Pro Java EE 5 Performance Management and Optimization

Steven Haines

Apress®

Pro Java EE 5 Performance Management and Optimization

Copyright © 2006 by Steven Haines

ISBN-13: 1-59059-610-2

ISBN-10: 978-1-59059-610-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Mark Gowdy, Dilip Thomas
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editors: Heather Lang, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert
Proofreader: Liz Welch
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski
Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 5

Performance Through the Application Development Life Cycle

"Okay, I understand how to gather metrics, but now what do I do with them?" John asked, looking confounded. "If I have application response time, instrumentation, and application server metrics, what should I have my developers do to ensure that the next deployment will be successful?"

"That is a very good question. At its core, it involves a change of mind-set by your entire development organization, geared toward performance. You'll most likely feel resistance from your developers, but if they follow these steps and embrace performance testing from the outset, then you'll better your chances of success more than a hundredfold," I said.

"I can deal with upset developers," John responded. "The important thing is that the application meets performance criteria when it goes live. I'll make sure that they follow the proper testing procedures; they have to understand the importance of application performance. I just can't face the idea of calling the CEO and telling him that we failed again!"

"Don't worry, I've helped several customers implement this methodology into their development life cycle, and each one has been successful. It is a discipline that, once adopted, becomes second nature. The key is to get started now!"

"Tell me more," John stated calmly, in contrast with his stressed demeanor. I knew that John had seen the light and was destined for success in the future.

Performance Overview

All too often in application development, performance is an afterthought. I once worked for a company that fully embraced the Rational Unified Process (RUP) but took it to an extreme. The application the company built spent years in architecture, and the first of ten iterations took nearly nine months to complete. The company learned much through its efforts and became increasingly efficient in subsequent iterations, but one thing that the organization did not learn until very late in the game was the importance of application performance. In the last couple of iterations, it started implementing performance testing and learned that part of the core architecture was flawed—specifically, the data model needed to be rearchitected. Because object models are built on top of data models, the object model also had to change. In addition, all components that interact with the object model had to change, and so on. Finally, the application had to go through another lengthy QA cycle that uncovered new bugs as well as the reemergence of former bugs.

That company learned the hard way that the later in the development life cycle performance issues are identified, the more expensive they are to fix. Figure 5-1, which you may recall from Chapter 1, illustrates this idea graphically. You can see that a performance issue identified during the application's development is inexpensive to fix, but one found later can cause the cost to balloon. Thus, you must ensure the performance of your application from the early stages of its architecture and test it at each milestone to preserve your efforts.

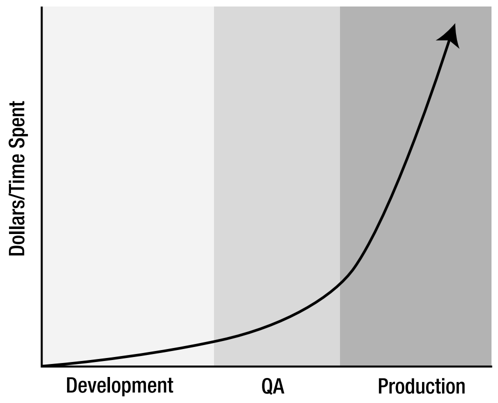


Figure 5-1. *The relationship between the time taken to identify performance issues and the repair costs*

A common theme has emerged from those customer sites I visit in which few or no performance issues are identified: these customers kept in mind the performance of the application when designing the application architecture. At these engagements, the root causes of most of the application problems were related to load or application server configuration—the applications had very few problems.

This chapter formalizes the methodology you should implement to ensure the performance of your application at each stage of the application development, QA, and deployment stages. I have helped customers implement this methodology into their organizations and roll out their applications to production successfully.

Performance in Architecture

The first step in developing any application of consequence is to perform an architectural analysis of a business problem domain. To review, application business owners work with application technical owners to define the requirements of the system. Application business owners are responsible for ensuring that when the application is complete it meets the needs of the end users, while application technical owners are responsible for determining the feasibility of options and defining the best architecture to solve the business needs. Together, these two groups design the functionality of the application.

In most organizations, the architecture discussions end at this analysis stage; the next step is usually the design of the actual solution. And this stage is where the architectural process needs to be revolutionized. Specifically, these groups need to define intelligent SLAs for each use case, they need to define the life cycles of major objects, and they need to address requirements for sessions.

SLAs

As you may recall from earlier in this book, an intelligent SLA maintains three core traits. It is

- Reasonable
- Specific
- Flexible

An SLA must satisfy end-user expectations but still be reasonable enough to be implemented. An unreasonable SLA will be ignored by all parties until end users complain. This is why SLAs need to be defined by both the application business owner and the application technical owner: the business owner pushes for the best SLAs for his users, while the application technical owner impresses upon the business owner the reality of what the business requirement presents. If the business requirement cannot be satisfied in a way acceptable to the application business owner, then the application technical owner needs to present all options and the cost of each (in terms of effort). The business requirement may need to be changed or divided into subprocesses that can be satisfied reasonably.

An intelligent SLA needs to be specific and measurable. In this requirement, you are looking for a hard and fast number, not a statement such as "The search functionality will respond within a reasonable user tolerance threshold." How do you test "reasonable"? You need to remove all subjectivity from this exercise. After all, what is the point in defining an SLA if you cannot verify it?

Finally, an intelligent SLA needs to be flexible. It needs to account for variations in behavior as a result of unforeseen factors, but define a hard threshold for how flexible it is allowed to be. For example, an SLA may read "The search functionality will respond within three seconds (specific) for 95 percent of requests (flexible)." The occasional seven-second response time is acceptable, as long as the integrity of the application is preserved—it responds well most of the time. By defining concrete values for the specific value as well as the limitations of the flexible value, you can quantify what "most of the time" means to the performance of the application, and you have a definite value with which to evaluate and verify the SLA.

■**Note** Although you define specific performance criteria and a measure of flexibility, defining either a hard upper limit of tolerance or a relative upper limit is also a good idea. I prefer to specify a relative upper limit, measured in the number of standard deviations from the mean. The purpose of defining an SLA in this way is that on paper a 3-second response time for 95 percent of requests is tolerable, but how do you address drastically divergent response time, such as a 30-second response time? Statistically, this should not be grossly applicable, but it is a good safeguard to be aware of.

An important aspect of defining intelligent SLAs is tracking them. The best way to do this is to integrate them into your application use cases. A use case is built from a general thought, such as "The application must provide search functionality for its patient medical records," but then the use case is divided into scenarios. Each scenario defines a path that the use case may follow given varying user actions. For example, what does the application do when the patient exists? What does it do when the patient does not exist? What if the search criterion returns more than one patient record? Each of these business processes needs to be explicitly called out in the use case, and each needs to have an SLA associated with it.

The following exercise demonstrates the format that a proper use case containing intelligent SLAs should follow.

USE CASE: PATIENT HISTORY SEARCH FUNCTIONALITY

Use Case

The Patient Management System must provide functionality to search for specific patient medical history information.

Scenarios

Scenario 1: The Patient Management System returns one distinct record.

Scenario 2: The Patient Management System returns more than one match.

Scenario 3: The Patient Management System does not find any users meeting the specified criteria.

Preconditions

The user has successfully logged in to the application.

Triggers

The user enters search criteria and submits data using the Web interface.

Descriptions

Scenario 1:

1. The Patient Management
2. . . .

Scenario 2:

3. . . .

Postconditions

The Patient Management System displays the results to the user.

SLAs

Scenario 1: The Patient Management System will return a specific patient matching the specified criteria in less than three seconds for 95 percent of requests. The response time will at no point stray more than two standard deviations from the mean.

Scenario 2: The Patient Management System will return a collection of patients matching the specified criteria in less than five seconds for 95 percent of requests. The response time will at no point stray more than two standard deviations from the mean.

Scenario 3: When the Patient Management System cannot find a user matching the specified criteria, it will inform the user in less than two seconds for 95 percent of requests. The response time will at no point stray more than two standard deviations from the mean.

The format of this use case varies from traditional use cases with the addition of the SLA component. In the SLA component, you explicitly call out the performance requirements for each scenario. The performance criteria include the following:

- *The expected tolerance level*: Respond in less than three seconds.
- *The measure of flexibility*: Meet the tolerance level for 95 percent of requests.
- *The upper threshold*: Do not stray more than three standard deviations from the observed mean.

With each of these performance facets explicitly defined, the developers implementing code to satisfy the use case understand their expectations and can structure unit tests accordingly. The QA team has a specific value to test and measure the quality of the application against. Next, when the QA team, or a delegated performance capacity assessor, performs a formal capacity assessment, an extremely accurate assessment can be built and a proper degradation model constructed. Finally, when the application reaches production, enterprise Java system administrators have values from which to determine if the application is meeting its requirements.

All of this specific assessment is possible, because the application business owner and application technical owner took time to carefully determine these values in the architecture phase. My aim here is to impress upon you the importance of up-front research and a solid communication channel between the business and technical representatives.

Object Life Cycle Management

The most significant problem plaguing production enterprise Java applications is memory management. The root cause of 90 percent of my customers' problems is memory related and can manifest in one of two ways:

- Object cycling
- Loitering objects (lingering object references)

Recall that object cycling is the rapid creation and deletion of objects in a short period of time that causes the frequency of garbage collection to increase and may result in tenuring short-lived objects prematurely. The cause of loitering objects is poor object management; the application developer does not explicitly know when an object should be released from memory, so the reference is maintained. Loitering objects are the result of an application developer failing to release object references at the correct time. This is a failure to understand the impact of reference management on application performance. This condition results in an overabundance of objects residing in memory, which can have the following effects:

- Garbage collection may run slower, because more live objects must be examined.
- Garbage collection can become less effective at reclaiming objects.
- Swapping on the physical machine can result, because less physical memory is available for other processes to use.

Neglecting object life cycle management can result in memory leaks and eventually application server crashes. I discuss techniques for detecting and avoiding object cycling later in this chapter, because it is a development or design issue, but object life cycle management is an architectural issue.

To avoid loitering objects, take control of the management of object life cycles by defining object life cycles inside use cases. I am not advocating that each use case should define every `int`, `boolean`, and `float` that will be created in the code to satisfy the use case; rather, each use case needs to define the major application-level components upon which it depends. For example, in the Patient Management System, daily summary reports may be generated every evening that detail patient metrics such as the number of cases of heart disease identified this year and the common patient profile attributes for each. This report would be costly to build on a per-request basis, so the architects of the system may dictate that the report needs to be cached at the application level (or in the application scope so that all requests can access it).

Defining use case dependencies and application-level object life cycles provides a deeper understanding of what should and should not be in the heap at any given time. Here are some guidelines to help you identify application-level objects that need to be explicitly called out and mapped to use cases in a dependency matrix:

- Expensive objects, in terms of both allocated size as well as allocation time, that will be accessed by multiple users
- Commonly accessed data
- Nontemporal user session objects
- Global counters and statistics management objects
- Global configuration options

The most common examples of application-level object candidates are frequently accessed business objects, such as those stored in a cache. If your application uses entity beans, then you need to carefully determine the size of the entity bean cache by examining use cases; this can be extrapolated to apply to any caching infrastructure. The point is that if you are caching data in the heap to satisfy specific use cases, then you need to determine how much data is required to satisfy the use cases. And if anyone questions the memory footprint, then you can trace it directly back to the use cases.

The other half of the practice of object life cycle management is defining when objects should be removed from memory. In the previous example, the medical summary report is updated every evening, so at that point the old report should be removed from memory to make room for the new report. Knowing when to remove objects is probably more important than knowing when to create objects. If an object is not already in memory, then you can create it, but if it is in memory and no one needs it anymore, then that memory is lost forever.

Application Session Management

Just as memory mismanagement is the most prevalent issue impacting the performance of enterprise Java applications, HTTP sessions are by far the biggest culprit in memory abuse. HTTP is a stateless protocol, and as such the conversation between the Web client and Web server terminates at the conclusion of a single request: the Web client submits a request to the Web server (most commonly `GET` or `POST`), and then the Web server performs its business logic, constructs a response, and returns the response to the Web client. This ends the Web conversation and terminates the relationship between client and server.

In order to sustain a long-term conversation between a Web client and Web server, the Web server constructs a unique identifier for the client and includes it with its response to the request; internally the Web server maintains all user data and associates it with that identifier. On subsequent requests, the client submits this unique identifier to identify itself to the Web server.

This sounds like a good idea, but it creates the following problem: if the HTTP protocol is truly stateless and the conversation between Web client and Web server can only be renewed by a client interaction, then what does the Web server do with the client's information if that client never returns? Obviously, the Web server throws the information away, but the real question relates to how long the Web server should keep the information.

All application servers provide a session time-out value that constrains the amount of time user data is maintained. When the user makes any request from the server, the user's time-out is reset, and once the time-out has been exceeded, the user's stateful information is discarded. A practical example of this is logging in to your online banking application. You can view your account balances, transfer funds, and pay bills, but if you sit idle for too long, you are forced to log in again. The session time-out period for a banking application is usually quite short for security reasons (for example, if you log in to your bank account and then leave your computer unattended to go to a meeting, you do not want someone else who wanders by your desk to be able to access your account). On the other hand, when you shop at Amazon.com, you can add items to your shopping cart and return six months later to see that old book on DNA synthesis and methylation that you still do not have time to read sitting there. Amazon.com uses a more advanced infrastructure to support this feature (and a heck of a lot of hardware and memory), but the question remains: how long should you hold on to data between user requests before discarding it?

The definitive time-out value must come from the application business owner. He or she may have specific, legally binding commitments with end users and business partners. But an application technical owner can control the quantity of data that is held resident in memory for each user. In the aforementioned example, do you think that Amazon.com maintains everyone's shopping cart in memory for all time? I suspect that shopping cart data is maintained in memory for a fixed session length, and afterward persisted to a database for later retrieval.

As a general guideline, sessions should be as small as possible while still realizing the benefits of being resident in memory. I usually maintain temporal data describing what the user does in a particular session, such as the page the user came from, the options the user has enabled, and so on. More significant data, such as objects stored in a shopping cart, opened reports, or partial result sets, are best stored in stateful session beans, because rather than being maintained in a hash map that can conceivably grow indefinitely like HTTP session objects, stateful session beans are stored in predefined caches. The size of stateful session bean caches can be defined upon deployment, on a per-bean basis, and hence assert an upper limit on memory consumption. When the cache is full, to add a new bean to it, an existing bean must be selected and written out to persistent storage. The danger is that if the cache is sized too small, the maintenance of the cache can outweigh the benefits of having the cache in the first place. If your sessions are heavy and your user load is large, then this upper limit can prevent your application servers from crashing.

Performance in Development

Have you ever heard anyone ask the following question: "When developers are building their individual components before a single use case is implemented, isn't it premature to start performance testing?"

Let me ask a similar question: When building a car, is it premature to test the performance of your alternator before the car is assembled and you try to start it? The answer to this question is obviously "No, it's not premature. I want to make sure that the alternator works before building my car!" If you would never assemble a car from untested parts, why would you assemble an enterprise application from untested components? Furthermore, because you integrate performance criteria into use cases, use cases will fail testing if they do not meet their performance criteria. In short, performance matters!

In development, components are tested in *unit tests*. A unit test is designed to test the functionality and performance of an individual component, independently from other components that it will eventually interact with. The most common unit testing framework is an open source initiative called JUnit. JUnit's underlying premise is that alongside the development of your components, you should write tests to validate each piece of functionality of your components. A relatively new development paradigm, Extreme Programming (`www.xprogramming.com`), promotes building test cases prior to building the components themselves, which forces you to better understand how your components will be used prior to writing them.

JUnit focuses on functional testing, but side projects spawned from JUnit include performance and scalability testing. Performance tests measure expected response time, and scalability tests measure functional integrity under load. Formal performance unit test criteria should do the following:

- Identify memory issues
- Identify poorly performing methods and algorithms
- Measure the coverage of unit tests to ensure that the majority of code is being tested

Memory leaks are the most dangerous and difficult to diagnose problems in enterprise Java applications. The best way to avoid memory leaks at a code level is to run your components through a *memory profiler*. A memory profiler takes a snapshot of your heap (after first running garbage collection), allows you to run your tests, takes another snapshot of your heap (after garbage collection again), and shows you all of the objects that remain in the heap. The analysis of the heap differences identifies objects abandoned in memory. Your task is then to look at these objects and decide if they should remain in the heap or if they were left there by mistake. Another danger of memory misusage is object cycling, which, again, is the rapid creation and destruction of objects. Because it increases the frequency of garbage collection, excessive object cycling may result in the premature tenuring of short-lived objects, necessitating a major garbage collection to reclaim these objects.

After considering memory issues, you need to quantify the performance of methods and algorithms. Because SLAs are defined at the use case level, but not at the component level, measuring response times may be premature in the development phase. Rather, the strategy is to run your components through a *code profiler*. A code profiler reveals the most frequently executed sections of your code and those that account for the majority of the components' execution times. The resulting relative weighting of hot spots in the code allows for intelligent tuning and code refactoring. You should run code profiling on your components while executing your unit tests, because your unit tests attempt to mimic end-user actions and alternate user scenarios. Code profiling your unit tests should give you a good idea about how your component will react to real user interactions.

Coverage profiling reports the percentage of classes, methods, and lines of code that were executed during a test or use case. Coverage profiling is important in assessing the efficacy of unit tests. If both the code and memory profiling of your code are good, but you are exercising only 20 percent of your code, then your confidence in your tests should be minimal. Not only do you need to receive favorable results from your functional unit tests and your code and memory performance unit tests, but you also need to ensure that you are effectively testing your components.

This level of testing can be further extended to any code that you outsource. You should require your outsourcing company to provide you with unit tests for all components it develops, and then execute a performance test against those unit tests to measure the quality of the components you are receiving. By combining code and memory profiling with coverage profiling, you can quickly determine whether the unit tests are written properly and have acceptable results.

Once the criteria for tests are met, the final key step to effectively implementing this level of testing is automation. You need to integrate functional and performance unit testing into your build process—only by doing so can you establish a repeatable and trackable procedure. Because running performance unit tests can burden memory resources, you might try executing functional tests during nightly builds and executing performance unit tests on Friday-night builds, so that you can come in on Monday to test result reports without impacting developer productivity. This suggestion's success depends a great deal on the size and complexity of your environment, so, as always, adapt this plan to serve your application's needs.

When performance unit tests are written prior to, or at least concurrently with, component development, then component performance can be assessed at each build. If such extensive assessment is not realistic, then the reports need to be evaluated at each major development milestone. For the developer, milestones are probably at the completion of the component or a major piece of functionality for the component. But at minimum, performance unit tests need to be performed prior to the integration of components. Again, building a high-performance car from tested and proven high-performance parts is far more effective than from scraps gathered from the junkyard.

Unit Testing

I thought this section would be a good opportunity to talk a little about unit testing tools and methods, though this discussion is not meant to be exhaustive. JUnit is, again, the tool of choice for unit testing. JUnit is a simple regression-testing framework that enables you to write repeatable tests. Originally written by Erich Gamma and Kent Beck, JUnit has been embraced by thousands of developers and has grown into a collection of unit testing frameworks for a plethora of technologies. The JUnit Web site (`www.junit.org`) hosts support information and links to the other JUnit derivations.

JUnit offers the following benefits to your unit testing:

- *Faster coding*: How many times have you written debug code inside your classes to verify values or test functionality? JUnit eliminates this by allowing you to write test cases in closely related, but centralized and external, classes.
- *Simplicity*: If you have to spend too much time implementing your test cases, then you won't do it. Therefore, the creators of JUnit made it as simple as possible.
- *Single result reports*: Rather than generating loads of reports, JUnit will give you a single pass/fail result, and, for any failure, show you the exact point where the application failed.
- *Hierarchical testing structure*: Test cases exercise specific functionality, and test suites execute multiple test cases. JUnit supports test suites of test suites, so when developers build test cases for their classes, they can easily assemble them into a test suite at the package level, and then incorporate that into parent packages and so forth. The result is that a single, top-level test execution can exercise hundreds of unit test cases.
- *Developer-written tests*: These tests are written by the same person who wrote the code, so the tests accurately target the intricacies of the code that the developer knows can be problematic. This test differs from a QA-written one, which exercises the external functionality of the component or use case—instead, this test exercises the internal functionality.
- *Seamless integration*: Tests are written in Java, which makes the integration of test cases and code seamless.
- *Free*: JUnit is open source and licensed under the Common Public License Version 1.0, so you are free to use it in your applications.

From an architectural perspective, JUnit can be described by looking at two primary components: `TestCase` and `TestSuite`. All code that tests the functionality of your class or classes must extend `junit.framework.TestCase`. The `test` class can implement one or more tests by defining `public void` methods that start with `test` and accept no parameters, for example:

```
public void testMyFunctionality() { ... }
```

For multiple tests, you have the option of initializing and cleaning up the environment before and between tests by implementing the following two methods: `setUp()` and `tearDown()`. In `setUp()` you initialize the environment, and in `teardown()` you clean up the environment. Note that these methods are called between each test to eliminate side effects between test cases; this makes each test case truly independent.

Inside each `TestCase` "test" method, you can create objects, execute functionality, and then test the return values of those functional elements against expected results. If the return values are not as expected, then the test fails; otherwise, it passes. The mechanism that JUnit provides to validate actual values against expected values is a set of `assert` methods:

- `assertEquals()` methods test primitive types.
- `assertTrue()` and `assertFalse()` test Boolean values.
- `assertNull()` and `assertNotNull()` test whether or not an object is null.
- `assertSame()` and `assertNotSame()` test object equality.

In addition, JUnit offers a `fail()` method that you can call anywhere in your test case to immediately mark a test as failing.

JUnit tests are executed by one of the `TestRunner` instances (there is one for command-line execution and one for a GUI execution), and each version implements the following steps:

1. It opens your `TestCase` class instance.

2. It uses reflection to discover all methods that start with "test".

3. It repeatedly calls `setUp()`, executes the test method, and calls `teardown()`.

As an example, I have a set of classes that model data metrics. A metric contains a set of data points, where each data point represents an individual sample, such as the size of the heap at a given time. I purposely do not list the code for the metric or data point classes; rather, I list the JUnit tests. Recall that according to one of the tenets of Extreme Programming, we write test cases before writing code. Listing 5-1 shows the test case for the `Metric` class, and Listing 5-2 shows the test case for the `DataPoint` class.

Listing 5-1. *DataPointTest.java*

```
package com.javasrc.metric;

import junit.framework.TestCase;
import java.util.*;

/**
 * Tests the core functionality of a DataPoint
 */
public class DataPointTest extends TestCase
{
  /**
   * Maintains our reference DataPoint
   */
  private DataPoint dp;

  /**
   * Create a DataPoint for use in this test
   */
  protected void setUp()
  {
    dp = new DataPoint( new Date(), 5.0, 1.0, 10.0 );
  }
```

```
/**
 * Clean up: do nothing for now
 */
protected void tearDown()
{
}

/**
 * Test the range of the DataPoint
 */
public void testRange()
{
  assertEquals( 9.0, dp.getRange(), 0.001 );
}

/**
 * See if the DataPoint scales properly
 */
public void testScale()
{
  dp.scale( 10.0 );
  assertEquals( 50.0, dp.getValue(), 0.001 );
  assertEquals( 10.0, dp.getMin(), 0.001 );
  assertEquals( 100.0, dp.getMax(), 0.001 );
}

/**
 * Try to add a new DataPoint to our existing one
 */
public void testAdd()
{
  DataPoint other = new DataPoint( new Date(), 4.0, 0.5, 20.0 );
  dp.add( other );
  assertEquals( 9.0, dp.getValue(), 0.001 );
  assertEquals( 0.5, dp.getMin(), 0.001 );
  assertEquals( 20.0, dp.getMax(), 0.001 );
}

/**
 * Test the compare functionality of our DataPoint to ensure that
 * when we construct Sets of DataPoints they are properly ordered
 */
public void testCompareTo()
{
  try
  {
      // Sleep for 100ms so we can be sure that the time of
      // the new data point is later than the first
      Thread.sleep( 100 );
    }
    catch( Exception e )
    {
    }

    // Construct a new DataPoint
    DataPoint other = new DataPoint( new Date(), 4.0, 0.5, 20.0 );

    // Should return -1 because other occurs after dp
    int result = dp.compareTo( other );
    assertEquals( -1, result );

    // Should return 1 because dp occurs before other
    result = other.compareTo( dp );
    assertEquals( 1, result );

    // Should return 0 because dp == dp
    result = dp.compareTo( dp );
    assertEquals( 0, result );
  }
}
```

Listing 5-2. *MetricTest.java*

```
package com.javasrc.metric;

import junit.framework.TestCase;
import java.util.*;

public class MetricTest extends TestCase
{
  private Metric sampleHeap;

  protected void setUp()
  {
    this.sampleHeap = new Metric( "Test Metric",
                                  "Value/Min/Max",
                                  "megabytes" );
    double heapValue = 100.0;
    double heapMin = 50.0;
    double heapMax = 150.0;
    for( int i=0; i<10; i++ )
    {
      DataPoint dp = new DataPoint( new Date(),
                                    heapValue,
                                    heapMin,
                                    heapMax );
      this.sampleHeap.addDataPoint( dp );
      try
      {
        Thread.sleep( 50 );
      }
      catch( Exception e )
      {
      }

      // Update the heap values
      heapMin -= 1.0;
      heapMax += 1.0;
      heapValue += 1.0;
    }
  }

  public void testMin()
  {
    assertEquals( 41.0, this.sampleHeap.getMin(), 0.001 );
  }

  public void testMax()
  {
    assertEquals( 159.0, this.sampleHeap.getMax(), 0.001 );
  }

  public void testAve()
  {
    assertEquals( 104.5, this.sampleHeap.getAve(), 0.001 );
  }

  public void testMaxRange()
  {
    assertEquals( 118.0, this.sampleHeap.getMaxRange(), 0.001 );
  }

  public void testRange()
  {
    assertEquals( 118.0, this.sampleHeap.getRange(), 0.001 );
  }
  public void testSD()
  {
    assertEquals( 3.03, this.sampleHeap.getStandardDeviation(), 0.01 );
  }

  public void testVariance()
  {
    assertEquals( 9.17, this.sampleHeap.getVariance(), 0.01 );
  }

  public void testDataPointCount()
  {
    assertEquals( 10, this.sampleHeap.getDataPoints().size() );
  }
}
```

In Listing 5-1, you can see that the `DataPoint` class, in addition to maintaining the observed value for a point in time, supports minimum and maximum values for the time period, computes the range, and supports scaling and adding data points. The sample test case creates a `DataPoint` object in the `setUp()` method and then exercises each piece of functionality.

Listing 5-2 shows the test case for the `Metric` class. The `Metric` class aggregates the `DataPoint` objects and provides access to the collective minimum, maximum, average, range, standard deviation, and variance. In the `setUp()` method, the test creates a set of data points and builds the metric to contain them. Each subsequent test case uses this metric and validates values computed by hand to those computed by the `Metric` class.

Listing 5-3 rolls both of these test cases into a test suite that can be executed as one test.

Listing 5-3. *MetricTestSuite.java*

```
package com.javasrc.metric;

import junit.framework.Test;
import junit.framework.TestSuite;

public class MetricTestSuite
{
  public static Test suite()
  {
    TestSuite suite = new TestSuite();
    suite.addTestSuite( DataPointTest.class );
    suite.addTestSuite( MetricTest.class );
    return suite;
  }
}
```

A `TestSuite` exercises all tests in all classes added to it by calling the `addTestSuite()` method. A `TestSuite` can contain `TestCases` or `TestSuites`, so once you build a suite of test cases for your classes, a master test suite can include your suite and inherit all of your test cases.

The final step in this example is to execute either an individual test case or a test suite. After downloading JUnit from `www.junit.org`, add the `junit.jar` file to your `CLASSPATH` and then invoke either its command-line interface or GUI interface. The three classes that execute these tests are as follows:

- `junit.textui.TestRunner`
- `junit.swingui.TestRunner`
- `junit.awtui.TestRunner`

And as these package names imply, `textui` is the command-line interface and `swingui` is the graphical interface. `awtui` provides a batch interface to executing unit tests. You can pass an individual test case or an entire test suite as an argument to the `TestRunner` class. For example, to execute the test suite that we created earlier, you would use this:

```
java junit.swingui.TestRunner com.javasrc.metric.MetricTestSuite
```

Unit Performance Testing

Unit performance testing has three aspects:

- Memory profiling
- Code profiling
- Coverage profiling

This section explores each facet of performance profiling. I provide examples of what to look for and the step-by-step process to implement each type of testing.

Memory Profiling

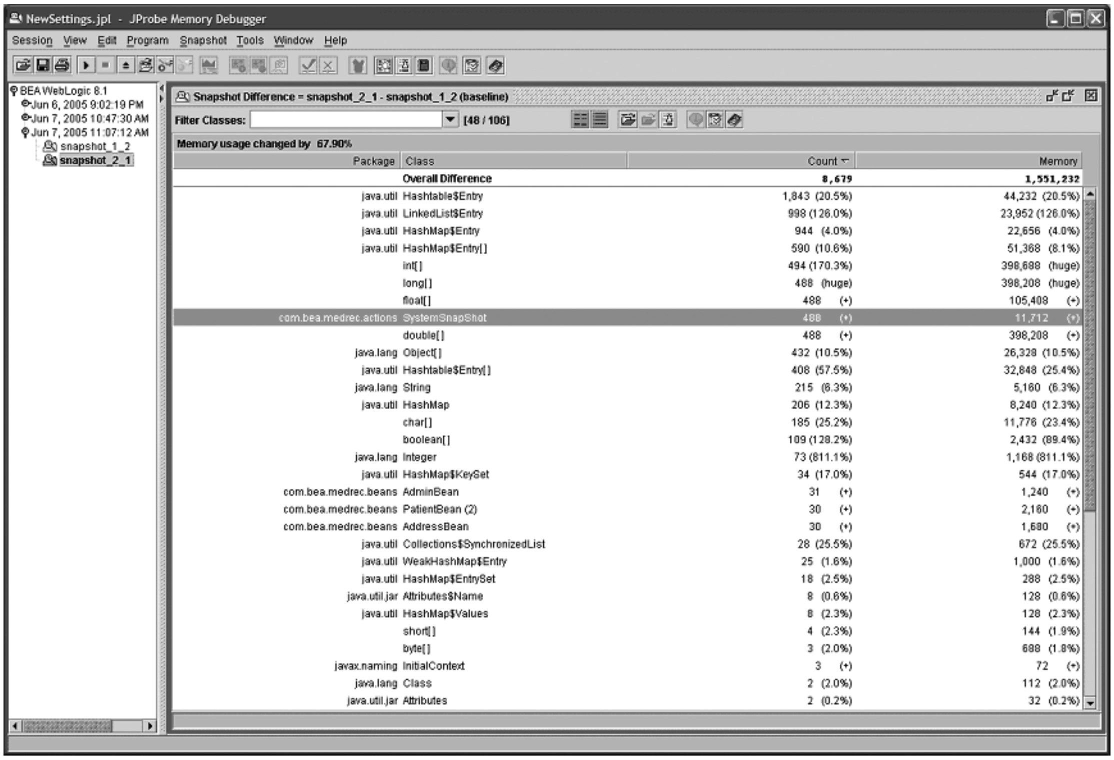
Let’s first look at memory profiling. To illustrate how to determine if you do, in fact, have a memory leak, I modified the BEA MedRec application to capture the state of the environment every time an administrator logs in and to store that information in memory. My intent is to demonstrate how a simple tracking change left to its own devices can introduce a memory leak.

The steps you need to perform on your code for each use are as follows:

1. Request a garbage collection and take a snapshot of your heap.
2. Perform your use case.
3. Request a garbage collection and take another snapshot of your heap.
4. Compare the two snapshots (the difference between them includes all objects remaining in the heap) and identify any unexpected loitering objects.
5. For each suspect object, open the heap snapshot and track down where the object was created.

■ **Note** A memory leak can be detected with a single execution of a use case or through a plethora of executions of a use case. In the latter case, the memory leak will scream out at you. So, while analyzing individual use cases is worthwhile, when searching for subtle memory leaks, executing your use case multiple times makes finding them easier.

In this scenario, I performed steps 1 through 3 with a load tester that executed the MedRec administration login use case almost 500 times. Figure 5-2 shows the difference between the two heap snapshots.



Snapshot Difference = snapshot_2_1 - snapshot_1_2 (baseline)

Filter Classes: [48 / 106]

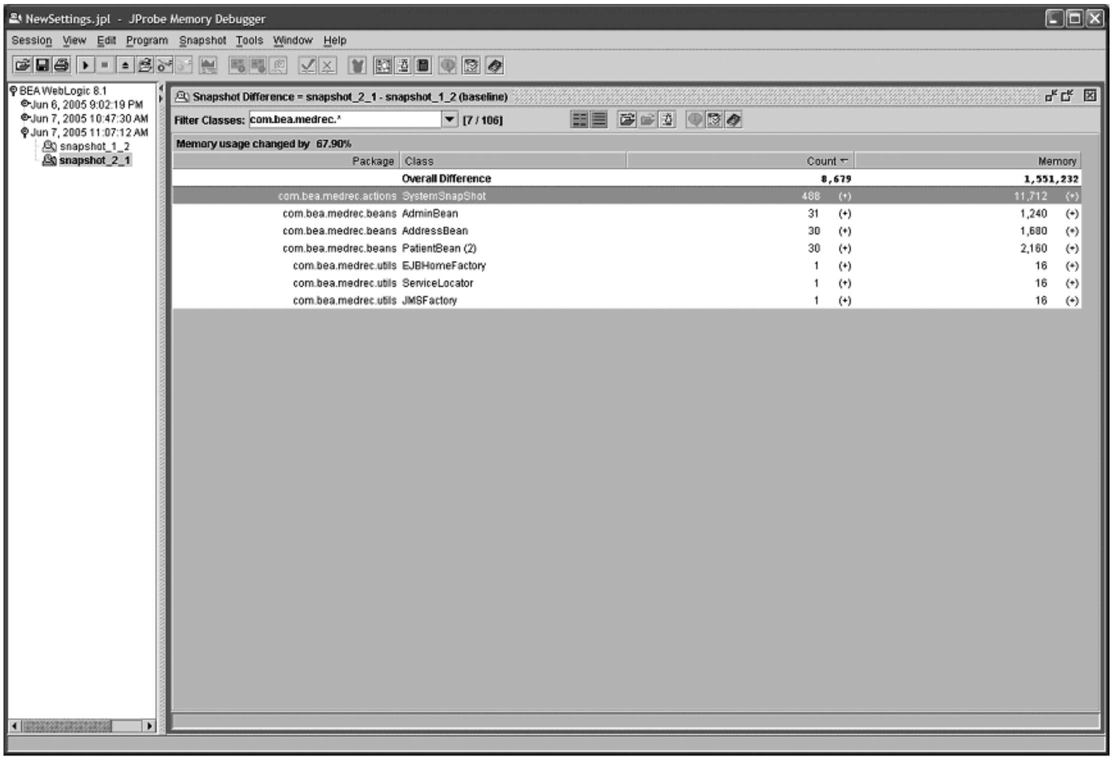
Memory usage changed by 67.90%

Package	Class	Count	Memory
	Overall Difference	**8,679**	**1,551,232**
java.util	Hashtable$Entry	1,843 (20.5%)	44,232 (20.5%)
java.util	LinkedList$Entry	998 (126.0%)	23,952 (126.0%)
java.util	HashMap$Entry	944 (4.0%)	22,656 (4.0%)
java.util	HashMap$Entry[]	590 (10.6%)	51,368 (8.1%)
	int[]	494 (170.3%)	398,688 (huge)
	long[]	488 (huge)	398,208 (huge)
	float[]	488 (+)	105,408 (+)
com.bea.medrec.actions	SystemSnapShot	488 (+)	11,712 (+)
	double[]	488 (+)	398,208 (+)
java.lang	Object[]	432 (10.5%)	26,328 (10.5%)
java.util	Hashtable$Entry[]	408 (57.5%)	32,848 (25.4%)
java.lang	String	215 (6.3%)	5,160 (6.3%)
java.util	HashMap	206 (12.3%)	8,240 (12.3%)
	char[]	185 (25.2%)	11,776 (23.4%)
	boolean[]	109 (128.2%)	2,432 (89.4%)
java.lang	Integer	73 (811.1%)	1,168 (811.1%)
java.util	HashMap$KeySet	34 (17.0%)	544 (17.0%)
com.bea.medrec.beans	AdminBean	31 (+)	1,240 (+)
com.bea.medrec.beans	PatientBean (2)	30 (+)	2,160 (+)
com.bea.medrec.beans	AddressBean	30 (+)	1,680 (+)
java.util	Collections$SynchronizedList	28 (25.5%)	672 (25.5%)
java.util	WeakHashMap$Entry	25 (1.6%)	1,000 (1.6%)
java.util	HashMap$EntrySet	18 (2.5%)	288 (2.5%)
java.util.jar	Attributes$Name	8 (0.6%)	128 (0.6%)
java.util	HashMap$Values	8 (2.3%)	128 (2.3%)
	short[]	4 (2.3%)	144 (1.9%)
	byte[]	3 (2.0%)	688 (1.8%)
javax.naming	InitialContext	3 (+)	72 (+)
java.lang	Class	2 (2.0%)	112 (2.0%)
java.util.jar	Attributes	2 (0.2%)	32 (0.2%)

Figure 5-2. *The snapshot difference between the heaps before and after executing the use case*

Figure 5-2 shows that my use case yielded 8,679 new objects added to the heap. Most of these objects are collection classes, and I suspect they are part of BEA's infrastructure. I scanned this list looking for my code, which in this case consists of any class in the `com.bea.medrec` package. Filtering on those classes, I was interested to see a large number of `com.bea.medrec.actions.SystemSnapShot` instances, as shown in Figure 5-3.

Note The screen shots in this chapter are from Quest Software's JProbe and PerformaSure products.


NewSettings.jpl - JProbe Memory Debugger
Session View Edit Program Snapshot Tools Window Help
BEA WebLogic 8.1
Jun 6, 2005 9:02:19 PM
Jun 7, 2005 10:47:30 AM
Jun 7, 2005 11:07:12 AM
snapshot_1_2
snapshot_2_1
Snapshot Difference = snapshot_2_1 - snapshot_1_2 (baseline)

Filter Classes: com.bea.medrec.* [7 / 106]

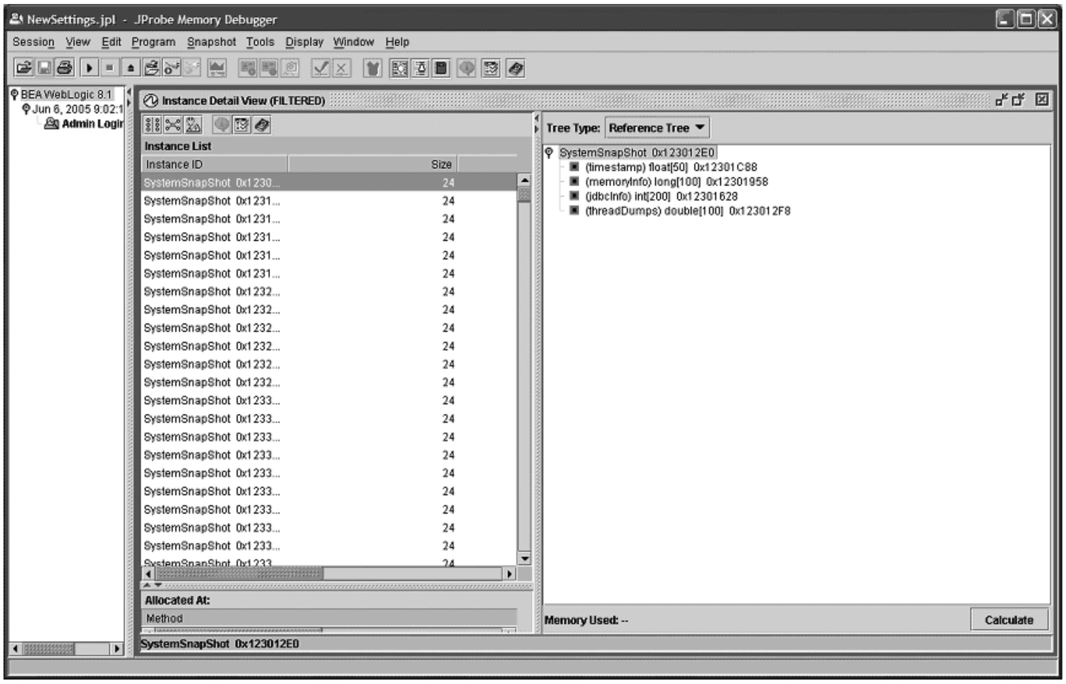
Memory usage changed by 67.90%

Package	Class	Count		Memory	
	Overall Difference	8,679		1,551,232	
com.bea.medrec.actions	SystemSnapShot	488	(+)	11,712	(+)
com.bea.medrec.beans	AdminBean	31	(+)	1,240	(+)
com.bea.medrec.beans	AddressBean	30	(+)	1,680	(+)
com.bea.medrec.beans	PatientBean (2)	30	(+)	2,160	(+)
com.bea.medrec.utils	EJBHomeFactory	1	(+)	16	(+)
com.bea.medrec.utils	ServiceLocator	1	(+)	16	(+)
com.bea.medrec.utils	JMSFactory	1	(+)	16	(+)

Figure 5-3. *The snapshot difference between the heaps, filtered on my application packages*

Realize that rarely is a loitering object a single simple object; rather, it is typically a subgraph that maintains its own references. In this case, the `SystemSnapShot` class is a dummy class that holds a set of primitive type arrays with the names `timestamp`, `memoryInfo`, `jdbcInfo`, and `threadDumps`, but in a real-world scenario these arrays would be objects that reference other objects and so forth. By opening the second heap snapshot and looking at one of the `SystemSnapShot` instances, you can see all objects that it references. As shown in Figure 5-4, the `SystemSnapShot` class references four objects: `timestamp`, `memoryInfo`, `jdbcInfo`, and `threadDumps`. A loitering object, then, has a far greater impact than the object itself.

Next, let's look at the referrer tree. We repeatedly ask the following questions: What class is referencing the `SystemSnapShot`? What class is referencing that class? Eventually, we finally find one of our classes. Figure 5-5 shows that the `SystemSnapShot` class is referenced by an `Object` array that is referenced by an `ArrayList` that is finally referenced by the `AdminLoginAction`.


NewSettings.jpl - JProbe Memory Debugger
Session View Edit Program Snapshot Tools Display Window Help
BEA WebLogic 8.1
Jun 6, 2005 9:02:1
Admin Logir
Instance Detail View (FILTERED)
Allocated At:
Method
Memory Used: --
Calculate

Tree Type: Reference Tree

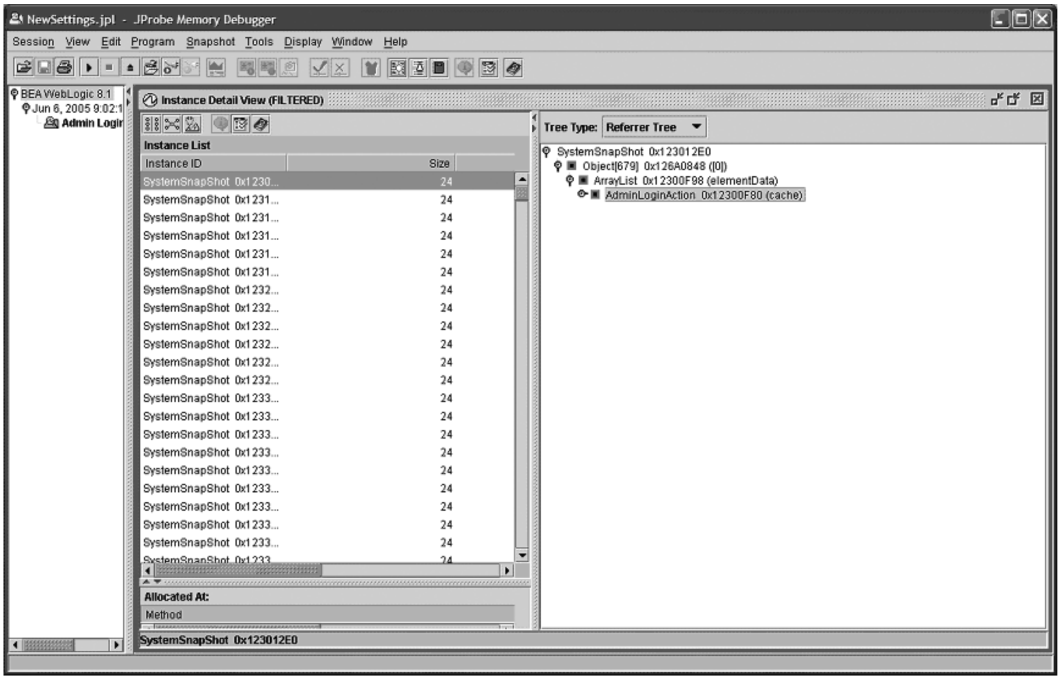
```
SystemSnapShot 0x123012E0
  (timestamp) float[50] 0x12301C88
  (memoryInfo) long[100] 0x12301958
  (jdbcInfo) int[200] 0x12301628
  (threadDumps) double[100] 0x123012F8
```

Instance List

Instance ID	Size
SystemSnapShot 0x1230...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24

SystemSnapShot 0x123012E0

Figure 5-4. *The SystemSnapShot class references four objects: timestamp, memoryInfo, jdbcInfo, and threadDumps.*


NewSettings.jpl - JProbe Memory Debugger
Session View Edit Program Snapshot Tools Display Window Help
BEA WebLogic 8.1
Jun 6, 2005 9:02:1
Admin Logir
Instance Detail View (FILTERED)
Allocated At:
Method

Tree Type: Referrer Tree

```
SystemSnapShot 0x123012E0
  Object[679] 0x126A0848 ([0])
    ArrayList 0x12300F98 (elementData)
      AdminLoginAction 0x12300F80 (cache)
```

Instance List

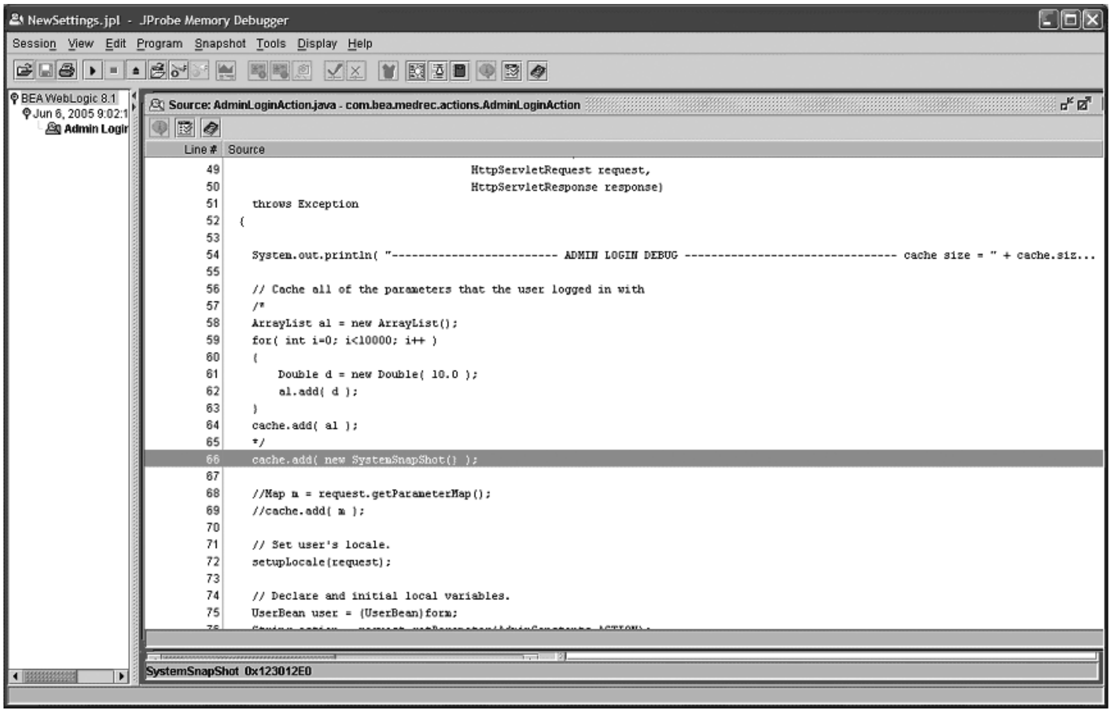
Instance ID	Size
SystemSnapShot 0x1230...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1231...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1232...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24
SystemSnapShot 0x1233...	24

SystemSnapShot 0x123012E0

Figure 5-5. *Here we can see that the AdminLoginAction class created the SystemSnapShot, and that it stored it in an ArrayList.*

Finally, we can look into the `AdminLoginAction` code to see that it creates the new `SystemSnapShot` instance we are looking at and adds it to its cache in line 66, as shown in Figure 5-6.

You need to perform this type of memory profiling test on your components during your performance unit testing. For each object that is left in the heap, you need to ask yourself whether or not you intended to leave it there. It's OK to leave things on the heap as long as you know that they are there and you want them to be there. The purpose of this test is to identify and document potentially troublesome objects and objects that you forgot to clean up.


NewSettings.jpl - JProbe Memory Debugger
Session View Edit Program Snapshot Tools Display Help
BEA WebLogic 8.1
Jun 6, 2005 9:02:1
Admin Logir
Source: AdminLoginAction.java - com.bea.medrec.actions.AdminLoginAction
Line # Source
SystemSnapShot 0x123012E0

```
49                                     HttpServletRequest request,
50                                     HttpServletResponse response)
51      throws Exception
52    {
53
54      System.out.println( "------------------------ ADMIN LOGIN DEBUG ------------------------------- cache size = " + cache.siz...
55
56      // Cache all of the parameters that the user logged in with
57      /*
58      ArrayList al = new ArrayList();
59      for( int i=0; i<10000; i++ )
60      {
61          Double d = new Double( 10.0 );
62          al.add( d );
63      }
64      cache.add( al );
65      */
66      cache.add( new SystemSnapShot() );
67
68      //Map m = request.getParameterMap();
69      //cache.add( m );
70
71      // Set user's locale.
72      setupLocale(request);
73
74      // Declare and initial local variables.
75      UserBean user = (UserBean)form;
```

Figure 5-6. *The AdminLoginAction source code*

Code Profiling

The purpose of code profiling is to identify sections of your code that are running slowly and then determine why. The perfect example I have to demonstrate the effectiveness of code profiling is a project that I gave to my Data Structures and Algorithm Analysis class—compare and quantify the differences among the following sorting algorithms for various values of *n* (where *n* represents the sample size of the data being sorted):

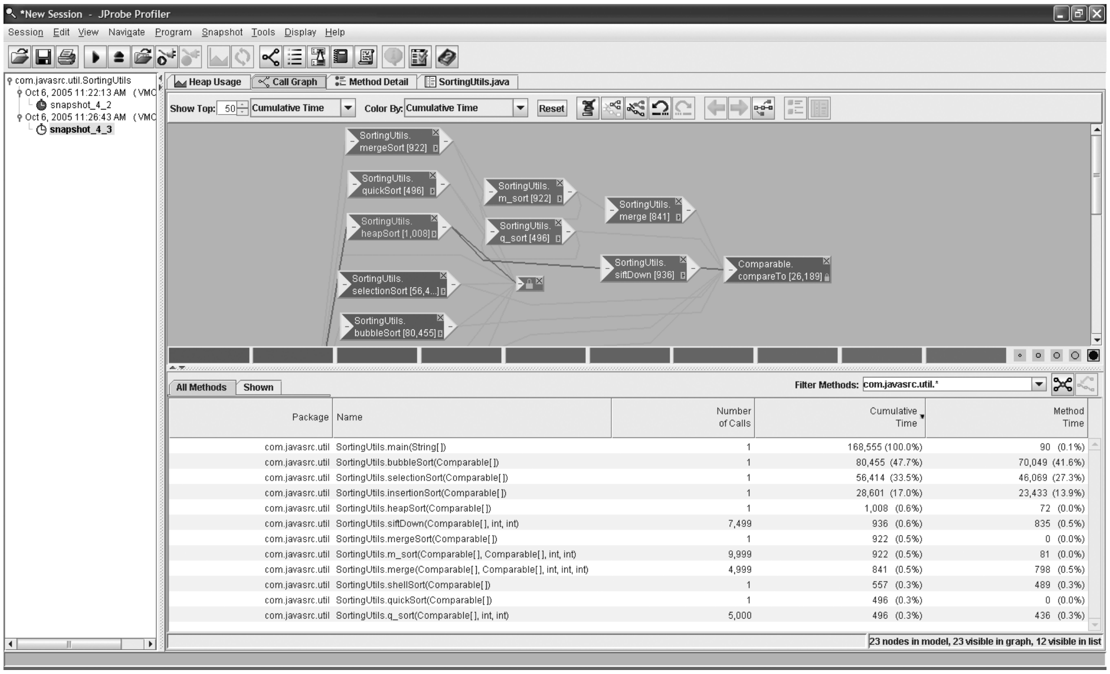
- Bubble sort
- Selection sort
- Insertion sort
- Shell sort

- Heap sort
- Merge sort
- Quick sort

As a quick primer on sorting algorithms, each of the aforementioned algorithms has its strengths and weaknesses. The first four algorithms run in $O(N^2)$ time, meaning that the run time increases exponentially as the number of items to sort, *N*, increases; specifically, as *N* increases, the amount of time required for the sorting algorithm to complete increases by N^2. The last three algorithms run in $O(N \log N)$ time, meaning that the run time grows logarithmically: as *N* increases, the amount of time required for the sorting algorithm to complete increases by $N \log N$. Achieving $O(N \log N)$ performance requires additional overhead that may cause the last three algorithms to actually run slower than the first four for a small number of items. My recommendation is to always examine both the nature of the data you want to sort today and the projected nature of the data throughout the life cycle of the product prior to selecting your sorting algorithm.

With that foundation in place, I provided my students with a class that implements the aforementioned sorting algorithms. I really wanted to drive home the dramatic difference between executing these sorting algorithms on 10 items as opposed to 10,000 items, or even 1,000,000 items. For this exercise, I think it would be useful to profile this application against 5,000 randomly generated integers, which is enough to show the differences between the algorithms, but not so excessive that I have to leave my computer running overnight.

Figure 5-7 shows the results of this execution, sorting each method by its cumulative run time.


*New Session - JProbe Profiler
Session Edit View Navigate Program Snapshot Tools Display Help
com.javasrc.util.SortingUtils
Oct 6, 2005 11:22:13 AM (VMC
snapshot_4_2
Oct 6, 2005 11:26:43 AM (VMC
snapshot_4_3
Heap Usage
Call Graph
Method Detail
SortingUtils.java
Show Top: 50
Cumulative Time
Color By: Cumulative Time
Reset
SortingUtils. mergeSort [922]
SortingUtils. quickSort [496]
SortingUtils. heapSort [1,008]
SortingUtils. m_sort [922]
SortingUtils. q_sort [496]
SortingUtils. merge [841]
SortingUtils. siftDown [936]
Comparable. compareTo [26,189]
SortingUtils. selectionSort [56,4...]
SortingUtils. bubbleSort [80,455]
All Methods
Shown
Filter Methods: com.javasrc.util.*

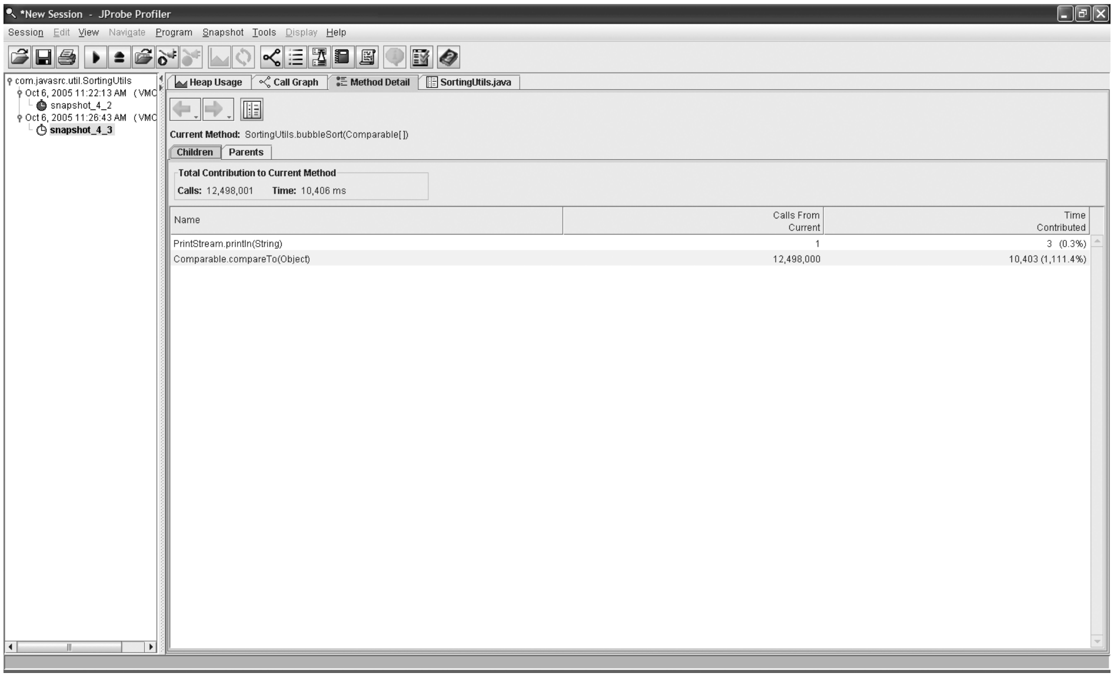
Package	Name	Number of Calls	Cumulative Time	Method Time
com.javasrc.util	SortingUtils.main(String[ ])	1	168,555 (100.0%)	90 (0.1%)
com.javasrc.util	SortingUtils.bubbleSort(Comparable[ ])	1	80,455 (47.7%)	70,049 (41.6%)
com.javasrc.util	SortingUtils.selectionSort(Comparable[ ])	1	56,414 (33.5%)	46,069 (27.3%)
com.javasrc.util	SortingUtils.insertionSort(Comparable[ ])	1	28,601 (17.0%)	23,433 (13.9%)
com.javasrc.util	SortingUtils.heapSort(Comparable[ ])	1	1,008 (0.6%)	72 (0.0%)
com.javasrc.util	SortingUtils.siftDown(Comparable[ ], int, int)	7,499	936 (0.6%)	835 (0.5%)
com.javasrc.util	SortingUtils.mergeSort(Comparable[ ])	1	922 (0.5%)	0 (0.0%)
com.javasrc.util	SortingUtils.m_sort(Comparable[ ], Comparable[ ], int, int)	9,999	922 (0.5%)	81 (0.0%)
com.javasrc.util	SortingUtils.merge(Comparable[ ], Comparable[ ], int, int, int)	4,999	841 (0.5%)	798 (0.5%)
com.javasrc.util	SortingUtils.shellSort(Comparable[ ])	1	557 (0.3%)	489 (0.3%)
com.javasrc.util	SortingUtils.quickSort(Comparable[ ])	1	496 (0.3%)	0 (0.0%)
com.javasrc.util	SortingUtils.q_sort(Comparable[ ], int, int)	5,000	496 (0.3%)	436 (0.3%)

23 nodes in model, 23 visible in graph, 12 visible in list

Figure 5-7. *The profiled methods used to sort 5,000 random integers using the seven sorting algorithms*

We view the method response times sorted by cumulative time, because some of the algorithms make repeated calls to other methods to perform their sorting (for example, the `quickSort()` method makes 5,000 calls to `q_sort()`). We have to ignore the `main()` method, because it calls all seven sorting methods. (Its cumulative time is almost 169 seconds, but its exclusive method time is only 90 milliseconds, demonstrating that most of its time is spent in other method calls—namely, all of the sorting method calls.) The slowest method by far is the `bubbleSort()` method, accounting for 80 seconds in total time and 47.7 percent of total run time for the program.

The next question is, why did it take so long? Two pieces of information can give us insight into the length of time: the number of external calls the method makes and the amount of time spent on each line of code. Figure 5-8 shows the number of external calls that the `bubbleSort()` method makes.


*New Session - JProbe Profiler
Session Edit View Navigate Program Snapshot Tools Display Help
com.javasrc.util.SortingUtils
Oct 6, 2005 11:22:13 AM (VMC
snapshot_4_2
Oct 6, 2005 11:26:43 AM (VMC
snapshot_4_3
Heap Usage
Call Graph
Method Detail
SortingUtils.java
Children
Parents

Current Method: SortingUtils.bubbleSort(Comparable[])

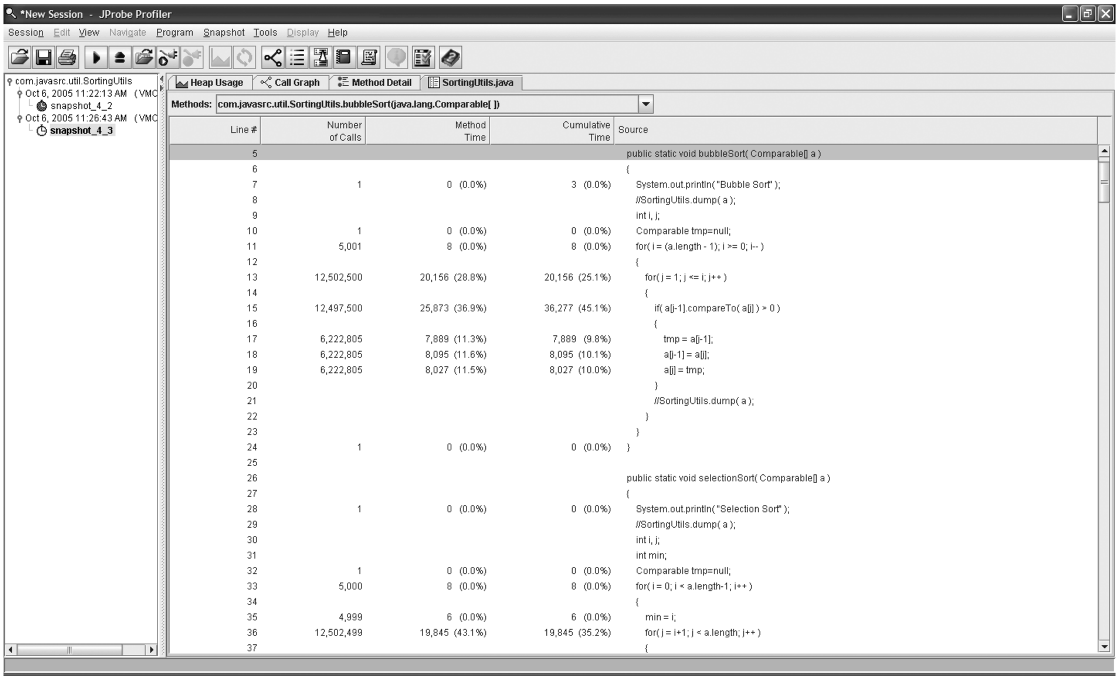
Total Contribution to Current Method

Calls: 12,498,001 Time: 10,406 ms

Name	Calls From Current	Time Contributed
PrintStream.println(String)	1	3 (0.3%)
Comparable.compareTo(Object)	12,498,000	10,403 (1,111.4%)

Figure 5-8. *The number of external calls that the bubbleSort() method makes*

This observation is significant—in order to sort 5,000 items, the bubble sort algorithm required almost 12.5 million comparisons. It immediately alerts us to the fact that if we have a considerable number of items to sort, bubble sort is not the best algorithm to use. Taking this example a step further, Figure 5-9 shows a line-by-line breakdown of call counts and time spent inside the `bubbleSort()` method.


*New Session - JProbe Profiler
Session Edit View Navigate Program Snapshot Tools Display Help
com.javasrc.util.SortingUtils
Oct 6, 2005 11:22:13 AM (VMC
snapshot_4_2
Oct 6, 2005 11:26:43 AM (VMC
snapshot_4_3
Heap Usage
Call Graph
Method Detail
SortingUtils.java

Methods: com.javasrc.util.SortingUtils.bubbleSort(java.lang.Comparable[ ])

Line #	Number of Calls	Method Time	Cumulative Time	Source
5				public static void bubbleSort(Comparable[] a)
6				{
7	1	0 (0.0%)	3 (0.0%)	System.out.println("Bubble Sort");
8				//SortingUtils.dump(a);
9				int i, j;
10	1	0 (0.0%)	0 (0.0%)	Comparable tmp=null;
11	5,001	8 (0.0%)	8 (0.0%)	for(i = (a.length - 1); i >= 0; i--)
12				{
13	12,502,500	20,156 (28.8%)	20,156 (25.1%)	for(j = 1; j <= i; j++)
14				{
15	12,497,500	25,873 (36.9%)	36,277 (45.1%)	if(a[j-1].compareTo(a[j]) > 0)
16				{
17	6,222,805	7,889 (11.3%)	7,889 (9.8%)	tmp = a[j-1];
18	6,222,805	8,095 (11.6%)	8,095 (10.1%)	a[j-1] = a[j];
19	6,222,805	8,027 (11.5%)	8,027 (10.0%)	a[j] = tmp;
20				}
21				//SortingUtils.dump(a);
22				}
23				}
24	1	0 (0.0%)	0 (0.0%)	}
25				
26				public static void selectionSort(Comparable[] a)
27				{
28	1	0 (0.0%)	0 (0.0%)	System.out.println("Selection Sort");
29				//SortingUtils.dump(a);
30				int i, j;
31				int min;
32	1	0 (0.0%)	0 (0.0%)	Comparable tmp=null;
33	5,000	8 (0.0%)	8 (0.0%)	for(i = 0; i < a.length-1; i++)
34				{
35	4,999	6 (0.0%)	6 (0.0%)	min = i;
36	12,502,499	19,845 (43.1%)	19,845 (35.2%)	for(j = i+1; j < a.length; j++)
37				{

Figure 5-9. *Profiling the bubbleSort() method*

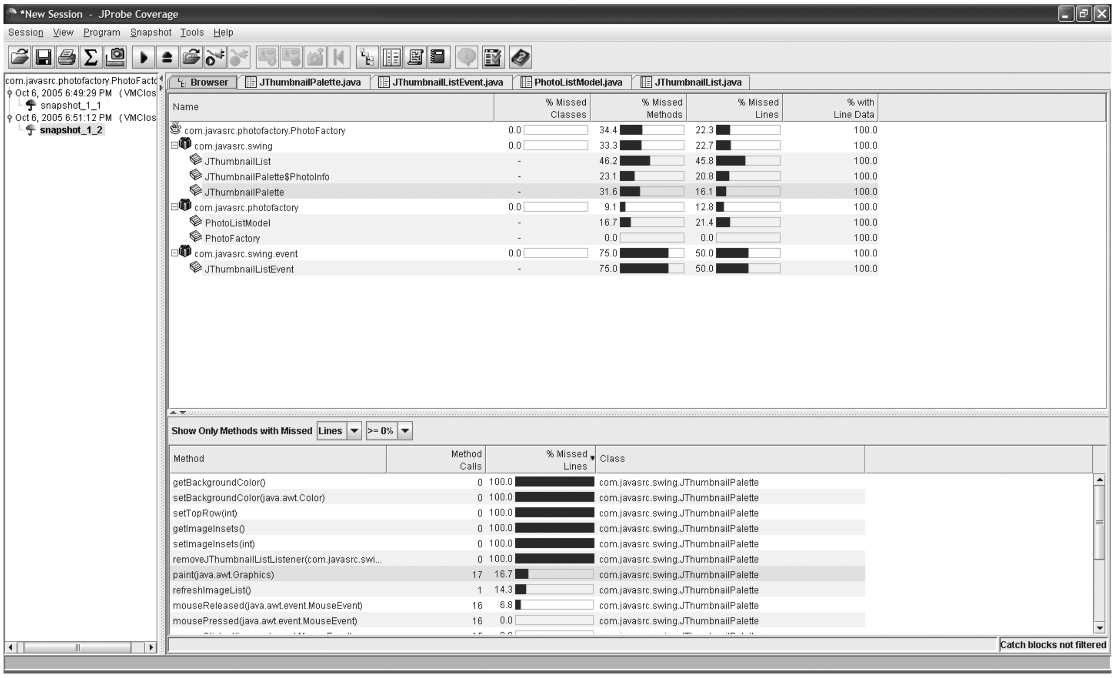
By profiling the `bubbleSort()` method, we see that 45 percent of its time is spent comparing items, and 25 percent is spent managing a `for` loop; these two lines account for 56 cumulative seconds. Figure 5-9 clearly illustrates the core issue of the bubble sort algorithm: on line 15 it executes the `for` loop 12,502,500 times, which resolves to 12,479,500 comparisons.

To be successful in deploying high-performance components and applications, you need to apply this level of profiling to your code.

Coverage Profiling

Identifying and rectifying memory issues and slow-running algorithms gives you confidence in the quality of your components, but that confidence is meaningful only as long as you are exercising all—or at least most—of your code. That is where coverage profiling comes in; coverage profiling reveals the percentage of classes, methods, and lines of code that are executed by a test. Coverage profiling can provide strong validation that your unit and integration tests are effectively exercising your components.

In this section, I'll show a test of a graphical application that I built to manage my digital pictures running inside of a coverage profiler filtered according to my classes. I purposely chose not to test it extensively in order to present an interesting example. Figure 5-10 shows a class summary of the code that I tested, with six profiled classes in three packages displayed in the browser window and the methods of the `JThumbnailPalette` class with missed lines in the pane below.


*New Session - JProbe Coverage
Session View Program Snapshot Tools Help
com.javasrc.photofactory.PhotoFacto
Oct 6, 2005 6:49:29 PM (VMClos
snapshot_1_1
Oct 6, 2005 6:51:12 PM (VMClos
snapshot_1_2
Browser
JThumbnailPalette.java
JThumbnailListEvent.java
PhotoListModel.java
JThumbnailList.java

Name	% Missed Classes	% Missed Methods	% Missed Lines	% with Line Data
com.javasrc.photofactory.PhotoFactory	0.0	34.4	22.3	100.0
com.javasrc.swing	0.0	33.3	22.7	100.0
JThumbnailList	-	46.2	45.8	100.0
JThumbnailPalette$PhotoInfo	-	23.1	20.8	100.0
JThumbnailPalette	-	31.6	16.1	100.0
com.javasrc.photofactory	0.0	9.1	12.8	100.0
PhotoListModel	-	16.7	21.4	100.0
PhotoFactory	-	0.0	0.0	100.0
com.javasrc.swing.event	0.0	75.0	50.0	100.0
JThumbnailListEvent	-	75.0	50.0	100.0

Show Only Methods with Missed Lines >= 0%

Method	Method Calls	% Missed Lines	Class
getBackgroundColor()	0	100.0	com.javasrc.swing.JThumbnailPalette
setBackgroundColor(java.awt.Color)	0	100.0	com.javasrc.swing.JThumbnailPalette
setTopRow(int)	0	100.0	com.javasrc.swing.JThumbnailPalette
getImageInsets()	0	100.0	com.javasrc.swing.JThumbnailPalette
setImageInsets(int)	0	100.0	com.javasrc.swing.JThumbnailPalette
removeJThumbnailListListener(com.javasrc.swi...	0	100.0	com.javasrc.swing.JThumbnailPalette
paint(java.awt.Graphics)	17	16.7	com.javasrc.swing.JThumbnailPalette
refreshImageList()	1	14.3	com.javasrc.swing.JThumbnailPalette
mouseReleased(java.awt.event.MouseEvent)	16	6.8	com.javasrc.swing.JThumbnailPalette
mousePressed(java.awt.event.MouseEvent)	16	0.0	com.javasrc.swing.JThumbnailPalette

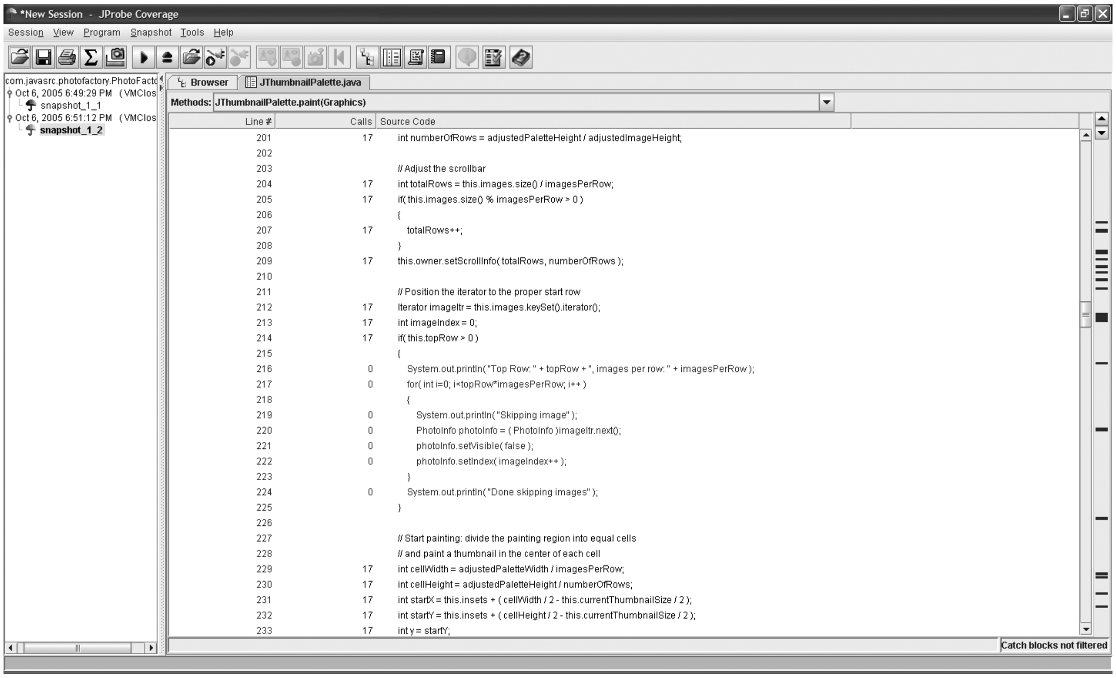
Catch blocks not filtered

Figure 5-10. *Coverage profile of a graphical application*

The test exercised all six classes, but missed a host of methods and classes. For example, in the `JThumbnailPalette` class, the test completely failed to call the methods `getBackgroundColor()`, `setBackgroundColor()`, `setTopRow()`, and others. Furthermore, even though the `paint()` method was called, the test missed 16.7 percent of the lines. Figure 5-11 shows the specific lines of code within the `paint()` method that the test did not execute.

Figure 5-11 reveals that most lines of code were executed 17 times, but the code that handles painting a scrolled set of thumbnails was skipped. With this information in hand, the person needs to move the scroll bar, or configure an automated test script to move it, to ensure that this piece of code is executed.

Coverage is a powerful profiling tool, because without it, you may miss code that your users will encounter when they use your application in a way that you do not expect (and rest assured, they definitely will).


*New Session - JProbe Coverage
Session View Program Snapshot Tools Help
com.javasrc.photofactory.PhotoFacto
Oct 6, 2005 6:49:29 PM (VMClos
snapshot_1_1
Oct 6, 2005 6:51:12 PM (VMClos
snapshot_1_2
Browser
JThumbnailPalette.java
Catch blocks not filtered

Methods: JThumbnailPalette.paint(Graphics)

```
Line #   Calls   Source Code
  201      17    int numberOfRows = adjustedPaletteHeight / adjustedImageHeight;
  202
  203            // Adjust the scrollbar
  204      17    int totalRows = this.images.size() / imagesPerRow;
  205      17    if( this.images.size() % imagesPerRow > 0 )
  206            {
  207      17      totalRows++;
  208            }
  209      17    this.owner.setScrollInfo( totalRows, numberOfRows );
  210
  211            // Position the iterator to the proper start row
  212      17    Iterator imageItr = this.images.keySet().iterator();
  213      17    int imageIndex = 0;
  214      17    if( this.topRow > 0 )
  215            {
  216       0      System.out.println( "Top Row: " + topRow + ", images per row: " + imagesPerRow );
  217       0      for( int i=0; i<topRow*imagesPerRow; i++ )
  218              {
  219       0        System.out.println( "Skipping image" );
  220       0        PhotoInfo photoInfo = ( PhotoInfo )imageItr.next();
  221       0        photoInfo.setVisible( false );
  222       0        photoInfo.setIndex( imageIndex++ );
  223              }
  224       0      System.out.println( "Done skipping images" );
  225            }
  226
  227            // Start painting: divide the painting region into equal cells
  228            // and paint a thumbnail in the center of each cell
  229      17    int cellWidth = adjustedPaletteWidth / imagesPerRow;
  230      17    int cellHeight = adjustedPaletteHeight / numberOfRows;
  231      17    int startX = this.insets + ( cellWidth / 2 - this.currentThumbnailSize / 2 );
  232      17    int startY = this.insets + ( cellHeight / 2 - this.currentThumbnailSize / 2 );
  233      17    int y = startY;
```

Figure 5-11. *A look inside the JThumbnailPalette's paint() method*

Performance in Quality Assurance

The integration of components usually falls more on development than on QA, but the exercise usually ends at functional testing. Development ensures that the components work together as designed, and then the QA team tests the details of the iteration's use cases. Now that your use cases have performance criteria integrated, QA has a perfect opportunity to evaluate the iteration against the performance criteria. The new notion that I am promoting is that an application that meets all of its functional requirements but does not satisfy its SLAs does not pass QA. The response by the QA team should be the same as if the application is missing functionality: the application is returned to development to be fixed.

Performance integration testing comes in two flavors:

- Performance integration general test
- Performance integration load test

QA performs the integration general test under minimal load; the amount of that load is a subset of the expected load and defined formally in the test plan. For example, if the expected load is 1,500 simultaneous users, then this test may be against 50 users. The purpose of this test is to identify any gross performance problems that might occur as the components are integrated. Do not run a full-load test, because in a failed full-load test it may be difficult to identify the root cause of the performance failure. If the load is completely unsustainable, then all aspects of the application and environment will most likely fail. Furthermore, if the integrated application cannot satisfy a minimal load, then there is no reason to subject it to a full load.

After the application has survived the performance integration general test, the next test is the performance integration load test. During this test, turn up the user load to the expected user load, or if you do not have a test environment that mirrors production, then use a single JVM scaled down appropriately. For example, if you are trying to support 1,500 users with four JVMs, then you might send 400 users at a single JVM. Each use case that has been implemented in this integration is tested against the formal use case SLAs. The performance integration load test is probably the most difficult one for the application to pass, but it offers the ability to tune the application and application server, and it ensures that the performance of the application stays on track.

Balanced Representative Load Testing

Probably the most important aspect of performance tuning in integration or staging environments is ensuring that you are accurately reproducing the behavior of your users. This is referred to as *balanced representative load testing*. Each load scenario that you play against your environment needs to represent a real-world user interaction with your application, complete with accurate think times (that is, the wait time between requests). Furthermore, these representative actions must be balanced according to their observed percentage of occurrence.

For example, a user may log in once, but then perform five searches, submit one form, and log out. Therefore the logon, logoff, and submission functionalities should each receive a balance of one-eighth of the load, and the search functionality should receive the remaining five-eighths of the load for this transaction. If your load scripts do not represent real-world user actions balanced in the way users will be using your application, then you can have no confidence that your tuning efforts are valid. Consider this example if the actions were not balanced properly (say each action receives one-fourth of the load). Logon and logoff functionalities may be far less database-intensive than search functionality, but they may be much heavier on a JCA connector to a Lightweight Directory Access Protocol (LDAP) server. Tuning each function equally results in too few database connections to service your database requests and extraneous JCA connections. A simple misbalance of respective transactions can disrupt your entire environment.

There are two primary techniques to extracting end-user behaviors: process access log files or add a network device into your environment that monitors end-user behavior. The former is the less exact of the two but can provide insight into user pathways through your Web site and accurate think times. The latter is more exact and can be configured to provide deeper insight into customer profiling and application logic.

Production Staging Testing

Seldom will your applications run in isolation; rather, they typically run in a shared environment with other applications competing for resources. Therefore, testing your applications in a staging environment designed to mirror production is imperative. As the integration test phase is split into two steps, so is the production staging test:

- Performance production staging general test
- Performance production staging load test

The general test loads the production staging environment with a small user load with the goal of uncovering any egregiously slow functionality or drained resources. Again, this step is interjected before performing the second, full-load test, because a full-load test may completely break the environment and consume all resources, thereby obfuscating the true cause of performance issues. If the application cannot satisfy a minimal amount of load while running in a shared environment, then it is not meaningful to subject it to excessive load.

Identifying Performance Issues

When running these performance tests, you need to pay particular attention to the following potentially problematic environmental facets:

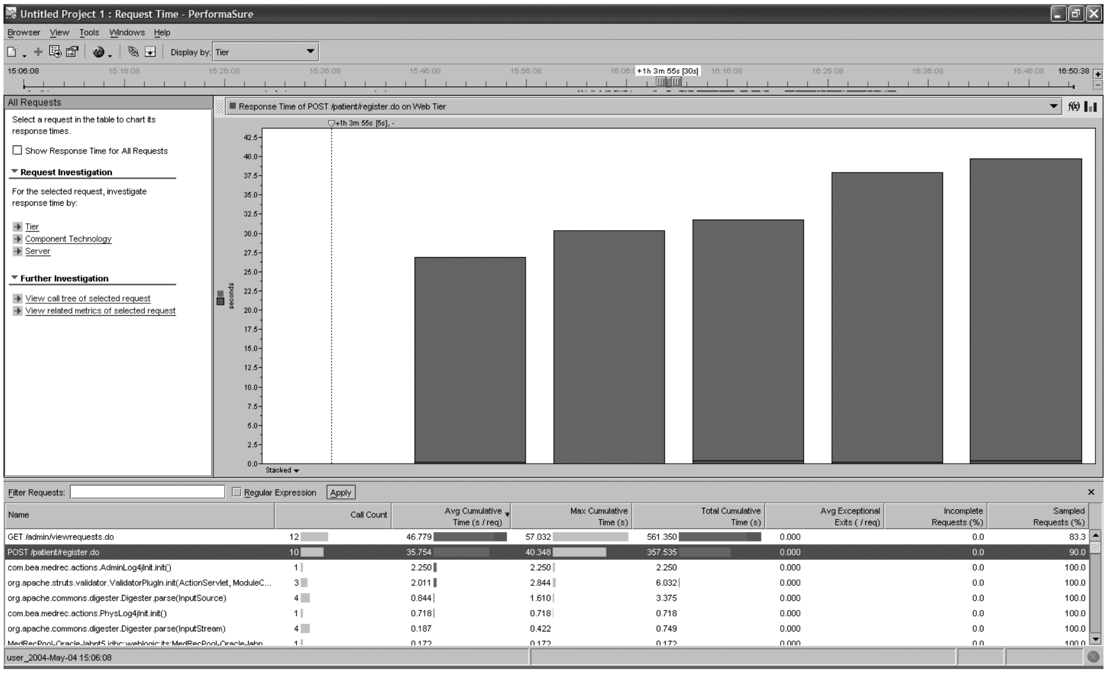
- Application code
- Platform configuration
- External resources

Application code can perform poorly as a result of being subjected to a significant user load. Performance unit tests help identify poorly written algorithms, but code that performs well under low amounts of user load commonly experiences performance issues as the load is significantly increased. The problems occur because subtle programmatic issues manifest themselves as problems only when they become exaggerated. Consider creating an object inside a servlet to satisfy a user request and then destroying it. This is no problem whatsoever for a single user or even a couple dozen users. Now send 5,000 users at that servlet—it must create and destroy that object 5,000 times. This behavior results in excessive garbage collection, premature tenuring of objects, CPU spikes, and other performance abnormalities. This example underscores the fact that only after testing under load can you truly have confidence in the quality of your components.

Platform configuration includes the entire environment that the application runs in: the application server, JVM, operating system, and hardware. Each piece of this layered execution model must be properly configured for optimal performance. As integration and production staging tests are run, you need to monitor and assess their performance. For example, you need to ensure that you have enough threads in the application server to process incoming requests, that your JVM's heap is properly tuned to minimize major garbage collections, that your operating system's process scheduler is allotting enough CPU to the JVM, and that your hardware is running optimally on a fast network. Ensuring proper configuration requires a depth of knowledge across a breadth of technologies.

Finally, most enterprise-scale applications interact with external resources that may or may not be under your control. In the most common cases, enterprise applications interact with one or more databases, but external resources can include legacy systems, messaging servers, and, in recent years, Web services. As the acceptance of SOAs has grown, applications can be rapidly assembled by piecing together existing code that exposes functionality through services. Although this capability promotes the application architect to an application assembler, permitting rapid development of enterprise solutions, it also adds an additional tier to the application. And with that tier comes additional operating systems, environments, and, in some circumstances, services that can be delivered from third-party vendors at run time over the Internet.

The first step in identifying performance issues is to establish monitoring capabilities in your integration and production staging environments, and record the application behavior while under load. This record lists service requests that can be sorted by execution count, average execution time, and total execution time. These service requests are then tracked back to use cases to validate against predefined SLAs. Any service request whose response time exceeds its SLA needs to be analyzed to determine why that's the case. Figure 5-12 shows a breakdown of service requests running inside the MedRec application. In this 30-second time slice, two service requests spent an extensive amount of time executing: `GET /admin/viewrequests.do` was executed 12 times, accounting for 561 seconds, and `POST /patient/register.do` was executed 10 times, accounting for 357 seconds.


Untitled Project 1 : Request Time - PerformaSure
Browser View Tools Windows Help
Display by: Tier
All Requests
Response Time of POST /patient/register.do on Web Tier
+1h 3m 55s [30s]
seconds
Stacked
Filter Requests:
Regular Expression
Apply

Select a request in the table to chart its response times.

Show Response Time for All Requests

Request Investigation

For the selected request, investigate response time by:

- Tier
- Component Technology
- Server

Further Investigation

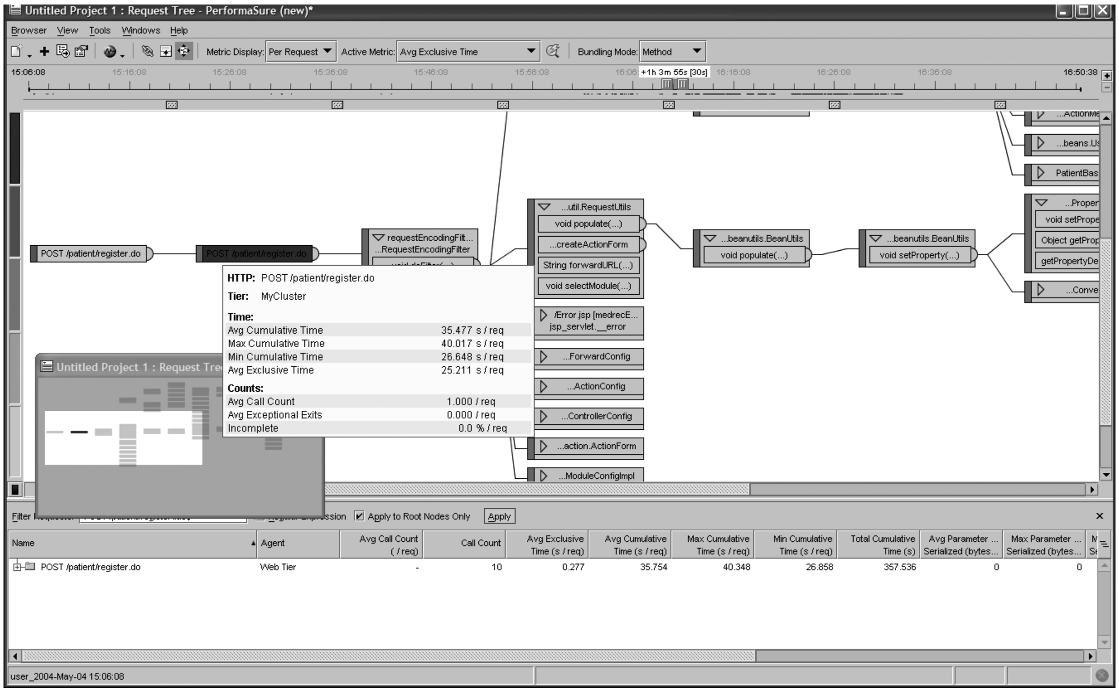
- View call tree of selected request
- View related metrics of selected request

Name	Call Count	Avg Cumulative Time (s / req)	Max Cumulative Time (s)	Total Cumulative Time (s)	Avg Exceptional Exits (/ req)	Incomplete Requests (%)	Sampled Requests (%)
GET /admin/viewrequests.do	12	46.779	57.032	561.350	0.000	0.0	83.3
POST /patient/register.do	10	35.754	40.348	357.535	0.000	0.0	90.0
com.bea.medrec.actions.AdminLog4jInit.init()	1	2.250	2.250	2.250	0.000	0.0	100.0
org.apache.struts.validator.ValidatorPlugIn.init(ActionServlet, ModuleC...	3	2.011	2.844	6.032	0.000	0.0	100.0
org.apache.commons.digester.Digester.parse(InputSource)	4	0.844	1.610	3.375	0.000	0.0	100.0
com.bea.medrec.actions.PhysLog4jInit.init()	1	0.718	0.718	0.718	0.000	0.0	100.0
org.apache.commons.digester.Digester.parse(InputStream)	4	0.187	0.422	0.749	0.000	0.0	100.0
MedRecPool-Oracle-Jabnt5 jdbc:weblogic:jts:MedRecPool-Oracle-Jabn	1	0.172	0.172	0.172	0.000	0.0	100.0

user_2004-May-04 15:06:08

Figure 5-12. *Breakdown of service requests running inside the MedRec application*

As shown in Figure 5-13, looking at the average exclusive time for each method that satisfies the `POST /patient/register.do` service request, the HTTP POST at the WebLogic cluster consumed on average 35.477 seconds of the 35.754 total service request average, which is important because the request passed quickly from the Web server to the application server, but then waited at the application server for a thread to process it. The remainder of the request processed relatively quickly.



Metric Display: Per Request Active Metric: Avg Exclusive Time Bundling Mode: Method

HTTP: POST /patient/register.do
Tier: MyCluster

Time:	
Avg Cumulative Time	35.477 s / req
Max Cumulative Time	40.017 s / req
Min Cumulative Time	26.648 s / req
Avg Exclusive Time	25.211 s / req

Counts:	
Avg Call Count	1.000 / req
Avg Exceptional Exits	0.000 / req
Incomplete	0.0 % / req

☑ Apply to Root Nodes Only Apply

Name	Agent	Avg Call Count (/ req)	Call Count	Avg Exclusive Time (s / req)	Avg Cumulative Time (s / req)	Max Cumulative Time (s / req)	Min Cumulative Time (s / req)	Total Cumulative Time (s)	Avg Parameter ... Serialized (bytes...	Max Parameter ... Serialized (bytes...
POST /patient/register.do	Web Tier	-	10	0.277	35.754	40.348	26.858	357.536	0	0

Figure 5-13. *Breakdown of response time for the POST /patient/register.do service request for each method in a hierarchical request tree*

Figure 5-14 shows a view of the performance metrics for the application server during this recorded session. This screen is broken into three regions: the top region shows the heap behavior, the middle shows the thread pool information, and the bottom shows the database connection pool information.

Figure 5-14 confirms our suspicions: the number of idle threads during the session hit zero, and the number of pending requests grew as high as 38. Furthermore, toward the end of the session, the database connection usage peaked at 100 percent and the heap was experiencing significant garbage collection.

This level of diagnosis requires insight into the application, application server, and external dependency behaviors. With this information, you are empowered to determine exactly where and why your application is slowing.

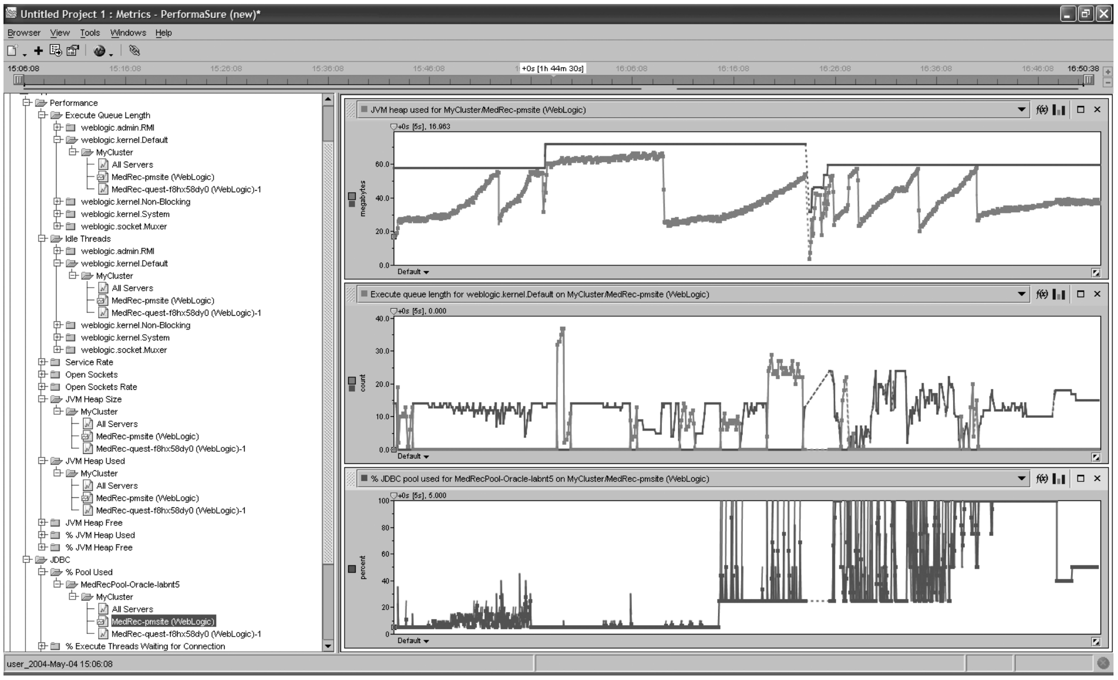

Untitled Project 1 : Metrics - PerformaSure (new)*
Browser View Tools Windows Help
15:06:08
+0s [1h 44m 30s]
16:50:38
Performance
Execute Queue Length
Idle Threads
Service Rate
Open Sockets
Open Sockets Rate
JVM Heap Size
JVM Heap Used
JVM Heap Free
% JVM Heap Used
% JVM Heap Free
JDBC
% Pool Used
MedRecPool-Oracle-labnt5
MyCluster
All Servers
MedRec-pmsite (WebLogic)
MedRec-quest-f8hx58dy0 (WebLogic)-1
% Execute Threads Waiting for Connection
JVM heap used for MyCluster/MedRec-pmsite (WebLogic)
megabytes
Execute queue length for weblogic.kernel.Default on MyCluster/MedRec-pmsite (WebLogic)
count
% JDBC pool used for MedRecPool-Oracle-labnt5 on MyCluster/MedRec-pmsite (WebLogic)
percent
Default
user_2004-May-04 15:06:08

Figure 5-14. *Performance metrics for the application server during this recorded session*

Summary

In this chapter, you learned how to integrate proactive performance testing throughout the development life cycle. The process begins by integrating performance criteria into use cases, which involves modifying use cases to include specific SLA sections that include performance criteria for each use case scenario. Next, as components are built, performance unit tests are performed alongside functional unit tests. These performance tests include testing for memory issues, and code issues, and the validation of the coverage of tests to ensure that the majority of component code is being tested. Finally, as components are integrated and tested in a production staging environment, application bottlenecks are identified and resolved.

In the next chapter, we'll look at a formal performance tuning methodology that allows you to maximize your tuning efforts by tuning the application and application server facets that yield the most significant improvements. By the end of the next chapter, you'll be empowered to bring your application and environment to within 80 percent of their ideal configuration, regardless of your deployment environment.