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 in place in releases 6.08 and 6.10 of Screen Control Language (SCL). It is already feasible to develop object-oriented systems in SAS just as you might in C++ or Smalltalk.

Discussions of object-oriented programming in SAS often focus on small "toy" problems. Although this technique helps the novice to understand every aspect of the example, it gives a limited and artificial view 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 and changing 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, and object-oriented SCL can be helpful here as well.

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

Advantages of OO

Change is constant in the world of computer programming, because of improved understanding of the problem, changed requirements, debugging, and enhancements. Object-oriented programming helps you manage change, contributing to other advantages of object-oriented programming including control of complexity, reuse of code, and smooth transition from analysis to design:

Complexity: If you can change code easily, you can begin with a simple program and elaborate upon it in stages. This incremental development strategy is discussed further below.

Reuse: Reusing code in a different, unanticipated context is a form of change.

Transition from Analysis to Design: A well-established principle in software development is to separate analysis (determining what needs to be done) from design (determining how to do it). Object-oriented programming supports this principle by allowing design decisions to be postponed and revised.

This paper focuses on object-oriented strategies for managing change.

Incremental Development

Traditional "waterfall" methodologies assume a predominantly one-way flow from analysis to design to implementation to testing. The analysis phase is completed before coding begins. This requires committing to important decisions early, when you have limited knowledge of their implications.

Incremental development (Cockburn, 1995) strategies begin by developing a simplified prototype version. This prototype is then incrementally enhanced based upon what has been learned so far. This process can be repeated or "iterated" many times: analyze-design-code-test, analyze-design-code-test.

Object-oriented programming is especially suited to incremental development (and vice-versa) because it is easy to make changes. Because the penalty for design mistakes is relatively small, you can tackle complicated projects without having to understand everything at the beginning.

Avoiding "Ripple Effects"

A small stone dropped into the water will affect the entire pond. In the same way, a small change in the specifications of a poorly designed system can require major efforts to implement. The problem is "ripple effects": local changes which require changes in their neighbors (which then require changes in their neighbors).

Laurence J. Peter (1986, page 155) says "The more complex the system, the more open it is to total breakdown." My definition of total breakdown is a system which cannot be fixed because any change would break something else.

To avoid "total breakdown", systems can be designed to be receptive to change. One way to do this is to construct systems out of modular sections which can be changed without adversely affecting other modules. Such systems are also easier to
Subroutines

The traditional way to "divide and conquer" computer programs has been through the use of subroutines (known in SAS as "methods" and "macros"). Subroutines let applications share code. There is only one copy of the code to maintain, which ensures consistency, saves effort, improves quality, and minimizes testing.

Subroutines also protect applications from changes in the subroutine code. When an application calls a subroutine, the two programs communicate through a limited and well-defined interface. There is only one copy of the code to maintain, which minimizes testing.

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

```
call method ('Driver', 'eligible', age, drivtrn, ok);
```

The first argument identifies the SCL entry containing the method code: DRIVER.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, AGE and DRIVTRN (driver training status), and one output parameter OK (eligibility for a driver's license). Because subroutines contain no persistent data, we know that the result OK is solely determined by the ELIGIBLE algorithm using the parameters AGE and DRIVTRN.

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 the 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. This is a disaster!

Suppose driver license policy changes: High school dropouts are ineligible for a license until age 18. The ELIGIBLE subroutine now requires another parameter: HIGHSCHL (dropout status). Every program calling the ELIGIBLE subroutine will need to be modified to read:

```
call method ('Driver', 'eligible', age, drivtrn, h highschl, ok);
```

Some of the programs that call the ELIGIBLE program may themselves need to be modified to obtain the HIGHSCHL information.

Unfortunately, subroutine signatures often need to be modified to handle new situations or make additional distinctions. If a modified subroutine requires additional data, the signature will need to be changed accordingly. Remember: Exposed data details lead to ripple effects.

The Solution

We need a way to pass additional information to a subroutine without modifying the signature. This can be done if a parameter contains 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 data elements are needed by the subroutine, just add them to the list. For example, assuming an SCL list DRIVER1 contains elements named SEX, AGE, and COUNTRY for a certain automobile driver, we could have:

```
call method ('Driver', 'eligible', driver1, ok);
```

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

```
call method ('Driver', 'eligible', driver1, result);
```

The SCL lists for different drivers may contain different elements. The SCL list DRIVER1 might refer to a suspended driver and contain different elements than SCL lists DRIVER2 and DRIVER3 which refer to drivers with active licenses.

Encapsulation

I have been emphasizing the importance of separating internal implementations from external interfaces. Code is hidden within a subroutine, and data parameters are hidden within a data list. Both are intimately related: Changes to data and code occur together. So instead of considering code to be the internals of a subroutine, let's consider the internal contents of the data list and the code of its associated subroutines to be part of the same thing: an object. In a shift of perspective, instead of passing data parameters into a subroutine, a message is sent to an object.
Because the object Driver1 contains its own data, we do not need to specify input parameters. Because the object contains its own code, we do not need to specify the SCL entry containing the method code. We only specify the name of the method — the most minimal interface possible, and the least susceptible to change.

Once we have shifted our perspective to the object-oriented viewpoint, 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 uses of that data. Structured analysis methods use functional decomposition. Object-oriented analysis recognizes that data and function are bound up or encapsulated together.

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 automobile drivers from different states, even though the call looks the same.

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

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

As more variations are introduced, the SELECT 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. Because changes to one implementation cannot damage other separate implementations, less code needs to be retested. OO lets you extend the system (such as adding another value of STATE) without disturbing any previously tested code.

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 that 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.

Classification

Systems typically have many objects sharing the same rules of behavior. For the sake of sanity (and easy code maintenance), SCL and most other object-oriented languages group objects sharing the same methods into classes. Each object is referred to as an instance of the appropriate class. Objects are attached to methods by class definitions, so each object can execute all methods defined for its class. The SAS System provides a variety of predefined classes, and lets programmers create customized classes.

Inheritance

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

Inheritance features let you extend an existing class with new or modified methods. Classes are organized in a hierarchy of superclass-subclass relationships: Methods added or changed in the superclass are automatically inherited by any subclasses. Inheritance lets classes share code in an orderly fashion.

If a subclass inherits from more than one superclass, this is called multiple inheritance. Multiple inheritance is not supported by SAS, but can be simulated to some degree by manually forwarding messages from one class 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.

During the execution of a method, the automatic SCL variable _SELF_ points to the target object.

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. Documents are not available for general use until all problems have been resolved.

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. All documents are always available to everyone, with problems flagged.

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**: the application of an Edit Specification to a Document.
- **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.
- **Problem Resolver**
  - Locates Fields with error Flags.
  - Updates data Fields.
  - Overrides Edits and Flags.
  - Checks updated results.
- **Data User**
  - Reads Document Fields.
  - Reads Document Edits.

**Scenario**

A typical scenario or sequence of events might be:

2. Data Entry Clerk enters the Field data for each Document.
3. Problem Resolver
   a. Locates Fields flagged in error.
   b. Updates Field values,
      Overrides Field Flags,
      Overrides Edits.
   c. repeats processes 3a and 3b.
4. Data User reads Field values (with Flags).

**OBJECT-ORIENTED ARCHITECTURE**

The Interactive Data Editor example will help us explore some of the common patterns of cooperation and interaction between objects.

**Document-centered 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 object first, and then choose 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 partition it into more manageable modules. Often, one module (the server) can be interpreted as providing services to other modules (the clients). Although the term "client-server" has
come to imply a particular hardware configuration, this need not be the case.] The relationship is asymmetrical: the client module knows about and needs the server module, but the server module is independent of the client module.

One of the first decisions made during the design of the IDE was to separate the user interface Viewer which displays the data from the Document class which manages the data. Each Viewer is a client of a Document which provides it with data management services. 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;
- allows for future support of multiple or alternative Viewer components, and
- eases maintenance, coding, and testing by partitioning a complex system into smaller simpler components.

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.

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. Viewers do not work with the data directly; rather, they fulfill their responsibilities by collaborating with their associated Document.

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

- **Viewer**
  - Presents Field data (using Document).
  - Updates Fields (using Document).
  - Presents Field flags (using Document).
  - Overrides Edits (using Document).
  - Locates flagged Fields (using Document).

- **Document**
  - Reads Field data.
  - Updates Fields.
  - Reads Field flags.
  - Overrides Edits.
  - Locates flagged Fields.
  - Broadcasts notice of changes in the Document and its contents.

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 by having one object contain pointers to other objects. This **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 directly between a given Widget and the corresponding Field. But when the Widget is initialized, it obtains the address or **object id** of the appropriate Field by asking the Document to locate a certain Field Name:

```
call send (Document, 'get_field', `fieldname', `field_object');
```

The widget then remembers the object id returned by the parameter **FIELD_OBJECT**, for future reference.

**Frameworks**

Even though there are many different subclasses of Field associated with many subclasses of edits, we can write a general Update framework method that can apply to any Field. Changes in overall policy should be made here:

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**

   Notify all objects depending upon this value (such as Edits Flags) that the value has changed.

For each subclass of Field, we must write implementations of the Validate, Read, Update, and Propagate methods (or take the default). Changes in the behavior of a particular subclass should be made in the subclass methods.
Delegation

Every Document of the same Form shares the same editing specifications. Thus, it makes sense to manage the editing specifications as part of an Edit Specification object rather than for each Document separately. Responsibility for determining the Edit results for each document is delegated from the Document to the Edit Specification.

The Edit Specification object supports a method EVALUATE which requires a Document object as an input parameter. The result of the edit is a joint product of the Edit Specification and the Document which is being edited. The EVALUATE method obtains data from the specified document and then determines the result of the edit.

In the driver's license example, the rules for determining eligibility are applied to data from individual drivers, but the rules themselves only vary from one state to another. Thus, it makes sense to delegate the ELIGIBLE method from the DRIVER object to a STATE_DRIVER_POLICY class.

Lazy Computation

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 editor 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, the SAS System does not yet provide 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 component 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 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) objects dependent upon this decision from the rest of the system, so the manner of data storage can 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 criteria such as the ID of the respondent. This is also how documents containing flagged fields are located.

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 iterative prototyping.

Audit Trail

An audit trail feature (a list of changes to values with a datetime stamp) is easy to implement because only one class of objects is affected: Value. Whenever a value is updated, the previous value is stored on a history list. A new method GET_HISTORY_LIST is defined to provide access to previous values.

The Value class (with audit trail) can then be modified to respond to the Read method by returning the value as of any specified date. This enables the entire system to appear as it did on any date in the past simply by changing a virtual date setting. This could help the user understand actions taken in the past.
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 difficult problem concerned with complexity and change, then "OO" 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. Between 1985 and 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


ANNOTATED BIBLIOGRAPHY


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.


This exciting new book (a companion to the above) examines the relationship between business engineering and object-oriented software engineering. An excellent case study illustrates the use of my favorite object-oriented methodology, Responsibility-Driven Design. The book concludes with a thought-provoking analysis of the life cycle of object-oriented software.


This book is the original description of the Responsibility-Driven Design (RDD) methodology. RDD emphasizes the allocation of responsibilities to classes and the definition of client-server collaborations between classes.


This article introduces the concept of "Patterns", a systematic way of communicating good practices from one programmer to another, inspired by the work of architect Christopher Alexander.

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

This handbook of solutions to common problems in object-oriented design is the most useful computer book in my library. Twenty-three patterns of relationships between objects are presented in a standardized format which discusses the problem addressed, a recommended solution, and the context in which the solution is appropriate. Examples are in Smalltalk or C++ but for the most part are readily adaptable to SAS. The target audience is assumed to understand object-oriented fundamentals.

Object-Oriented User Interfaces


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.


If you want your SAS applications on Windows 3.1 to look like other Windows applications, this is the primary reference.

SAS: Object-Oriented SCL


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.


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.


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


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.


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 Grant Blank, David Biesack, Sally Goostrey, Tony Ettwein, Mark Olson, and Peter Lord 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 earlier version of this paper was presented at the 1994 Midwest SAS Users Group conference.

SAS and SAS/AF 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

194