The Singleton Class in Screen Control Language

Kevin Tyler Brown, Trilogy Consulting Corporation, Kalamazoo, MI

ABSTRACT

A class in object-oriented programming is an object which can create other objects. These are termed instances of the class. There is a particular type of class that controls the instantiation of its objects. That class is called the Singleton class.

A Singleton class permits only one instance to exist. For example, you might use a Singleton to create a print spooler. A system may have many printers, but only one print spooler.

SAS® has no built-in capabilities for creating Singleton classes. This paper will show you how you can create a Singleton class and how Singletons can help your SCL programming.

INTRODUCTION

"What's a Singleton and why would I ever need one?", is probably a good question to start with. A Singleton manages its own creation. It allows only one object to be created from it, and that object is the only instance that will exist at any given time. A Singleton is easily accessible anywhere in the program. Examples of a Singleton class could be: one teacher for a class of students, one print spooler for any number of printers, one window manager program in a system. If you have a group of objects that need a controller object, the controller is a singleton. You could be asking "Why haven't I used a singleton?" , but it is not explicitly provided in the SAS System. This paper will show you how to create and use Singletons to expand the capabilities of the SAS System.

CREATING THE SINGLETON

The following models show the progressive development of the Singleton class design.

A call to the Singleton class in its truest form and our desired goal would be:

```plaintext
   model_c = LoadClass ( 'model' );
   call send (model_c, '_new_', model_o);
```

This would create the object model_o, if it did not already exist.

MODEL # 1

In the first model we set up a stand alone method that performs all Singleton class functions. It allows single point access from anywhere in the program, storing the Singleton object ID into the environment list as a named item. A _NEW_ message to the Singleton class will first check if the object already exists. If it does not, it will create the object and add the object's ID to the environment list. The method will then pass the Singleton object ID back to the requesting client.

The limitations of this approach are the use of an external method, the use of the environment list, and an initial call at the beginning of the program to set up the Singleton class. Using this approach, subclasses do not have the Singleton property. This model may not be the cleanest, but it provides Singleton capabilities to SAS releases 6.08 and above.

MODEL # 2

Recent additions to the SAS programming language allow us to move the method function described above into the class definition. In 6.10, classes are objects and support methods of their own, including the _NEW_ method. The _NEW_ class method is executed before the _INIT_ and gives us a way to modify the class definition to control the creation of objects before they're created. It should be remembered that we are only affecting the _NEW_ in the class definition and not the _NEW_ method in the object.

The class methods can only be accessed or changed at runtime. Class methods can be changed using the _SETINSTANCE_METHOD_ method. The _NEW_ method of the class definition is changed dynamically at runtime.

```plaintext
call send (class_, '_set_instance_method_',
   '_new_', 'CLASS.CATALOG.DEF', '_new_');
```

Overriding the _NEW_ method with the _SETINSTANCE_METHOD_ happens once at run-time, at the beginning of the program and is the key to Singleton behavior. The revised _NEW_ code will check for a previous instantiation of the Singleton object. If a Singleton object exists, the Singleton
class will return the ID of that Singleton object. If no Singleton object exists, one will be created.

/* Get instance list */
call super (_, _get_instance_list_, instance_1);
/* Check validity of the list and singleton */
if listlen (instance_1) < 0 then
/* Valid and there is at least 1 singleton */
/* Return it's Object ID to the client */
out_instance_o = getitem (instance_1);
else
/* Create the first and only singleton */
call super (_, _new_, out_instance_o);

This model has more of an object-oriented design, but has a few open ends. The overriding the Singleton class _NEW_ method still happens at the beginning of the program. Also, the overriding is done outside the class, which is similar to Model #1 in that an external method is used to set up the first Singleton class. Although once these steps are accomplished, we have a true Singleton class. The creation of other Singleton classes also requires manual intervention because subclasses do not inherit the Singleton behavior. This missing piece hold us back from a true object-oriented program. This leads us to the next step, Model #3.

MODEL # 3
Model #3 takes Model #2 and modifies the _NEW_ method within a class-object relationship. This allows us to subclass,

```
.Singleton Class
          _______________  
           |         |   |         |
           v         v   v         v
.Singleton Object
          _______________  
           |         |   |         |
           v         v   v         v
            |   set_instance_method
            |                     |
            v   _NEW_          v
            |   get_instances_   |
            |___________________|
```

create on-demand, and utilize a total object-oriented approach to the Singleton class. This is accomplished by overriding the _INIT_ method of the object definition, illustrated by the diagram above. The code insert for the Singleton object is described below.

```
.NEW_

._NEW_ method
Has been altered

._NEW_

._INIT_

.optional = arg_1, 8;
call super (_, _init_, _arg_1);
_class_ = getitem (instance_1);

./ Overwrite the _NEW_ in the class def. */
call send (_, _set_instance_method_,
'new', 'libname.catname.code',
'new_');

endmethod;
```

When creating a Singleton object,

```
single_c = loadclass ('single');
call send (single_c, _new_, single_o);
```

the first call to the Singleton class would create a Singleton object and run the _INIT_ for that object. The _INIT_ in turn modifies the _NEW_ method of the Singleton class definition as in Model #2.

Model #3 now can be subclassed, called on-demand throughout the program and perform all tasks required by the construct of a Singleton. New Singletons can be created easily throughout the program.

MODEL # 4
One problem remains to be solved leading us to Model #4, which is the preferred method for SAS release 6.10 and above. In using the Singleton class as a controller class that manages and stores objects (see the paper by Andrew Norton, Persistent Storage of SCL Data Objects), a problem arose when all references to the Singleton were resolved, the Singleton object did not terminate. A direct call to _TERM_ was the only way to end the existence of the Singleton. This leads us to the final evolution, Model #4.

In this model, programs are responsible for sending a DELETE message to the Singleton when they are no longer needed. The object is not actually deleted until all users are finished. The final Singleton class incorporates a reference counter that tracks use of a Singleton object. When all references are released (reference count = 0) the Singleton is terminated. An analogy can be made to a file open() and a file close(). If you reference the Singleton then you should de-reference it when you’re finished. The same principle as: “If you’re the last out of a room, TURN OFF THE LIGHTS!”. The reference count instance variable is initiated in the object’s
_INIT_ method. The reference count instance variable is put on
the Singleton class _self_ list, and initialized to one. From this
time on, the class definition will control the reference count.

_self_ = _self_;  
_INIT_: 
    method
        optional = in_arg_1
        ;
        catentry = searchpath('single2.scl');
        call super (_self_, '_init_', in_arg_1);
        _class_ = getitemm (_self_, 'CLASS_');
        call send (_class_, 'set_instance_method_',
            'NEW', catentry, 'NEW_');
        call send (_class_, 'set_instance_method_',
            'DELETE', catentry, 'DELETE_');
        /* Set up the reference count here */
        rc = getitemm (_class_, 1, 'REF_CNT_');
        endmethod;

The changes in the Singleton code for the _NEW_ method are
slight. The counter is incremented when accessed and saved on
the _self_ list.

_NEW_: 
    method
        out_instance_o = 0
        ;
        out_instance_o = 0;
        call send (_self_, '_get_instance_list_',
            instance_1);
        if listlen (instance_1) gt 0 then do;
            out_instance_o = getitem (instance_1);
            /* Get the reference count off of the _self_
                list */
            Ref_cnt = getitemm (_self_, 'REF_CNT_');
            /* Add to the Ref_cnt */
            rc = setitemm (_self_, count+1,
                'REF_CNT_');
        end;
        else do;
            call super (_self_, 'NEW', out_instance_o);
            /* Reset the Ref_cnt */
            rc = setitemm (_self_, 1, 'REF_CNT_');
        end;
        endmethod;

The DELETE method decrements the count until all references
have been deleted. At that time it terminates the Singleton
object.

DELETE: 
    method
        Ref_cnt = getitemm (_self_, 'REF_CNT_');
        Ref_cnt = Ref_cnt - 1;
        rc = setitemm (_self_, count - 1, 'REF_CNT_');
        /* Do we terminate singleton yet */
        if ref_cnt gt 0 then
            return;
        call send (_self_, '_get_instance_list_',
            instance_1);
        /* First terminate the instance object */
        /* By definition there should only be one
            instance */
        if listlen (instance_1) gt 0 then
            call send (getitem (instance_1),
                'TERM_');
        endmethod;

Now when a client requests use of a Singleton object, the
Singleton class will create the instance if it does not exist, and
increment the reference count. When the client is finished, it
tells the Singleton class, that it is finished by sending a DELETE
message.

_INIT_: 
    single_c = loadclass ('single');
    call send (single_c, 'NEW', single1_o);
    call send (single_c, 'NEW', single2_o);
    call send (single_c, 'DELETE');
    call send (single_c, 'DELETE');
    return;

CONCLUSION
It is now as easy to use a Singleton class as any other class.
Completely self-contained, it can be put into a library of tools,
and used on demand. Object-oriented programming within the
SAS System benefits everybody in application development
trying to simplify programs, and Singleton classes play a key
role.

REFERENCES
Erich Gamma, Richard Helm, Ralph Johnson, and John
Vlissides. (1995), Design Patterns: Elements of Reusable Object-
Oriented Software, Reading MA: Addison Wesley Publishing
Company, 127-134.

SAS/AF Software Course Notes, Cary, NC: SAS Institute Inc.

Objects", Proceedings of the Twenty-First Annual SAS Users

ACKNOWLEDGMENTS
I would like to thank David Biessack of SAS Institute, who
conceived the original implementation of Model #2 within the
SAS System.
AUTHOR CONTACT

Kevin T. Brown
Object Technology Group
Trilogy Consulting Corporation
5148 Lovers Lane
Kalamazoo MI 49002
Phone: 616-344-4100
e-mail brown@net-link.net