Subject: Re: IDL Object for creating a Singleton
Posted by J.D. Smith on Mon, 22 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>
> Phillip David (pdavid@earthling.net) writes:
>
>> I've been doing a fair amount of reading on OO design and methodology. One of
>> the books I've had pointed out to me was 'Design Patterns', which deals with
>> OO algorithm design quite nicely. One of the patterns the authors describe is
>> a Singleton -- A class for which only one instance exists. If the sole
>> instance exists, the constructor function (object's init function in IDL) will
>> return a reference to the existing object. If none exists, a new object is
>> created and returned.
>>
>> This structure is particularly useful when there is no reason to have multiple
>> objects around. I am trying to create a user preferences object that fits
>> this criteria very nicely. I'd like to be able to store user preferences for
>> screen size in an object. If that object exists, other screens can size
>> themselves appropriately. The book gives sample C++ code that relies on class
>> variables and class methods (i.e., not associated with a specific instance of
>> the object, but with the object itself). IDL doesn't seem to have any way to
>> do something similar. Does anyone know of a way to do this?
>>
>> My first attempt uses a common block called preferencesclassvars to store an
>> object reference. This common block is called from a function that, while not
>> part of the object, is stored in the same file. If the reference contained in
>> the common block is not an object reference, then I know the object hasn't
>> been initialized, and I set it to a new instance of the object. Otherwise, I
>> return the object reference from the common block.
>>
>> Is there a better way to achieve the techniques of class variables and class methods?
>
> Humm. Objects are turning out to be not only to be fun and powerful to
> program, but synchronistic as well. This is the second time in two
> days the _Design Patterns_ book (which I have not seen or read) was
> mentioned to me and in an almost identical context. I was talking
> to a friend about writing a "lurker" object, a singleton object like
> Philip's screen preferences object, that lurks in the background
> waiting for something to happen. In this case, I wanted to
> update all the open windows when a color table was changed on a
> 24-bit display.
>
> This wonderful idea suffers from other unsolvable (so far)
> problems, but I did manage to get it started up. Like Philip,
> I had to resort to a common block to initialize it properly.
> Since common blocks are never my preferred solution, and since

> this thing reeks of a hack, I'm also looking for something more
> elegant. I'm afraid I am swimming in programming space that
> is a little out of my depth. Perhaps someone like JD can throw
> both Philip and me a life-line. :-)
>
> Cheers,

I think you've both run head-on into the limits of IDL's OO implementation. The foremost among these is the lack of controllable access. All class members are equivalent to "protected" -- visible to a class plus any derived classes. Especially frustrating is the absence of public data members, which requires one to set up GetProperty and SetProperty methods which do *nothing* other than access otherwise "public" data.

Protected class methods, although less frustrating, can limit the amount of cooperative programming possible for large projects (since no mechanism is provided to prevent accidental reliance on internal, and therefore potentially changing, class implementation).

But one must keep in mind that RSI wanted to provide the basic OO functionality, not a totally robust OO environment. I think they did an O.K. job with this. Polymorphism is well-implemented since all methods are "virtual protected", and all objects (read object references) are the equivalent of object pointers in C++ ... there is no such concept as static typing of a method (just as there is no concept of static typing of a variable) in IDL. Encapsulation, though rigid, accomplishes the basics of the principle.

In any case, as to your specific problem, I haven't seen the book you reference, but I'd bet they achieve the "singleton" functionality with a static data member -- a variable which is shared by all instances of the defining class, but only initialized once. There is no comparable functionality in IDL. I believe the only way to implement this *is* with a common block (uhgg ... but remember that XManager, everybody's favorite procedure, is implemented with a bunch of common blocks). The issue is then one of locality and longevity -- we must ensure nothing else alters our common block, and that, after the singleton's life is over, the common block doesn't contain data. The latter is easy enough, the former might be impossible.

I've been tinkering around with various ways to implement this type of structure for a while... unfortunately, you cannot affect what gets returned from Obj_New (more on this later). You could of course achieve the same functionality with other, uglier ways.

An implementation is below. It defines a super-class, Singleton, which is only meant to be inherited from. This class maintains a common block

variable "slist", in which are kept the various singleton objects. Only one list is needed for as many different types of Singleton's you may want, during a session, and, as you kill Singleton objects, the list is cleaned.

********** Singleton Abstract Class -- must be inherited
****

```
pro Singleton::Cleanup
  common Singleton, slist
  if ptr_valid(slist) then begin
    wh=where(NOT obj_isa(*slist, obj_class(self)),cnt)
    if cnt ne n_elements(*slist) then $
     if cnt eq 0 then ptr_free, slist else *slist=(*slist)[wh]
  endif
end

pro Singleton::Init, Object=obj
  common Singleton, slist

  ;; Add this type to our singleton list, if we need to.
  if ptr_valid(slist) then begin
    ;; clean up the list .. removing any danglers
    wh=where(obj_valid(*slist),cnt)
    if cnt eq 0 then begin
      ptr_free,slist
      slist=ptr_new([self])
    endif else begin
      *slist=(*slist)[wh]
      ;; find us on the list
      wh=where(obj_isa(*slist,obj_class(self)),cnt)
      if cnt eq 0 then begin
        *slist=[*slist,self] ;not yet on list -- add us
      endif else begin       ;we are already on the list !
        abort=self
        self=(*slist)[wh[0]]
        obj_destroy,abort
      endelse
    endelse
  endif else slist=ptr_new([self])
  obj=self
  return,1
end

pro Singleton__define
  struct={Singleton, $
          NULL:0b }           ;I am forced to include something in
                              ;this abstract class
end
```

```
 ********************************************************** ***************
```

And a User preferences object based on this ...

```
****************** UsrPref derived Class
*********************************
pro sUsrPref::PrintPrefs
  print,'XSIZE: ',self.Prefs.xsize
  print,'YSIZE: ',self.Prefs.ysize
  print,'CTABL: ',self.Prefs.ctabl

end

pro sUsrPref::SetPrefs, XSIZE=xsize, YSIZE=ysize, CTABL=ctabl
  if n_elements(xsize) ne 0 then self.Prefs.xsize=xsize
  if n_elements(ysize) ne 0 then self.Prefs.ysize=ysize
  if n_elements(ctabl) ne 0 then self.Prefs.ctabl=ctabl
end

pro sUsrPref__define
  struct={USRPREF, $          ;A Structure to hold the preferences
          XSIZE:0, $          ;x size of screen
          YSIZE:0, $          ;y size of screen
          CTABL:0}            ;preferred color table
  class={sUsrPref, $          ;the sUsrPref Class
          INHERITS Singleton,$ ;make it a singleton
          Prefs: {USRPREF}}    ;pointer to user pref structure
end
 ********************************************************** ***************
```

The problems with this are:

1. You have to put some data in the Simpleton class, which really needs no data (since it stores things in the common block). Required data members, egad.

2. You are forced to call the derived class as null=obj_new('sUsrPref',Object=sup), which will always yield a valid object "sup" (the same one each time), but a valid object "null" only once. If "self" were fully by-reference we could skip this awkward keyword. Alas.

3. A heap variable is created and then destroyed when one of this current type is already on the list. This is wasteful and slow. Another possiblity to avoid this is to make a function to use instead of

obj_new, call it singleton()... something like:

```
function singleton, oType, _EXTRA=e
  common Singleton, slist

  if ptr_valid(slist) then begin
    ;; clean up the list .. removing any danglers
    wh=where(obj_valid(*slist),cnt)
    if cnt eq 0 then begin
      ptr_free,slist
      obj=obj_new(oType,_EXTRA=e)
      slist=ptr_new([obj])
    endif else begin
      *slist=(*slist)[wh]
      ;; find us on the list
      wh=where(obj_isa(*slist,oType),cnt)
      if cnt eq 0 then begin
        obj=obj_new(oType,_EXTRA=e)
        *slist=[*slist,obj] ;not yet on list -- add us
      endif else begin       ;we are already on the list !
        obj=(*slist)[wh[0]]
      endelse
    endelse
  endif else begin
    obj=obj_new(oType,_EXTRA=e)
    slist=ptr_new([obj])
  endelse
  return,obj
end
```

This would make Singleton::Init unecessary (could remove it). Now you would simply say:

```
sup=singleton('sUsrPref')
```

To get a new/current instance of the preferences object. However, this is not a perfect replacement for obj_new() because only keyword parameters (through inheritence) are permitted in the initted object.

Anyway, if something better comes to me, I'll let you know.

JD

J.D. Smith |*| WORK: (607) 255-5842
Cornell University Dept. of Astronomy |*| (607) 255-4083
206 Space Sciences Bldg. |*| FAX: (607) 255-5875
Ithaca, NY 14853 |*|