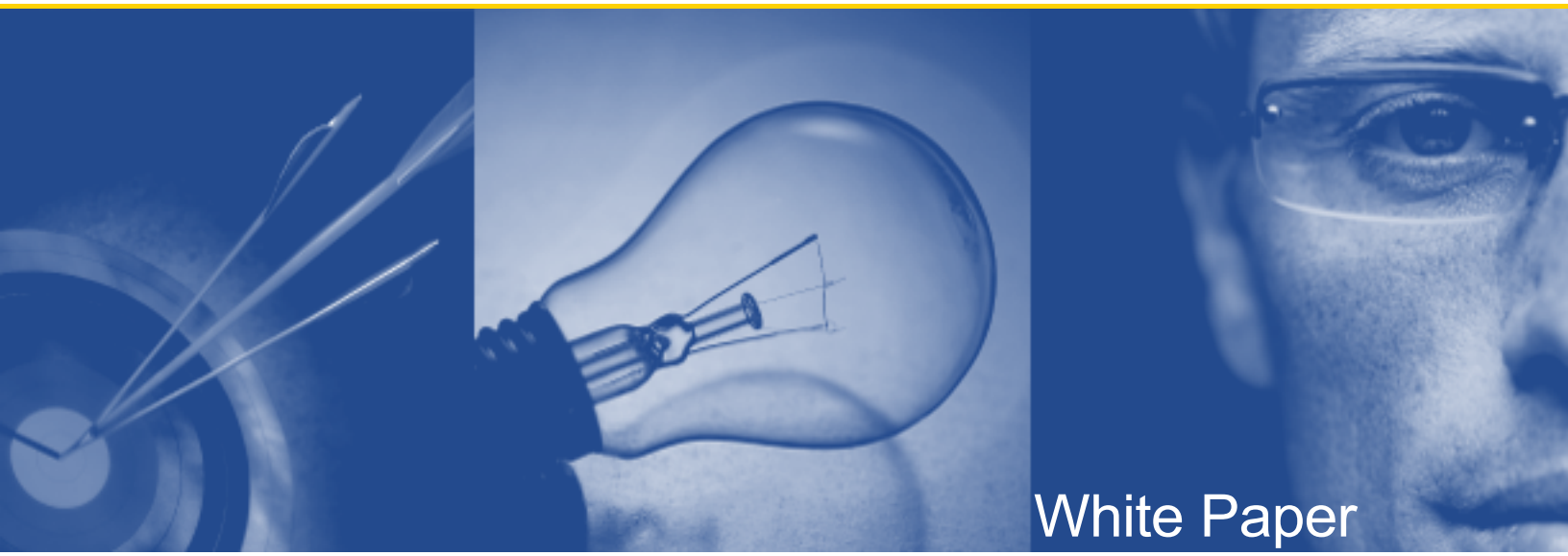
QUEST SOFTWARE®

Advanced Memory Analysis

Written by
Steven Haines,
CEO/Geek, GeekCap, In



WARRANTY

The information contained in this document is subject to change without notice. Quest Software makes no warranty of any kind with respect to this information. QUEST SOFTWARE SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTY OF THE MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Quest Software shall not be liable for any direct, indirect, incidental, consequential, or other damage alleged in connection with the furnishing or use of this information.

TRADEMARKS

World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
e-mail: info@quest.com

Please refer to our Web site (www.quest.com) for regional and international office information.

Created—April, 2009

CONTENTS

ABSTRACT

Java memory analysis is one of the most challenging disciplines in all of performance management because of the nature of Java memory leaks. As Java memory leaks are, in essence, a reference management issue, there is no hard and fast way to easily identify them. In C/C++, a memory leak is more definitive for a profiling tool to identify because orphaned memory is definitively a memory leak. Objects in Java that are growing boundlessly may or may not truly be memory leaks, but may behave as expected for the first few (or even first few hundred) megabytes.

In this paper, I delve into the world of Java memory leaks and present an easy-to-follow process that will help you navigate the Java Virtual Machine heap and find memory leaks. Then, by employing state-of-the-art tools, I demonstrate how to analyze that information, not only to identify objects that are leaking, but to understand **why** they are leaking, **who's** holding onto their references and **what** needs to be done to resolve them.

INTRODUCTION

Memory analysis, and specifically identifying the root cause of memory leaks, is one of the most challenging concerns that Java administrators, architects and developers face. Java memory leaks are not like their C++ predecessors, as illustrated in Figure 1.

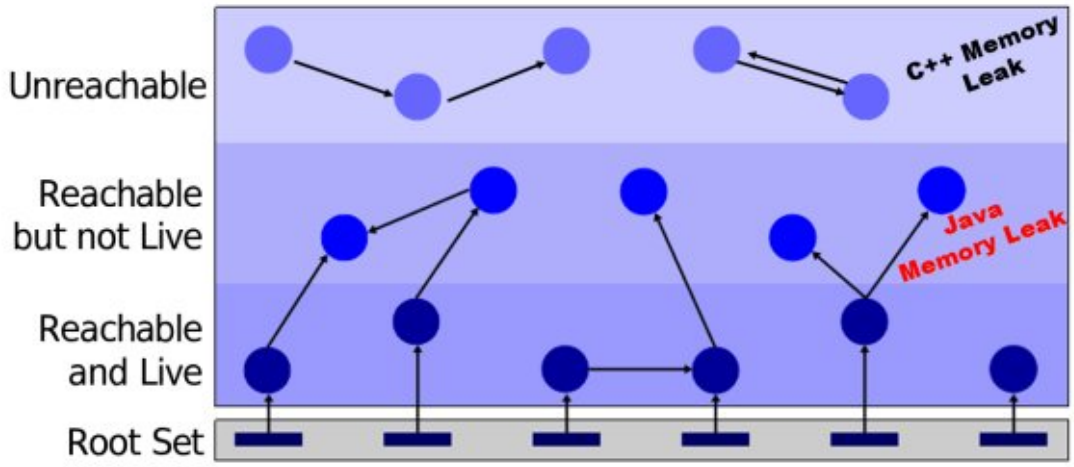


Figure 1 Java Versus C++ Memory Leaks

A C++ memory leaks occurs as follows:

1. An application allocates memory, such as through a call to malloc(), and assigns that memory to a variable
2. The variable is either set to null, assigned to another memory region, or goes out of scope without first releasing the memory, for example by invoking free()
3. The memory has been allocated, but not freed, so it has been “leaked” into the application’s process space

Java avoids these C++ style memory leaks through garbage collection by a process called the “reachability test”. The reachability test performs the following steps:

1. When garbage collection begins, the garbage collector identifies all objects in the “root set”, which essentially consists of all objects visible by all live threads
2. It traverses across all references maintained by each root set object
3. It repeats the process for all object references until it finds leaf objects (objects that do not reference any other objects)
4. During this process, it maintains an internal array that maps to all memory locations in the heap: when it finds a path from the root set to an object, it “marks” the memory spaces used by that object

5. Once the reachability test has completed, it “sweeps” the heap, or frees all memory locations that have not been “marked” by the reachability test
6. Finally, if the heap is measurably fragmented, it “compacts” the heap, or defragments it

In short, if there is not a path from the root set to an object, such as in the case of a C++ style memory leak, the memory is reclaimed by the Java Virtual Machine. So if the garbage collector eliminates C++ style memory leaks, then why are we so concerned about memory leaks in Java?

Garbage collection, and specifically the reachability test, is a double-edged sword: while it ensures that if there is not a path from the root set to an object that the object is collected, it also ensures that if there is a path from the root set to an object, that it is not collected. *And that is what constitutes a Java memory leak: a path exists from the root set to an object that the application does not intend to exist.*

Essentially, a Java memory leak is a reference management issue. All objects, except primitive types and wrapper classes, are references to objects, and actions like adding objects to Collections classes increases the number of references to an object. For example, if you create an object, add it to an ArrayList, and then delete your reference (by setting your reference to null), the ArrayList maintains a reference to your object. In most cases this is what you want (otherwise you would not have added it to the ArrayList), but the problem occurs if you never remove it from the ArrayList. It is common for an application to retrieve an object from a collection class by calling one of its get() methods, using the object, and then discarding the object, fully believing that the object will be deleted.

Unfortunately, the get() methods return new references to objects, so when the application deletes its reference, the collection continues to maintain a reference, which means that the garbage collector cannot reclaim it. Calling remove() on the collection class is the only effective way to remove an object from the collection class and delete its reference.

There are two approaches to addressing memory leaks:

- Proactive
- Reactive

Several of my other white papers on memory analysis have focused on the proactive approach: examine memory usage at a unit test-level and resolve memory issues as they are introduced into an application. This process involves capturing snapshots of a Java heap before and after a unit test runs and then analyzing the differences between the two snapshots. This is very effective methodology whose importance cannot be emphasized strongly enough. However, what should you do if you need to identify the root cause of a memory leak in an existing application?

If you are thrust into an environment that has a memory leak or your proactive measures did not detect the memory leak then you are forced to engage the situation using reactive strategies.

This paper explores memory leaks from a reactive standpoint: *it provides you with a strategy for identifying the root cause of a memory leak in an application that you may have already deployed to production.*

MEMORY ANALYSIS STRATEGY

It would be nice if you could simply point a tool at your production environment and analyze every object in your heap, including tracing information such as what line of code created it, what objects are referencing it and what objects it is referencing. Unfortunately, the overhead to gather all of this information in an application handling production load is too great, so you are going to need to recreate a memory leak in a development environment.

In order to recreate a memory leak in a development environment, you are going to need to reproduce a scaled down version of what your users are doing and subject your application to a light, but still significant, amount of load. Here is an outline of the whole process:

1. Analyze the behavior of your users
2. Construct load scenarios that mimic your users' behavior
3. Start your application server in a mode that supports performance analysis
4. Capture a snapshot of your heap (without tracing information)
5. Subject your application to a "light" version of the load scenarios
6. Capture another snapshot of your heap (to compare with the first one)
7. Enable tracing
8. Subject your application to the load scenarios using a single user running multiple times
9. Capture a heap snapshot (with tracing information)
10. Analyze the heap differences to find good starting points for analysis
11. Analyze the heap trace-enabled snapshot

In this section I review the pre-analysis steps (1 and 2) and in the next section I describe, in detail, how to run the test and perform the analysis.

Analyzing User Behavior

Before you can mimic what your users are doing, you need to fully understand what they are doing and in what proportion they doing it. I refer to this as constructing "balanced and representative" service requests. Balanced means that in the load you simulate, the proportion of different service requests matches that of your users. For example, if a user logs in once, reads 10 articles, and logs out, then your load test should balance the requests as follows: 1 login to 10 articles to 1 logout. Representative means that you are reproducing the top 80 percent or more of your users' actions. It is important to generate balanced and representative service requests because these are the actions that your users are performing to cause the memory leak.

Definition: a service request is any action that triggers your application, such as making a Web request or putting a message on a message queue.

Identifying user behavior is actually much easier than it may appear. If your application is Web-based and accepts standard HTTP requests, then your best option is to review access logs. Most Web servers and application servers provide a mechanism that can be enabled to log user requests. There are several commercial as well as free tools to help you analyze those logs. For example, Quest Software has a freeware product called Funnel Web Analyzer that can help you sift through access logs. The purpose is to sort user requests by their execution count to identify the top 80 percent of requests and then compute the balance of those requests based on their relative execution counts.

If your application is more complicated and you are not able to differentiate between service requests by their Uniform Resource Identifier (URIs) alone then you may need to employ an end user management solution. Foglight's end user management solution allows you to dissect requests based on the payload contained within a request.

Regardless of how you obtain the data, the important thing is that you need to obtain this information to be able to reproduce your users' behavior.

Constructing Load Scenarios

With balanced and representative service requests in hand, you are ready to construct load scenarios. There are several commercial and open source load testing tools – if your company has one then use it, if not, then consider an open source tool from Apache called JMeter. JMeter does a very good job of generating synthetic Web requests that either GET data from a server or POST data to a server. JMeter allows you to create load scripts manually if you know the URLs or you can record your own web actions using JMeter's proxy server. Finally, JMeter supports distributed load testing, which means that you can install JMeter on multiple machines and control all of them from a single workstation. I use JMeter in the example in the next section.

If you want more information about using JMeter, I wrote a detailed article series on Load Testing and specifically on using JMeter on InformIT.com.

During a load test in a development environment, I would not recommend increasing the load beyond 10 users—you can if you need to, but be aware that while a development machine is powerful, you may have dozens of other applications (such as your IDE) running and you do not want to bring your machine to a standstill. Additionally, most load testing tools allow you to configure "think time" between requests. This is the amount of time that you would expect a user to read your page before clicking on another link. If you ignore think time, then your 10 user load could easily crash your application.

MEMORY ANALYSIS IN PRACTICE

In this section I present a sample application based on the Spring framework and Spring MVC (Spring's Model View Controller Web architecture) that runs inside Tomcat and is experiencing a memory leak. The following outlines the process for identifying the root cause of the memory leak:

1. Configure Tomcat to work with Quest Software's JProbe
2. Start Tomcat and take a baseline memory snapshot
3. Load test the application
4. Take a subsequent memory snapshot
5. Enable code tracing, object allocation data, and garbage collection data
6. Load test the application with one user walking through the load scenarios five times
7. Capture a detailed memory snapshot
8. Analyze the results

Sample Application: GeekNews

The sample application is a mini Content Management System (CMS) called GeekNews. GeekNews presents a list of article summaries on the homepage and article content on a drilldown page, see Figure 2. It is built using Spring and Spring MVC and we recently added a caching layer that sits between the GeekNewsService bean and the underlying data access object to improve performance. The problem is that we outsourced the development of the caching layer to John, a recent college graduate, that, while very good at writing code, does not understand the potential problems with memory management. When the new cache-equipped GeekNews application was launched to production, within two days the application ran out of memory and we experienced the dreaded java.lang.OutOfMemoryError! Now we have to find it…

This application was built for JavaWorld.com in an article series I wrote on Spring MVC. If you are interested in learning about Spring MVC, or you just want to see how this application was built, please visit www.javaworld.com.

Figure 2 GeekNews Mini CMS Application

Introducing JProbe

JProbe is a code, memory, and coverage profiling tool developed by Quest Software that allows us to perform each of the aforementioned steps. It provides an extremely low overhead mode in which it does not trace any information, but still allows you to capture heap snapshots—which works perfectly for the load testing phase of our analysis. Furthermore, it provides very detailed memory analysis capabilities that allow you to trace an object down to the line-of-code that allocated it – which allows us to trace down the root cause of the memory leak.

Integrating JProbe with Tomcat is a simple task: when you start JProbe, choose "New Settings/Configuration" and add a new configuration for Apache Tomcat (and your version.) It will prompt you for your Tomcat installation directory and startup script (which is usually catalina.bat or catalina.sh) and then it will prompt you for your WAR file(s) so that it can configure filters from which to analyze your code. Next, you need to choose a JProbe Analysis type: memory, performance (code), or coverage – choose "Memory". The end result is that JProbe will create a new startup script for Tomcat that causes it to listen for JProbe connections and capture traces and snapshots. For example, in my environment it created a startWithJProbe.sh file that is a replacement for catalina.sh. Therefore it is launched as follows:

```
./startWithJProbe.sh start
```

The JProbe reference documentation is your authority on setting up these configurations, but Figure 3 shows a screenshot collage of the wizard pages to perform these steps.

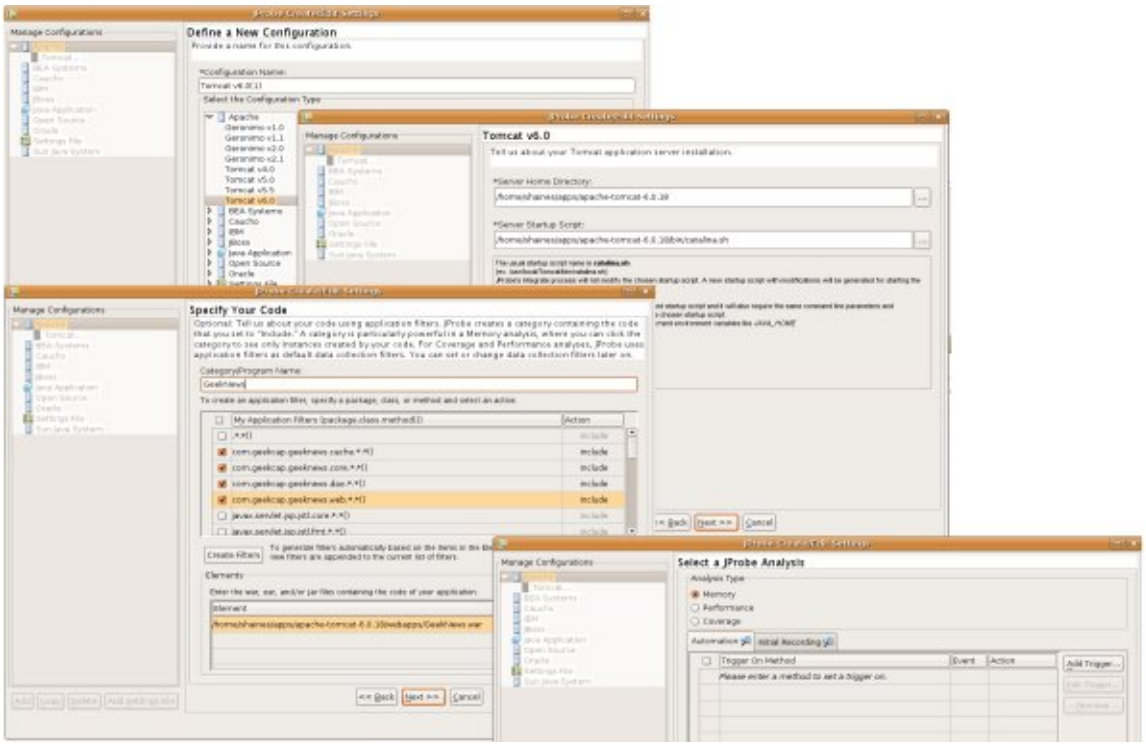


Figure 3 Setting up a JProbe Tomcat Configuration

Launch Tomcat with your modified script and attach to it with JProbe by pressing the "Attach to a Running Session" toolbar button:

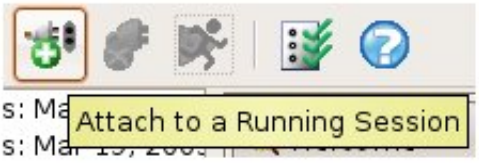


Figure 4 Attach to a Running Session Toolbar Button

Before starting the load test, ensure that data recording is off by pressing on the "Specify What Data to Record During the Session" toolbar button in the "Runtime Summary" tab (far right):

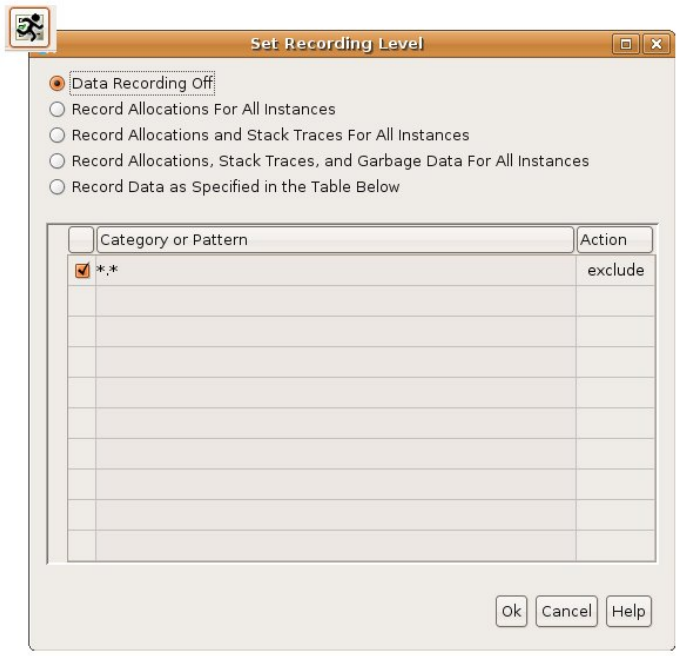


Figure 5 Telling JProbe not to capture data (yet)

Finally, capture a baseline snapshot from that same toolbar:

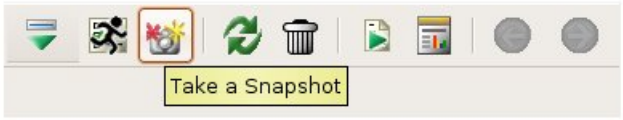


Figure 6 Take a Snapshot Toolbar Button

Performing the Load Test

With JProbe watching Tomcat with data recording turned off and a baseline snapshot, you're ready to start the load test. If you're following along with me using JMeter, Figure 7 shows two screen snippets: the load configuration to generate 10 concurrent users executing the service requests five times each and the summary report, showing the results of the load test.

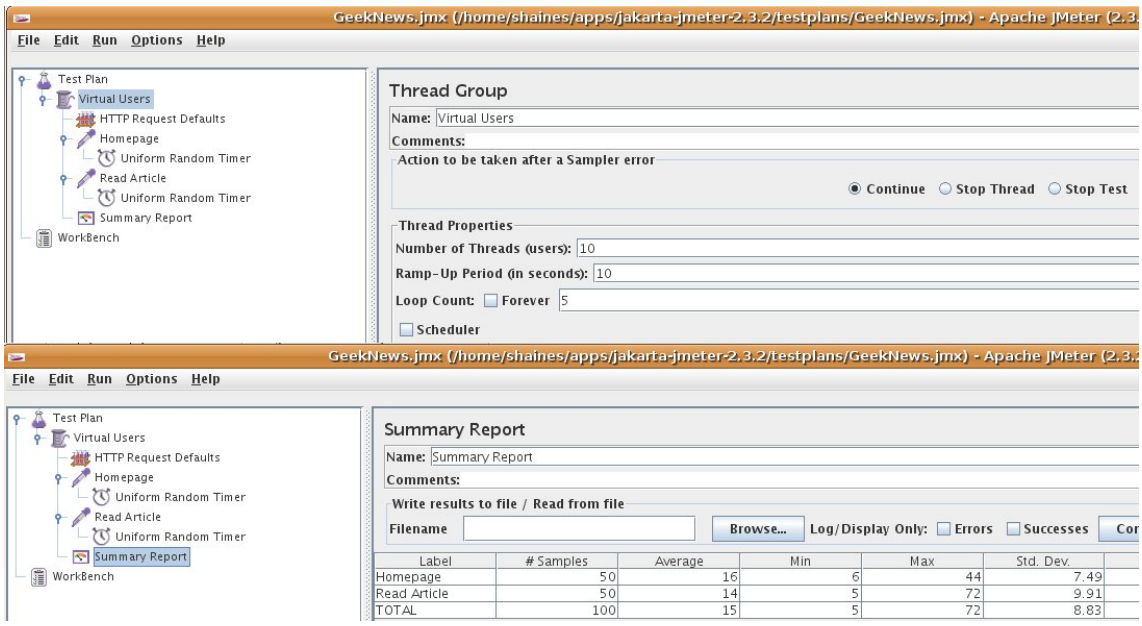


Figure 7 JMeter Running a Load Test

Recall that in the load test we want to generate a decent amount of load, but within the constraints of what a development environment can handle (one machine with JProbe, an IDE, Tomcat and a load tester.) You can, however, setup Tomcat on a separate machine and connect to it remotely with JProbe, so there is nothing stopping you from generating massive load. If the simple approach works, then great, but if you still cannot reproduce the memory leak on your local machine, then you may need to setup a distributed environment and connect remotely.

In this test, the 10 users executed two requests five times each, which resulted in 100 total samples. After this load test is complete, we need to do the following:

1. Capture another memory snapshot, which we'll use later for snapshot differencing
2. Reconfigure JMeter (or your load test tool) to launch the same test with one virtual user (set "Number of Threads (users)" to "1" in Figure 7)
3. Reconfigure JProbe to record detailed information by selecting "Record Allocations, Stack Traces, and Garbage Collection Data For All Instances" from the "Set Recording Level" dialog box (see Figure 5)

Analyzing the Results

After launching the load tester with one user, capture a final snapshot so that we can analyze the results. At this point you have two options:

1. Dive into the detailed snapshot and start hunting around
2. Calculate the difference between the first two snapshots so that you know where to start your investigations.

Hopefully the load test will call significant attention to the memory leak, but in case you need some help determining where the start, let's first ask JProbe to calculate the difference between the two memory snapshots that we captured during the load test (one before and one after the load test.)

To have JProbe compute the differences between two heap snapshots, right-click on the second one and choose "Snapshot Differencing…" It will prompt you to select a "Baseline Snapshot" from which to compare, so choose the first one and press "OK". JProbe will open the "Memory Difference" tab, which allows you to see the difference between the heap before the load test started and after it completed. Figure 8 shows an example of the snapshot differencing.

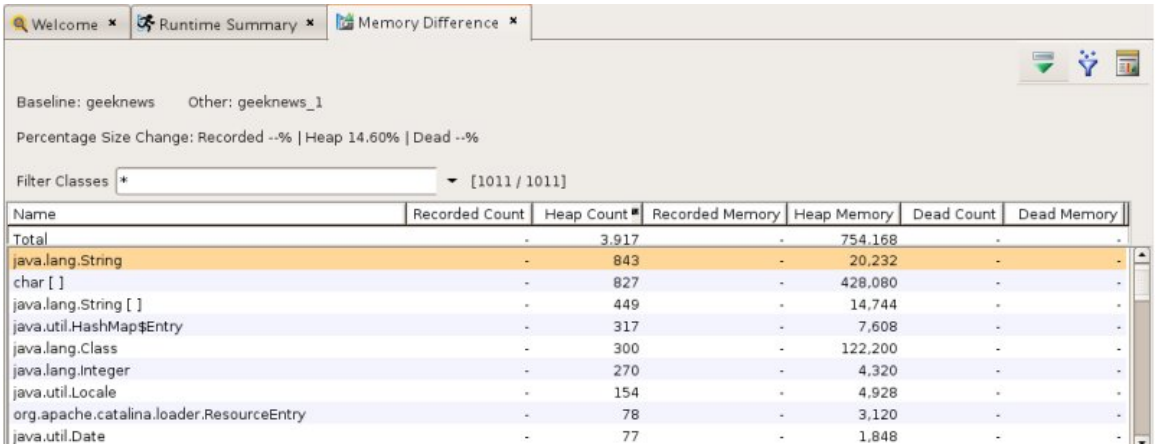


Figure 8 JProbe Performing Snapshot Differencing

The snapshot differencing can be sorted by the number of objects added to the heap ("Heap Count") or by the amount of memory used by those objects ("Heap Memory"). In Figure 8, there were 843 new Strings added to the heap, which accounted for 20K worth of memory. We'll get to it soon, but note that the 420K was occupied by character arrays is mostly consumed by Strings.

The challenge with the heap difference is that it shows everything in the heap, which can result in an unmanageable amount of data to analyze (1011 classes of objects in this small application.) It might be easier to hone in on objects that are in your application's package structure. A filter can be applied to the classes in the "Filter Classes" text box in the following form:

```
com.geekcap.*
```

Enter the root of your package structure, followed by an asterisk, and JProbe will display only your classes. Figure 9 shows how this affects the test application.

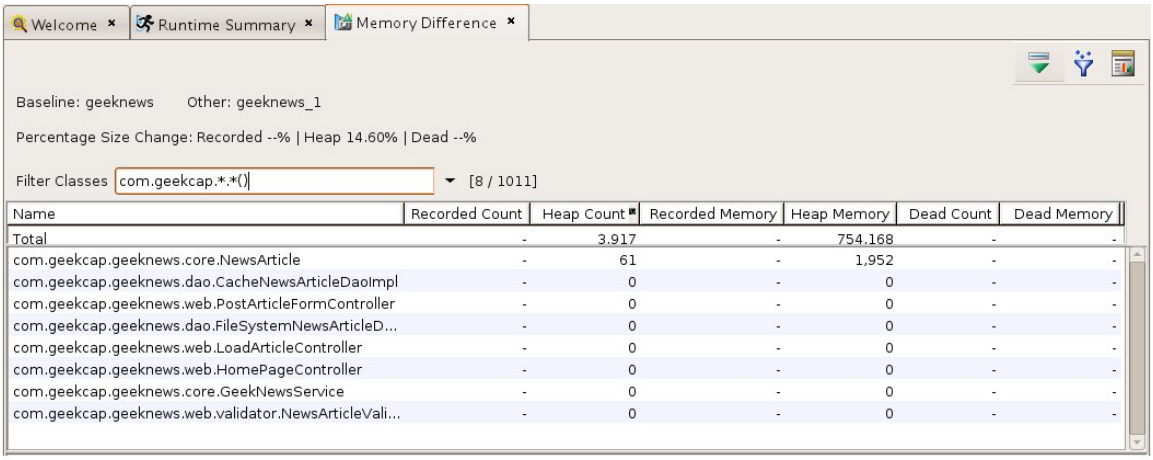


Figure 9 JProbe's Heap Differencing Filtering on GeekCap classes

Your applications will probably be much larger than this, but as you can see, the filter reduced the heap contents from 1011 classes of objects to eight. And one of those, the NewsArticle, looks suspicious. But before we jump to any conclusions, let's open the detailed snapshot and investigate this suspicious NewsArticle.

Right-click on the detailed snapshot and choose "Open Instances View", which is shown in Figure 10.

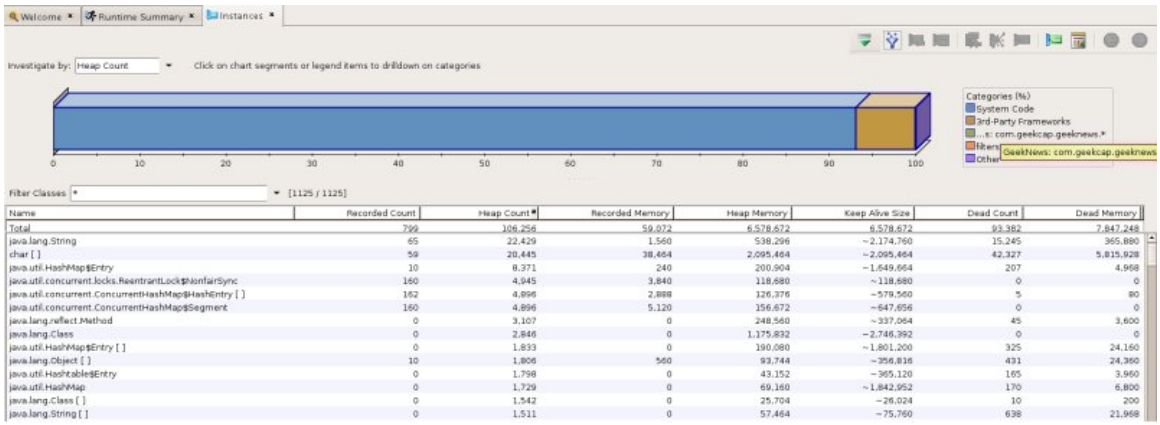


Figure 10 Heap Instances View

The Heap instances view shows you all classes of objects in the heap, including the "Keep Alive Size", which reports the amount of memory required to keep those objects alive. The keep-alive count differs from the heap memory in that it, not only includes the memory used by the class itself, but also includes the memory that its children occupy. Notice that the String class requires 538K of memory, but its keep-alive size it over 2MB: this is the result of the underlying character arrays (and other things) that it maintains internally.

You can choose to investigate the heap by the number of objects in the heap (which is the default) or you can investigate by the amount of used memory. The bar chart reports the "investigate by" metric (heap count in the example) grouped by the various components that JProbe identifies in your application. In this example, the majority of memory is occupied by system code, but if you hover over, and click on, any of the components in the legend on the right, the instances at the bottom of the screen and the bar chart will be filtered by that component. Again, sorting through 1125 objects is a daunting task, so click on your application's package (in my case "GeekNews com.geekcap.geeknews.*"), which is shown in Figure 11.

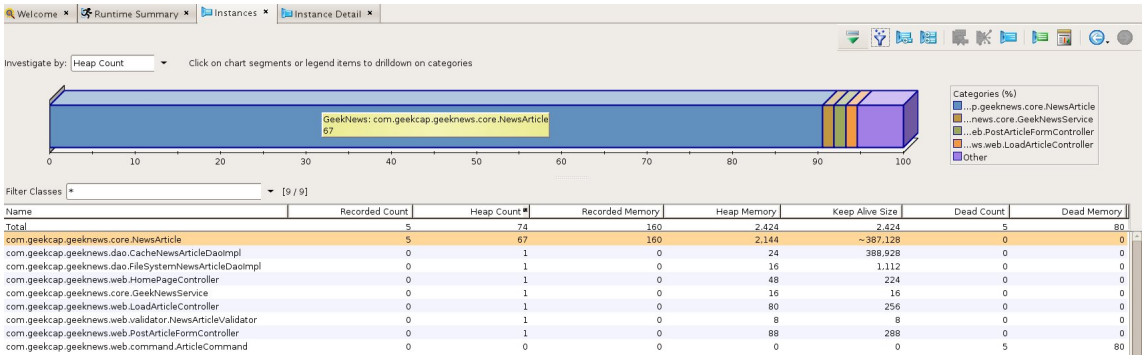


Figure 11 Heap Instances Filtered for GeekNews

Notice that there are 67 NewsArticle instances in memory accounting for 387K worth of memory. Your application will probably be more complicated that this, but the point is that you can, at-a-glance, see where your application is using all its memory, and by looking at the keep-alive count, you can see the impact of those objects on your application. And if you have a memory leak and you have load tested your application long enough, you should see it here.

The NewsArticle class looks suspicious, so it's time to investigate it. We'll learn more about this object later, but the pressing question is: why are these things in the heap? Right-click on the NewsArticle object in the table and choose "Why Are These Live?" This is shown in Figure 12.

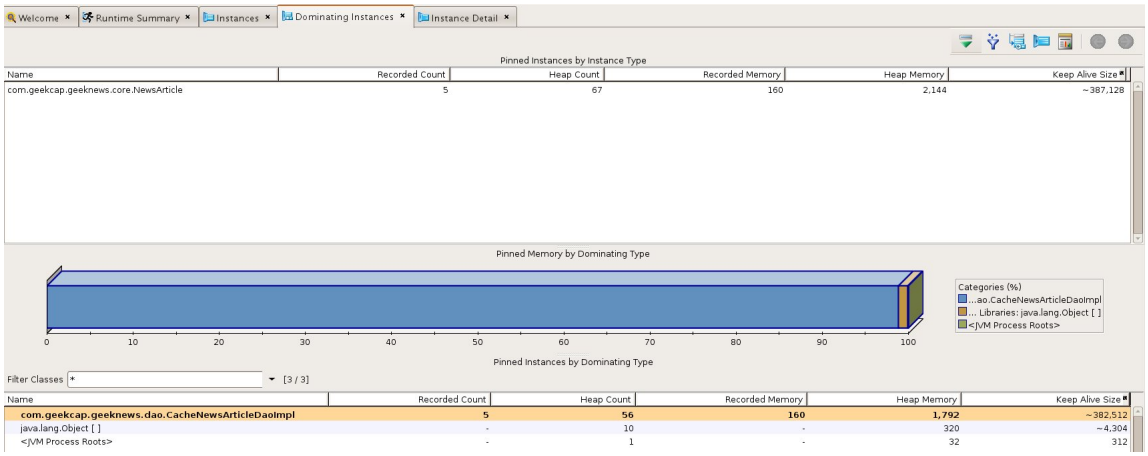


Figure 12 NewsArticle Dominator View

The Dominator view displays the list of objects that are maintaining references to the NewsArticle. In this example, the CacheNewsArticleDapImpl class is maintaining 56 copies, an Object array is maintaining 10 copies, and the JVM root set is maintaining one reference. The CacheNewsArticleDaoImpl is the apparent object that is leaking memory, so let's investigate the actual instances to see if we can figure out where the problem is. Right-click on the CacheNewsArticleDaoImpl object and choose "Show this Dominator's Pinned Instances", shown in Figure 13.

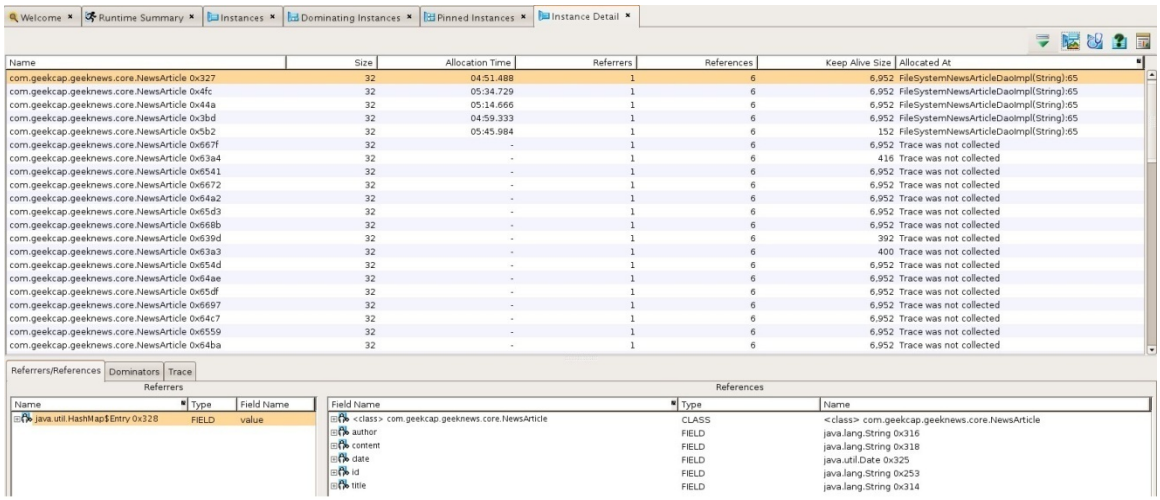


Figure 13 NewsArticle Instance Detail

You might have noticed that the "Recorded Count" was 5 while the "Heap Count" was 56. The reason is that during our load test we created 51 instances of the NewsArticle object, and we captured snapshots of those instances without tracing information. Then we launched the load test with a single user executing our service requests five times—those five instances account for the "Recorded" instances. Figure 13 has an "Allocation Time" for five of the instances, which are the times in which those instances, after the load test, were allocated. Those instances are the ones that we can use JProbe to trace down to the class, and line-of-code, in which they were allocated. The remaining instances are only useful to help us identify the memory leak.

Selecting one of the instances from the table updates the bottom sections with information about that particular object instance. Click on one that has an allocation time and then review the information on the bottom of the screen. The first tab reports the objects that are holding a reference to this instance as well as the objects that it is referencing. In this example, the NewsArticle instance is being referenced by a HashMap and it maintains references to four Strings and a Date, which represent the contents of an article.

The "Dominators" tab shows the object stack that is maintaining the reference to this object, which is shown in Figure 14.

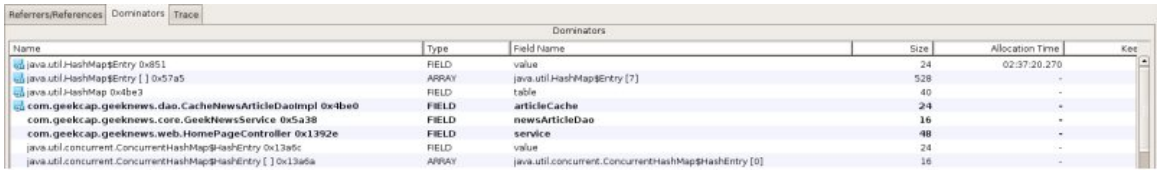


Figure 14 Dominators Tab

From this we can see that the HashMap is referenced by the CacheNewsArticleDaoImpl, which is referenced by the GeekNewsService, which is referenced by the HomePageController, which is ultimately referenced by Spring. And knowing the architecture of the application, the NewsService is a Spring service bean that is injected into the HomePageController.

Finally, the Trace tab reveals exactly where this object was created, shown in Figure 15.

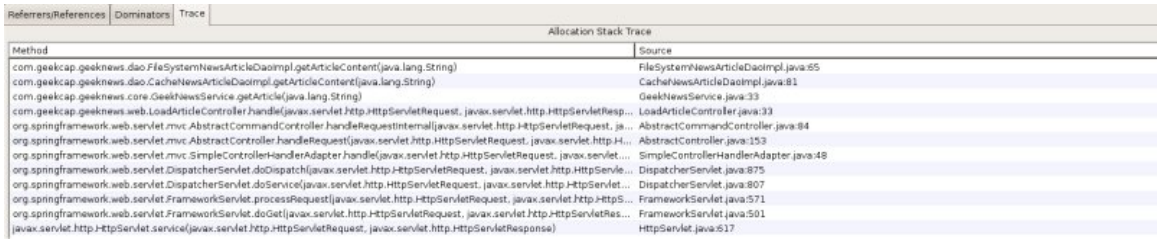


Figure 15 Trace Tab

The Trace tab presents the stack trace through which the object instance was created. If you want to review the source code that created it, right-click on the instance and choose “Show Allocated At Source”, which is shown in Figure 16.

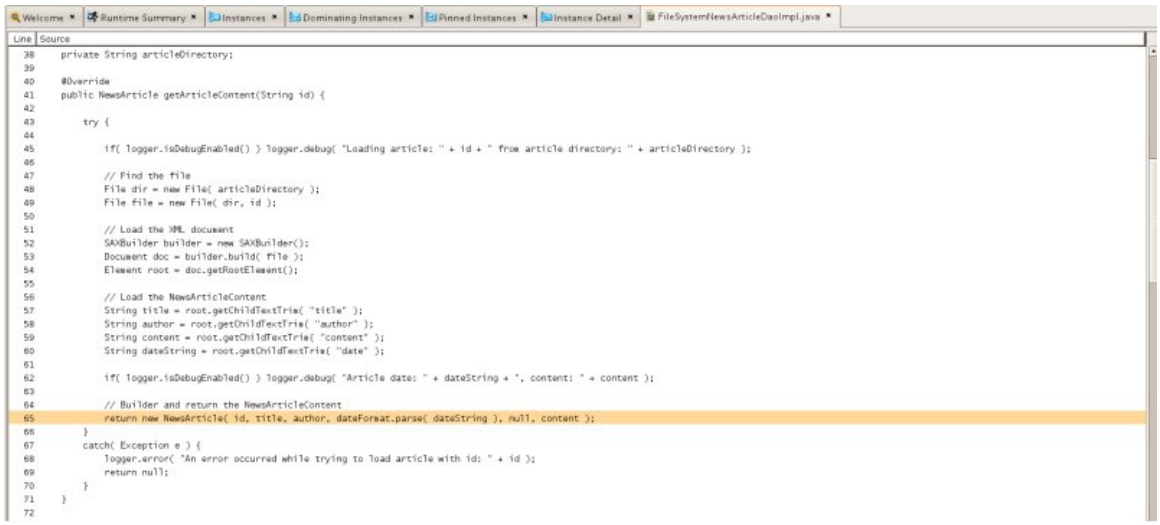


Figure 16 FileSystemNewsArticleDaoImpl java

Figure 16 shows that the getArticleContent() method loads an article from an XML file and then creates and returns a NewsArticle instance.

Summarizing Analysis Results

Reviewing the load test snapshots revealed that NewsArticle instances were growing in proportion with the number of requests being executed. Using this information, we tracked down the NewsArticle instances in the detailed heap snapshot and learned the following:

- A simple load test that exercised two service requests 50 times each yielded over 2MB of NewsArticle data
- NewsArticles are being created in the FileSystemNewsArticleDaoImpl class
- References to NewsArticles are being maintained by a HashMap that is contained within the CacheNewsArticleDaoImpl class – the cache is keeping them alive

This could be our expected behavior: articles should be cached for quick access. But from my understanding of the application I know that there is not even 2MB worth of articles in the application, let alone being accessed by the load test script. As with all performance analysis, and memory analysis in particular, you need to apply your own knowledge of your application and environment in order to determine what is and what is not problematic.

After investigating the cache, I found that John, our recent college grad, did not manage his cache keys properly. As a result each request for an article was not only loaded from the file system, but was added to the cache. I updated the cache code and re-ran the load test capturing before and after snapshots. The snapshot difference is shown in Figure 17.

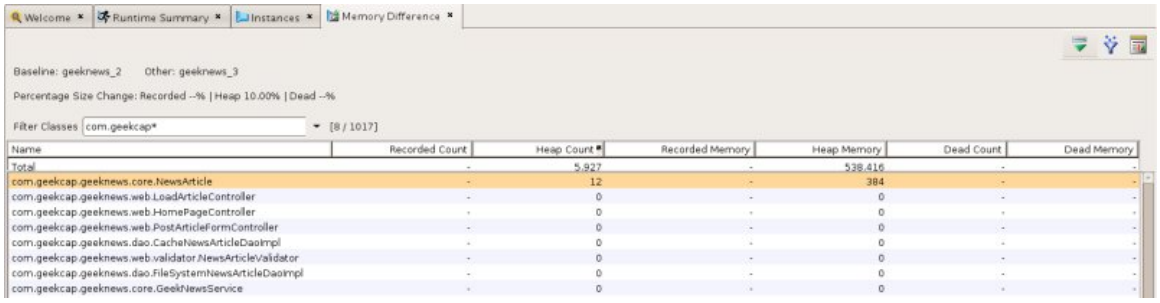


Figure 17 Cache Problem Resolved

If you compare this snapshot difference with the one in Figure 9, you'll notice that the change reduced the number of NewsArticles from 61 to 12. Memory leak resolved!

SUMMARY

Identifying the root cause of memory leaks is challenging at best and down right impossible at worst, but with tools like JProbe, the impossible becomes more attainable. In this paper, I reviewed how the Java Virtual Machine and garbage collection work and what constitutes a Java memory leak. From this knowledge I developed a plan for hunting down memory leaks.

Through a sample application with a memory leak, I proposed a plan and demonstrated how to use JProbe to realize that plan. I started by load testing the sample application using JMeter while JProbe silently watched, adding no observable overhead. JProbe captured two memory snapshots: one before the load test and one after. The difference between the two snapshots quickly identified objects that were growing disproportionately. Then, enabling JProbe's tracing capabilities and running a load test with a single user, I was able to capture a snapshot that revealed the relationships between objects, the amount of memory that they (and their children) were consuming, the objects were maintaining references to them, and their allocation points, down to the line-of-code. In short, JProbe's low-overhead memory analysis identified the objects that were leaking and JProbe's deep recording and analysis capabilities equipped me with everything I needed to be able to resolve the problem.

If you have a production memory leak, by using JProbe, you no longer have an excuse for not being able to find it!

About the Author

Steven Haines is the founder and CEO of GeekCap, Inc., which provides e-learning solutions for software developers, in the form of adaptive brain-based online courses for geeky subjects like Java Performance Tuning, Spring, Hibernate, as well as core computer science concepts such as data structures, algorithm analysis, and SQL. Previously he was the Java EE Domain Expert at Quest Software, defining the expert rules and views used to monitor the performance of enterprise Java applications and application servers. During his tenure at Quest Software he created a Java EE performance tuning professional services organization where he improved the performance of and resolved critical production issues for many Fortune 500 companies. He is the author of *Pro Java EE 5 Performance Management* and *Optimization* (Apress), *Java 2 Primer Plus* (Sams), and *Java 2 From Scratch* (Que), and shares author credits on *Java Web Services Unleashed*. Steven is the Java host on Pearson Education's online IT resource, Informit.com, where he posts weekly articles and blogs, he is a Java Community Editor on InfoQ.com, and he is a regular contributor to JavaWorld.com. Steven has taught all facets of Java programming at the University of California, Irvine and Learning Tree University. Steven can be reached at steve@geekcap.com.

ABOUT QUEST SOFTWARE, INC.

Quest Software, Inc., a leading enterprise systems management vendor, delivers innovative products that help organizations get more performance and productivity from their applications, databases, Windows infrastructure and virtual environments. Quest also provides customers with client management through its ScriptLogic subsidiary and server virtualization management through its Vizioncore subsidiary. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 100,000 customers worldwide meet higher expectations for enterprise IT. Visit www.quest.com for more information.

Contacting Quest Software

Phone:	949.754.8000 (United States and Canada)
Email:	info@quest.com
Mail:	Quest Software, Inc. World Headquarters 5 Polaris Way Aliso Viejo, CA 92656 USA
Web site:	www.quest.com

Please refer to our Web site for regional and international office information.

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract. Quest Support provides around the clock coverage with SupportLink, our web self-service. Visit SupportLink at http://support.quest.com

From SupportLink, you can do the following:

- Quickly find thousands of solutions (Knowledgebase articles/documents).
- Download patches and upgrades.
- Seek help from a Support engineer.
- Log and update your case, and check its status.

View the ***Global Support Guide*** for a detailed explanation of support programs, online services, contact information, and policy and procedures. The guide is available at: http://support.quest.com/pdfs/Global Support Guide.pdf