Performance Testing Methodology

Best Practices Documentation

written by
Steven Haines
Senior PSO Consultant, Java
Applications and Database Management
Quest Software, Inc.

Last revised: October 2005

TABLE OF CONTENTS

ABSTRACT

This document presents a detailed methodology that defines processes for effectively implementing performance testing in enterprise Java environments. It explains how to follow performance requirements from architecture artifacts through unit testing, integration testing and production staging. It details the process required to implement a formal capacity assessment that quantifies specifically when you need to add extra resources to your environment. Furthermore, it qualifies the necessary steps to validate that your test bed mimics the behavior of your end users.

Providing a complete solution, this document describes how Quest's Application Management Suite for Java and Portals integrates into this methodology to ensure your success at every stage of the application development lifecycle. Equipped with this methodology and Quest's Application Management solutions, you can be confident that your applications will meet performance criteria when presented to your users.

Finally, this document emphasizes the importance of automation by creating a repetitive test process that quickly reports on the quality of your application code. Only automation can assure that processes are properly followed and application components are tested both accurately and consistently.

INTRODUCTION

Performance testing has become an afterthought in the software development community. IDG Research reports that only 20 percent of production enterprise Java applications meet their performance requirements. If 80 percent of all production enterprise Java applications are failing to meet their service level agreements (SLAs), then significant effort needs to be made to determine why this is happening and to resolve this issue.

The key to making a transition to successful SLA fulfillment is the adoption of a formal **Performance Testing Methodology**. This document details the methodology and identifies the toolsets used at each testing phase to ensure the successful enterprise application performance.

There are two main categories of testing: functional testing and performance testing. This document is dedicated to performance testing and, as such, all references to testing in this document are assumed to be “**performance testing**” unless otherwise specified.

PREREQUISITES

Quantifying Performance Requirements

In order to quantify performance requirements, it is assumed that you will have defined SLAs. Key stakeholders should define SLAs systematically after a deep problem analysis.

The primary drivers of the SLA should be the **application business owner** and the **application technical owner**. The application business owner, who is sometimes the application product manager, analyzes the business cases and brings customer requirements to the SLA. In essence, as long as the SLA is satisfied, customer needs will be satisfied as well. The application technical owner, who is sometimes the application architect, analyzes the technical requirements necessary to solve the use case and brings a “reality check” to the SLA. The responsibility of the technical business owner, therefore, is to ensure that the service-level is attainable.

An effective SLA exhibits three key properties:

1. It is specific.
2. It is flexible.
3. It is realistic.

An effective SLA must be a specific value. Stating that a use case must complete in about five seconds is not definitive and therefore difficult to verify—5.25 seconds is considered *about* five seconds. It is an exact value that enables quality assurance to test with before moving the application to production. When the application is in production, it provides alert criteria for both active as well as passive monitoring.

An effective SLA must also be flexible in the context of its distributed variance. The use case must adhere to the specific SLA value for a predefined percentage of time, allowing for a measurable degree of flexibility in anticipation of unexpected conditions. For example, consider the popular search engine that you use on a daily basis. When you execute a search, do you consider it acceptable that your results are presented in less than two seconds 95 percent of the time? Are you willing to accept a seven second response time in one out of every 20 searches you perform? This level of variance is acceptable. However, if 10 out of 20 searches returned your results in seven seconds, there is a chance you would change search engines.

Not only must a SLA be specific, yet flexible, it must also be realistic. You can ensure this by requiring the SLA to be defined by **both** the application business owner and the application technical owner. This is called out specifically as a key property of an effective use case because most of the time, SLAs are defined solely by the application business owner—without the opinion of the application technical owner. When the technical team receives the performance requirements, they simply ignore them, but an unrealistic SLA is worse than none at all.

Know Your Users

The most important thing you can do to ensure the success of your tuning efforts is to take the time to get to know your users and understand their behavior inside your applications. Seldom do you tune your application servers in a production environment. Rather, you generate test scripts representing virtual users, execute load tests against a pre-production environment and tune it. When your pre-production environment is properly tuned, you can then safely move the configuration information to production.

Most corporations cannot adequately reproduce production load on a pre-production environment. If you work for one of these companies, do not lose hope. Most of the larger companies that I visit do not have a firm understanding of their user behavior and cannot generate representative load on their test environments.

There are two common excuses that I hear:

1. Production load is too large for pre-production.
2. I do not have any way of knowing what my end users are really doing.

To address point number 1, we can build a scaled-down version of production in pre-production and scale up the configuration of our production deployment. It is not as effective as having pre-production mirror production, but sometimes it is not affordable to do so.

To address point number 2, I will show you how you can gather end-user behavior in the following section.

Because we try to tune our environment in pre-production to validate settings before moving them to production, it naturally follows that we are tuning our environment to support the load test scripts that are executed against the environment. The process of tuning an enterprise application is to first implement some best-practice settings, load test the application, observe its behavior and adjust the configuration parameters appropriately. It is an iterative process where we try to hone in on the optimal configuration settings. Some changes will yield improvements, and some will actually degrade performance. That is another reason why performance tuning should not be left until the end of a development lifecycle—it is time-intensive.

Given that we tune our application servers to our load scripts, what does that tell you about them? It tells you that load scripts really need to represent real-world user behavior. Consider tuning a Web search engine. I can write a test script that searches for apples and bananas all day, but is that what end users are doing? I can tune my environment to be the best “apples and bananas” search engine in the world, yielding world class performance, but what happens when someone searches for BEA or IBM? In my application, I could have grouped technical companies in a separate database from fruits and vegetables, so that piece of code would never be executed in pre-production and my tuning efforts would be in vain.

The better solution is to determine the top 1000 or 10,000 search phrases and their frequencies. Then, compute the percentage of time that each is requested and build test scripts that request those phrases in that percentage. For the remaining percentage balance, you might connect the load-test generator to a dictionary that queries for a random word.

The difficult part of writing user-representative load scripts is the process of discovering how your users are using your applications. It is not an exact science, but for reasonably reliable results, the first step is to look at your access logs. Now, I would not recommend doing this by hand, because the task is insurmountable for even a Web application of medium size. There are plenty of commercial or free tools that will analyze your access logs for you.

They will show you the following about your service requests:

- Sort service requests by percentage of time requested and display that percentage.
- Zoom in and out of your analysis time period to present finer or lesser granular results.
- Identify peak usage times of the day, week, month and year.
- Track bytes transferred and mean time of requests.
- Identify and categorize the originators of requests against your application (internal, external, geographic location).
- Summarize the percentage of requests that were successful.
- Summarize HTTP errors that occurred.
- Summarize customer loyalty, such as return visitors and average session length.
- Track page referrals from other sites.

Regardless of the software that you choose to analyze your access logs, it is important that you perform the analysis and use this information as a starting point for building your test scripts. Access logs are somewhat limited in what they report and may not suffice in certain instances, such as when you use a single URL as the front controller for your application and differentiate between business functions by embedded request parameters. In this case, you need a more advanced tool that can monitor your usage and partition business functions by request parameters.

Access logs give you part of your solution. The next step requires a deeper understanding of the application itself. For example, when a particular service request is made, you need to know the various options that control the behavior of that service request. The best sources of that information are application use cases and the architect responsible for that functionality. Remember that the goal of exercise is to identify real-world user behavior, so your research needs to be thorough and complete. Errors at this stage will lead to the aforementioned “apples and bananas” search engine anomaly.

For a more holistic approach to capturing a more reliable and detailed understanding of end-user behavior, you may want to consider Quest's User Experience Monitor (UEM). UEM sits between your end users and your Web servers. It captures, **in real-time**, every single request that passes though your environment. It provides the most comprehensive set of data describing exactly what the real users are doing, including: connection speeds, desktop browser versions and aggregation by geography/domain. All of this is provided with zero overhead to the application via passive network sniffing technology.

Before leaving this subject, you should know that the biggest mistake that I see my customers make in defining load test scripts: ***users do not know how to log out of your system***. I do not care how big you make your logout button; at most, 20 percent of your users are going to use it. I believe that this behavior is the result of the late adoption of the Web as a business deployment platform. Commercial Web sites dominated the Internet through its emergence and mass growth.

As such, users became accustomed to exiting a Web site in one of two ways:

1. Leaving the current site and traversing to another.
2. Closing the browser window.

Because this nature is ingrained in their Web usage patterns, you cannot depend on users to properly log out of your Web site. Therefore, when you develop test scripts, you need to determine the percentage of users that log out properly and the percentage that do not, and then develop your test scripts accordingly. One large-scale automotive manufacturer that I worked with struggled with this problem for more than a year. Their application servers crashed every few days so they became accustomed to simply rebooting their application servers nightly to reset the memory. After interviewing them and looking at their HTTP Session usage patterns, we discovered an inordinate number of lingering sessions.

We reviewed their load test scripts and sure enough, each test scenario included the user properly logging off. They tuned their environment with this supposition and when it was not correct, their tuning efforts could not account for the amount of lingering session memory. They adjusted their test scripts, retuned their environment, and have not been forced to restart their application servers because of "out-of-memory" errors since.

PERFORMANCE TESTING PHASES

Performance testing must be conducted at the following specific points in the development lifecycle:

- Unit Test
- Application Integration Test
- Application Integration Load Test
- Production Staging Test
- Production Staging Load Test
- Capacity Assessment

Current software engineering methodologies break the development effort into iterations. Each iteration specifies a set of use cases that must be implemented. The typical pattern is that the first iteration implements the framework of the application and ensures that the communication pathways between components are functional. Subsequent iterations add functionality to the application and build upon the framework established during the first iteration.

Because iterations are defined by the use cases (or sections of use cases) that they implement, each iteration offers specific criteria for performance testing. The use cases define additional test steps and test variations to the SLAs quality assurance should test against. Therefore, all of the following performance test phase discussions should be applied to each iteration. The controlling factor that differentiates the work performed during the iteration is the set of use cases.

Unit Tests

Performance unit testing must be performed by each developer against their components prior to submitting their components for integration. Traditional unit tests only exercise functionality but neglect performance.

Performance unit testing means that the component needs to be analyzed during its unit test by the following tools:

- Memory profiler
- Code profiler
- Coverage profiler

The memory profiler runs a garbage collection and records a snapshot of the heap **both** before the use case begins and after it completes. From this, we can see the memory impact of the use case and the list of specific objects left in memory by the use case. The developer needs to review those objects to ensure that they are supposed to stay in memory after the use case terminates. If objects are inadvertently left in the heap after the use case completes, then this represents a Java memory leak and we refer to these as **loitering objects***,* sometimes referred to as **lingering object references**.

The next memory issue to look for is referred to as **object cycling**. Fine-grained heap samples recorded during the use case, combined with creation and deletion counts, show you the number of times an object was created and deleted. If an object is created and deleted rapidly, then it could be placing too much demand on the JVM. Each object that is created and deleted can only be reclaimed by a garbage collection, and object cycling dramatically increases the frequency of garbage collection. This typically happens with the creation of an object inside of a loop or nested loop.

Consider the following:

```
for( int i=0; i<object.size(); i++ ) {
   for( int j=0; j<object2.size(); j++ ) {
      int threshold = system.getThreshold();
      if( object.getThing() – object2.getOtherThing() > threshold ) {
         // Do something
      }
   }
}
```

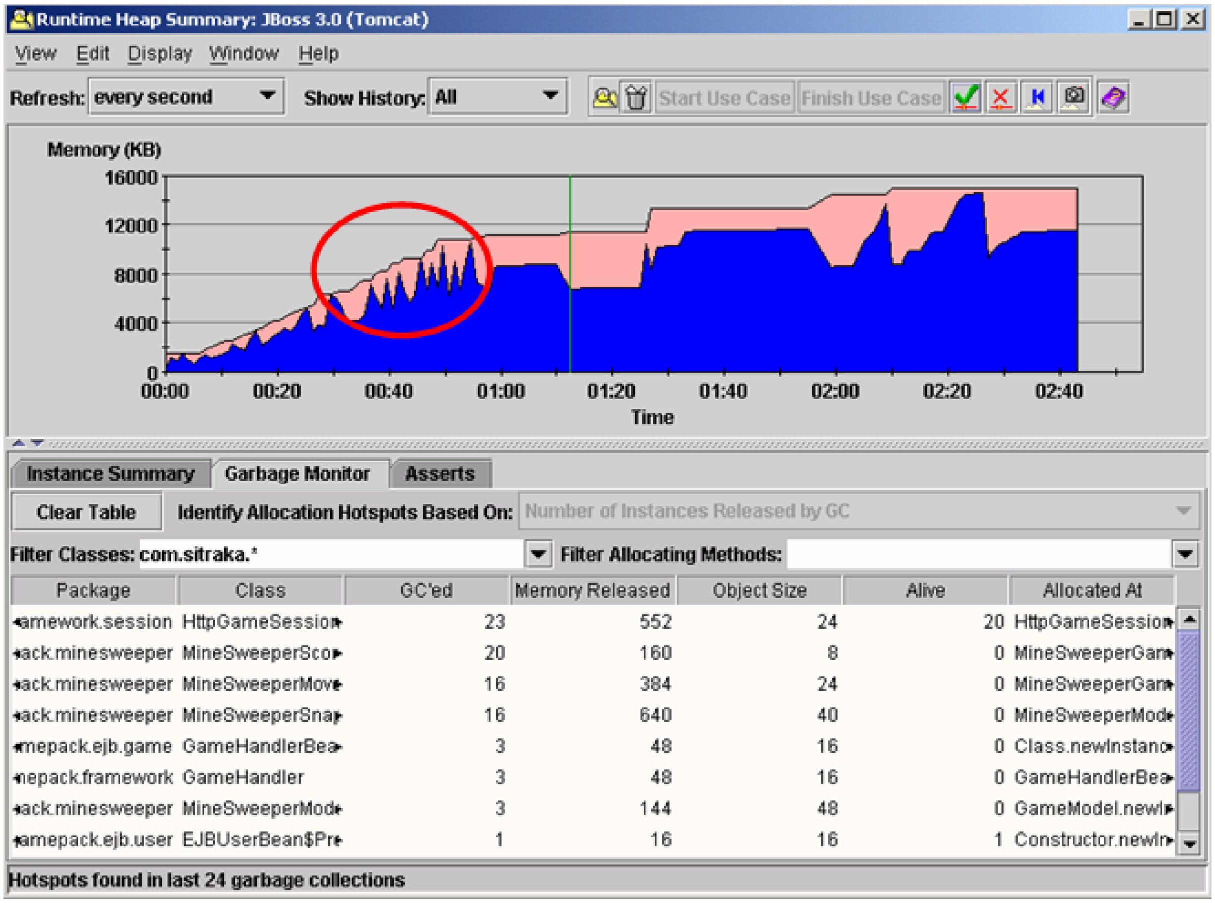
In this case, the outer loop iterates over all of the items in object, and for each item it iterates over the collection of object2's items. If object contains 1000 items and object2 contains 1000 items, then the code defined in the inner loop will be executed 1000 * 1000 times, or 1 million times. The way that the code is written, the threshold variable is allocated and destroyed every time the inner loop runs (it is destroyed as its reference goes out of scope). If you look at this code inside a memory profiler, you will see one million threshold instances created and destroyed.

The code could be rewritten to remove this condition by writing it as follows:

```
int threshold = system.getThreshold();
for( int i=0; i<object.size(); i++ ) {
   for( int j=0; j<object2.size(); j++ ) {
      if( object.getThing() – object2.getOtherThing() > threshold ) {
         // Do something
      }
   }
}
```

Now, the threshold variable is allocated once for all one million iterations. The impact of the threshold variable went from being significant to being negligible.

One other common scenario where we see object cycling in Web-based applications is in the creation of objects inside the context of a request. On an individual basis, it is not problematic, but as soon as the user load increases substantially, the problem quickly becomes apparent. The decision that you have to make is whether the object needs to be created on a per-request basis, or if it can be created once and then cached for reuse in subsequent requests. If the answer to this question is the latter, then you can stop that object from cycling. Figure 1 shows a visualization of the heap when object cycling occurs.



Refresh: every second

Show History: All

Identify Allocation Hotspots Based On: Number of Instances Released by GC

Filter Classes: com.sitraka.*

Filter Allocating Methods:

Package	Class	GC'ed	Memory Released	Object Size	Alive	Allocated At
amework.session	HttpGameSession	23	552	24	20	HttpGameSession
ack.minesweeper	MineSweeperSco	20	160	8	0	MineSweeperGam
ack.minesweeper	MineSweeperMove	16	384	24	0	MineSweeperGam
ack.minesweeper	MineSweeperSnap	16	640	40	0	MineSweeperMode
mepack.ejb.game	GameHandlerBea	3	48	16	0	Class.newInstanc
nepack.framework	GameHandler	3	48	16	0	GameHandlerBea
ack.minesweeper	MineSweeperMode	3	144	48	0	GameModel.newI
amepack.ejb.user	EJBUserBean$Pre	1	16	16	1	Constructor.newIn

Hotspots found in last 24 garbage collections

Figure 1: Object cycling can be identified visually by looking at a fine-grained heap sample. The circled region of the heap points to a time when memory was rapidly created and freed, which indicates the potential that objects are cycling.

Application Integration Test

After components have been through unit tests and deemed satisfactory to add to the application, the next step is to integrate them into a single application. The integration phase occurs at the conclusion of each iteration, and the primary focus is determining if disparate components can function together to satisfy the iteration use cases. After functional integration testing is complete and the application satisfies the functional aspects of the use cases, then next step is to run performance tests against the integrated whole.

This test is not a load test, but rather a small scale set of virtual users. The virtual users are performing the functionality that we defined earlier: attempting to simulate end users through balanced and representative service requests. The user load for the test is defined and documented in the Performance Test Plan by a joint decision between the application technical owner and the application business owner. The purpose of this test is not to break the application, but to identify application issues such as contention, excessive loitering objects, object cycling and poor algorithms, which can occur in any application when it is first exposed to multiple users.

In addition, to identify functional issues within the application that result from load and obvious performance issues, this test is the first one that holds the use case to its SLAs. If the application cannot satisfy its use case under light load, then there is no point in subjecting it to a load test.

Application Integration Load Test

Now that the application is properly integrated, has passed all of its functional requirements and has been able to satisfy its SLAs under a small load, it is time to execute a performance load test against it. This test is a full load test with the number of projected users that the application is expected to eventually support in production.

This test should be performed in two stages:

1. Executed with minimal monitoring.
2. Executed with detailed monitoring.

In the first test, the goal is to see if the code holds up to its SLAs while under real levels of load. When we deploy the application to production, it will have a minimal amount of monitoring enabled. In this first test, we give the application every chance to succeed.

In the second test, we enable detailed monitoring, either for the entire application or in a staged approach (with filters to only capture a subset of service requests), so that we can identify performance bottlenecks. Even applications that meet their SLAs can have bottlenecks. If we identify and fix them at this stage, then they do not have the opportunity to grow larger in subsequent iterations.

This phase of the performance test plan represents our first opportunity to performance tune the application. This is quite a change from traditional approach of waiting to tune until the application is finished. We are now trying to tune our application when its functionality is simplistic. If we build our application on a solid foundation, we ensure success.

Production Staging Test

Our performance tuning and management task would be greatly simplified if our applications could always run in isolation, where we had full use of application server, operating system and hardware resources. Unfortunately, it is expensive to add hardware and software licenses for each new application that we develop, so we are forced to deploy our applications to a shared environment. This means that while our integration load tests helped us tune our applications, we still need a real-world testing environment that will mimic a production deployment.

This imposes quite a task on quality assurance. Not only do they need to manage test scripts for your applications, but also for all applications running in the shared environment. It is imperative that quality assurance implements an automated solution that produces repeatable and measurable results.

Just as with the Application Integration Test, this is not a load test, but rather a test to identify resources that applications may be competing for. The load is minimal and defined in the Performance Test Plan. If contention issues arise, then deep analysis is required to identify the problem. But this is the very reason that the test is automated and performed by adding one component at a time. When your application arrives in this test bed, the test bed has already passed this test in the past, so the problem can be isolated to something in your application or something in your application in conjunction with another application. Either way, your application is the only change between a working test bed and a failing test bed, which presents a good starting point for problem diagnosis.

Production Staging Load Test

When it finally appears that your application has successfully integrated into the shared environment, it is time to turn up the user load to reflect production traffic. If your application holds up through this test and meet its SLAs, then you can have confidence that you are headed in the right direction. If it fails to meet its SLAs in this test, then you need to enable deeper monitoring, filter on your application's service requests and identify new bottlenecks.

It is important to realize that simply dropping your new application into an existing tuned environment is not sufficient. Rather, you need to retune the new environment to continue supporting the existing applications and load. This may mean resizing shared resources such as the heap, thread pools, JDBC connection pools, etc.

Capacity Assessment

When you've finally made it to this stage, you have a very competent application iteration in your hands. This final stage of performance testing captures the capacity of your application. In this stage, you generate a load test on the entire environment, combining the expected usage of your application with the observed production behavior of the existing environment. In other words, you start with the Environment Load Test and then start scaling the usage up in the same proportion as the Environment Load Test. All the while, you are testing for all SLAs.

You continue to increase the load slowly until the system resources saturate, throughput begins to degrade and response time increases dramatically. During this test, you record the load at which each use case exceeds its SLA and then pay close attention to the response time of each use case. It is important to know the rate at which performance degrades for each use case as it will feed back later into capacity planning.

The capacity assessment gives you the total picture of your application (and environment) so you can assess new architectural considerations. Furthermore, recording capacity assessments on a per-iteration basis (and correlating them) provides insight into both application code that was added at any specific iteration and measures the capabilities and growth of your development team.

PERFORMING A FORMAL CAPACITY ASSESSMENT

Many people throw the terms “capacity assessment”, “capacity planning”, “trending” and “forecasting” around without really understanding what they mean. When someone says, “capacity planning”, they are typically referring to the time that their applications cease satisfying their SLAs and they are forced to buy more hardware.

I have been “preaching” for a long time that additional hardware is usually not the right solution, though if you have enough funds, it is an effective one. By proactively implementing a systematic methodology to learn the capacity of your environment, you can avoid this reactive troubleshooting and make educated decisions that are specific to your environment before problems impact your end users.

A capacity assessment is more than a load test. You need the following components before you are ready to perform a capacity assessment:

- **Balanced Representative Service Requests**: you need to know your users. Specifically you need to know what they do and in what percentage (balance) they do it.
- **Hard SLAs**: you need to explicitly define SLAs for your key service-requests.
- **Expected Load**: you need to know the number of simultaneous users that your application needs to support. This includes their typical behaviors such as think time in order to setup your load tests appropriately.
- **Graduated Load Generator**: you need a load generation application that will climb up to your expected load in a reasonable amount of time and then slowly crawl up.
- **SLA Evaluation**: this functionality can be built into your load generator or be accomplished through synthetic transactions, but the focus is on monitoring the response time of your service requests against their respective SLAs.
- **Resource Utilization Monitor**: you capture the performance of application server and operating system resource utilizations to determine resource utilization saturation points as well as which resources that give out first. This information can help you in your tuning efforts to determine where you need to perform more tuning.

With all of these capabilities in hand, it is time to start loading your application. Configure your load generator to generate your expected usage in a reasonable amount of time (it could be as short as 10 minutes or as much as an one hour or more depending on the user behavior that you have observed in your production environment). While you are increasing load to the expected usage, capture the response time of your service-requests and evaluate them against their SLAs.

Once you reach your expected user load, it is time to determine the size of the steps that you want to monitor. The size of a step is the measurable increase in user load between sampling intervals—it defines the granularity of accuracy of your capacity assessment. Consider that your expected user load is 1000 users—you might define a step as 25 or 50 users. Pick a time interval in which to increase steps and record the response times of your service requests at these intervals.

Continue this pattern for each service request until the response time of each exceeds its SLA. Note this time and start recording response times at a tighter interval. The purpose for increasing the sampling interval is to better identify how a service request degrades after it has reached its capacity. From these degrading figures, we want to attempt to plot the response times to determine the order of the degradation: is it a linear, exponential or worse? The point is that we need to understand the implications of missing or incomplete SLAs.

For example, if we miss our SLA at 1500 users, but only increase our response time by 50 percent over the next 500 users, it is better than if our response times triple every 100 users and then the entire application server crashes at 1800 users? This helps us understand and mitigate the risk of changes in user behavior.

For each service request, we compile this information and note the capacity of our application at the lowest common denominator: the service request that first consistently misses its SLA. The next section in the capacity analysis report describes the behavior of the degrading application. From this report, business owners can determine when they require additional resources.

While this is going on, you need to monitor the utilization of your application server and operating system resources. You need to know the utilizations of your thread pools, heap, JDBC connection pools, other back-end resource connection pools (e.g. JCA and JMS), and caches, as well as CPU, physical memory, disk I/O and network activity.

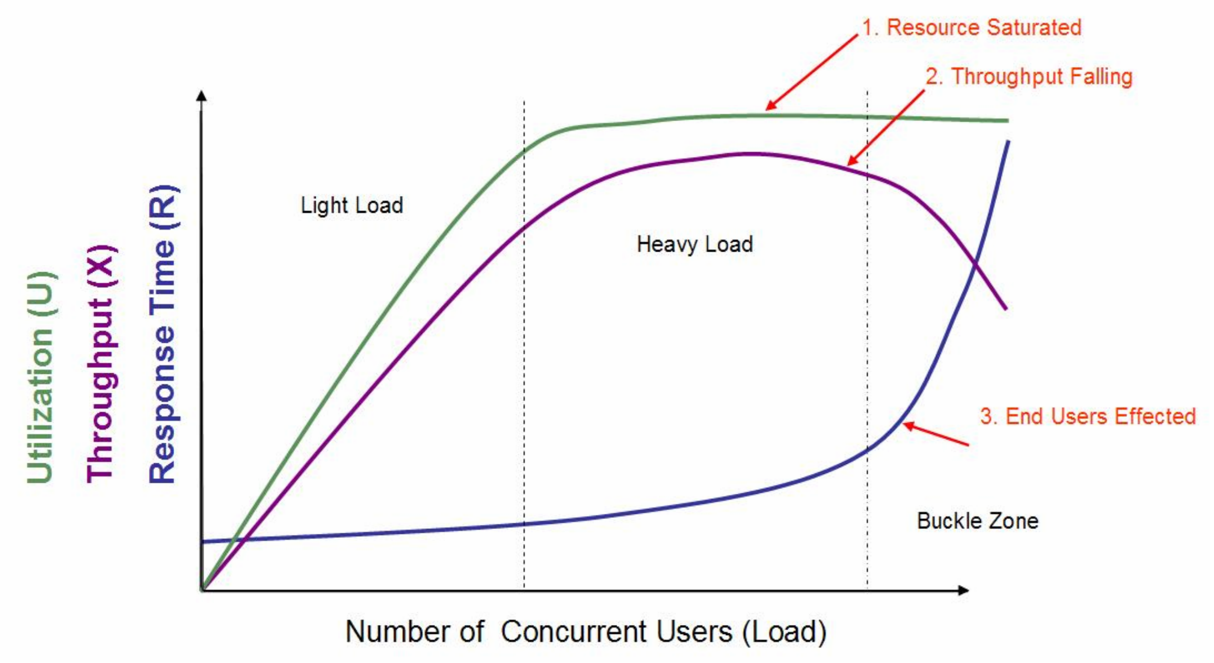


Figure 2: The relationship between user load, service request response time, and resource utilization.

Figure 2 relates the user load with service request response time and resource utilization. You can see that as user load increases, response time increases slowly and resource utilization increases almost linearly. This is because the more work you are asking your application to do, the more resources it needs. Once the resource utilization is close to 100 percent, however, an interesting thing happens – response degrades with an exponential curve. This point in the capacity assessment is referred to as the **saturation point**. The saturation point is the point where all performance criteria are abandoned and utter panic ensues. Your goal in performing a capacity assessment is to ensure that you know where this point is and that you will never reach it. You will tune the system or add additional hardware well before this load occurs.

A formal capacity assessment report therefore includes the following:

- The current/expected user load against the application
- The capacity of the application under balanced and representative service requests
- The performance of key service requests under current load
- The degradation patterns of each service request
- The system saturation point
- Recommendations

After you have gathered this data and identified the key points (e.g. met SLA, exceeded SLA, degradation pattern, saturation point), it is time to perform deeper analysis and generate recommendations. Try to classify your application into one of the following categories:

- **Extremely Under-utilized System**: system can support greater than 50 percent additional load.
- **Under-utilized System**: under current/expected load, all service requests are meeting their SLAs and the system can easily support more than 25 percent additional load.
- **Nearing Capacity**: the application is meeting its SLAs but its capacity is less than 25 percent above current load.
- **Over-utilized System**: the application is not meeting its SLAs.
- **Extremely Over-utilized System**: system saturated at the current/expected load.

In the extremely under-utilized system, you may consider reducing hardware and servers to conserve licensing costs. This determination can only be made after interviews with application business owners to determine if the additional capacity is required.

In the under-utilized system, you can sleep well at night as your environment can support any reasonable additional load that it might receive, but it is not so under-utilized that you would want to cut resources.

In the nearing capacity system, you need to spend some hard time with the application business owner to determine expected changes in user behavior, forecasted changes in usage patterns, planned promotions, etc. to decide if additional resources are required.

In the over-utilized system, you need more resources. But at this point, it is still the decision of the application business owner. How badly is the application missing its SLAs? What is the degradation pattern? Is the current application behavior acceptable? Are projected changes in usage patterns going to significantly degrade application performance?

In the extremely over-utilized system, you have undoubtedly received user complaints and are in that state of utter panic that I mentioned earlier. You need significant tuning and possibly additional resources such as hardware to save your users.

Performing a capacity assessment on your environment should be required for all of your application deployments and should occur at the end of significant application iterations. A proper capacity assessment captures the performance of your application at the current load, the capacity of your system (when the first service request exceeds its SLA), the degradation patterns of your service requests and the saturation point of your environment. From this information, you can generalize conclusions that trigger discussions about modifying your environment. Without a capacity assessment, you are running blind, hoping that your applications won't fall over with the next promotion or during the holiday season. Urge your management to permit this exercise. I can assure you that the resulting calm nights of sleep will more than make up for this argument.

INTEGRATING TOOLS INTO A FORMAL PERFORMANCE TESTING METHODOLOGY

Quest Software's Application Assurance Suite for Java and Portals provides specific functionality that covers code quality and optimization, performance and availability and change and configuration management.

This paper is concerned with providing the capability to deliver optimized and bug-free code to production, both in a proactive sense and as a response to troubleshooting issues in production. Quest's Application Assurance Suite for Java and Portals consists of two products: JProbe® and PerformaSure™. They integrate to provide development teams with a complete performance and memory testing solution.

Below, I will describe how each component of the Application Assurance Suite for Java and Portals integrates into the above-mentioned phases.

Unit Test

The unit test domain is owned by JProbe, which provides profilers in each of the required categories:

- Memory Profiling
- Code Profiling
- Coverage Profiling

JProbe can be scripted to analyze the performance of component code and generate resultant reports that detail the quality of the code. Or it can be used interactively. It is suggested that developers use JProbe interactively after each major change to the code base to test for memory anomalies (lingering object references and object cycling), as well as to visually identify long running algorithms. Furthermore, the coverage profiler should be included to ensure that the entirety of the code is, in fact, being tested.

JProbe's scripting features can be integrated into various build technologies (such as Ant) and it is suggested that the code is subject to immediate and automated testing immediately prior to released to integration. The purpose is to collate reports for examination prior to the onset of significant integration efforts. This helps ensure that the integrated solution is assembled from working components.

All outsourced components need to run through a similar automated analysis. This is even more important than in-house code because in-house code can be more easily periodically reviewed and the environment controlled.

Integration Test

Upon the successful completion of functional integration testing, performance integration testing should be completed using PerformaSure. At this stage, the user load is very minor in comparison to projected production usage. Therefore, it is suggested that PerformaSure be configured to record the behavior of the entire application at a full detail level. If this requirement is not feasible as a result of the size and complexity of the application, then a staged approach may be employed.

The staged approach to integrating PerformaSure into a large scale or high traffic application is as follows:

1. Execute the test with PerformaSure configured to capture all service requests at a component level with very coarse-grained samples (e.g. five minute samples).
2. This session will reveal the general performance of the application, identifying the major enterprise Java components being utilized, along with their respective response times. From this list, you can partition poor performing service requests into groups of five.
3. For each group of five identified service requests, configure PerformaSure to record at full detail with average granularity (e.g. 30 seconds or one minute samples), but with explicit filters only to include the identified service requests.
4. If the granularity for a specific service request is still too coarse, then configure PerformaSure to record with a finer granularity (e.g. 10 seconds) filtered on a specific service request.
5. If the root cause of a slow running service request is determined to be the result of application code issues, then a JProbe Launcher file can be exported for the offending class at line-of-code level.
6. JProbe can then record and profile the offending class to expose its performance, as well as the performance of its children (classes and methods that it calls), to resolve the performance problems.
7. Repeat until all problems are resolved.

The result of the performance integration test should be either a green light to move to performance integration load testing, or a set of poor performing service requests with pinpointed root causes of their respective performance anomalies.

Integration Load Test

As mentioned above, Quest's Application Assurance solutions are clustered into three sets—code quality and optimization, performance and availability and change and configuration management. Production support teams are traditionally dependent on performance management solutions. However, this methodology is relying on an expanded set of integrated tools in order to be as thorough and proactive as possible.

Quest's Performance Management Suite for Java and Portals, part of the performance and availability solution set, consists of Foglight® and PerformaSure. The Foglight component provides 24x7 unattended monitoring of all key service level indicators, while PerformaSure provides transaction diagnostics across a distributed environment. In this integration, the monitor – Foglight – launches the diagnostic tool – PerformaSure – when a performance issue occurs in the Java production "stack," in order to pinpoint the source.

During the performance integration load test, the goal is to ramp the application up to projected user load. Therefore, the performance impact of monitoring software must be mitigated. The tools of choice for this test are Foglight and PerformaSure running in a staged capacity. The initial tests should be executed with Foglight only and the Response Time Agent configured to launch PerformaSure to gather detailed information about service requests that exceed their SLAs. Offending requests should then be analyzed and the test repeated until it is passed.

If the entire test fails to meet its SLAs, then the aforementioned staged PerformaSure implementation needs to be employed to identify the root causes of performance problems.

Production Staging Test

The production staging test mimics a production deployment in which your application runs in a shared environment with other applications, potentially competing for resources. Again, as with the integration load testing, the tools of choice for this test are Foglight and PerformaSure. Foglight provides a very low overhead solution that records historical information and identifies slow running service requests and depleting resources—it is appropriate for a long running test. Finally, as performance problems are identified, PerformaSure is the tool of choice for diagnosing the root causes (a problem that cannot be definitively diagnosed using PerformaSure may be taken to JProbe).

The intelligent configuration of PerformaSure can mitigate performance impact and manage session size. As such, all attempts should be taken to utilize PerformaSure's detailed monitoring capabilities throughout the production-staging test. Intelligent configuration includes configuring things like:

- **Varying Degrees of Monitoring Levels**: for example, component-level recording can provide deep diagnostic information at a lesser overhead than detailed-level recording.
- **Longer Sampling Time Slices**: PerformaSure aggregates performance metrics on a configurable interval; so shorter intervals provide finer granularity of diagnostics, but increase overhead. Longer intervals can used to lessen the overall impact.

- **Reducing Sampling Overhead**: PerformaSure has the ability to reduce its impact on your environment dynamically. By default, it is configured to capture as many samples as possible, but it can be configured to capture fewer samples to mitigate overhead.
- **Custom Components**: through a regular expression class-matching engine, PerformaSure can group classes together into named components. For example, all classes that start with org.apache.struts are automatically classified as "Apache Struts". You are free to define custom components for each of your major application components, such as: logging classes, auditing classes, data persistence classes, etc. When recording at a component level, you are still able to glean detailed diagnostic information.

We have worked with many customers to utilize the advanced PerformaSure tuning configuration parameters to enable multi-hour sessions that span the entirety of the production/ staging test.

Production Staging Load Test

The environment or staging load test mimics production in deployment topology as well as user load. This environment should be monitored as though it was production using the following tools.

Foglight with the following cartridges:

- Application Server Cartridge
- Foglight Transaction Recorder (FTR)
- Synthetic Transaction Player
- Response Time Agent

System agents, database agents, any external dependency agents, appropriate Spotlights and PerformaSure configured to record detailed information on slow running transactions (either from FTR or the Response Time Agent).

This is a long-running test that intends to reproduce the production environment, so post mortem needs to include an analysis of the historical data captured by Foglight. This analysis needs to identify trends in resource utilization, as well as identify potential subtle memory leaks.

Capacity Assessment

The capacity assessment is the most taxing test that the application environment will be exposed to. As such, it needs to maintain minimal monitoring overhead. Monitoring must not be removed from the environment. Monitoring will always be running in a production environment, and removing monitoring would invalidate the results of a capacity assessment and may create a false sense of security in user adaptation.

The monitoring suite mirrors that of the Environment Load Test. The purpose is to run the test in a realistic environment, but with enough monitoring to identify the exact resources that first saturate and those that impact the degradation model.

Production Testing Validation

An important aspect of performance testing is properly understanding your usage patterns. As previously mentioned, performance tuning is only valid for the test bed to which it is tuned—if the test bed is flawed then the performance tuning is invalidated and the capacity assessment is unreliable.

Therefore it is of the utmost importance that proper usage patterns be identified. From a coarse level view, access log analyzers can provide aggregate information detailing user behavior. From a fine level view, Quest's User Experience Monitor (UEM), mentioned earlier in this document, provides definitive analysis of both user behavior as well as the user experience. All applications that run in a production environment need to be monitored using UEM and the results fed back into the formal Performance Test Plan to determine the coverage and effectiveness of test scripts. UEM can also be run in the Environment Load Test and compared to production to extract the efficiency and applicability of test scripts.

AUTOMATION

The only sure-fire way to ensure that performance tests are always conducted consistently is to automate the process at each test phase.

Automation in Unit Tests

Process documentation may require that developers implement performance testing in their unit tests. When time is at a premium, testing is the first step to be skipped. Functional testing must take a higher priority, so performance testing may be bypassed completely in lieu of a couple hours of relaxation spent browsing the Web.

This methodology proposes automated generation of performance reports against unit tests. The developer implements his/her component and writes unit tests (using something like JUnit or Cactus). Then, in the build process, the unit tests can be replayed against the application running in a scripted version of JProbe. JProbe then generates reports detailing the performance of the component code (Code Profiling), the impact of the component on memory (Memory Profiling) and the breadth of the unit test (Coverage Profiling.)

These reports can be reviewed to ensure that the unit test is, in fact, exercising a high percentage of the code and that there are not any gross memory or algorithm errors. JProbe integrates into Ant to allow this automated process to be tightly integrated into build procedures.

Designate a core individual on each development team who is given the responsibility and authority to:

1. Review the automated performance testing report output generated as part of each final build for transition from unit level testing to integration testing.
2. Then, make a “pass/fail” decision based on that report. Individual developers are then be engaged if the performance report indicates any significant issues with line-level timings, memory usage or insufficient unit level test code coverage.

These reports should then be archived for future reference including difference comparison from one release cycle of the code to the next.

Automation in Integration Tests

A load generator drives automated integration tests. The primary differences between the integration test and the integration load test are:

1. The amount of load.
2. The level of monitoring.

PerformaSure has a command-line interface that can be triggered by your load generator to start and stop session recording. This allows PerformaSure sessions to be recorded by the load generator, eliminating the need for manually starting and stopping session recording, as well as manually configuring recording nodes and recording criteria (filters, components, etc.). This ensures that as integration tests are performed, the resulting sessions are consistent and prime for comparison.

Automation in Production Staging Tests

Production staging tests mimic a true production environment, and when under significant load, certain levels of monitoring may be prohibitive to the performance of the application. As such, the best mechanism to capture relevant diagnostic information is to configure Foglight, its response time agent and its transaction playback agent (Foglight Transaction Recorder), to automatically start and stop PerformaSure recordings with context when it observes a performance abnormality. Foglight can be configured to start and stop PerformaSure session recordings using the abovementioned command-line interface. When the application is effectively servicing its requests and meeting its SLAs, then deep-level diagnostics may not be required. However, when a request fails to meet its SLAs, deep-level diagnostics are of paramount importance.

As a standard part of the Integration and Production Staging Test cycles:

1. Record a PerformaSure session for each set of load tests, using the same recording criteria that were used during the similar performance testing activity for the previous release cycle of the code.
2. Open the matching PerformaSure recording session from the previous release cycle and visually compare that side by side with the new recording taken using the current version of the code.
3. Compare the longest running service requests to identify any significant, undesirable changes in request response time.

When all anomalies have been resolved, archive the final PerformaSure recording session for future reference.

BEST PRACTICES TROUBLESHOOTING: MEMORY ISSUES

Java provides robust memory management through advanced garbage collection algorithms, but even the best memory management algorithm cannot prevent human error. Back in the days of C/C++ programming, memory management was in the hands of the programmer and as such they became very cognizant of the implications of not properly de-allocating memory. If memory was not properly de-allocated, it was lost for the duration of the application. In some cases, an inappropriate pointer could lead to an operating system crash. Java, therefore, created the notion of a virtual machine and a "sandbox" in which its applications would run. Because Java manages the memory, it can assure that memory requests stay within the virtual machine. When objects can no longer be referenced, Java can automatically reclaim that memory. The end result is that operating systems are safer from rogue applications and C/C++ memory leaks are avoided. The byproduct, however, is that modern Java developers that do not come from a programming background that required strict memory management, and are therefore not as cognizant of memory issues.

While C/C++ style memory leaks are avoided in Java Virtual Machines, there are two problems that commonly occur in Java applications:

- Object Cycling
- Lingering References

Every time a garbage collection occurs, the garbage collector must determine what objects are valid (or currently referenced) and free those objects. The process is referred to as "mark and sweep". The garbage collector traverses the heap and marks objects that are currently in use and then sweeps away the dead objects. Some garbage collection algorithms follow this phase with a "compact" phase that compresses the heap memory for best performance and future allocation.

This points to two inherent limitations to these procedures:

1. If objects are rapidly created and destroyed then the garbage collector will be required to run more frequently.
2. If object references are not properly destroyed then the garbage collector cannot free the objects.

We refer to the first problem as **object cycling** and the second as **lingering references**.

Object Cycling

Object cycling can be detected in the heap either through a visual tool that samples memory very frequently (once a second or less). You will observe very narrow choppy marks in the heap or through garbage collection logs. The key indicator to look for when reading garbage collection logs is frequent minor collections that reclaim small amounts of memory.

Consider that the performance impact of garbage collection on your application is the result of two factors:

- The frequency of garbage collection.
- The duration of each garbage collection.

Object cycling impacts the frequency of garbage collection. You can therefore detect it by calculating the frequency of garbage collection. You can enable verbose garbage collection reporting in your Java Virtual Machine by passing the Java Runtime Engine (java.exe on Windows or java on Unix/Linux) the following command line parameter:

```
-verbose:gc
```

Other options will help you identify heap performance at a deeper level, but this option provides enough information to empower you to calculate the frequency and type of garbage collections.

For example:

```
30.971: [GC 11241K->3574K(130176K), 0.0124588 secs]
31.691: [GC 11766K->3936K(130176K), 0.0104734 secs]
32.536: [GC 12128K->4349K(130176K), 0.0073750 secs]
33.009: [GC 12541K->5472K(130176K), 0.0175936 secs]
33.862: [GC 13664K->6202K(130176K), 0.0115339 secs]
35.999: [GC 14394K->7062K(130176K), 0.0151237 secs]
```

In this scenario, minor garbage collections are running every couple seconds and reclaiming relatively small amounts of memory. This indicates that the application may be cycling objects. Now, the difficult part is determining what objects are cycling and how to avoid this behavior. JProbe allows you to profile your code while it is running and report back the number of times an object is created. These creation counts can help you quickly identify where problems exist.

Once you have identified objects that are cycling, the next step is to determine if those objects do, in fact, need to be created each time or if they can be cached. In a Web application, this may mean storing the value in the Session or Application context (ServletContext is used for holding objects in application scope). For standalone applications, this may mean storing objects in memory variables or in your own caching infrastructure. The core software development concept here is proper object lifecycle management—defining object life cycles inside your use cases (when an object is created, how long it lives, and where it is destroyed) will help you manage this issue.

Loitering objects (Lingering object references)

While object cycling involves creating and destroying objects rapidly, loitering objects refers to creating objects and never destroying them. In other words, you leave a lingering reference to that particular object. This happens somewhat frequently when using Java collections classes. When you are finished using an object that is in a collection, you must be sure to remove it from the collection.

This problem is compounded by that fact that loitering objects do not usually tend to be individual objects, but rather a sub-graph of objects. For example, consider leaving a reference to a car in memory. It is not just a car. It also includes an engine object that includes a radiator, alternator, muffler, etc. So while a single reference may not seem significant, the repercussions can be substantial.

In order to locate loitering objects, you must visually observe the behavior of your heap and look for an upward trend, analyze garbage collection logs or use an advanced memory profile tool. When visually observing the heap, the natural pattern of the heap is to have peaks and troughs as objects are created and destroyed. Most applications follow a memory growth pattern (that could be hours) that climbs to a critical mass and then oscillates between peaks and troughs, such that the troughs remain relative constant. If your application is “leaking memory” and continually holding on to object references, then when your application reaches its critical mass, the heap will continue increasing until you run out of memory. The key is to watch the troughs and try to plot a line through them. After reaching what you believe to be the critical mass, if the line you plot through the troughs is increasing, there are probably references that your application is not properly freeing.

Garbage collection logs can be very helpful here too. When detecting loitering objects, instead of looking for rapid minor collections, look for frequent occurrences of major garbage collections that are for the most part ineffectual. The typical pattern is to see major collections occurring more frequently, freeing less memory and taking longer to execute.

Detecting lingering references is relatively easy. Let your application run and if it runs out of memory. If it does, then there is a strong possibility that you are not properly managing object references. Identifying the objects that are lingering in memory is a far more difficult task. There are two options: review your entire application (line-by-line) until you find out where the problem is or use a memory-profiling tool. JProbe allows you to profile the memory usage of your application. Specifically you perform the following steps:

1. Start your application inside the memory profiler.
2. Run a garbage collection to clean up memory.
3. Start a use case.
4. Perform your business case.
5. End the use case.

6. Run a subsequent garbage collection to remove all temporary objects from memory.
7. Look at the differences between the heaps to see what objects are left in memory.

The information that you acquire by running your application through a memory profiler will help you better define your object lifecycles.

Summary

The main point of this section is to promote a stronger awareness of object lifecycles. You need to have a clear understanding of when objects are created and destroyed and the best way to do that is to move the object lifecycle definitions into your use cases. This ensures that your quality assurance department will not pass your application code if your objects are not properly managed.

In this section, we identified the two primary causes of memory issues in Java applications: object cycling and loitering objects. We reviewed mechanisms that can be employed to identify both of these conditions and discussed methods to resolve them. A strong awareness of object lifecycles, as well as a good set of tools, can help improve both the performance and the availability of your applications.

Quest’s Application Assurance Suite for Java and Portals supports an extremely proactive and thorough performance testing methodology. Using our solutions for Application Assurance, Performance Management and End User Performance Management, you can make certain that your applications will meet and exceed expectations once released to production.

ABOUT THE AUTHOR

Steven Haines is currently the J2EE Domain Expert/Architect at Quest Software, defining the expert rules used to monitor the performance of J2EE applications and application servers. He is the author of three Java books: *The Java Reference Guide* (InformIT/Pearson, 2005), *Java 2 Primer Plus* (SAMS, 2002) and *Java 2 From Scratch* (QUE, 1999). In addition to contributing chapters and coauthoring other books, as well as technical editing countless software publications, he is also the Java Host on InformIT.com. As an educator, he has taught all aspects of Java at Learning Tree University as well as at the University of California, Irvine. By day he works as a Java EE 5 Performance Architect at Quest Software, defining performance tuning and monitoring software as well as managing and performing Java EE 5 performance tuning engagements for large-scale Java EE 5 deployments, including those of several Fortune 500 companies.

ABOUT QUEST SOFTWARE, INC.

Quest Software, Inc. delivers innovative products that help organizations get more performance and productivity from their applications, databases and infrastructure. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 18,000 customers worldwide meet higher expectations for enterprise IT. Quest Software can be found in offices around the globe and at www.quest.com.

Contacting Quest Software

Mail:	Quest Software, Inc. World Headquarters 5 Polaris Way Aliso Viejo, CA 92656 USA
Web site	www.quest.com
Email:	info@quest.com
Phones:	1.800.306.9329 (Inside U.S.)
	1.949.754.8000 (Outside U.S.)

Please refer to our Web site for regional and international office information. For more information on Quest's Application Assurance Suite for Java and Portals or other Quest Software solutions, visit http://www.quest.com/application%5Fassurance/.

Trademarks

All trademarks and registered trademarks used in this guide are property of their respective owners.