Interapplication Communication through Object-Oriented Programming with SAS/AF® Software

Joe Carter, SAS® Institute Inc., Cary, NC
Randy Pierce, SAS Institute Inc., Cary, NC
Duane Ressler, SAS Institute Inc., Cary, NC

ABSTRACT

The introduction of object-oriented programming (OOP) features in SAS/AF software provides applications developers with enhanced capabilities for achieving greater communication between applications. Classes that are developed within an organization can be shared across the organization more effectively with the use of application tools.

This paper advocates utilizing OOP features to develop an application tool set that establishes a protocol for managing communications between classes and applications. The importance of assigning a person to administrate classes and applications is also emphasized.

INTRODUCTION

The advent of OOP features in SAS/AF software has allowed applications developers to develop their own CLASS entries. This opportunity has caused applications developers to seek to understand the advantages that OOP offers them over traditional programming approaches. OOP also presents its own dilemmas, such as what should be done when different applications' classes cannot communicate with each other.

The ability of applications to communicate with one another is known as *interapplication communication*. Interapplication communication is important when you have functionality that you want to share between applications. Such communication across applications has been limited in the past due to the difficulties in maintaining a consistent communication protocol between applications. For example, an application might not know how to access other applications' classes or have access rights to them. The concept of interapplication communication means that applications can communicate with each application's classes without worrying about the location of classes or access rights issues.

HOW OOP CAN ENHANCE INTERAPPLICATION COMMUNICATION

Advantages of using OOP can be shown in contrasting traditional and OOP methodologies for how applications handle reading, updating, and showing relationships between data.

Reading Data

In the past, applications could read data only from another application if they knew the names of librefs (and how to allocate them), data sets, and variable names. If related data was stored in multiple data sets, applications reading the data would have to know how to merge the data correctly. This situation required close coordination between developers of other applications that shared this data, and it also meant that changes to how another application worked could adversely affect any other application that tried to read its data. For example, if a data set in one application were renamed, all other applications reading that data would have to be updated.

Using OOP features, a given application can contain a set of classes that allocate librefs appropriately and access the correct data sets. Provided the data maintained by the application is stable, other applications can use these classes to read this data, even if a given application were to change its librefs or data set names. Since each application reading an application's data does so through classes maintained by the application, developers of the application can rest assured that an application's data is being accessed correctly.

Updating Data

Prior to the introduction of OOP, making SAS data sets available for read only was often preferred to allowing an application to update data maintained by a different application. There were fears that there was no way to guarantee that the updated data was correct and that there were few ways to provide error checking on the updated data. One compromise was to get the other application to write out a transaction data set that a different application could read. Some of the problems with using transaction data sets are that other applications do not really know that the updates have occurred or that lots of intermediate transaction data sets have to be created to allow each application to see what the other one has done or has requested.

With OOP, classes that read data can be extended to include methods that update the data. The methods that update the data can perform all kinds of error checking and validation. An application knows that all values came through its own set of methods and were validated by its own classes. The risk associated with providing update access to data sets may be reduced to the point where you can safely allow other applications to directly update data in your application.

Showing Relationships Between Data

Data in one application frequently needs to know how it is related to data in another application. This is usually done by storing a *key* value of another application on each record in the other application. A bi-directional link is established if both applications store the key value of the other on each record. Using these key values has its limitations when you consider a record that may be related to many applications, such as employee records. A great deal of overhead is required to have an employee record maintain key values for all applications that have data related to an employee. Storing all the key values in the employee record also implies a one-to-one relationship between an employee and a record in another application. Everything becomes more complicated when a given employee can be related to two or more records in another application. There is also the risk associated with storing so many key values across applications, as well as the maintenance burden of keeping these keys updated.

By using OOP techniques, you can develop a class that generalizes the relationship between any two pieces of data. The key values of the two pieces of data are kept in an *association object*. Neither piece of data needs to store the key value of the other. With this capability, users can relate any two pieces of data. Applications developers no longer need to store multiple key

values of related information. Another advantage to using OOP techniques is that every association between each piece of data is a two-way link, so that any piece of data can seek out associations between itself and also related to itself.

DEVELOPING AN APPLICATION TOOL SET

With the advantages of using OOP features to enhance the functionality of your applications, you can take several approaches in managing your classes. For example, you can:

* put all your CLASS entries in a single SAS catalog that acts as a central repository
* register each CLASS entry but allow the CLASS entries to be stored in any number of SAS catalogs
* store the CLASS entries in any number of SAS catalogs and not register them

The approach presented in this paper for resolving the issue of accessing classes is to utilize an Object-Oriented Application Tool Server (OOATS) system.

DEVELOPING AN OOATS SYSTEM

An OOATS system is a SAS/AF SCL application that provides capabilities for registering classes and providing services to client applications for accessing the registered classes. Applications communicate with OOATS through a gateway program, which may be an SCL entry or other program. OOATS acts as both a class manager and an application manager.

The OOATS system allows any number of applications to be developed and managed with its classes, methods, and services. As new classes and methods are developed, they are registered with OOATS. You can think of OOATS as being analogous to a special kind of public library where everyone contributes their own personal books and from then on accesses these books through the library, following a certain protocol for checking in and checking out the books. As with a public library, there is the need for standardization.

The OOATS system manages and provides access to all classes and methods under its domain through a communication *protocol*, which establishes how classes and applications are accessed. OOATS also provides a registry of all available classes and provides information on access to any of the available classes, on whatever host platform they reside. OOATS communicates with client applications, whether they are SAS data steps, procedure steps, host commands, or other application types.

Interpreting Queries

Once you have established access and communication between the classes, applications need to be able to interpret the results of queries sent to each other. Since each application can return data in any format it chooses, each application might need to document extensively how it interprets the values returned by the application classes.

With an OOATS system, however, an application can return an instance of some well-known generic class to the communicating application. For example, if a project management application returns an instance of the FILE class to an application attempting to communicate with it, the communicating application knows that it can print, mail, or edit the returned value. The communicating application can also send that returned instance to some other application for further processing. This interapplication communication works since both applications follow the protocol OOATS uses.

Handling Error Conditions

Another advantage of an OOATS system is having a common set of tools for handling error conditions. If each application has a different way of responding to error conditions, other applications cannot necessarily understand these error flags. If an application cannot understand a method sent to one of its classes and there is no standard protocol for returning this information to the communicating application, the error condition might cause a program halt. If every method can return an error code, each method should return a consistent one. If one method returns an error code in a string of `$40`, all methods should conform to this standard.

IMPORTANCE OF THE CLASS MANAGER

The introduction of an OOP methodology into an applications development environment can result in the development of classes that not only cannot communicate with each other but even produce deleterious effects from their interactions. One way to reduce these miscommunications between classes is to assign a person as the *class manager*.

Having multiple people work on an OOATS system can require a great deal of coordination. Because it is in the interest of an organization to minimize duplication of effort, one of the responsibilities of the class manager is to track class development efforts to prevent duplication of functionality in the classes. The class manager should seek consensus on which classes to incorporate into the OOATS system and enforce that classes managed by the OOATS system maintain a consistent protocol.

The role of the class manager is critical to the success of an OOATS environment. The class manager must understand the OOATS system thoroughly and also act as *palace guard* to protect the intellectual assets stored in the CLASS entries. Because the class manager should have a solid understanding of all classes and applications—and the interactions between them—the class manager should be involved in all activities that involve adding, removing, or updating any classes or applications within the OOATS system.

CONCLUSION

OOP features in SAS/AF software make tremendous capabilities available for creating class libraries and developing applications that utilize those classes. The challenge is to let these classes communicate with each other across applications. This paper introduced the concept of an OOATS system to help facilitate communication between these classes and applications. The role of a class manager was also highlighted as being crucial to the success of an OOATS system.

ACKNOWLEDGMENTS

SAS, SAS/AF, SAS/GRAPH, SAS/OR, and SAS/SHARE are trademarks or registered trademarks of SAS Institute Inc. in the USA and other countries.