

Designing Complex Systems Using SCL Data Objects

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

INTRODUCTION

"Object-oriented" has become the buzzword of the moment. The recent marketing blitz has bleached it of all meaning, but the underlying concepts go back 25 years. Although these concepts developed incrementally from earlier software design principles, the end result is revolutionary — once you have learned the new way of thinking, you can't go back.

A complete object-oriented language is developing within the SAS® System. Many of the pieces are already in place in release 6.08 of Screen Control Language (SCL), and it is feasible to develop object-oriented systems in SAS just as you might in C++ or Smalltalk.

Most discussions of object-oriented programming in SAS focus on small "toy" problems. Although this technique helps the neophyte to understand every aspect of the example, it gives a limited and artificial view of the use of these powerful new features. This paper is different. It focuses on how objects are used in constructing large-scale complex systems. I will not be spending much time discussing the nuts and bolts of how individual objects can be implemented in SAS. Instead, I will be discussing what objects can do, and how they can be used in combination to tame complex systems.

Object-oriented programming is a natural progression from traditional programming, so I will start there. Terms of special significance are italicized the first time they appear.

WHAT'S SO GREAT ABOUT OO?

Object-oriented programming (OO) has become especially popular for the development of Graphical User Interface (GUI) applications. These applications can be very complex, because the user is given a great deal of freedom. Of course, batch processes can also be very complicated; object-oriented SCL can be helpful here as well.¹

In general, object-oriented languages excel at complex tasks, and are less effective for simple and repetitive tasks.

Handling Complexity

The solution to complexity is to *modularize*: separate the program into pieces, get each smaller piece working and then integrate the modules together. This solution works because smaller modules are generally less complex than larger modules.

Consider the following numerical analogy. Suppose we have 100 items (modules, variables, whatever) in a module. There are $100 \times 99 / 2 = 4950$ pairs of independent items to be

coordinated, an indicator of overall complexity. Let's define the function $C(n) = n(n-1)/2$.

If we can divide these items into two modules, each module has $C(50) = 1225$ pairs to be coordinated, for a total of 2450 intra-module pairs. The aid that modularization provides in dealing with complexity depends upon how loosely coupled the modules are; the savings comes from not having to consider each interaction between items in different modules individually. If the modules are each highly cohesive and perform a single useful action, then integrating them to form a more complex module or system may be no more difficult than combining atomic items, for a total complexity of $C(2) + 2 \times C(50) = 2451$.

Although this mathematical model can only suggest the savings to be gained, pursuing this line of inquiry yields the following conclusions:

- Using more modules gives greater savings, up to the point that coordinating between the modules becomes too difficult. For example, with 10 modules of 10 items, the complexity is $C(10) + 10 \times C(10) = 495$.
- Similar savings can be achieved for maintenance. If we do not have modules, we have to consider each of 100 items for changes. If we have 10 modules of 10 items, we can consider which of the 10 modules needs changing, and then which of the 10 items within.
- Nesting modules within modules can offer still greater savings. Suppose we have 5 modules of 5 modules of 4 items: $C(5) + 5 \times C(5) + 5 \times 5 \times C(4) = 210$.

The above discussion has assumed that each item must be treated individually, but often we have many items of the same type that are each processed in the same way. By structuring the program into repeated parameterized module calls, we can achieve even greater savings. Suppose we have just 1 module of 10 items, which is called 10 times. The complexity is only $C(10) + 1 \times C(10) = 90$. Instead of having to debug ten modules, we only have to debug one module which is then used ten times.

Nesting	Complexity
1 x 100	4950
2 x 50	2451
10 x 10	495
5 x 5 x 4	210
10 + 10	90

This strategy for handling complex problems can be summed up by the phrase:

Divide and Conquer.

Subroutines

The traditional way to "divide and conquer" computer programs has been through the use of *subroutines*. "Subroutine" is not a SAS term: in Screen Control Language they are called "methods" and in the Macro Language they are called "macros". I am going to continue to use the term "subroutine" rather than "method" to distinguish stand-alone modules from the object-oriented *methods* to be discussed later. In SCL, the former are invoked with CALL METHOD and the latter with CALL SEND or CALL NOTIFY.

Input data are passed into a subroutine which then produces output. Internally, the subroutine has its own protected code and its own independent *scope* of local variables, so side-effects are unlikely. As long as the subroutine returns the correct output for the provided input data, then the details of implementation on either side (calling program or called subroutine) are irrelevant. How the subroutine works (the internal implementation) has been separated from what the subroutine does (the external interface). Integration only needs to be concerned with what is externally visible.

Testing is simplified because the behavior of the subroutine is dependent upon the parameters alone and does not change from one context to another. Similarly, maintenance becomes much easier; once you have located and corrected the appropriate subroutine, there is no need to re-examine programs using the subroutine. There is a natural division of responsibilities: the subroutine knows what to do with the input and the caller knows what to do with the results.

In Screen Control Language, a typical subroutine call might be:

```
call method ('Patient','eligible',
            sex, age, is_eligible);
```

The first argument identifies the SCL entry containing the method code: PATIENT.SCL. The second argument identifies the labeled block of code to be executed: ELIGIBLE. The remaining parameters (if any) are defined by the programmer. In this case there are two input parameters, SEX and AGE, and one output parameter IS_ELIGIBLE. Because subroutines contain no persistent data, we know that IS_ELIGIBLE is a function of SEX and AGE.

This technology has been well-established for many years, and you might not see any flaws with it until a better way has been described.

The Problem

If you write a really good subroutine, it might get used over and over again, in many different contexts. It might be used years after you wrote it, or by people you have never met. This is success!

The internal code of this subroutine can be changed without causing difficulty for its *client* programs. But if the *signature* or form of the subroutine call changes, you might need to locate, revise, and test every use of the subroutine.

Unfortunately, subroutine signatures often need to be modified to handle new situations or make additional distinctions. All data values needed by subroutines are (typically) passed as parameters. If a modified subroutine requires additional data, the signature will need to be changed accordingly. For example, the ELIGIBLE method may need to be changed so that it uses different rules for patients in different countries:

```
call method ('Patient','eligible',
            sex, age, country, is_eligible);
```

The Solution

We need a way to pass additional information to a subroutine without modifying the signature. This can be done if a parameter can contain a bundle of data rather than an individual variable. In some languages this can be done with a compound data structure. In Screen Control Language, you can use an SCL list (SAS Institute, 1991). If more parameters are needed by the method, just add them to the list. For example, assuming an SCL list PATIENT1_L contains elements named SEX, AGE, and COUNTRY for a certain patient, we could have:

```
call method ('Patient','eligible',
            patient1_l, is_eligible);
```

For different patients, we can pass different lists which may contain different elements. The SCL list PATIENT1_L might refer to a female patient and contain different elements than SCL lists PATIENT2_L and PATIENT3_L which refer to male patients.

No harm is done if the list includes additional information not used by the subroutine. Because a variety of subroutines will probably need some patient data, we could maintain one list for each patient containing all attributes of that patient. Such a list could be used by any method needing data about that patient:

```
call method ('Patient','evaluate',
            patient1_l, evaluation);
```

Encapsulation

I have been emphasizing the importance of separating internal implementations from external interfaces. Code is hidden within subroutines, and data is hidden within data list parameters. Both are intimately associated: changes to data and code occur together. So rather than considering code to be the internals of a subroutine, let's consider the internal contents of the data list and the code of associated subroutines to be part of the same thing: an *object*. In a shift of perspective, instead of a call invoking a subroutine and passing it data parameters, a *message* is *sent* to an object which contains both the method code and the data it uses:

```
call send (patient1, 'eligible', is_eligible);
```

We no longer need to specify the SCL entry containing the method code because this information is internally associated with the target object, PATIENT1.

Once this shift of perspective or *paradigm* has occurred, we can modularize systems in terms of objects rather than data alone or

function alone. Relational database design decomposes systems in terms of data, without regard for the function using that data. Structured analysis methods use functional decomposition. Object-oriented analysis recognizes that data and function are bound up or *encapsulated* together.

Association

It is often the case that one object cannot fulfill its *responsibilities* without the assistance of other objects. Objects can contain references pointing to other associated objects, and can *collaborate with* these other objects by sending them messages. For example, a Patient object may be associated with several Visit objects.

Polymorphism

Now we can discuss method code as part of the internals of an object rather than standing on its own. I mentioned before that the data attributes (*instance variables*) contained within one object could be different from another. In the same way the code executed when a given message is received can differ from one object to another. For example, the ELIGIBLE method could execute entirely different code for male patients than for female patients, even though the call looks exactly the same.

This crucial feature is called *polymorphism* in object-oriented jargon. Polymorphism makes *frameworks* possible. Conventional subroutines are typically used as standard *components*: standard structures that are used by applications as generic building blocks. Frameworks are exactly the opposite: standard structures which call application-specific modules to implement the details. For example, the SAS/AF® FRAME entry processor is a framework which executes the user-specified INIT, MAIN, and TERM labels. Object-oriented programming provides both framework and component capabilities.

In conventional programming, the conditional programming statements would be used to specify exceptions and variations in processing:

```
case (STATE);
  when ('IL') then do; ... end;
  when ('MI') then do; ... end;
  otherwise      do; ... end;
end;
```

As more variations are introduced, the CASE statement must be further extended. Each time the code is modified there is the chance of an error damaging the existing code, so the method must be retested.

Object-oriented programming offers an alternative: different implementations of a method for different objects (or classes of objects). Because changes to one implementation cannot damage other separate implementations, less code needs to be retested. Similarly, adding another case such as another STATE value can be accomplished without disturbing any previously tested code.

Classification

For the sake of sanity (and easy code maintenance), SCL and most other object-oriented languages group objects into *classes*. Each object is referred to as an *instance* of the appropriate class and can execute all methods defined for that class. Classes can be defined by the user in addition to those provided with the SAS System. The "class" concept recognizes that systems typically have many objects with similar properties.

Inheritance

It may have occurred to you that if you had alternative versions of a method, there would probably be substantial amounts of code duplicated between one version and another. Duplicate code is bad not only because it requires extra maintenance and testing but also because it introduces the risk of discrepancies between "identical" blocks of code.

Inheritance features permit objects to be instances of several classes. The classes are organized in a hierarchy of *superclass-subclass* relationships: methods added or changed in the superclass are automatically inherited by any subclasses.

If a subclass has more than one superclass, this is called *multiple inheritance*. Multiple inheritance is not supported by SAS, but can be simulated by manually specifying that messages received by one class be forwarded to another.

Although inheritance is one of the most distinctive features of object-oriented programming, the current consensus in the field is that inheritance should be used in moderation.

SAS Implementation

Let's review the object-oriented concepts in terms of their implementation in SAS. Objects are implemented as SCL lists containing data and a link to one class definition. Each class definition maintains a list of supported methods with the location of the source code corresponding to each method. Each class definition (except for the root object SASHELP.FSP.OBJECT) also contains a pointer to one parent class. If a method sent to an instance of a class is undefined for that class, the ancestors of that class are searched until the method is found.

SAS supports both *Widget* objects, which are displayed on SAS/AF FRAME entries during interactive applications, and data objects which function as non-display data structures.

EXAMPLE: INTERACTIVE DATA EDITOR

Now let's look at an example currently under development at Trilogy Consulting Corporation: an interactive data editor that identifies potential data problems and allows their correction.

Traditional "data cleaning" programs require repeated "cleaning passes" to print edit reports. Sometimes changes in response to these reports introduce or reveal additional problems, so the process is repeated until no problems remain.

The Interactive Data Editor (IDE), by contrast, maintains continuous records of the status of each edit. It gives immediate feedback in response to a change, indicating which problems have been resolved and which new problems have been introduced.

Some basic terminology used in the IDE specifications:

- **Respondent:** a source of information (typically a person or organization).
- **Variable:** a question to be asked of Respondents.
- **Form:** a set of Variables to be processed together.
- **Document:** an instance of a Form returned from a Respondent.
- **Field:** an instance of a Variable, part of a Document.
- **Edit Specification:** a rule estimating whether a Field or group of Fields is likely to contain a data error.
- **Edit:** an instance of an Edit Specification.
- **Edit Override:** A manually-entered exception specifying that the edit does not apply in this instance.
- **Flag:** an attribute of a Field summarizing the overall likelihood of a data error in that Field, based upon the Edits associated with that Field.
- **Flag Override:** A manually-entered exception forcing a particular Flag value.
- **Constraint:** a rule that always must be satisfied, thereby limiting the updates that can be made.

Actors

Four classes of users or *actors* will use the IDE system:

- **Specification Writer**
 - Adds, deletes, and modifies Edit Specifications.
- **Data Entry Clerk**
 - Enters Documents into the system.
- **Data User**
 - Reads Document Fields.
 - Reads Document Edits.
- **Problem Resolver**
 - Locates Fields with error Flags.
 - Updates data Fields.
 - Overrides Edits and Flags.
 - Checks updated results.

Scenario

A typical *scenario* or sequence of events might be:

1. Specification Writer defines Forms, Variables, and Edit Specifications.
2. Data Entry Clerk enters the Field data for each Document.
3. Initial Edit and Flag values are derived from the Field data.
4. Data User reads Field values (with Flags).
5. Problem Resolver
 - a. Locates Fields flagged in error.
 - b. Updates Field values, Overrides Field Flags, Overrides Edits.
 - c. repeats processes 5a and 5b.

OBJECT-ORIENTED ARCHITECTURE

There isn't sufficient space to review the entire Interactive Data Editor design, so I will highlight selected points of interest from an object-oriented perspective.

Document-centric Processing

In the early days of interactive applications, only one program could be run at a time. The application would be opened first, and then the data to be processed would be identified. Only one set of data (e.g. word processing documents, charts, spreadsheets) could be processed at a time. Later enhancements permitted two or more sets of data to be swapped or simultaneously processed.

More advanced operating systems (such as Microsoft® Windows™ or OS/2® from IBM®) allow the user to select the data first, and then pick an application from a context-sensitive menu of applications appropriate for that data. The operating system itself provides multitasking capabilities, so each data object can be processed independently. Multitasking capabilities for SAS/AF are provided by the AFAPPLICATION command (SAS Institute, 1991:4).

To be compatible with this approach, the IDE design requires each instance of the Document class to be able to open an interactive application (Viewer) with which users can view and edit the contents.

Client-Server Relationships

Given a complex system to design, one of the first tasks is to chop it into more manageable chunks. In order to control complexity as I described before, we want to construct modules that are highly cohesive internally yet only loosely coupled to each other.

A "client-server" metaphor is useful in this regard (I do not mean to imply any particular hardware configuration by this term). Often one system can be interpreted as providing services to other systems. The relationship is asymmetrical: the client module knows about and needs the server module, but the server module is independent of the client module.

The server module responds to requests by client modules, but does not directly initiate communications with clients. Instead, servers are only responsible for *broadcasting* announcements of server *events* to all subscribing clients. It is the client's responsibility to respond appropriately when a broadcast is *received*.

One of the first decisions made during the design on the IDE was to separate the user interface Viewer which displays the data from the Document class which manages the data. The Viewer has a client relationship to the Document (regarding a data management *contract*). This separation:

- allows the Document to function independently or serve other classes of clients,
- allows Viewer specifications to be changed without disturbing the Document code, and
- allows for future support of multiple or alternative Viewer components.

You might recognize this as the Model-View-Controller paradigm of Smalltalk (Goldberg, 1990). The IDE Viewer class combines the functions of both the View (user display) and Controller (user input).

Each class of objects in the system has assigned responsibilities. The responsibilities of the different classes of human actors were described above. To carry out those responsibilities, a class may use or collaborate with other classes. In the case of the Interactive Data Editor, the human actors fulfill their responsibilities by collaborating with one or more Viewers. Each Viewer, in turn, fulfills its responsibilities by collaborating with a Document.

All data management responsibilities supported by the Viewer class are *delegated* to the Document class by forwarding requests and responses. This serves to isolate the internal details of one subsystem from another.

Within the IDE, the following classes have been assigned the responsibilities noted:

- Viewer
 - Reads Field data (using Document).
 - Updates Fields (using Document).
 - Reads Field flags (using Document).
 - Overrides Edits (using Document)
 - Locates flagged Fields (using Document).
 - Responds to changes in the associated Document.

□ Document

- Reads Field data.
- Updates Fields.
- Reads Field flags.
- Overrides Edits.
- Locates flagged Fields.
- Broadcasts notice of changes.

At this point you are probably thinking that this is a tremendous volume of messages to be passing back and forth, and it is. The object-oriented paradigm is intended for a world in which computer power is much cheaper (and less vulnerable to error) than human power; the advance of technology continues to move us in that direction. Object-oriented applications are organized into many small independent pieces, improving ease of comprehension, debugging, maintenance, and reuse at the potential cost of computer efficiency. Sometimes object-oriented solutions are surprisingly efficient, but this is not the primary goal.

Composition

Objects may be composed of other objects in a *PART_OF* relationship. This is implemented by having one object contain pointers to other objects. The *PART_OF* relationship differs from ordinary association relationships because certain methods (such as Create, Delete, and Move) are transitively applied to all parts (sub-objects).

In the IDE example, Fields are part of Documents. This relationship generally is mirrored by a relationship between Widgets and Frames in the user interface. Most communication usually travels between a given Widget and the corresponding Field. But rather than the two communicating directly, I am having the Widget communicate with the Viewer, the Viewer communicate with the Document, and the Document communicate with the Field. This decision was made to preserve the internal independence of the Viewer and Document, so that Fields need not be concerned with display details such as extended table handling.

A typical scenario might be:

- 1) A User modifies a Widget on the screen, and the Widget sends a message to the Viewer:
`call send (Viewer, 'widget_changed',
self, newvalue);`
- 2) The Viewer determines the appropriate field name, and sends a message to the Document:
`call send (Document, 'change_field',
fieldname, newvalue);`
- 3) The Document delegates the change to the corresponding Field object:
`call send (Field, 'change', newvalue);`

Frameworks

Even though there are many different subclasses of Field

associated with many subclasses of edits, we can write a general Change method that can apply to any Field:

1) Validate

Validate update against any constraints. If the update is invalid, return with error message.

2) Read

Read existing value. If new value is the same as the existing value, then return (no message).

3) Update

Update field.

4) Propagate

Instruct all derived fields depending upon this value (such as Edits) to Refresh themselves.

For each subclass of Field, we must write implementations of the Validate, Read, Update, and Propagate methods.

Any modification of Change handling which applies to all subclasses of Field can be made in this centralized framework method.

Triggers

In database jargon, a *trigger* specifies an action that takes place automatically when another action is invoked, such as an edit flag being reevaluated when the data field changes.

In the IDE, whenever a Field is modified, it sends a Refresh message to all Edits dependent on the Field. These Edits then update themselves based upon their stored Update methods.

This technique can be used for other types of derived values as well, such as generated sums. Because derived values may be based upon other derived values, triggered messages may ripple through several layers of objects.

Deferred Derived Fields

Triggers are nice, but often imply redundant processing. Deferred derived values can be much more efficient, and are completely interchangeable with triggers (that is, their external interfaces are identical).

Suppose we have a derived sum based upon 10 fields. When the fields are first entered, 10 different messages will instruct the derived sum to refresh itself, even though no one will read the derived sum until later. When I developed a similar data editor² in 1986, elimination of these duplicate messages was a major source of difficulty.

Later I read an article by Rizwan Mithani (1994) which led me to realize that derived fields do not need to be recomputed until there is a request to read the field. This is easily implemented

using objects. The Refresh message sets an internal marker indicating that the value will need recomputing. When a Read request is received, the value is recomputed and stored, then the marker is cleared. Subsequent Read requests simply retrieve the stored value. Using this design, derived values are computed no more often than they need to be.

Object Databases

Of course we would like to be able to save our Documents and continue processing on another day. Unfortunately, SAS does not yet have an Object DBMS. Nevertheless, there are two other ways to store and retrieve objects:

- Objects can be stored as SCL lists.

SAS objects are implemented as SCL lists. These lists can be stored as SLIST catalog entries, and later retrieved.

The tricky part is that each object is linked to its class definition and possibly to other objects. If we want to allow the class definition to be modified while the object is in storage, we need to separate the object from its class definition and reattach it when we retrieve the object.

Similarly, if we want to be able to store and retrieve each object separately, we need to change embedded object pointers into object names, and change them back to object pointers on retrieval. This could be done by defining a Store method for each object which would store the target object and instruct each part object to store itself. A Library Manager would determine whether an object has already been stored. Retrieval would work in an analogous manner.

- Objects can serve as a *wrapper* around SAS data sets.

Remember that the implementation of a method is hidden within an encapsulated object. We have been assuming that the data for each object is stored on the `_SELF_` list within that object. But we could construct the various methods so that they read and write from a SAS data set observation (or more than one) rather than from an SCL list. If necessary, an object could be based upon observations from several data sets.

This approach seems preferable for a Data Editor because the data would be maintained in the form of a SAS data set which could be read by standard SAS procedures.

The availability of two strong alternatives suggests that we should isolate (that is, encapsulate) code and data structures dependent upon this decision from the rest of the system, so it could easily be changed later. In the IDE, details of Document storage and retrieval are encapsulated within a Document Library class which checks-out and checks-in Document objects.

The Document_Library class is also responsible for locating desired documents based on specified criteria. This is how documents with unresolved edit flags are located, as well as retrieving documents based on external identifiers.

One of the nicest features of object-oriented programming is that different components and frameworks can be independently developed. For example, development of the Document Editor can begin without knowing how the data values will be permanently stored. Object-oriented programming is ideal for incremental development and prototyping.

Future Enhancements

One enhancement that is relatively easy to add is an audit trail. Only one class of objects is affected: Value. Whenever a value is updated, the previous value could be added to a history list. No changes to the external interface of Value would be needed.

The Value class (with audit trail) could then be modified to respond to the Read method by returning the value as of any specified date. This would enable the entire system to appear as it did on any date in the past simply by changing a virtual date setting. This could be useful in interpreting actions taken in the past.

A more difficult enhancement is support for cross-Documents edits. The Edit objects would not belong to any one document, but would be associated with several documents. Each Edit could not be evaluated until all of the documents that it used were available.

CONCLUSION

Not all object-oriented systems are this complex. In fact, many are quite simple. If you have a relatively simple problem, object-oriented techniques can make it even easier to understand, debug, and maintain. If you have a complicated problem, then although "OO" isn't the silver bullet claimed by the hype machines, it could help you divide and conquer.

ENDNOTES

1. Screen Control Language can be executed in the batch mode of SAS using PROC DISPLAY (Norton, 1991) or the DMSBATCH system option introduced in release 6.09 (SAS Institute, 1993).
2. In 1985 to 1988, I developed a similar data editor as part of the Integrated Post-secondary Education Data System (IPEDS) for the National Center for Education Statistics (NCES). This system was developed by Derek Drummond, Brenda Matthews, and myself while at ORI, Inc., using the Model 204 database management system.

REFERENCES

Goldberg, Adele (1990), "Information models, views, and controllers: the key to reusability in Smalltalk-80 lies within MVC," *Dr. Dobbs Journal*, Volume 15 number 7 (July 1990).

Mithani, Rizwan (1994), "Modeling databases with objects and rules: a new dimension for distributed processing," *Object Magazine*, Volume 3 Number 5 (January, 1994), page 59.

Norton, Andrew A. (1991), "Screen Control Language versus Macros in Batch Environments," pages 1325-1330 in *Proceedings of the Sixteenth Annual SAS* Users Group International Conference*. Cary, NC: SAS Institute, Inc. Reprinted in SAS Institute (1994), *SAS*Macro Facility Tips and Techniques, Version 6, First Edition*, pages 106-111. Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1991), *SAS* Technical Report P-216, SAS/AF* Software, SAS/FSP* Software, and SAS* Screen Control Language: Changes and Enhancements, Release 6.07*. Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1993), *SAS* Technical Report P-252, SAS* Software: Changes and Enhancements, Release 6.09*, pages 2-3. Cary, NC: SAS Institute, Inc.

ANNOTATED BIBLIOGRAPHY

I have found these references especially useful. The SAS bibliography emphasizes on references which discuss SCL data objects rather than just SAS/AF Frame widgets.

Object-Oriented Analysis

Taylor, David A. (1990), *Object-Oriented Technology: A Manager's Guide*. Reading, MA: Addison-Wesley.

This very concise book presents an accurate yet highly readable overview of the basic principles of object-oriented programming. I recommend it as the starting point for anyone interested in the topic.

Yourdon, Edward (1994), *Object-Oriented Systems Design: An Integrated Approach*. Englewood Cliffs, NJ: Yourdon Press.

The structured-programming guru reviews management issues associated with introducing object-oriented technology into an organization, and summarizes the common features of the leading Object-Oriented Analysis methodologies. I like this book much better than his earlier books on object-oriented analysis.

Wirfs-Brock, Rebecca, Brian Wilkerson and Lauren Wiener (1990). *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.

This is the only book devoted to my favorite object-oriented analysis method, known as CRC (Class-Responsibility-Collaborator) or Responsibility Driven Design. CRC emphasizes the allocation of responsibilities to classes and the definition of client-server relationships ("collaborations") between classes.

Object Magazine. SIGS Publications, New York.

SIGS publishes *Object Magazine*, *Journal of Object-Oriented Programming*, *C++ Report*, *The Smalltalk Report*, and *Report on Object Analysis and Design (ROAD)*. *Object Magazine* is the most general and accessible.

Object-Oriented User Interfaces

IBM® Corporation (1992). *Object-Oriented Interface Design: IBM Common User Access Guidelines*. Carmel, IN: Que.

Common User Access (CUA) is IBM's standard for user interfaces, as exemplified by OS/2®. Even if you aren't using OS/2, this book is valuable for the excellent discussion of object-oriented user interface design principles. A new revision of CUA is under development.

Microsoft® Corporation (1992). *The Windows™ Interface: An Application Design Guide*. Redmond, WA: Microsoft Press.

Users are often more aware of the user interface than of the underlying implementation. So if you implement a SAS application for use on Windows 3.1, you might want it to look like other Windows applications. This is the primary reference.

SAS: Object-Oriented SCL

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

Chapters 7 ("Creating Customized Classes and Methods"), 9 ("SCL Functions to Support Object-Oriented Programming"), 10 ("The CLASS Entry"), and 14 ("The Object Class") form the primary reference for object-oriented programming in Screen Control Language.

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

Chapters 5 ("Using Subclasses and Methods") and 6 ("Communicating Between Objects") focus on widget objects within SAS/AF FRAME entries, but also serve as a useful supplement for the topic of SCL data objects.

Biesack, David J. (1993), "Object-Oriented User Interface (OOGUI) Using FRAME Entries in SAS/AF® Software: Part IV," pages 1404-1411 in *Proceedings of the Eighteenth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Source code for an appointment calendar application using two classes of data objects (called "utility classes").

Pierce, Randy (1994, forthcoming), "The Benefits of Object-Oriented Application Development Using the SAS® System," in *Proceedings of the Nineteenth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Covers essentially the same ground as the "What's So Great About OO?" section of this paper, with an approach more explicitly connected to traditional SAS programming techniques. Includes a detailed glossary.

SAS Institute (1993), *Building SCL Applications Using FRAME Entries Course Notes*. Cary, NC: SAS Institute, Inc.

Although this book has little to say about user-defined objects, it is indispensable as an orientation to SAS/AF FRAME Entry programming.

ACKNOWLEDGEMENTS

Thanks to Peter Lord and Grant Blank for reviewing earlier drafts of this paper, and to Derek Drummond and Brenda Matthews for their contributions to the design and development of the IPEDS data editor.

NOTES

An updated version of this paper is scheduled to be presented at the 1995 SAS Users Group International conference in Orlando, Florida, as part of the Advanced Tutorials section.

SAS, SAS/AF, and SAS/FSP are registered trademarks of SAS Institute, Inc. IBM and OS/2 are registered trademarks of International Business Machines Corporation. ® 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
Trilogy Consulting Corporation
5148 Lovers Lane
Kalamazoo, MI 49002
(616) 344-4100 voice
(616) 344-6849 fax
Internet: 76350.1604@compuserve.com