Kill Objects Dead!

Doug Hennig

Dangling object references are among the hardest bugs to track down, and can cause a wide range of problems. This month's article looks at the cause of these problems, presents a couple of tools to help track them down, and discusses techniques for preventing them.

Yuanitta Morhart, a senior consultant at Stonefield Systems Group, came into my office one day recently and asked me to look at a problem she was having. "Every time I run and close this form, I get an 'unknown datasession' left open," she exclaimed, exasperated. The problem was obviously a case of some object not being destroyed, but she'd spent a while trying to track down which object was the culprit without any success. I too spent some time on the problem, and what we did to finally solve it is the basis for this month's column.

Dangling Objects

First, a little background. When you create an object using code such as:

```
loObject = newobject('MyClass1', 'test.vcx')
```

the class is really instantiated somewhere in VFP's memory space, and a reference to that object is stored in the specified variable. VFP keeps an internal counter of how many references there are to an object, and when the last one is destroyed, the object itself is destroyed. So, in the following code:

```
loObject1 = newobject('MyClass1', 'test.vcx')
loObject2 = loObject1
release loObject1
release loObject2
```

the object isn't actually destroyed until the last statement (you can see this because I put a WAIT WINDOW in the Destroy method of the class). That's because the second statement doesn't create a copy of the object (as beginners to object-oriented programming might expect) but instead creates a second reference to the same object.

So far, so good; we just need to be sure to release all references to an object to make sure it's destroyed, right? Well, it can get a little complicated. Imagine the following:

```
loObject1 = newobject('MyClass1', 'test.vcx')
loObject2 = newobject('MyClass1', 'test.vcx')
loObject1.oObject = loObject2
release loObject2
release loObject1
```

Notice that you don't see the Destroy method of the second object fire when you release loObject2. Instead, you see the Destroy method of both objects fire on the last statement. That's because the first object contains a reference to the second; when loObject1 is released, that's the last reference to the first object, so it gets destroyed, which destroys the last reference to the second object.

Now look at a worse situation:

```
loObject1 = newobject('MyClass1', 'test.vcx')
loObject2 = newobject('MyClass1', 'test.vcx')
loObject1.oObject = loObject2
loObject2.oObject = loObject1
release loObject1
release loObject2
```

Guess what? Although the loObject1 and loObject2 variables are gone, the objects are still alive (you won't see the Destroy methods fire at all). This is because the first object contains a reference to the second, and the second contains a reference to the first. So, nuking the variables doesn't get rid of the last reference to these objects; this type of situation is sometimes called a “deadly embrace”. To make matters worse, there’s no way to reference these objects any more; they’re orphaned in memory. You can’t even find them using the AINSTANCES() function. The only thing you can do is get rid of the objects using CLEAR ALL, which of course does a lot more than just destroy those two objects—it blows away everything.

You’re probably wondering “who would ever want a deadly embrace kind of object design?” I’ll give you an example of one: the report objects I described in my June 1999 column. The main class, SFReportFile, has object references to SFReportBand objects (for example, the oDetailBand property contains a reference to the SFReportBand object used as the detail band in the report), and SFReportBand objects contain a reference to the SFReportFile object in their oReport property because they need to use some services in SFReportFile. If you instantiate an SFReportFile object into a memory variable and then release the variable, the SFReportFile, SFReportBand, and other SFReport* objects still exist in memory because they’re locked in a deadly embrace.

How do you know when you have a problem caused by objects not being destroyed? Some of the symptoms are:

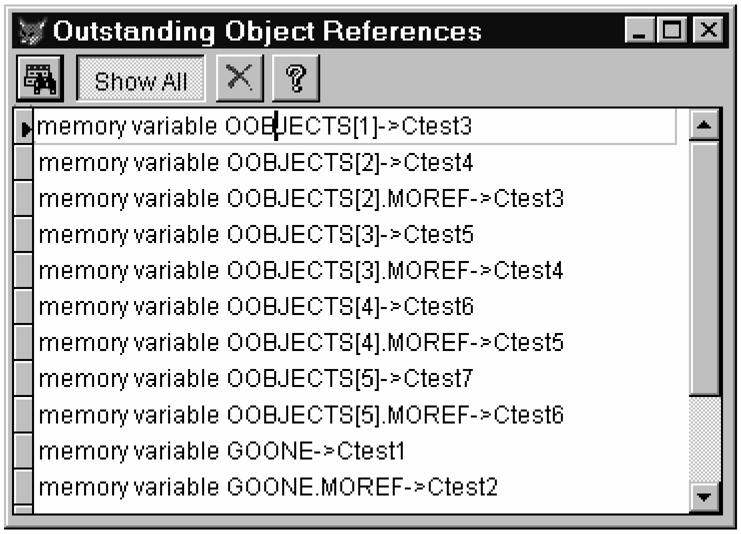
- A form doesn’t go away when you click on its Close box or a “Close” button. This is caused by something holding a reference to the form or some object on the form.
- Performance degrades as inaccessible objects chew up memory and virtual memory gets used more and more.
- “Unknown” datasessions exist after a form with a private datasession is closed. This is caused by the form instantiating some object (so it lives in the form’s datasession) but not destroying it when the form is destroyed. The form is truly gone, but since some object still lives in the datasession created by the form, the datasession can’t be destroyed. You may even find open tables or cursors in this datasession if anything opening or creating them didn’t close them again upon release.
- Although I’ve personally never seen this (“Sure”, you’re thinking <g>), I suspect unreleased object references may eventually lead to a GFP.

Hunting Down Objects

Once you’ve determined that you likely have unreleased objects, how do you find out which ones they are? One way is to go through all your source code and look for every object that gets created, then ensure it’s properly destroyed. This isn’t the greatest approach, since there may be a lot of complex code to search through, and in the case of third-party tools or libraries, you may not have all the source code.

One of the ways you can look for object references is to use the AINSTANCES() function. Although it won’t work on inaccessible objects, it can help you locate references to objects that may not be obvious otherwise. Rather than using AINSTANCES() directly, I prefer to use a tool called ObjRef, written by FoxPro guru David Frankenbach and available from his Web site (http://www.geocities.com/ResearchTriangle/9834/mainfram.htm). Dave’s tool goes through _SCREEN, _SCREEN.Forms, _VFP, _VFP.Objects, and public variables, looking for object references, and displays any it finds in the form shown in Figure 1.

Figure 1. The ObjRef utility.



```
memory variable OOBJECTS[1]->Ctest3
memory variable OOBJECTS[2]->Ctest4
memory variable OOBJECTS[2].MOREF->Ctest3
memory variable OOBJECTS[3]->Ctest5
memory variable OOBJECTS[3].MOREF->Ctest4
memory variable OOBJECTS[4]->Ctest6
memory variable OOBJECTS[4].MOREF->Ctest5
memory variable OOBJECTS[5]->Ctest7
memory variable OOBJECTS[5].MOREF->Ctest6
memory variable GOONE->Ctest1
memory variable GOONE.MOREF->Ctest2
```

While ObjRef is useful for locating object references in public places, it can't help with inaccessible objects (since by definition they're inaccessible <g>). I tried putting WAIT WINDOW statements in the Destroy method of classes, but found it extremely tedious watching every object get destroyed and difficult to figure out from a list of which objects were instantiated which ones didn't get destroyed. That's when I came up with the idea of an object logger.

LOG.PRG is a very simple program: it either adds a record to a log table (in the case of an instantiation) or modifies an existing record (in the case of a destruction). Here's the code for this program:

```
lparameters tcID, ;
  tcName, ;
  tcClass, ;
  tlInit
local lnSelect, ;
  llOpened, ;
  lcID, ;
  lnStack, ;
  lcName, ;
  lnLen, ;
  lnI, ;
  lcModule

* Open or create the log file.

lnSelect = select()
do case
  case not file('LOG.DBF')
    create table LOG free (ID C(10), NAME C(40), ;
      CLASS C(40), INSTFROM C(128), INIT L, DESTROY L)
    index on ID tag ID
    llOpened = .T.
  case not used('LOG')
    select 0
    use LOG order ID again
    llOpened = .T.
  otherwise
    select LOG
    set order to ID
endcase

* If this is an instantiation, figure out where the
* object was instantiated from. Start from the method
* that called us, and work back.

lcID = lower(tcID)
do case
```

```
  case tlInit
    lnStack = program(-1) - 1
    lcName  = upper(tcName) + '.'
    lnLen   = len(lcName)
    for lnI = lnStack to 1 step -1
      lcModule = program(lnI)
      if left(lcModule, lnLen) = lcName
        lcModule = program(lnI - 1)
        if lcModule = 'MAKEOBJECT'
          lcModule = program(lnI - 2)
        endif lcModule = 'MAKEOBJECT'
        exit
      endif left(lcModule, lnLen) = lcName
    next lnI
    insert into LOG values (lcID, lower(tcName), ;
      lower(tcClass), lcModule, .T., .F.)

* This is a destroy, so try to find the existing record
* in the log and update it.

  case seek(lcID)
    replace DESTROY with .T.
endcase

* Clean up and exit.

if llOpened
  use in LOG
endif llOpened
select (lnSelect)
```

LOG.PRG accepts four parameters: an identifier for the object (we'll discuss this in a moment), the object's name, the object's class, and a flag indicating if the object is being instantiated or destroyed. LOG either opens or creates (if it doesn't exist) a log table with columns for the object identifier, name, class, name of the program or method that instantiated the object, whether the object instantiation was logged, and whether the object destruction was logged. If the last parameter is .T. (indicating the object is being instantiated), LOG figures out which program or method did the instantiation by looking at the program stack and grabbing the program prior to the one that called LOG. Presumably, the call to LOG would be in the Init method of the object, but it also might be in a parent class for the object, so we need to find a method that contains the object name; also, since I use MAKEOBJECT.PRG to instantiate objects, this code skips that too. LOG then adds a record to the log table. If the last parameter is .F., LOG looks for an existing record for the specified object and sets its DESTROY column to .T.

To enable object logging, I added the following code to the Init method of SFCustom (the base class used for all custom classes):

```
This.Tag = sys(2015)
do LOG with This.Tag, This.Name, This.Class, .T.
```

This.Tag is set to a unique value, then LOG.PRG is called with This.Tag as the identifier for the object. The reason for an identifier for the object is that you may have several objects with the same name. This is common when you have a collection or array containing many instances of the same object. In this case, they all have the same name (the name of the class) so identifying them in the log table is impossible.

The Destroy method of SFCustom includes this code:

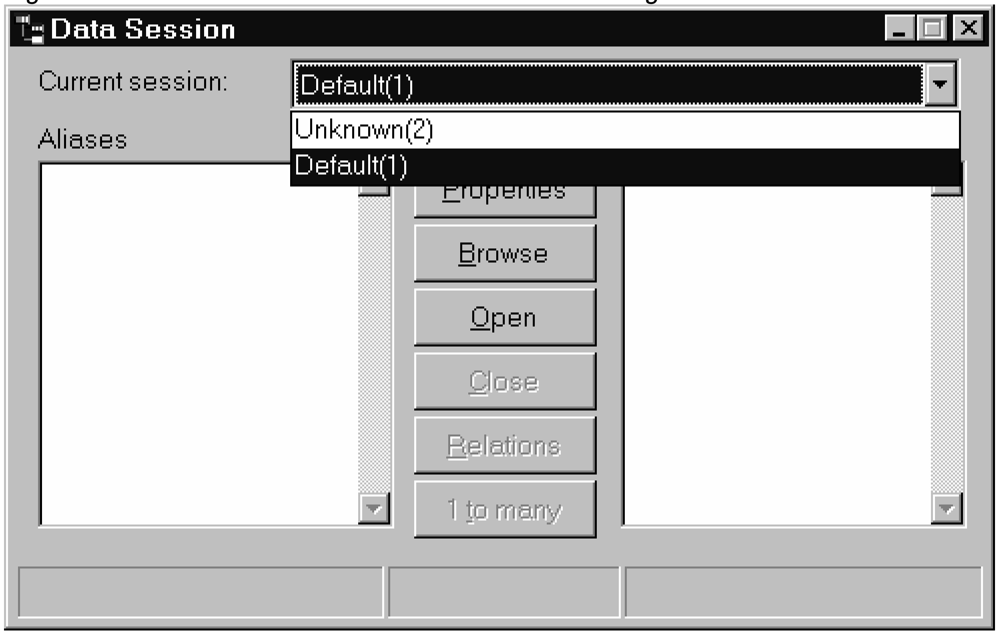
```
do LOG with This.Tag, This.Name, This.Class
```

This.Tag is passed again so LOG.PRG can find the correct record in the log table.

To see this in action, run the CUSTREP program. It instantiates an SFReportFile object (which is based on SFCustom, so object instantiation and destruction will be logged) and programmatically creates a report. When it's finished, open and browse the log table. You'll find a lot of records in there (SFReportFile instantiates many other objects), and interestingly, none of them have DESTROY set to .T., meaning that they're all still alive somewhere but inaccessible. Type CLEAR ALL in the Command window and again open and browse the log table. This time, DESTROY is set to .T. for all records. This proves that the objects didn't get destroyed until the CLEAR ALL command was issued.

To see a better example, erase LOG.DBF and run the CUSTREP form and click on the Report button. Close the form and type SET. As you can see in Figure 2, although the form is gone, its datasession lives on. If you open and browse the log table, you'll see the same results as before (not surprising, since the button's Click method contains the same code as CUSTREP.PRG). Type CLEAR ALL and the unknown datasession goes away.

Figure 2. An "unknown" datasession exists after running CUSTREP.SCX.



Fixing the Problem

Once I'd determined that SFReportFile was the culprit in our form (all the other objects were instantiated from it), it didn't take long to find out why it wasn't being destroyed when the variable referencing it was released. As I mentioned earlier, SFReportFile and several of the objects it instantiates form a deadly embrace. So, if releasing the variable containing the object doesn't cause the object to be destroyed, how do we destroy it?

The answer is surprisingly easy: tell the object to release itself. I added a Release method to all the classes in SFCTRLS.VCX, our base class library, that don't natively have this method. Then, in the Release method, I put this code (the last line isn't required if the class has a native Release method):

```
if This.lRelease
  nodefault
  return .F.
endif This.lRelease
This.Cleanup()
release This
```

This code first checks a custom property called lRelease (which I added to the class and set to .F.) and doesn't carry on if it's .T. This prevents this code from doing things twice if, during the destruction of this object, another object gets destroyed and that object calls the Release method of this object. It then calls a custom Cleanup method and releases the object.

The Destroy method also calls Cleanup, because it could be that this object is being properly destroyed but some reference to another object it contains may not be. Here's the code for Cleanup:

```
if This.lRelease
  return .F.
endif This.lRelease
```

```
This.lRelease = .T.
This.ReleaseMembers()
This.oHook = .NULL.
This.oMenu = .NULL.
```

As with Release, it doesn’t do anything if we’re already in a “being released” state. It then sets lRelease to .T., calls a custom ReleaseMembers method, and specifically nukes two properties that may contain object references. ReleaseMembers is an abstract method (it contains no code in the base class) that makes it easier to destroy other properties in subclasses: rather than overriding Cleanup (and having to check for This.lRelease in the subclass), simply put the code to destroy those properties in ReleaseMembers. How do you know which properties need to be destroyed? If you use a naming convention in which properties that contain object references start with “o”, simply look in the Property sheet for proeprties starting with “o” and add code to destroy them in ReleaseMembers.

What’s the best way to destroy the properties: setting them to .NULL. or calling their Release methods? It depends (don’t you hate that answer <g>). My rule of thumb is this: if the class instantiated an object itself (that is, a call to CREATEOBJECT(), NEWOBJECT(), or my MAKEOBJECT() was used in this class), then call its Release method. This ensures that if that object contains a reference to this object, that object will destroy its reference to this object, which will allow this object to be destroyed. Of course, you should ensure the object exists before calling its Release method. Otherwise, set the property to .NULL. Why the distinction? What if ObjectA creates ObjectB and sets a property of ObjectB to ObjectC? If ObjectB called the Release method of ObjectC when it gets released, it would destroy ObjectC even though it didn’t create it. What if ObjectA needs to continue using ObjectC after ObjectB has been destroyed? Hence the rule: if you didn’t create the object, don’t release it; just nuke the property referencing the object.

SFReportFile is a subclass of SFCustom, and it has properties that hold references to other objects, so its ReleaseMembers method looks like this:

```
local lnI
if vartype(This.oTitleBand) = 'O'
  This.oTitleBand.Release()
endif vartype(This.oTitleBand) = 'O'
if vartype(This.oPageHeaderBand) = 'O'
  This.oPageHeaderBand.Release()
endif vartype(This.oPageHeaderBand) = 'O'
if vartype(This.oDetailBand) = 'O'
  This.oDetailBand.Release()
endif vartype(This.oDetailBand) = 'O'
if vartype(This.oSummaryBand) = 'O'
  This.oSummaryBand.Release()
endif vartype(This.oSummaryBand) = 'O'
for lnI = 1 to alen(This.aGroupbands)
  if vartype(This.aGroupBands[lnI]) = 'O'
    This.aGroupBands[lnI].Release()
  endif vartype(This.aGroupBands[lnI]) = 'O'
next lnI
for lnI = 1 to alen(This.aVariables)
  if vartype(This.aVariables[lnI]) = 'O'
    This.aVariables[lnI].Release()
  endif vartype(This.aVariables[lnI]) = 'O'
next lnI
```

SFReportBand (a subclass of SFCustom that’s instantiated by SFReportFile) sets its oReport property, which contains a reference to the SFReportFile object, to .NULL. So, when SFReportFile is released, it tells all of the objects it references to release themselves, which in turn nuke their references to the SFReportFile object, which then can itself be destroyed since nothing else holds a reference to it.

If you modify CUSTREP.PRG or the Click method of the command button in CUSTREP.SCX, you’ll find a commented out line at the end that calls the Release method of loReport (the variable holding the reference to the SFReportFile object). Uncomment this line and run the program or form again. This time, you’ll find the log contains no entries with DESTROY set to .F., and in the case of the form, there’s no “unknown” datasession left over.

Of course, once you’ve tracked down the problem, you’ll want to remove the calls to LOG in the Init and Destroy methods of your classes, or else performance will be dragged down needlessly.

Conclusion

Dangling object references are among the most difficult bugs to solve in any programming language, and VFP is no exception. Hopefully, you'll get some ideas from this article about tracking these problems down and designing your objects to prevent them in the first place.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). He can be reached at dhennig@stonefield.com.