**Simplified identification of memory problems with ANTS Memory Profiler’s Instance Categorizer**

An application’s memory usage is made up of instances, each of which belongs to a class. For example, an application may store a person’s name in memory as a single instance of the string class.

Each instance is held in memory by being referenced by other objects, which in turn are referenced by other objects, forming chains of references back to a set of objects called GC roots. GC roots are special system objects which are never freed from memory. The other objects in the reference chain need not necessarily be members of the same class as the instance in question, and an instance can be referenced by multiple objects, forming multiple paths back to multiple GC roots.

For any instance, we can draw an Instance Retention Graph, which shows all the chains of references back to GC roots which cause that instance to be held in memory.

![Instance Retention Graph](image)

**Figure 1. An Instance Retention Graph**

When an object no longer needs another object it is referencing, it is important to remove the reference to that object. If this isn’t done correctly the object being referenced will never be removed from memory, and the application will experience a memory leak. This results in significant performance degradation as a result of increased resource usage, and in many situations can cause an application to crash altogether.

In the example of Figure 1, our instance is being held in memory because it is referenced by object A, which itself is being referenced by objects B and C. Object C is held in memory by object D, which in turn is referenced by GC root 1 and object E. Objects B and E are respectively held in memory by GC roots 2 and 3.

When searching for memory leaks, we examine an instance’s Instance Retention Graph, and analyze each link in the chain to understand whether there is a genuine reason for the reference to still be
required. If, for example, object B still needed access to the name stored in the instance, this would be a legitimate reason for the instance being held in memory and we would know there wasn’t a problem here. Conversely, if object A was a Form which should have been closed, but was still being held onto by object B because the reference hadn’t been removed properly, we would have discovered a memory leak. We would then go into our code and remove that reference to fix the leak. To free an instance from memory, we need to break all paths back to the GC roots.

Unfortunately even a trivial application might generate tens of thousands of instances. Examining each Instance Retention Graph would not be possible in a reasonable period of time. In the past, this meant that choosing which instances to examine was a semi-random process, involving a significant element of trial and error.

ANTS Memory Profiler 7 introduces a feature to remove the uncertainty from the process of locating suspect instances: the Instance Categorizer. For any selected class, the Instance Categorizer intelligently scans the Instance Retention Graph of every instance in that class and groups together sets of instances which have similar Instance Retention Graphs. For every set of instances discovered, a chain of references is drawn showing that set of instances’ shortest path back to a GC root.

Why is this useful? Instances which have similar Instance Retention Graphs have a strong tendency to have been created by an application for similar reasons. For example, if a Form were to retrieve 1000 names from a database and store each of those names as an instance in memory, they would all have nearly identical Instance Retention Graphs. In effect the Instance Categorizer allows us to visualize these instances as a single block of memory with a common reason for being held in memory. If they were all being held in memory because of an error disposing of the Form, identifying that error as being shared by those 1000 instances amongst the thousands of others might not be possible without the Instance Categorizer grouping them together. The memory leak would never be discovered.

Let’s look at another example. Over the course of a few minutes an application creates 24 instances of a class. There are a few different reasons for those instances to have been created; as such they each have different Instance Retention Graphs. If we were able to see each of those instances’ Instance Retention Graphs side by side they might look a little like Figure 2. Even for this simplistic example, picking out similar Instance Retention Graphs may take some time.

Figure 2. The instances held in memory.
The Instance Categorizer identifies instances with similar Instance Retention Graphs and groups them together in sets – in this case there are 3 different sets. There are also 3 instances whose Instance Retention Graphs aren’t similar to any other instances. These instances are not grouped with any others.

![Instance Retention Graphs](image)

*Figure 3. Sorting the instances into sets.*

Next, the Instance Categorizer displays the shortest chain of references back to a GC root for each of the sets of instances identified.

![Chains of References](image)

*Figure 4. The chains of references for categorized instances.*

Now that we are only required to examine 3 chains of references instead of 24, it is easy to look at each chain and identify whether those instances are being held in memory unnecessarily. In most real-world scenarios, those chains would be found by analyzing thousands of instances, not 24, and the Instances Retention Graphs are frequently far more complex than those shown.

For each of the chains, we can also generate a list of instances which belong to that set. We can then generate a full Instance Retention Graph for any of these instances, and from there identify the precise issue keeping it held in memory. Resolving that issue will likely result in all the instances in that same set being freed, improving the application’s memory usage, and ultimately leading to better performing code.