Subject: Re: Object Madness or Restoring Nightmares
Posted by JD Smith on Thu, 04 Mar 2004 17:57:18 GMT
View Forum Message <> Reply to Message

On Wed, 03 Mar 2004 22:06:13 -0700, David Fanning wrote:

> David Fanning writes:
>
>> One of what I used to think of as the "advantages" of my
>> Catalyst Library is that it is an object hierarchy. If
>> you "draw" the top-level object, all the objects below
>> get "drawn" automatically. This means widgets appear,
>> images get drawn in windows, coordinate systems get set
>> up, etc. Neat.
>>
>> Similarly, if you "destroy" the top-level object, all the
>> objects below in the hierarchy get destroyed. No memory
>> leaks, no great effort involved. Very, very neat.
>
> Ah, here is the thing about this hierarchy that you
> should know. This is an object *containment* hierarchy.
> The top-level object is a container that holds all the
> other objects. Every object (except the top object) is
> both contained in a container and can (potentially)
> contain other objects. (All objects in my system
> inherit IDL_CONTAINER.)
>
> If you pick any object whatsoever out of this web,
> you can (apparently easily to judge from how fast
> IDL does it) traverse the entire object hierarchy.
> I can see that this is the reason IDL *must* save
> everything when I save even a single object that
> belongs in the hierarchy.
>
> What I can't see at the moment is a way out of
> this mess.

Why not implement a set of methods in your top-level which leverages the inherent connectedness to detach unnecessary objects before saving? The only technical problem is where to stick the detached objects while you save (you can't stick them somewhere else in the object: you'll be back to the same problem). This can be accomplished by dynamically building a list of objects and their detached components and propagating it all the way up to a variable at the top stack level at the time of the call. Something like this:

```
pro topClass::Detach, detachlist, RECORD=rec
  if obj_valid(self.parent) then begin
```

```
    ;; Add to or create this object's "detached" record
    if n_elements(rec) gt 0 && ptr_valid(rec.Detached) then $
       *rec.Detached=create_struct('parent',self.parent,*rec.Detach ed) $
    else rec={Object:self, Detached: ptr_new({parent: self.parent})}
    self.parent=obj_new()
  endif
  ;; Stow our detached set on the list
  if n_elements(rec) gt 0 then begin
    if n_elements(detachlist) gt 0 then detachlist=[detachlist,rec] else
    detachlist=[rec]
  endif
  if ptr_valid(self.children) then $
    for i=0,n_elements(*self.children)-1 do $
      (*self.children)[i]->Detach,detachlist
end

pro topClass::Reattach, detached
  self.parent=(*detached).parent
  ptr_free,detached
end

pro topClass::ReattachList, list
  for i=0,n_elements(list)-1 do $
    if obj_valid(list[i].Object) && ptr_valid(list[i].Detached) $
    then list[i].Object->Reattach,list[i].Detached
end

pro topClass::Save, file, COMPRESS=comp
  self->Detach,list             ;all our vital info is now stashed in list
  catch, serr
  if serr ne 0 then begin       ;it failed!
    catch,/CANCEL
    self->ReattachList,list
    message,'Error Saving to File: '+file
  endif
  save,self,FILENAME=file,COMPRESS=comp
  catch,/CANCEL
  self->ReattachList, list
end
```

Now, when you say 'obj->Save, file', it will detach its unnecessary parts, stowing them on the list for safe keeping, and then recursing down to its children and so on, thus sending a propagating wave of detachment all the way down the tree hierarchy. Then the object and its children will be saved, and then everything will be re-attached by iterating over the detached list. Notice how I was careful to reattach everything in case of error too.

Now suppose some lower class has more than just the parent that it needs
to detach, e.g. some widget ids, irrelevant to keep track of, since of
course they will change. Then you can simply overload the Detach and
Reattach methods like so:

```
pro lowerClass::Detach,list, RECORD=rec
  if n_elements(rec) gt 0 && ptr_valid(rec.Detached) then $
     *rec.Detached=create_struct('widget_info',self.widget_info,* rec.Detached)$
  else rec={Object:self, Detached: ptr_new({widget_info: self.widget_info})}
  self.widget_info=ptr_new()
  self->topClass::Detach,list,RECORD=rec
end

pro lowerClass::Reattach, detached
  self.widget_info=(*detached).widget_info
  self->topClass::Reattach, detached
end
```

Here I allow for even further sub-classing with the same RECORD keyword.
Anyway, this (or rather some tested and debugged version of this),
should serve well enough to strip out all those troublesome back links
for saving. Of course, when you restore the object from disk, none of
the parent references will be valid, but perhaps this is not a problem,
if you're just using the data embedded in this structure. Another
option which is fancier but doable is to only trim parents which point
"above" you in the tree hierarchy. I leave that one as an exercise ;).

JD
