
                    -1, '_region_');
   call send (in_canvas_o,
              '_get_object_size_',
              width, height);
   dummy = insertn (region_l, 0, -1,
                    'ulx');
   dummy = insertn (region_l, 0, -1,
                    'uly');
   dummy = insertn (region_l, width, -1,
                    'lrx');
   dummy = insertn (region_l, height, -1,
                    'lry');
   dummy = insertc (region_l, 'pixels',
                    -1, 'units');
   /* draw a graph widget */
   graph_c = loadclass
              ('sashelp.fsp.graph');
   call send (graph_c, '_new_', graph_o,
              defaults_l);
   ignored = dellist (defaults_l, 'y');
   call send (graph_o, '_set_graph_',
              'demo.example4.gchart.grseg');
   endmethod;
```

DRAW method of the DISPLAY class

```
REFRESH: /* _REFRESH_ method */
   method
   ;
   display_o = getnitemn (_self_,
                          'display_o');
   if display_o then
      call send (display_o, 'draw',
                 _self_);
   call super (_self_, _method_);
   endmethod;
```

REFRESH method of the CANVAS class

```
POSTINIT: /* _POSTINIT_ method */
   method
   ;
   call super (_self_, _method_);
   call send (_self_, '_refresh_');
   endmethod;
```

POSTINIT method of the CANVAS class

```
SETDISP: /* SET_DISPLAY method */
   method
      in_display_o 8
   ;
   /* first remove any old widgets */
   child_l = getniteml (_self_,
                        'children');
   do c = 1 to listlen (child_l);
      child = getitemc (child_l, c);
      call send (_self_, '_get_widget_',
                 child, widget_o);
      call send (widget_o, '_term_');
   end;
   /* now set new display object */
   dummy = setnitemn (_self_, in_display_o,
                      'display_o');
   /* on next refresh, display_o will
      draw itself into the composite */
   call send (_self_, '_need_refresh_');
   endmethod;
```

SET_DISPLAY method of the CANVAS class

'DELEGATION' INTERFACE

The Delegation interface is a relationship between many delegators and one or more delegees. The Delegee knows methods which may be executed upon request of a Delegator. The Delegator knows where to locate the Delegees.

Delegation offers the benefit of dynamically changing the method (Uses) interface.

Delegation can be accomplished manually, by defining a method that takes no action except to invoke the same method upon another specified object. SAS allows automatic delegation to be specified, so that if one object does not know how to execute a method, it will request another object to do so.
An object may delegate to a list of several delegees. The first object on the search list to support the requested method wins.

Delegation is interesting because:

1) It is dynamic (the delegee objects can be changed at run time), allowing the methods supported by an object to change during execution, and

2) It can delegate to multiple objects, permitting the simulation of multiple inheritance through a technique known as "mixins."

Example 1: The State Pattern

Suppose you want an object of one class (perhaps, CATERPILLAR) to change into an object of another class (BUTTERFLY), upon receiving a METAMORPHOSE message. A first attempt might look like:

```
MORPH: /* METAMORPHOSE method */
      /* Incorrect!         */
   method
   ;
   butterfly_c = loadclass ('butterfl');
   call send (butterfly_c, '_new_',
              butterfly_o, 0, _self_);
   _self_ = butterfly_o;
   endmethod;
```

METAMORPHOSE method of the CATERPILLAR class

The problem with this approach is that the Butterfly object has a different Object ID than the Caterpillar object, even though one is supposed to change into the other. Any references to the Caterpillar object will not be updated to point to the new Butterfly object.

The solution is described in the State pattern of the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma 1995, page 305). We use this book as our standard handbook of best practices in object oriented programming.

Use <u>two</u> objects to describe a Caterpillar, one for the constant parts (call it Lepidopteran), and one for the changeable state (the caterpillar part). Let everyone refer to the object using the ID of the constant part, but have that object delegate any unknown methods (such as CRAWL) to the Caterpillar state object.

When the constant part receives a METAMORPHOSE message, replace the Caterpillar state object with a Butterfly state object. The Butterfly composite object can be referenced with the same ID as the former Caterpillar, but it can FLY now and not CRAWL anymore.

Note that when I create the Butterfly State object, I inform it of the address (_SELF_) of the Lepidopteran object. This lets the Butterfly State object access all the methods of itself (such as its name) rather than just those methods defined in the state object. This technique is based upon the Client-Specified Self pattern of Viljamaa (1995.)

```
MORPH: /* METAMORPHOSE method */
   method
   ;
   /* create butterfly state object */
   /* pass it _self_ in case it needs
      access to this object */
   butstate_c = loadclass ('butstate');
   call send (butstate_c, '_new_',
              butterfly_state_o, 0,
              _self_);
   /* get rid of the old caterpillar
      state object */
   call send (av_state_o, '_term_');
   /* set the current state to
      butterfly.  This object delegates
      methods such as FLY to
      AV_STATE_O */
   av_state_o = butterfly_state_o;
   endmethod;
```

METAMORPHOSE method of the LEPIDOTERAN class

Example 2: Mixins

C++ programmers often miss multiple inheritance when they program in SAS. But it can often be simulated with the use of mixin objects (Gamma 1995, page 16) and delegation. (The Client-Specified Self pattern can come in handy here as well.)

Suppose, for example, that we have a CAR class and we have a BOAT class, and we want an AMPHIBIOUS_VEHICLE class with properties of both. We can create both a CAR and a BOAT object as well as our AMPHIBIOUS_VEHICLE object, and have the later delegate to both of its components. So if we ask the AMPHIBIOUS_VEHICLE object to DRIVE, the DRIVE method of the CAR component will be executed.

The _NEW_ method would create the CAR_O and BOAT_O objects to be "mixed in". These would later be deleted by the _TERM_ method.

```
_NEW_:
   method;
   call super (_self_, _method_);
   /* delegates to CAR_O */
   car_c = loadclass ('car');
   call send (car_c, '_new_', av_car_o);
   /* delegates to BOAT_O */
   boat_c = loadclass ('boat');
   call send (boat_c, '_new_',
              av_boat_o);
   endmethod;
```

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.

These are not just theoretical concepts. We are using them more and more. The learning curve may be difficult, but these techniques enhance the user interface and simplify the architecture of applications.

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.

McNee, Amy Turske (1997), "The Evolutionary Data Warehouse: An Object-Oriented Approach," in *Proceedings of the Twenty-Second Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

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

Norton, Andrew A. (1996b), "Persistent Storage of SCL Data Objects," in *Proceedings of the Twenty-First Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Viljamaa, Panu (1995), "Client-Specified Self," in *Pattern Languages of Program Design*, edited by James O. Coplien and Douglas C. Schmidt. Reading, MA: Addison-Wesley.

ACKNOWLEDGEMENTS

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: aanorton@trilogy-cnslt.com