Taking out the garbage

Derek Williams

"One person's trash is another person's treasure."

"Memory leak!": it's a scary phrase, yet so many object-oriented programmers use it. In a general sense, it means that the memory size of a running program continually grows so less room is available to create new objects. The cause is usually *dangling instances*: objects that are no longer needed but are still around, using valuable memory space. For C++, dangling instances usually mean that the programmer forgot to explicitly delete objects or failed to follow the protocols for who is responsible for deleting. For Smalltalk, dangling instances are usually objects that are considered trash to the programmer but treasure to the garbage collectors.

Thanks to the garbage collectors in nearly all Smalltalk implementations, Smalltalk programmers aren't burdened with having to explicitly delete objects: they simply create new objects as needed and let the garbage collectors remove obsolete objects and reclaim the space. The garbage collectors deem an object to be obsolete if it is no longer referenced (directly or transitively) from the "root" pointers that keep immortal objects around. But sometimes, programming errors can leave unintended, dangling references to an object, causing it to live much longer than it should.

In this article I introduce some common tools and techniques for detecting, diagnosing, and treating dangling instances problems. It is slanted toward the VisualWorks environment and its memory management architecture, but many of the concepts apply to other Smalltalk dialects.

DETECTION: DO YOU HAVE MEMORY PROBLEM?

You might have a memory problem if:

1. Your cursor frequently changes to a garbage collection or compaction cursor.
2. Your development image file grows progressively larger each time you save it, especially if you feel you've done nothing that would cause it to be larger.
3. The memory footprint of your image grows to be much larger than you think it should be. You might notice that less space is available to other programs or you might detect it with your finger on the *dynamically allocated footprint* pulse of Smalltalk.
4. You notice heavy thrashing: lots of disk I/O due to the operating system constantly swapping in and out the memory allocated for use by Smalltalk.
5. You see a "low space notifier" or "out of memory" error message.

These types of errors usually mean that memory problems have gotten out of hand. We'll look at some steps you can take to avoid getting to this point, but first let's see how each of these can occur. To understand these effects, it helps to know a little about how your Smalltalk virtual machine (a.k.a. object engine) manages memory.

The Smalltalk object engine divides the memory it uses to store objects into a number of separate regions or *spaces*. It does this to get the optimum benefit from different garbage collection schemes. For example, newly created objects that are less than 1 KB in size are stored into the Eden *subspace* in NewSpace. NewSpace is managed by the scavenger garbage collector, which uses a two-space copying algorithm. The scavenger runs as a background process, alternately copying surviving objects between Eden and the two SurvivorSpaces. The two-space algorithm works especially well for NewSpace, since most new Smalltalk objects live very short lives and can be easily discarded simply by not copying them forward.

Objects that survive NewSpace are *tenured* into OldSpace, which is managed by an incremental mark-and-sweep garbage collector. OldSpace is unique in that it dynamically grows as needed to accommodate the "mature" objects in the system. All other spaces are fixed in size when the memory policy is installed (typically at image startup).

You can control the balance between collecting garbage and thus reclaiming space and growing the size of OldSpace by changing memory policy parameters. When a request is made to allocate a new object and there simply isn't room for it, we say a *low space condition* has occurred. It is then up to the memory policy to decide what to do: to try to reclaim space by aggressively collecting garbage and compacting objects or to simply ask for more memory from the operating system. You can set a cap at which reclamation will be favored over growth and even limit the total memory size of the image.

Much of the partitioning of objects into spaces is done "under the covers" and not directly visible to your Smalltalk code. For example, you cannot find out which space a given object resides in or find all the objects in a particular space. But the parameters that control the

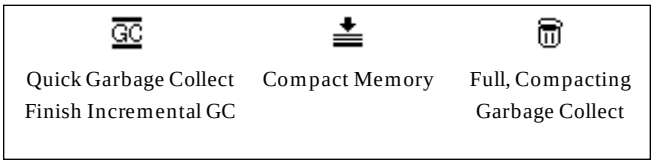

Figure 1. Cursors indicating you are running out of space.

memory management behaviors are available to your Smalltalk code and you can modify them. For example, you can set the space sizes based on your application needs, check various garbage collection statistics, check on the current size of OldSpace, explicitly invoke a garbage collector, etc. There's a lot more that can be said about how the garbage collectors work and how to tailor and tune the memory policy for your needs—that's not the purpose of this article. But you can start learning by reading the class comments and documentation methods in the classes ObjectMemory and MemoryPolicy or reading the Memory Management chapter in the VISUALWORKS USER'S GUIDE.[1] You should also read Kent Beck's article on garbage collection in the February 1995 issue of this publication. It explains how the garbage collection algorithms work in detail and covers the Visual Smalltalk environment.

Although we're not going to delve further into the object engine's memory management work, we can use our basic knowledge of how it operates to help detect dangling instance problems. Let's review those five "warning signs" and look at how each could occur.

Frequent cursor changes

The MemoryPolicy (through services in ObjectMemory) displays special cursors to show you when incremental garbage collection and compaction activities are occurring. Since these activities are typically in response to low space conditions, they provide a visual clue to the state of memory. Frequent collection or compaction cursors are often early indicators that you are running out of space. Figure 1 shows what these cursors look like.

Large image file size

Most Smalltalk applications are coded and unit tested using development images that are frequently saved to disk and then packaged into runtime images for further testing and deployment. Usually the runtime images are delivered to users who load, use, and exit them as needed, but never save them again.

Some memory leaks can go undetected in runtime images because the user starts at square one each time he or she reloads the image. Since each image save writes all the objects to disk, dangling instances have a way of stacking up in a development environment. During development, it's a good idea to occasionally look at the size of your image file. If it grows larger than you would expect, you have an early indicator of a potential memory problem.

Large memory footprint

You don't have to wait until you save your image to a file to determine its size. Sending ObjectMemory dynamicallyAllocatedFootprint will answer the total number of bytes of memory your image is currently using. You can send this message as often as you like (such as before and after testing an application scenario) to gather measurements.

There are other services on the ObjectMemory class to give you a view of the current sizes and state of memory. For example, since only OldSpace will grow in size, you may only be interested in oldBytes, rather than the total size.

Thrashing by the operating system

Recall that the image will grow in size until it reaches the limit you set or until the operating system refuses to give more memory to Smalltalk. You should set a cap so that Smalltalk is a good citizen and leaves plenty of room for other programs to run. If you don't and the image grows too large, switching back and forth between Smalltalk and other programs can lead to heavy swapping.

When running under MS Windows, you should be aware that Windows will often politely let Smalltalk have so much memory that it doesn't keep enough space for a good working set of its own. When this happens, even a basic operation like opening a new window can cause swapping.

Because thrashing is never a good thing, and because the object engine never gives space back to the operating system until you exit, you should choose your cap carefully. If you would like to start favoring reclamation over growth at *x* bytes and you never want your image to be larger than *y* bytes, you can set a fixed value with something like:

```
ObjectMemory installMemoryPolicy:
    (MemoryPolicy new
        setDefaults;
        growthRegimeUpperBound: x;
        memoryUpperBound: y;
        yourself).
```

Or you can get information about available or installed memory from the operating system and use this as a basis for setting a cap or controlling your own custom memory policy. Also, the Runtime Packager tool has a window you can use to set memory sizes when you build a runtime image.

Low space notifier

If things are really bad, i.e., you've run out of room for new objects, garbage collections do not reclaim enough space, and the image cannot grow any more, you may see a *low space notifier*.

The MemoryPolicy has no direct way of communicating with the user to report that space is running dangerously low. So, it invokes the low space notifier via the user interrupt signal when it needs to say "Emergency: No SpaceLeft" or "Space warning."

The user interrupt signal is the same mechanism used to invoke the emergency evaluator when Ctrl+Shift+C is pressed. If you've tried to disable the emergency evaluator for a runtime image or change the way it displays, you should make sure that you're not masking the low space

notifier. I've seen cases where low space conditions were simply reported as "user interrupts" and the end user had no idea what was happening.

DIAGNOSIS: FINDING THE CAUSE

So, based on the warning signs described above, you think you have a memory problem—now what? We want to find exactly which objects are not being cleaned up and why.

A quick scan of instance counts can help you find the dangling instances and narrow the set of classes to examine for potential problems. A code snippet like the one in Listing 1 can help.

Listing 1.

```
| dict |
"Faster than sending instanceCount to all classes"
ObjectMemory garbageCollect.
dict := IdentityDictionary new.
Object allSubclasses
    do: [ :ea | ea isMeta
            ifFalse: [ dict at: ea put: 0 ]].

ObjectMemory allObjectsDo:
    [ :ea | | count | (count := dict at: ea class ifAbsent: []) notNil
                                ifTrue: [ dict at: ea class put: count + 1 ]].
(dict reject: [ :ea | ea value < 200 ]) inspect.
```

You're now inspecting a dictionary that shows you counts for all classes having greater than 200 instances floating around. Choosing 200 as a cutoff is arbitrary—substitute whatever works for the situation or create your own list of classes to check. If you want to narrow the search, you can replace Object with another parent class. Once you have the inspector, you'll probably want to look further, e.g., you may want to sort by class name:

```
self associations asSortedCollection:
    [ :a :b | a key name < b key name ]
```

or by number of instances:

```
self associations asSortedCollection:
    [ :a :b | a value > b value ].
```

Once you've targeted a class that appears to have more instances than it should, send allInstances or allInstancesWeakly: and inspect the result.

This is a rather brute-force approach to tracking down runaway instances, but it's often all you need. You can wait until runaway instances start to get out of hand, interrupt your code if necessary, and run this snippet. For example, if you see 1,000 instances of your Scooter class and you were expecting only two or three, you have a good place to start.

If you can run through an application scenario to consistently create the problem or if you don't know whether or not you have a problem, then check at regular intervals. Periodic measurements taken with the code snippet in Listing 1 and displaying the value of ObjectMemory dynamicallyAllocatedFootprint can tell a lot.

It's a good idea to include some lightweight memory diagnostics like the above even in a runtime image you deliver to customers. For example, you might add a window somewhat off the beaten path to display footprint sizes or instance counts on demand. These snippets add very little to the size of a runtime image, and you may just find that your customers can create memory problems you never expected (e.g., by leaving an image running steadily for several weeks at a time).

The AllocationProfiler in Advanced Tools can show you which new objects a particular block of code allocates by tracking calls to methods that create new objects. Since it shows you only memory allocation and not reclamation, be prepared to look through the complete picture (remember, most new objects die quickly). But it is an easy tool to use and gives detailed information. To use it, simply send AllocationProfiler profile: and pass it a block to measure. You can learn more about the AllocationProfiler by reading the ADVANCED TOOLS USER'S GUIDE.[2]

Finally, I've implemented some of the above techniques in the Memory Diagnostics tool, which you can get from the University of Illinois Smalltalk archives (http://st-www.cs.uiuc.edu/).

TREATMENT: FINDING AND CLEANING UP REFERENCES

The measurements you took above should tell you at least two things: (1) what the dangling instances are and (2) what application scenario creates them. But often you need to look further: you want to know exactly where the dangling references are coming from.

You can determine this by inspecting one of the dangling instances and looking at the reference path to it. There are several ways to follow reference paths:

1. Manually follow the path of references by sending allOwners or allOwnersWeakly: and inspecting the result. This will show you the immediate references to your dangling object. Sending allOwners or allOwnersWeakly: to each of these references will show you the next level. You can continue this process until you start to see objects or methods that point to potential problems.
2. Use the ReferencePathCollector in Advanced Tools. If you're inspecting one of your dangling instances, you can send ReferencePathCollector allReferencePathsTo: self and inspect the result. Read the comments for class ReferencePathCollector for more information.
3. Use the PointerFinder tool written by Hans-Martin Mosner. You can get it from the author's web page at http://donald.heeg.de/pub/hmm-goodies/.

The reference path will often provide its own clues to exactly which portion of code caused the dangling instance and why it is not being cleaned up. Since it helps to know what to look for, here are some common causes.

1. Unbroken dependencies. In many cases, dangling

instances are due to one object being referenced as a dependent of another object that is still in use. The dependent object may no longer be needed, but it isn't collected because it is still referenced by the "parent" object. Usually this is caused by a failure to send removeDependent: or one of the related methods. Keeping track of all the places where dependencies are set and broken can sometimes be difficult, given all the different layers and frameworks that use them (dependency transformers, adapters, value models, etc.) and all the message variants.

The unbroken dependency problem is much easier to create if you add dependents to an object that does not track them in an instance variable (e.g., it does not override Object>>myDependents). In this case, the dependency connection is kept in the global DependentsFields dictionary. With "local" dependents, a failure to break dependencies will be cleaned up when the parent is no longer referenced. But when the dependencies are tracked in the DependentsFields dictionary, you have a new pair of references that will keep parent and dependent around. So, inspecting DependentsFields is often a good way to look for problems.

2. Overlooked object references. Dependency connections certainly aren't the only common ways that objects reference each other. Indeed, object connections are at the very heart of object design—those associations and aggregations we like so much. You may need to clean up some of your own object references through a release, finalization, or similar protocol. For example, you can use the release event for ApplicationModel classes to "nil out" or otherwise clean up references to objects referenced by instance variables.

   It's easy to forget about object caches held onto by class variables and class instance variables. While such caches are nice for boosting performance, don't forget about them. You may want to implement and send class-side "uninitialize" methods to clear out caches when necessary. If you're using ENVY, you may find yourself doing this as part of *removing* methods to unload an application.
3. Failure to copy. So many collection operations answer copies that we sometimes take it for granted and assume we always have either a shallow copy of a collection or a deep copy of the collection and its contents. This assumption can be dangerous and can not only indirectly be a source of dangling references, but also lead to other errors such as one client of a collection modifying its contents and affecting others. This is, by the way, why a method answering a literal string that is coded in it is generally a bad idea.

How you manage references and copies really depends on what you are trying to do, so it's hard to give general rules. But if you suspect a problem caused by a reference to a shared object rather than a copy, you can use the identity comparison (==) or compare object identifiers to see if the references really are to the same object. To see the object identifier, send asOop to the object and print or inspect the result.

Finally, now that you've found the dangling instances and cleaned up the cause, what do you do with all those "zombies" floating around?

It's best to start with a clean image and load your code into it. But if you're fond of your current image and want to keep it, you'll need to do your own clean up. When cleaning up an image, track back to the root cause and correct it. For example, you may have to remove dependency connections from an inspector on DependentsFields.

I often hear the suggestion to use become: String new to "morph" a dangling object so that it loses its instance variable links. Using become: should always be a last result and done with great care. And you should keep in mind how your Smalltalk implements it—whether it swaps pointers or copies state.

CONCLUSION

Now that you have the fear of runaway memory problems, take comfort: it's usually a rare occurrence. The garbage collectors do an amazing job of managing memory efficiently and the class libraries are tolerant of potential errors. Only rarely do I have to pull out this bag of tricks to help someone diagnose a memory problem. But by using some of these techniques, you'll have the diagnostics to easily watch for problems and, when you find one, the tools to track it down and fix it.

References

1. ParcPlace Systems. VISUALWORKS USER'S GUIDE, Sunnyvale, CA, 1994.
2. ParcPlace Systems. ADVANCED TOOLS USER'S GUIDE, Sunnyvale, CA, 1994.

Derek Williams has been developing vertical client/server applications for 11 years and using Smalltalk for the past 4 years. He can be reached at derek_wi@hboc.com.

continued on page 32