The Benefits of Object-Oriented Application Development Using the SAS® System

Randy Pierce, SAS Institute Inc., Cary, NC

ABSTRACT

The SAS System can now take full advantage of the object-oriented (OO) paradigm. There are volumes of materials describing the benefits of object-oriented programming in general. There are also manuals describing *how* to write OO applications with SAS software. This paper describes *why* you may want to use OO techniques when writing applications based on SAS software.

Object-oriented programming (OOP) is a refinement of modular programming. SAS Institute has provided OO constructs that build upon existing modular programming techniques available to Screen Control Language (SCL) programmers. As a result, well-written modular SCL programs are but a step away from becoming classes in a class hierarchy. A class hierarchy can act as a set of generic tools that can be applied to virtually any application.

INTRODUCTION

You do not need a thorough grounding in object-oriented programming in order to understand this paper. On the contrary, you will see how easy it is to convert a modular application based on the SAS System to an OO application. The process of converting an application can itself enhance your understanding of this new paradigm and how it applies to application development with the SAS System. It is expected that you are familiar with the SAS System and have been exposed to SCL.

Object-oriented programming is based upon modular programming. *Modular programming* is the process of breaking apart complex programs and then organizing the resulting smaller, simpler programs. The following sections address the potential drawbacks as well as the advantages of four ways to use modular programming with SCL.

IMPLEMENTING MODULES

This paper uses a simple human resource system (HRS) as an example of a modular application. The programs in question manipulate records describing employees of a company. Each employee record has a value for these five variables:

NAME	the name of the employee.
DEPT	the department to which the employee is assigned.
STARTDTE	the date the employee begins working for the company.
STOPDTE	the date the employee stops working for the company.
EMAILADR	the electronic mail address of the employee.

Suppose some SCL programs in the HRS become very difficult to maintain. It is imperative to minimize the maintenance costs for an application since most applications are in a maintenance phase longer than they are in any other phase of the development cycle. In an attempt to make the HRS easier to maintain, you may decide to use modular programming. Modular programming allows you to keep (and thus maintain) commonly used code segments in only one location regardless of the number of times those segments are executed. Since modular programming involves writing each commonly used code segment only once, you must first identify frequently executed code segments.

While examining the first program in the HRS, you notice several occurrences of code designed to transfer an employee from one department to another. Each occurrence prints a warning message if the user attempts to transfer an employee into the department to which they already belong. Otherwise, the new department value is assigned to the employee. These code segments might look like this:

```
if dept = newdept then
   put 'WARNING: ....';
else
   dept = newdept;
```

Discovering similar code segments is just the first step. Now you must decide how to implement these segments as a single segment that can be called when needed. A code segment designed to be called from other programs is called a *module*. Modular programming centers around creating and managing modules. To see why the OO approach is the best way to implement these modules, examine the following traditional alternative approaches and their limitations.

Labeled Sections

Since you found these lines repeated in a single program, you can eliminate all but one occurrence of this code segment by implementing the module as a labeled section. A labeled section can be executed with the SCL LINK statement. If the module were implemented as a labeled section, it might look like this:

```
transfer:
   if dept = newdept then
      put 'WARNING: ....';
   else
      dept = newdept;
   return;
```

If the 'transfer' module were implemented as a labeled section, it could be invoked with a line like this:

```
link transfer;
```

Labeled sections are ideal when working with one program. They can help segment a given program and divide it into smaller, simpler pieces. However, a labeled segment can only be executed within the program where it is defined. If you find other programs in the HRS which need to transfer employees, you will have to create a labeled section in those programs. All of the 'transfer' modules will be nearly identical, and probably need to be keep synchronized. Thus, labeled sections give you the power to write a module once per program, but they do not give you the power to write a module once per application. In order to further

ease your maintenance chores, you need to find a technique that allows many different programs to share a common module.

Macros

A traditional SAS programmer knows exactly what to do when writing modular code: use macros. Macros have long been used to implement DATA step and PROC step modules. Any base SAS program can call a macro once it has been defined. SCL programs can use this same technique. If the module were implemented as a macro, it might look like this:

```
%macro transfer(dept,newdept);
   if &dept = &newdept then
      put 'WARNING: ....';
   else
      &dept = &newdept;
   %mend transfer;
```

If the 'transfer' module were implemented as a macro, it could be called with a line like this:

```
%transfer(dept, newdept);
```

Note that the macro actually looks *more* complicated than the original code. Opting to implement modules as macros tends to lead to ampersands, percent signs, and various quoting problems. One of the objectives is to make already-complex code *easier* to read and understand.

Note that the implementation did not have to be so convoluted. You could eliminate some of these special characters (percent signs and ampersands) if the macro always used the same SCL variables when called. For example, you could have insisted the macro use the SCL variables DEPT and NEWDEPT. But forcing calling programs to know *how* a macro works and to use specific variable names for specific purposes makes the macro more difficult to use.

Worse yet, if two macros use the same SCL variable name for two different purposes, they become virtually impossible to call from the same SCL program. To help solve these variable name problems, the macro shown actually receives the names of all SCL variables that it can use. However, solving the name scope problem made the code more difficult to read and understand.

There is a second potential problem when using macros to implement modules. Since SCL is compiled (rather than interpreted like the SAS language), each and every SCL program that calls a macro must be recompiled whenever that macro is updated. Keeping track of which programs need to be compiled when a given macro is edited can cause logistical problems.

A third, although more technical, problem is that implementing modules as macros does not reduce the size of the compiled SCL code. When an SCL program references a macro, the lines generated by that macro are included to the compiled code as if those lines existed at that location in the program. If a complex program is growing so large that it is in danger of exceeding the size limit imposed by SCL, macros will be of little use in breaking apart the program.

Macros prove to be a better way to implement a module than labeled sections since you only need to write the module once. A module implemented as a macro can be used by many different programs. However, you trade several easy to maintain occurrences of a module (one each program which needs to execute that module) for a single less easily maintained occurrence. The single macro implementation will need additional information indicating which programs should be compiled when the macro is updated. Further, the statements in the macro will probably be more difficult to read than the statements in the original SCL code. You can significantly reduce your maintenance chores by keeping the implementation of a module simple yet allowing many programs to execute that implementation.

SCL Entries

Implementing each module in its own SCL program has become very popular. Since each SCL program has its own set of variables (and cannot affect variables in other SCL programs), there is no need for calling programs to rely on how called programs work or which variables they use. Instead, use the ENTRY statement to pass the values of variables from one SCL program to another. If the module were implemented as an SCL entry, it might look like this:

```
entry dept newdept $5;

init:
   if dept = newdept then
      put 'WARNING: ....';
   else
      dept = newdept;
   return;
```

If the 'transfer' module were implemented as an SCL entry, it could be called with a line like this:

```
call display('lib.cat.transfer.scl',curdept,
             newdept);
```

The example call assumes that the SCL entry is named TRANSFER and is stored in a catalog named CAT in a library currently accessed by the libref LIB. In fact, calls to any module implemented as an SCL entry require the calling program to know where that SCL entry was stored in addition to its name. Note also that the module name must be a valid catalog entry name.

The code comprising the module is, again, simple SCL code, void of unnecessary ampersands and percent signs. Updating this module no longer requires you to compile programs that use this module. Instead, only the SCL entry used to implement the module needs to be compiled if the lines of code for this module are updated. Note also that any program in the HRS can call this implementation of the module.

While this technique does combine the advantages of both the labeled section and the macro approaches, it still has some drawbacks. The major disadvantages of this technique apply equally well to our last option, so we will delay discussion of them until our final option is presented. Minor disadvantages to implementing modules as SCL entries stem from the isolation of each module in its own catalog entry.

Since each module is in a separate catalog entry, each module name must be unique among all modules stored in the same catalog. You can opt to give different modules the same name only if those modules are stored in different catalogs. For example, we can only have one TRANSFER.SCL entry per catalog.

Suppose there were a module designed to move funds from one account to another. The module written to move funds would either have to be named something other than TRANSFER or stored in a catalog other than the one storing the TRANSFER method written for employees. A naming convention could clarify the purpose of the methods, but at a cost.

If you use a naming convention, you quickly loose the ability to give modules descriptive names. For example, EMPTRANS and FUNDTRANS could both be stored in the same catalog. More often, however, developers use the catalog name to clarify the purpose of the modules. You could have two SCL entries named TRANSFER if one were stored in a catalog named EMPLOYEE and the other were stored in a catalog named FUNDS. If you opt to use multiple catalogs, you may find yourself pondering the location of a version of TRANSFER you need to call.

There is another minor problem associated with implementing modules as individual SCL entries. If two modules contain similar lines of code, you need a third SCL entry that both modules could call. If you extrapolate this idea, you can see how easy it is to have one SCL entry call another which calls another which calls another and so on. The proliferation of SCL entries can make the application appear more complex than it needs to, and the repeated calls to SCL entries can impede the application's performance.

Using an SCL entry to implement a module can greatly reduce your maintenance costs. You no longer need to keep careful track of the programs which execute this module as you do with the macro approach. Yet the implementation of the module is as simple as it was with the labeled section approach. However, the various naming conventions which tend to surround this technique are themselves troublesome. You could further reduce your maintenance tasks if you could give each module a more meaningful name and group related modules together.

Methods

You can implement a module as a method nearly the same as you would implement it as an SCL entry. In fact, the code actually resides in an SCL entry, but there are a few syntax changes. Each module

- has a descriptive label
- uses the SCL METHOD statement to declare its own parameters and starting point
- uses the SCL ENDMETHOD statement to declare its own ending point.

If the module were implemented as a method, it might look like this:

```
transfer: method dept newdept $5;
   if dept = newdept then
      put 'WARNING: ....';
   else
      dept = newdept;
   endmethod;
```

If the 'transfer' module were implemented as a method, it could be called with a line like this:

```
call method('tools.master.employee.scl',
            'TRANSFER',dept,newdept);
```

The invocation of a method contains the name of the SCL entry containing the method, the location of that SCL entry, and the name of the labeled section to execute.

A significant benefit to using the method approach rather than the SCL entry approach is that you gain an extra naming level. Rather than label the code 'init', 'main', or 'term' as you would when using SCL entries to implement a module, you can now use a more descriptive label. Further, each module name need only be unique among other modules stored in the same SCL entry.

The ability to store multiple modules in a single catalog entry leads you to group related code segments together. As a result, fewer catalogs are required. For example, a FUNDS.SCL entry containing methods named TRANSFER and INVEST, might reside in the same catalog as the EMPLOYEE.SCL entry containing methods named TRANSFER and RENAME.

Grouping modules together has another benefit as well. Modules that have been stored in a single SCL entry can link to labeled sections in that SCL entry. Thus, related modules can share common code without having to call yet another SCL entry. For example, suppose the TRANSFER and RENAME modules of the HRS shared a few lines of code. If these modules were implemented as methods in a single SCL entry, they could simply link to common lines of code.

At this point, the maintenance chores of writing and maintaining the modules has be reduced nearly as far as is possible. Each module is implemented only once, yet can be executed from any program. The implementation of the module is as simple as the original SCL code. Each module can be given a fairly descriptive name. Related modules can be grouped together to facilitate sharing code they might have in common. However, we have not considered how easy it is to use a module implemented with any of the described approaches. Reducing the maintenance required when updating the modules is only half the problem. You should also consider ways to make modules easy to use.

Comparison of Techniques

To summarize the implementation options for modular programming with SCL:

Labeled Sections

- are better than one large program
- can reduce the size of any one program
- can only be used by the program in which they are defined.

Macros

- are better than labeled sections
- can be used by more than one program
- have a very high maintenance cost
- do not reduce the size of programs that call them.

SCL Entries

- are better than macros
- reduce the size of the original program
- are easier to maintain than macros
- can lead to naming conventions or modules scattered over several catalogs.

Methods

- are better than SCL entries
- allow you greater flexibility for naming the module
- allow you to group related modules together.

Implementing modules as methods appears to be the optimal way to write modular code with SCL. However, modular programming has some limitations that are present in all implementations. Object-oriented code overcomes many of these limitations. The following sections show how simple it is to convert modular SCL programs into OO ones. You will also see how great the return is for such a small modification.

USING METHODS

This portion of the paper assumes that you have practiced modular programming. The two most viable approaches for implementing SCL modules are the SCL entry and method approaches. Weeknesses discussed in this portion of the paper apply to both implementations. Examples, however, are shown using the method implementation approach. Further, the paper assumes you have already written the following methods:

RENAME	to change the name of an employee.
TRANSFER	to change the department to which the employee is assigned.
HIRE	to assign the STARTDTE.
EMAIL	to email a message to the employee.

The following sections show how this modular implementation might be used. In the process, you can note problems that are addressed with the OO approach.

Parameters

When you call a method, you need to pass it all of the values that it requires. For example, the TRANSFER method undoubtedly needs the new department value. But the TRANSFER module already shown also requires the current department value. Suppose the TRANSFER method were enhanced to ensure the employee is currently employed at the company. In that case, it needs the STOPDTE and STARTDTE values. If the TRANSFER method automatically sends the employee email notifying them of the transfer, it needs the email address as well.

In practice, many methods make use of nearly all attributes of the piece of data they manipulate. TRANSFER uses most of the attributes for EMPLOYEE, including the forwarding of EMAILADR to the EMAIL method. To allow for future enhancements, it is usually good practice to pass all attributes of a piece of data to all methods that manipulate that type of data. In this case, we need to pass NAME, DEPT, STARTDTE, STOPDTE, and EMAILADR to all methods in EMPLOYEE.SCL.

There are at least two approaches for passing these parameters to each method. The first and most obvious approach is to place these parameters on the METHOD statement for each method. The TRANSFER method would look something like this:

```
transfer: method name $40 dept $5
                startdte stopdte 8
                emailadr $20
                newdept $5;
   :
   endmethod;
```

Repeating a similar statement for each method can cause tremendous maintenance problems. If the length of an attribute changes, or if additional attributes are added, numerous lines of code will be affected. Listing the parameters also requires calling programs to pass the right values in the right order. No errors would be generated at compile time if a program calling a module has two parameters out of order, but executing the call would certainly not perform as intended.

A slightly better approach is to pass a list of attributes to each method. In this case, the entire list of attributes is passed with a single variable: the list identifier. Methods can retrieve values from the list by name and can forward the list of values to other methods as well. A version of the transfer method using this technique, would look something like this:

```
transfer: method employee 8 newdept $5;
   dept = getnitemc(employee,'DEPT');
   startdte = getnitemn(employee,'STARTDTE');
   :
   rc = setnitemn(employee,newdept,'DEPT')
   endmethod;
```

However, this approach requires the methods to pull values out of the list before they can be used, and it requires the calling code to insert values into a list before calling the method. Likewise, the method must assign new values to named items in the list if it is to affect the values of attributes for the employee. Depending on the complexity of the method, your code may spend more time manipulating the list of values than the values themselves.

It would be nice if all methods in EMPLOYEE.SCL could automatically have at their disposal the values for all attributes of an employee. If this were possible, the methods could be simplified to the point of only receiving values not assigned to attributes of an employee. In other words, we would *like* to be able to write the TRANSFER method like this:

```
transfer: method newdept $5;
   if (dept = newdept) or
      (date() > stopdte) then
      put 'WARNING: ....';
   else
      dept = newdept;
   endmethod;
```

Note that this implementation only receives the new department value. The current values for all attributes of the employee are assumed to be available in SCL variables of the same names. Note also that this method affects the value of the attribute DEPT by simply assigning a value to the SCL variable of the same name. The OO approach makes passing attributes nearly effortless, as described later in this paper.

Generic Programs

As we have seen, an EMPLOYEE has a NAME attribute and a RENAME method exists to assign a new value as an employee's name. But many other types of data may have a NAME attribute (and thus their own RENAME method). Suppose you want to write one interface that will allow the user to rename any type of data.

Imagine a screen that prompts the user for a new name. When the user enters a new name, the screen should invoke the RENAME method. But which RENAME method? Keep in mind that invoking a method requires knowing both the name of the method and the name of the SCL entry containing that method. What would the code look like?

The code will probably need some sort of IF-THEN logic to ensure the right method is called. The name of the SCL entry containing the RENAME method depends upon the type of data being renamed. If you can somehow derive the data type, the code might look like this:

```
select(datatype);
   when('EMPLOYEE')
      call method('TOOLS.MASTER.EMPLOYEE',
                  'RENAME',newname);
   when('FILE')
      call method('TOOLS.TOOLS.FILE',
                  'RENAME',newname);
   when('CATALOG')
      call method('MISC.MASTER.CATALOG',
                  'RENAME',newname);
   otherwise
      put 'ERROR: ....';
   end;
```

If the implementation were similar to the one shown, then it would have to be updated each and every time a new data type were added. It would also have to be updated if the SCL entry containing the RENAME method for a given data type were moved or was itself renamed. Following strict naming guidelines would help reduce this problem. If the SCL entries were always named the same as the data type and they were all stored in the catalog named TOOLS.MASTER, then the invocation of the RENAME method would be as simple as

```
call method('TOOLS.MASTER.' || datatype,
            'RENAME',newname);
```

The real question is how the value for DATATYPE is determined. If you claim that that value was passed to the interface as a parameter, you have not answered the question. How did the calling code determine DATATYPE?

It would be convenient if no code had to use the value DATATYPE directly. You would probably like to ask a piece of data to rename itself and not have to worry about its data type, the location of the appropriate RENAME method, or anything else. Suppose an entire piece of data (an employee, for example) can be assigned to one variable. It would be nice to ask that variable to RENAME itself without having to use any additional information. Suppose the data to be renamed were assigned to a variable named DATA. The SCL behind the generic RENAME interface would be as simple as

```
call method(data,'RENAME',newname);
```

The OO approach makes writing generic code this easy. It allows for a DATA variable that not only knows the values for all of its attributes, but also knows its data type. Because the DATA variable knows its data type, it can use that information to determine the location of appropriate methods. We can now see how the OO approach has overcome problems associated with even the most robust implementation of modular programming.

USING CLASSES

The EMPLOYEE data type as previously discussed in this paper indicates that both writing and calling modules could be much easier if every data type knew both its attributes and its modules. If data types knew their attributes, these values could be passed automatically to their modules. If data types knew the locations of their modules, generic code could invoke a module by its name alone.

SAS has opted to store the module name and location information in an SCL list. The module names are the names of list items and the module locations are the values of those list items. Storing module locations as named items in a list gives you much more freedom for naming your modules. Note that the name of the labeled section containing the module is considered to be a portion of that module's location, rather than its name.

For example, the labeled section implementing a module to send an employee to SUGI might be named SUGI, but the name of a list item (and thus the name of the module) pointing to that labeled section could be SEND_TO_SUGI. Thus, module names no longer need to be valid catalog entry names (as with the SCL entry approach) or label names (as with the method approach). A table of module names and locations might look like

NAME	LOCATION
TRANSFER	EMPLOYEE:TRANSFER
HIRE	EMPLOYEE:HIRE
RENAME	EMPLOYEE:RENAME
EMAIL	EMPLOYEE:EMAIL
SEND_TO_SUGI	EMPLOYEE:SUGI

Here, the location is shown as the name of an SCL entry followed by a colon and the name of a labeled section in that SCL entry. The names of the librefs and catalogs containing the SCL entries are omitted due to space constraints, but these SCL entries could have different names or reside in different catalogs.

A stored list containing both the list of attributes and the list of modules is called a *class*. In SAS, such a list is saved in a CLASS catalog entry. Note that a CLASS contains information about a data type, but it does not contain information about any one member of the data type. For example, the EMPLOYEE class defines that members of that class have a name, department, email address and so on. But the employee class does not define the name, department, or email address for one particular employee.

The OO approach does allow for a list containing values for one employee. A list representing an occurrence of a data type is called an *instance*. Each instance has a value for every attribute its data type has. For example, an instance of the EMPLOYEE class has a value for NAME, DEPT, STARTDTE, STOPDTE, and EMAILADR. Instances must also be able to access the class describing their data type.

SAS has chosen to implement instances with SCL lists. The list contains named items. The name of each item is the name of an attribute, and the value of each item is the value of that attribute. There is an additional item named _CLASS_ in the list whose value is the class to which the instance belongs. An instance of

the employee class might look something like this:

NAME	VALUE
NAME	Oscar Oliver Peters
DEPT	MIS
STARTDTE	15MAY91
STOPDTE	.
EMAILADR	OOP@MVS
CLASS	TOOLS.MASTER.EMPLOYEE.CLASS

OO Messages

Using instances and classes can reduce some of the problems associated with modular programming. In particular, it is easier to pass parameters and write generic programs when using the OO approach. To execute a module in OOP, send a message to an instance. The name of the message is the same as the name of the module you wish to execute. But because the message is sent to an instance, the language immediately knows all of the attributes that should be passed as parameters and the location of the module being called.

Suppose Oscar (the employee previously shown) was about to transfer from MIS to Finance (FIN). Assume the SCL variable EMPLOYEE is the list identifier representing an instance of the EMPLOYEE class and currently holds Oscar's information. To send a message to the instance EMPLOYEE, you can use

```
call send(employee,'TRANSFER','FIN');
```

Use the SCL call routine SEND to send a message to an instance. In this case, the instance EMPLOYEE is sent the message TRANSFER. The value FIN is passed to the method as a parameter. Since an invocation of the SEND call routine does not contain the location of the method to execute, SAS must determine its location and what values will be automatically passed to it.

First, SAS must determine the class the instance belongs to. In this case, the EMPLOYEE belongs to the class TOOLS.MASTER.EMPLOYEE.CLASS. Then SAS must determine where the method named TRANSFER used with the employee class is stored. In this case, the labeled section named TRANSFER in SCL entry named TOOLS.MASTER.EMPLOYEE.SCL will be executed.

Note that the call to the TRANSFER method does *not* contain the location of the module. In fact, it does not always contain the name of the labeled section used to implement the module. The mapping of the module name TRANSFER to the labeled section named TRANSFER in the SCL entry TOOLS.MASTER.EMPLOYEE is saved in the class definition.

OO programmers use the term *method* to indicate a pairing of module name and location. A module need not be implemented as an SCL method to be used as a method of a class. For example, TRANSFER, HIRE, RENAME and SEND_TO_SUGI are all methods of the EMPLOYEE class not because of their implementation but because those names have been paired with locations of modules.

Since the class is the only thing referring to the location of a method, it is also the only thing that must be changed if a method is moved. Programs calling methods supply the method name, but not its location. When an instance is sent a message, its class determines the location of the method. As a result, moving a module from one location to another no longer requires you to find and change all calls to that module. For example, moving the TRANSFER method to another SCL entry, catalog, or library would not require you to alter the CALL statement shown above.

When using the OO approach, programmers can omit more than just the location of the method they wish to execute. While the traditional invocation of a module requires explicitly passing it all values it uses, an OO programmer need only pass values that are not assigned to attributes of the instance they wish to process. Values assigned to attributes of the instance to be processed will be automatically made available to the module executed. For example, consider the CALL statement shown above. Before the TRANSFER method is executed, SAS assigns values to all SCL variables in its SCL entry whose names and types are the same as the names and types of attributes in the employee class. Thus, Oscar's name is assigned to the SCL variable NAME, his department is assigned to the SCL variable DEPT, his email address is assigned to the SCL variable EMAILADR, and so on.

Its important to recognize that each instance will pass its own set of attribute values to methods when it is sent messages. This way different instances of a given class can represent different members of that class. Each instance must have attribute values that describe a particular member of the class to which it belongs. Since the values of attributes can differ from one instance to another, OO programmers refer to them as *instance variables*. For example, two different instances of the EMPLOYEE class probably have two different values for the instance variable NAME. For example, one instance might have the value 'Oscar', while another instance might have the value 'Juli.'

The TRANSFER method can use any instance variable of the employee class without having to receive it as a parameter. Note that when the method has completed, SAS automatically sets the instance variable values to the values remaining in the like-named SCL variables. So the TRANSFER method can assign a new value to Oscar's department by simply assigning a value to the SCL variable DEPT.

Also note how easy it is to write generic programs with the OO technique. The code executed with the generic rename interface previously discussed would be as simple as this:

```
call send(instance,'RENAME',newname);
```

SAS uses the definition of the class to which INSTANCE belongs in determining the location of the appropriate RENAME method and in determining what values should be passed automatically to that method. Thus, the data type of INSTANCE does not need to be taken into account at the application level. Instead, the language (in this case SCL), automatically determines the data type of INSTANCE and executes the proper method for you.

OO programmers use the term *polymorphism* to describe the ability to select a module based on more than just the module name. In OOP, polymorphism most frequently involves selecting a module based on both a module name and an instance. In order to fully utilize polymorphism, it is important that classes use the same method names for similar processes. For example, the

previously shown generic code uses polymorphism to select the appropriate module by supplying both the name of the module, RENAME, and the instance to be renamed. This generic program, however, is useless unless all classes use the name RENAME for modules that assign new names to instances of themselves.

Exploiting OO Techniques

At this point, we can see clear advantages to using instances and classes, and writing OO programs. We have seen that methods need not receive as many parameters as modular programming. We have also seen that generic programs can be written and then used by many applications. We have also seen that it is much easier to maintain OO code since calls to modules no longer depend on knowing the location of those modules.

However, we are still using modular programming techniques. We have used OOP to help manage our modules, but we have not altered our fundamental design. In fact, we have not yet mentioned some of the greatest strengths of the OO paradigm. Well-managed modular programming becomes OO programming when you begin to use inheritance and develop a class hierarchy.

Class Hierarchies

As we have seen, a class defines a data type in terms of both instance variables and methods. Many times, two or more classes differ only slightly. For example, suppose you start a summer intern program for college students. Student employees have all the attributes of other employees (NAME, DEPT, and so on), but have an additional attribute, SCHOOL, to indicate the college or university that student attends. Perhaps the student employee class also has a GRADUATE method that converts a student employee into a regular employee.

All methods and attributes pertaining to the EMPLOYEE class apply equally well to the STUDENT EMPLOYEE class. We would prefer to avoid duplicating portions of the EMPLOYEE class when defining the STUDENT EMPLOYEE class. If we consider the many possible types of employees (STUDENT, PART-TIME, FULL-TIME, CONTRACT, TEMPORARY, and so on) then we can see how difficult it would be to keep all of these classes synchronized when a method or attribute is added or changed for the EMPLOYEE class.

To overcome this problem, the OO approach provides a way for one class to reference another class as its parent. A class will inherit both the instance variables and methods defined in its parent class. The result is that any new class need only define attributes and methods that are different from the definitions contained in its parent.

The maintenance troubles alluded to with various types of EMPLOYEE classes can be averted by using inheritance. Rather than trying to repeat the methods and attributes of the EMPLOYEE class in all of the more specific classes, we can simply inherit them. For example, the STUDENT EMPLOYEE class need only define the SCHOOL attribute and GRADUATE method. Provided the STUDENT EMPLOYEE class names the EMPLOYEE class as its parent, it will automatically have all attributes and methods of the EMPLOYEE class.

OO programmers use the terms subclass and superclass to describe the parent-child relationship mentioned above. One class is a *subclass* of another if it names the other class as its parent. The STUDENT EMPLOYEE class is a subclass of the EMPLOYEE class. Similarly, one class is a *superclass* of another if the other class names it as its parent. The EMPLOYEE class is a superclass of the STUDENT EMPLOYEE class.

The design and implementation of a class hierarchy plays a critical role in the success of an OO system. The objective is to create a hierarchy so generic and flexible that the majority of the work required for any application has already been implemented in the form of classes within the hierarchy. Developers can spend their time concentrating on the few classes that are unique to the new application and have not yet been implemented. If the class hierarchy is particularly robust, the new classes can name existing classes as their parent and require relatively few new attributes and methods.

A key advantage OOP has over modular programming is the ability for one application to use modules originally designed for specifications in another application. For example, suppose one application requires the ability to print a file. In all probability, the developer will create a FILE class if one does not exist and implement a PRINT method if one does not exist. The PRINT method for the FILE class was created due to the needs of an application under development. However, future applications that might need to print files can take advantage of the newly available PRINT method of the FILE class.

Frequently, classes originally required for one application turn out to be applicable to other applications. When applications rely on commonly used generic classes, the distinction between one application and another blurs. Instead, the user is given many applications with similar interfaces that can communicate with each other through the common classes. For example, one application might use the FILE class to store information, and another application might use the FILE class to retrieve information. These two applications can effectively communicate with each other through a generic class even if neither one has direct knowledge of the other.

Additional Advantages

Most of the advantages discussed so far were from the programmer's perspective. There are quite a few advantages to OOP that pertain to the user. It is important to realize that users need programs that can fulfill their requirements. Most users do not create their programs, but acquire them instead. If the purchased or developed application is to be of any benefit to the user, they must accurately and precisely describe their requirements.

However, people tend to distort received information when they pass it to someone else. The distortion of a user's requirements is another factor in the usefulness of the program acquired for them. The number of transitions the requirements undergo on their way to the finished product affects the extent to which they are distorted. Thus, even the most precise and accurate requirements can lead to grossly inappropriate applications if they change hands several times before they become statements in a program.

Imagine a scenario where the user verbally describes some requirements to an assistant who writes them down. The written requirements are passed to a secretary that types them. The secretary passes the typed requirements to a program designer. The designer suggests some data structure and program flow which are given to a senior programmer. The senior programmer develops a 'psudo-code' model and gives it to a language expert. The language expert writes some 'core' modules and then lets an assistant write the rest of the program. Is it really difficult to understand why the code written by the assistant programmer acts so differently from the behavior described by the intended user?

To help solve this problem, it is imperative that the user be involved at each transition of the requirements. However, few

users can read a macro called by an SCL program and either confirm or deny that the program will meet their needs. The programmer necessarily separates the user from the programs, but OO techniques can narrow the gap between user requests and the resulting programs.

For example, instances know the class to which they belong and thus the methods to which they can respond. Programmers use this fact to write generic code. They can tell any instance to RENAME itself, and the instance in question will drive the appropriate version of the RENAME method. However, users can use this fact to get some minimal help for instances.

A user can ask any instance to list the methods that are valid for it and then execute a selected method. There should probably be some filtering of methods presented to the user, but the fact remains that instances know what they can do and should be able to provide a list of valid options to the user. A user requesting the ability to terminate employees might also be comforted when they see the item TERMINATE appear in the list of actions that can be performed. The employee in question, however, might not be so comforted!

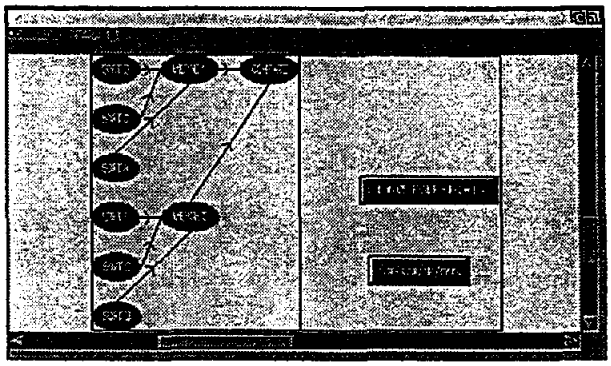
Another advantage is that OOP involves classifying data. Each CLASS of data has its own attributes and methods. The more a class resembles a real-life classification, the easier it will be for the user to interact with the system. It should make sense to a user that an instance of the FILE class has methods named PRINT, EDIT, and MAIL.

A similar advantage is that users can describe application requirements in terms very similar to those used by the programmer. For example, if the user requests that some mechanism exist to terminate an EMPLOYEE, the programmer can simply implement a TERMINATE method for the EMPLOYEE class. Since the implementation of requirements mimics the user's description of them, OO code can be easier to review. OO code can also be more accurate since there are fewer translations from user specifications to lines of code.

OO programs center around classes that closely mimic a particular type of objects in the real world. Since instances of these classes are surfaced to the user, the user tends to want to manipulate them as they would the real-world objects. Note that the term instance identifies something in the computer world, while the term *object* identifies something in the real world. These terms are frequently used as synonyms, but there are some important differences. An instance is used to represent an object. Many times an object (like the living, breathing employee named Oscar) exists both before and after the execution of a program, but an instance (like an instance of the EMPLOYEE class) can exist only during the execution of a program.

There is a further distinction between instances that represent stored data and those that represent portions of the interface. A *widget* is an instance of a class which the user can interact with directly. Most widgets are visual and appear on screens presented to the user. Push buttons, scroll bars, and text entry fields are examples of widgets. When a widget is acted upon by a user, it generally sends one or more messages to an instance. For example, a scroll bar with values 0 through 110 might send the message SET_AGE to an instance of the EMPLOYEE class when ever the user repositions the slider.

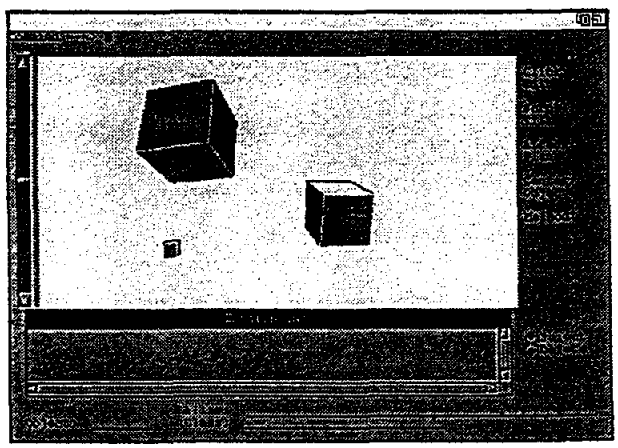
The user's desire to add widgets to the display, move them, and select them has led to various desk-top-like interfaces. A common implementation of this OO environment is an infinitely large plane on which two-dimensional shapes represent various objects.



Display 1 *Two dimensional view of objects*

In the environment shown in display 1, each shape/color pair represents a different class. For example, yellow rectangles might represent employees, while blue ellipses might represent projects. With the shown environment, the text appropriate for each instance is placed on the shape representing that instance. Another approach is to give each class a unique icon and place appropriate text centered beneath the icon.

There might be some merit to representing objects as three-dimensional shapes floating in space. One drawback to this approach is the need for appropriate controls to enable the user to manipulate the objects. A three dimensional environment requires all of the two dimensional controls (like move left/right and move up/down), but needs additional controls to move objects towards the front or back as well as to tilt, twist, and rotate the objects. Another drawback is the complexity of three-dimensional object manipulation algorithms. Algorithms manipulating objects in three dimensions generally requires more powerful hardware, less timely response, or less detailed images than do algorithms manipulating objects in only two dimensions.



Display 2 *Three dimensional view of objects*

In the environment shown in display 2, each plane (face of a cube, side of a tetrahedron, and so on) can represent an object. Thus a single cube can represent up to six different objects, all occupying roughly the same place on the display. By manipulating the cube, you can see up to three of the six faces at once. One motivation for using a three dimensional environment is that users initially find it more enjoyable than a two dimensional one. Another more substantial reason is that some data simply lends itself well to three dimensional representation.

OO Terms

This section of the paper is not intended to be used as a glossary. It does not contain precise, widely accepted definitions. Instead, this section clarifies some of the OO buzzwords you may have heard and relates them to modular programming as it is described in this paper. The terms are arranged in order from those most directly related to modular programming to those least directly related to modular programming.

- METHOD: A pairing of a module name and location. With SCL, most modules are implemented as SCL methods, but they can be implemented in other ways. In OO jargon, the term *method* has little to do with implementation of the module. Instead, it refers to the ability of a class to find a module when supplied with the name of a module. Methods eliminate the need for calling programs to know the location of modules, and thus calling code need not change even if modules move to other locations.
- INSTANCE VARIABLE: An attribute possessed by instances of a class. Instance variable values of one class are automatically passed to methods of that same class. NAME, DEPT, and EMAILADR are examples of instance variables of the EMPLOYEE class. Programs calling methods do not need to explicitly pass the values of instance variables, and thus do not need to change even if instance variables are added, removed, or have their names changed.
- CLASS: A definition of a type of data. It contains at least three items: a list of methods, a list of instance variables, and the name of a parent class. A class is used in two ways. It can either create an instance of or execute methods appropriate for the data type it represents. A class is assumed to have all instance variables defined in its parent. A class also acquires all methods in its parent that it does not redefine.
- INHERITANCE: The receiving of definitions found in another class. Inheritance allows any given class to define only those instance variables and methods which have not yet been defined in another class. Instance variables and methods which have already been defined in another class can be inherited by a new class if the new class names the other class as its parent.
- SUBCLASS: A class whose parent is the given class. Suppose that class A named class B as its parent. Class A is a SUBCLASS of class B. Since a subclass contains all methods and instance variables of its parent, it can be used anywhere its parent can be used. Note that any given class is a subclass of its parent, grand-parent, great grand-parent, and so on.
- SUPERCLASS: A class named as the parent of a given class. Suppose that class A named class B as its parent. Class B is the SUPERCLASS of class A. Note that any given class is a superclass of its children, grand-children, great grand-children, and so on.
- CLASS HIERARCHY: A set of classes in which one class is a superclass of all others in the set. A class hierarchy is the corner stone of OOP. A good class hierarchy contains enough generic classes to meet a large percentage of the requirements for any application.
- INSTANCE: An occurrence of a class. An instance has a value for all instance variables defined in its class. These values are passed automatically to any method that processes the instance. When a message is sent to an instance, it uses its class definition to find the appropriate module to execute.
- MESSAGE: An invocation of a method. When a message is passed to an instance, that instance uses its class to find the location of the module named after the message. The named module is executed and automatically receives all instance variable values.
- POLYMORPHISM: Selecting a module based on more than just the module name. Modular programming is not polymorphic, since calls to modules contain *only* the module name (the name of the labeled section, macro, SCL entry, method, and so on). OOP is polymorphic since two items are required when invoking a module: the object to be processed and the name of the module. Thus, two different objects of two different classes can both respond to the same message, but they might execute different modules. For example, the EMPLOYEE class and FUNDS class might route a TRANSFER message to two different modules. Because of polymorphism, it is much easier to write generic code with OOP than with modular programming.
- OBJECT: A real-world item that a class is based upon. A class named EMPLOYEE might try to emulate employee objects as they exist in reality. Generally, objects exist long before and after instances representing them exist. For example, an employee of the company exists even if there is currently no instance of the EMPLOYEE class whose instance variable values represent that employee object.
- WIDGET: A visual object in the user interface. Push buttons, list boxes, scroll bars, and windows are all examples of widgets.

CONCLUSION

Object-oriented programming is a logical extension of modular programming. OO modules are easier to maintain since they generally do not have to declare as many parameters as their modular counterparts. OO modules are easier to use since calling programs do not need to know the location of the modules. Removing the need for a calling program to know the location of a module makes writing generic tools much easier in OO code than in modular code. Gradually building a set of generic classes can make implementing an application much easier than it would have been using the modular approach.

You have much to gain if your applications are object oriented. First, you can become more closely involved with the implementation of your request (perhaps to the point of browsing the application layer of code) to ensure its accuracy. Second, the application presents you with a very familiar situation. You can interact with the images on your display as though you were interacting with the physical objects those images represent.

Modular programming was, and still is, an effective technique for building a simple application that has little in common with other applications. If applications continue to become more complex and if the need for applications to communicate with each other continues to increase, the need to use OO techniques will also increase.

REFERENCES

SAS Institute, Inc. (1993), *SAS/AF Software: FRAME Entry, Usage and Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute, Inc. (1993), *Getting Started with the FRAME ENTRY: Developing Object-oriented Applications, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS is a registered trademark or trademark 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.