Towards a Lightweight Tool for Locating Unintentional Object Retention in Java Programs

Chin-Hung Chien

Foxconn Electronics Inc., Taipei, Taiwan.

richard.chien@foxconn.com

Kung Chen and Ju-Bing Chen

Dept. of Computer Science, National Chengchi University, Taipei, Taiwan.

chenk@cs.nccu.edu.tw; g9405@cs.nccu.edu.tw

Abstract

Many Java programmers think that since there is a built-in garbage collector running behind the scene, they do not have to worry about freeing dynamically allocated memory. On the contrary, it is not unusual to see a Java program producing an OutOfMemory error after running for a period of time. Indeed, memory leaks do occur in Java programs and happen in a very different way than those in C or C++ programs. They are resulted from proliferation of objects which exist beyond their intended lifetime, the so-called *loitering objects*. These objects should have been removed by the garbage collector, but they persist because some active object unintentionally retains a reference to them. In Java terms, these loitering objects are reachable from objects in the *root set* and hence will not be garbage collected.

There are a variety of commercial or freely available profiling tools [1] which are designed to assist Java programmers in identifying loitering objects in their Java programs. As far as we know, all these tools attempt to dig out information about loitering objects from the heap of JVM. They generally take a snapshot-based approach by providing a detailed reference graph among the objects on the heap at a certain point during a program's execution [1]. Hopefully, by taking a series of snapshots on the heap, programmers may discover the creation of loitering objects by comparing the contents of two consecutive snapshots. The key to their success is that the programmer is able to identify the critical operation that generates loitering objects. This requires extensive knowledge about the application details or simply relies on the intuition. Sometimes it can be difficult or impossible to identify a critical operation in a sophisticated application; in particular, those Java applications running on the background without a GUI. Furthermore, even if the loitering objects are found, the programmer still needs to find out which other object is holding on to them in

order to solve the problem. Most profiling tools allow the object navigation from a loitering object to other objects through the associated references, which makes it easier to find out which objects are referring to the loitering objects. Nevertheless, knowing the referring objects does not necessarily mean that the programmer will be able to successfully locate the lines of code where unintended references are made. In most cases, it is a matter of luck and the profiling tools seem to help little.

Recently, we have been investigating the issues mentioned above and trying to develop a lightweight tool using AspectJ [2] for identifying loitering objects and further locating unintended object references in the code. Through its code weaving mechanism, AspectJ provides a good basis for building our tool mainly because it enables us to write the object activity monitoring code at a higher level of abstraction, namely in Java source program instead of bytecode or JVM heap (see Figure 1). We propose a two-stage approach to locate the unintended references in the code: first identifying the suspect classes of loitering objects and then tracing the references set to link those objects for locating those unintentional references.

In the first step, the numbers of object constructions and object destructions for each application class are counted, respectively. The object construction counts are easy to get with the help of AspectJ's constructor advice. To calculate the object destruction counts, we need to be notified of the object finalization event. This is achieved by making use of the introduction facility of AspectJ to insert a *finalize* method into those application classes. We weave in these object life monitoring aspects into a target application for calculating these counts[1]. After running the application for a period of time, the length of which is specified by the programmer depending upon the application scenario, the D/C (Destruction/Creation) ratio value for each application class is calculated. Apparently, an unexpected small D/C ratio value is a signpost of potential problems. Hence, given the D/C ratio values, the programmer should be able to identify the suspect classes of loitering objects.

The second step is further divided into two parts. In the first part, programmers choose some classes from the suspect classes and use our reference log builder to collect detailed reference set/unset records. In particular, we develop some logging aspects using the field set pointcut of AspectJ to record all the information about the references set to the objects of suspect classes, including the referred object, the referring object, the statement (line) in the code where the reference is made, and so on. The idea behind these aspects is that all unintended references are created through object fields. Furthermore, to avoid the performance degradation of the monitored application and therefore alter its behavior, we keep the aspect code as simple as possible to minimize the overheads that may be incurred by our reference logging

[1] These are static aspects that will consume only fixed amount space.

operations. Likewise, after running the application for a period of time, a considerable amount of information about object reference traces has been gathered and stored in the log file for off-line analysis. In the second part, an analytical tool is used to extract the reference patterns from the log file. With the reference patterns plus the precise location information describing where the references occur in the code, the programmer will be able to locate the code causing unintended references.

We have built a prototype version of our tools for testing the feasibility of our approach. They work fine with a few simple applications. However, we also realize that we must further enhance AspectJ to make our tools really feasible. Specifically, the field set pointcut of AspectJ does not expose the index value of an array field that is being set. But we need to get the index value while logging the reference set to an array element. A close look into the bytecode sequence for realizing the array element set instruction reveals that it is possible to expose these index values in the aspects woven into an application. Therefore, we are now investigating a way to enhance the AspectJ compiler to work around this problem.

References

[1] Wim De Pauw and Gary Sevitsky (2000), Visualizing Reference Patterns for Solving Memory Leaks in Java, *Concurrency: Practice and Experience*, Volume 12, Issue 14, Pages 1431 – 1454.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold, Getting Started with AspectJ, *Communications of ACM*, vol. 44, no. 10, pp 59-65, Oct. 2001. AspectJ website: http://www.eclipse.org/aspectj/

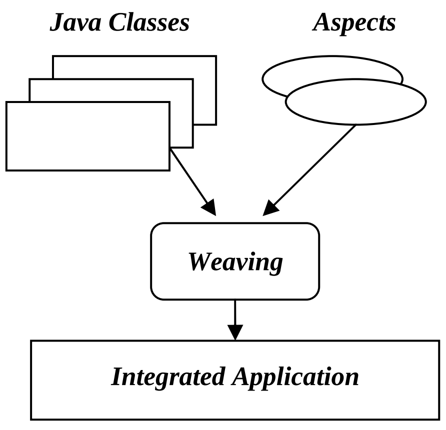

Figure 1: Aspect weaving in AspectJ