

Persistent Storage of SCL Data Objects

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

INTRODUCTION

SAS® applications traditionally store data in data sets. Although data sets are easy to use and efficient, there are some disadvantages: there is a sharp distinction between programs and data, every observation must be the same, and there is no automated navigation between observations.

In SAS release 6.08 and above, Screen Control Language provides object-oriented capabilities, including both visually displayed objects (called "widgets") and nonvisual objects (which I call "SCL data objects"). The latter can be used to represent complex data structures in memory. Many object-oriented techniques from languages such as Smalltalk adapted for use in SCL, as discussed in my paper "Designing Complex Systems using SCL Data Objects" (Norton, 1995).

As documented, SCL objects disappear as soon as the SAS/AF® or SAS/FSP® application shuts down. This paper introduces methods of storing these data objects persistently, so they can be used as a means of data storage.

WHY STORE DATA IN OBJECTS?

Object-oriented programming follows three main principles: encapsulation, polymorphism, and inheritance. These principles provide a good outline for discussing the differences between objects and data set observations.

Encapsulation

The "encapsulation" principle of object-oriented programming specifies that objects can only be accessed through method calls. This insures that changes to either the method code or the representation of the data are transparent to applications using the object.

It is therefore possible to customize any read or write operation. For example, you can implement audit trails, integrity constraints, or cache summarized results for future use.

The data of an object can be stored in any way that can be accessed by SCL code. Some objects contain their own data on an associated list called `_SELF_`, while others make use of linked objects or data sets.

Polymorphism

SAS data sets are rectangular: every observation looks and is treated exactly the same. The "polymorphism" principle of object-oriented programming allows objects to be similar at one level of abstraction, yet different at another level of abstraction.

For example, in a pharmaceutical study we may collect different

information for women than for men, and there may be different criteria for eligibility to participate. Object-oriented programming lets us hide these differences within each object and treat all related objects the same.

Inheritance

Object-oriented programming allows the extension of existing structures without the possibility of disrupting what already works or even the need to see the internal implementation. This is called "inheritance" -- new structures inheriting the properties of existing structures.

Inheritance goes hand-in-hand with polymorphism. Polymorphism allows new subclasses to be used within existing programs, provided that the new subclasses support all the inherited properties of their superclasses. Inheritance insures that this provision is maintained: If the superclass is modified, the properties of all subclasses are automatically modified to match.

Inheritance allows classes to be modified without disrupting any subclasses. This capability is vital due to the essential asymmetry of reusable tool development -- although every user of your tool knows about the tool, you may not know about every user.

The three principles of object-oriented programming help you deal with complexity and change. Because object-oriented systems are compartmentalized and incrementally modifiable, you can add complexity gradually, without having to understand the entire system at once.

In traditional database programming, exceptional situations are visible at a high level, in the form of extra variables and IF-THEN-ELSE statements. In object-oriented programming, general programs address the general situation, and special cases are hidden within subclasses.

GOALS OF AN OBJECT LIBRARY

It is not immediately obvious what the features of an Object Library should look like. This is commonly the case in object-oriented programming; the paradigm gives you so much flexibility that determining what you want to do takes more effort than determining how to do it.

Transparency

SCL objects ordinarily go away at the end of the AF application. I am developing SCL objects that will be saved permanently. In line with the object-oriented philosophy of polymorphism, I would like this difference to be as transparent as possible. The user should not need to know whether the object is in memory or saved, just use it.

Comprehensiveness

There should be no restrictions on what type of objects can be stored. In particular, we should be able to handle arbitrary complexity of object linkages (multiple links to an object, components and associations, bidirectional links).

Support for Inheritance

We should be able to store a mixture of different types of objects into the same Object Library. This provides support for the inheritance principle -- if we can save a class (such as Patient) we must be able to do the same operations on a subclass of that class (such as Female Patient).

What am I leaving out?

The techniques discussed in this paper store only the data, not the classes (method definitions). When an object is retrieved, it is linked to the current class definition, not the class definition at the time the object was stored.

This may or may not be desirable. It allows you to update the algorithms, like you might with a conventional database program. But it is possible for an updated program to become incompatible with some stored objects.

OBJECT LIBRARY MODELS

In order to store and retrieve objects, we need to have some identifier assigned to each object. When we retrieve an object, we send the identifier to an Object Library, which returns the desired object.

This is a convenient model for persistent storage, because you can also have an object library of transient objects in memory. In this case, the object library maintains a list of object IDs in memory, without responsibility for storing the objects. From the outside, the behavior of transient object libraries works like a persistent object library.

Once I began work on this project, I realized that there were two competing models of how persistent storage should work:

- 1) Objects in memory and in storage are separate. Storage of an object makes a copy of the object. You can update the object in memory without committing the changes to the stored object. You can also retrieve a stored object multiple times into separate copies, or store an object into multiple locations.
- 2) There is only one active copy of any object. If an object already resides in memory, the Object Library simply returns the object ID. For a stored object, the Object Library reconstructs the object, then returns the object ID. The user need not be concerned with the location of an object.

The present design is partway between these two models. The first model was more obvious, but I like the way persistent

storage is transparent to applications in the second model so I am working on recasting the classes in this new light. Further information will be available in the revised version of this paper to be presented at the 1996 SAS Users Group International conference.

USING PERSISTENT OBJECTS

Object-oriented programming separates the interface (external appearance) from the implementation (internal), so let's begin with how the object classes are used.

Wrapper Objects

The first class I will discuss will be the Wrapper class, which is a special (and simple) case. Classes descended from the Wrapper class obtain their data solely from one observation of a data set. The class determines the data set and key variable, the object instance determines the key variable value.

To use the Wrapper class to build a class of your own, override the `_INIT_` method, specifying the values of the instance variables `WRAPPED_DATASET` and `KEY_VARIABLE`.

Now program whatever methods you wish, remembering that only data stored on the data set observation will be persistent. The methods `GETVARN`, `GETVARC`, `PUTVARN`, and `PUTVARC` are available for your convenience. Consider the following example:

```
GETPRICE: /* GET_PRICE method */
method
  out_price 8;
  call send(_self, 'getvarn',
            'cost', out_price);
endmethod;
```

When the object receives a `GET_PRICE` message, it sends a `GETVARN` message to itself to read the data set variable `COST` from the current observation into the output parameter `OUT_PRICE`.

To create an instance of the `HOUSE` class, you might use:

```
house_c = loadclass('house');
arg_l = makelist();
dummy = insertn(arg_l, 7, -1, 'id value');
house7_o = instance(house_c, arg_l);
ignored = dellist(arg_l);
```

The `ARG_L` list is used to pass the `ID_VALUE=7` parameter to the `_INIT_` method. The object `HOUSE7_O` will wrap the data set observation `ID=7`.

In release 6.11, we will be able to use parameters of the `_NEW_` method instead of an `ARG_L` list.

Transient Objects

I am using the term "transient object" to refer to objects which store their instance variable data on their `_SELF_` list, and consequently would be transient or temporary unless the list was saved.

Storage or retrieval of a transient object requires collaboration with an Object Library class which issues object index numbers for storage and creates objects for retrieval.

Storing an object MYHOUSE_O might look like:

```
call send (objlib_o, 'store_as', my_house_o,  
           'My House');
```

Retrieving the object stored as 'My House' might look like:

```
call send (objlib_o, 'retrieve_as', 'My House',  
           house1_o);
```

Some objects contain links to other objects. In my implementation (discussed below), the UNSWIZZLE and SWIZZLE methods are used to store and retrieve these linked objects properly, and must be customized for each class. For ease of use I defined a STOROBJ ("Storable Object") class containing default methods suitable for simple classes.

I also developed an Object Library class which maintains the correspondence between the stored object and retrieved object as described in the second model above. If you delete objects and create new objects, these new objects may reuse the same object ID. The RESET method specifies that new objects are being stored, regardless of any reused object IDs.

As you can see, making use of these classes is straightforward. Now let's look at how these tools were programmed.

IMPLEMENTATION: WRAPPER OBJECTS

The data of an object is encapsulated within that object, and can only be accessed from the outside via object methods. Because the data representation is protected, the data can be stored in any way the designer wishes. One alternative is to store the data on the SCL list `_SELF` associated with each object. On the other hand, data could be stored elsewhere, such as in a SAS data set. Objects which store their data externally are called "wrapper" objects, because an object-oriented shell wraps around a data set core. Wrapper objects are automatically persistent, because data sets persist from one SAS/AF session to another.

In order to make it easier to develop wrapper objects, I have developed a WRAPPER class. This class handles transactions with the data set, so that programmers using the class will be shielded from this complexity.

The Wrapper class opens up the wrapped data set (positioned at the specified observation) when the object is first created. The ID of this opened data set is saved for future use. The GETVARC, GETVARN, PUTVARC, and PUTVARN methods simply use the corresponding built-in SCL functions on this opened data set.

Objects derived from the Wrapper class are not limited to simply reading and writing from the data set. You can calculate values, convert units, check for proper authorization, or maintain an audit trail. Because every read and write of a value executes a customized method, you can program whatever you

want.

The primary complexity arises because SCL data access methods work from a buffer rather than directly from the observation on the disk. If the observation is updated directly (for example, through the FSEDIT window), the buffer will not be updated to match. So I have included a REFRESH method which rereads the observation into memory.

Similarly, if the buffer is updated, the observation on disk will not be updated automatically. So I have included a COMMIT method which uses the Update function to update the observation.

During the evolution of the WRAPPER class, it has become more powerful and elaborate. Placing this functionality in a class rather than repetitiously hardcoding eases code maintenance.

IMPLEMENTATION: TRANSIENT OBJECTS

Wrapper objects are easy, because all of the instance data is already stored in persistent form. The more general situation of transient objects is more difficult, because to retrieve an object you need to recreate the object, load it with data, and reconstruct its links with other objects. This method would work with wrapper objects as well, if you want to use a consistent approach.

How an object is represented in SAS

In SAS, SCL objects are represented as SCL lists. SCL objects are identified by an object ID, which is actually the SCL list identifier. When you have composite objects, the components are linked using the object IDs.

The object list contains not only the data for that object, but also an item named `_CLASS_` which points to the class definition for that object. Class definitions indicate the location of each supported method, and the parent class. Class definitions are themselves objects and thus are also stored as lists.

These lists exist only for the duration of the AF application. SAS provides a mechanism for storing and retrieving SCL lists as catalog entries. Although there is no documented or official way to store and retrieve objects persistently, we can achieve this result by converting objects to and from a list representation. The main tricks involve replacing the data contents of a new object with the stored data, and linking the reconstructed object to the current class definition.

Converting Objects to Lists

I will start with the simplest case: an object containing no pointers to other objects. An object differs from a regular SCL list in two ways:

- 1) It contains an entry `_CLASS_` pointing to the class definition, and
- 2) The list ID is registered as an object within SAS.

The class definition itself is represented as a list, registered as a class object within SAS.

When we retrieve an object, we want to connect it to the current class definition. The original pointer to the old class definition may be obsolete. In order to locate the current class definition (through the LOADCLASS function), we need the class name. So before we store the object list, we add an item `_CURRENT_` containing the name of the class:

```
UNSWIZL: /* unswizzle method */
method
  out_l      8
  in_objlib_o 8 /* object library */
;
out_l = copylist (_self_, 'y');
class_c = getnitemn (_self_, '_class_');
current = getnitemc (class_c, '_current_');
dummy = insertc (out_l, current, 1, '_current_');
endmethod;
```

The resulting list is then saved into a catalog, using:

```
ignored = savelist ('catalog', entryname,
  unswizzled_l);
```

For retrieval, first we get the list:

```
ignored = fillist ('catalog', entryname, loaded_l);
```

Then we remove the class name from the list, locate that class definition, and create an instance of that class:

```
current = popc (loaded_l);
class_c = loadclass (current);
out_o = instance (class_c);
```

Finally, we remove the items from the object list and replace them with the items stored (`_CLASS_` and `_DESC_` are not stored so they get their value from the new object):

```
do i = listlen (out_o) to 1 by -1;
  if nameitem (out_o, i) not in ('_CLASS_', '_DESC_')
    then ignored = delitem (out_o, i);
end;
dummy = copylist (loaded_l, 'y', out_o);
```

We have thus performed surgery on the new object and replaced it with the stored "brain" of the old object.

Reconstructing Object Patterns

Objects often contain pointers to other objects, representing either component objects or associated objects. If we were to store these pointers in their original form, they would be useless when the object was retrieved in the future, because the component objects would no longer be in the original memory location.

As we save each object, we must also save all of the connected objects. The object pointers in the saved object must be converted into stored object indexes. The established term for this in the object-oriented DBMS field is "swizzling". Cattell (1991, page 166) explains "...references between objects may be *swizzled* — that is, turned into physical pointers — when objects are stored in memory."

When we retrieve an object, we must retrieve all of the

connected objects, swizzling the stored object indexes back into pointers to the correct reconstructed objects.

The Storable Object class handles swizzling and unswizzling by calling methods named `SWIZZLE` and `UNSWIZZLE`. By default, these methods process only the current object. When you subclass the Storable Object class for a specific application, you must extend the `SWIZZLE` method to retrieve each linked object and convert the associated pointer. Similar extensions are required for the `UNSWIZZLE` method so that it stores each linked object.

For example, suppose we had a container object. Inside this object is a list of component objects. The `UNSWIZZLE` method would locate each component object and store that object, then change the pointer to the component's object index:

```
item_l = getniteml (out_unswizzled_l, 'item_l');
do i = 1 to listlen (item_l);
  component_o = getitemn (item_l, i);
  call send (in_objlib_o, 'store', component_o,
    objindex);
  dummy = setitemn (item_l, objindex, i);
end;
```

The `SWIZZLE` method reverses this process, retrieving each component and changing the pointer to the ID of the reconstructed object:

```
item_l = getniteml (_self_, 'item_l');
do i = 1 to listlen (item_l);
  objindex = getitemn (item_l, i);
  call send (in_objlib_o, 'retrieve', objindex,
    component_o);
  dummy = setitemn (item_l, component_o, i);
end;
```

Recursion Checking

Now we run into another problem. It is possible to have more than one pointer to the same object, such as multiple objects linked to a shared object. When we retrieve these objects, we want them to be linked together in the original topology, rather than having duplicated objects. If objects A and B both pointed to X, then we should not have reconstructed A pointing to X1 and B pointing to X2. Similar problems arise from bidirectional pointers: parent pointing to child and child to parent.

If we simply retrieved each object as it was requested, we would end up with duplicate objects in memory. Duplicate objects are bad because they lead to disconnected navigational paths and inconsistencies in updating.

Luckily this problem is easy to solve. If we have one object in memory, we want to have exactly one corresponding object in storage. So the `STORE` method keeps a list of every object it has stored, and where it was stored. Requests to store an object a second time are intercepted and the existing stored object index is returned.

```
recursion_l = getniteml (_self_, 'recursion_l');
out_objindex = getnitemn (recursion_l,
  put(in_o, 8.), 1, 1, 0);
if out_objindex gt 0 then do; /* already stored */
  return;
end;
call super (_self_, 'store', in_o, out_objindex);
```

```

dummy = insertn (recursion_l, out_objindex, -1,
                put(in_o, 8.));
endmethod;

```

Retrieval works the same way: if the object already exists in memory, it is not reconstructed again. Instead, the existing object ID is returned.

```

RETRIEVE:
method
  in_objindex 8
  out_o       8;
  recursion_l = getniteml (_self_,
                          'recursion_l');
  position = searchn (recursion_l, in_objindex);
  if position > 0 then do;
    /* already in memory */
    out_o = input( nameitem (recursion_l,
                             position) , 8.);
  return;
  end;
  call super (_self_, 'retrieve', in_objindex,
              out_o);
  dummy = insertn (recursion_l, in_objindex, -1,
                  put(out_o, 8.));
endmethod;

```

This approach works well unless you delete an object in memory. SAS might reuse that object ID for another different object. If you tried to save that new object, it would be interpreted as a duplicate and not saved. To work around this problem, I have provided a RESET method which simply clears the list RECURSION_L.

CONCLUSION

These methods work efficiently and can be packaged into reusable generalized classes.

You should remember, however, that the "swizzling" method is undocumented and based upon internal features of object-oriented SCL which could be changed in the future. Isolating the code dependent upon these internal features into encapsulated objects should ease any modifications required by future releases of SAS.

Given this caveat, I hope you will find, as I have, that these methods substantially increase the flexibility and scope of SCL data objects.

REFERENCES

Cattell, R.G.G. (1991), *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Reading, MA: Addison-Wesley Publishing Company.

Norton, Andrew A. (1995), "Designing Complex Systems using SCL Data Objects," forthcoming in *Proceedings of the Twentieth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute, Inc.

ACKNOWLEDGEMENTS

Thanks to John Ellis and Dov Rosenberg for reviewing an earlier draft of this paper.

NOTES

A revised version of this paper will be presented at the 1996 SAS Users Group International conference.

SAS, SAS/AF, and SAS/FSP are registered trademarks of SAS Institute, Inc. ® 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