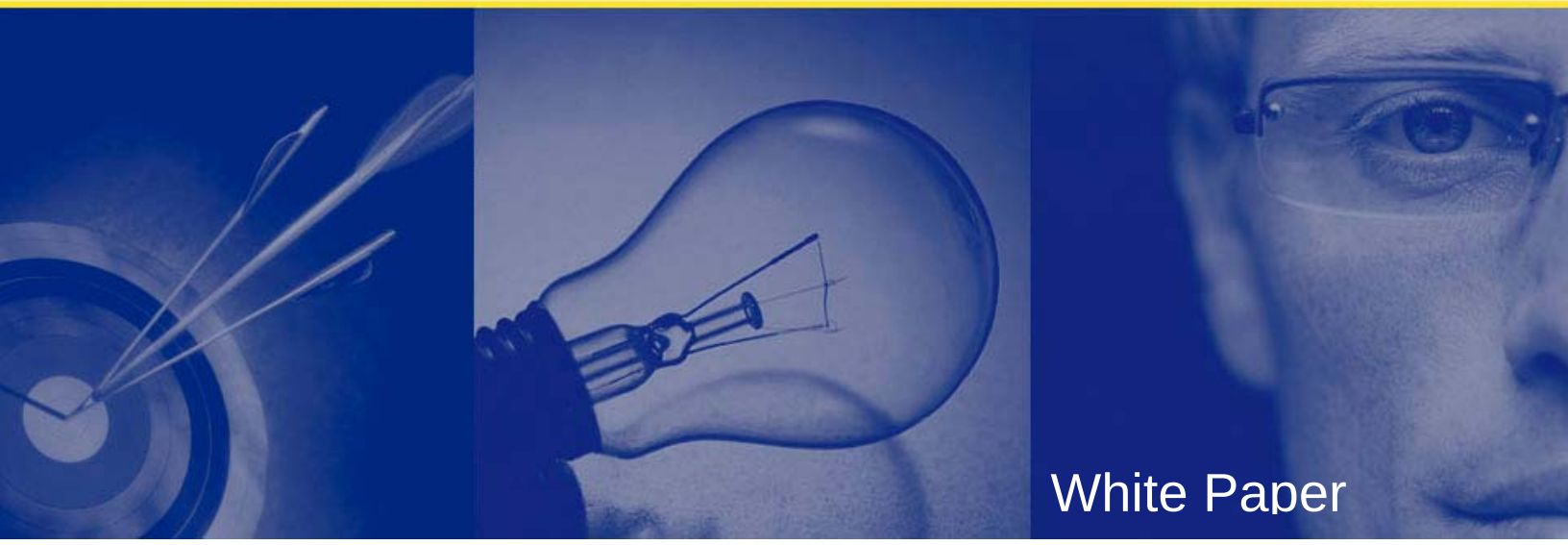
QUEST SOFTWARE®

Realizing Continuous Performance Management Best Practices Documentation

Written by
Steven Haines
Java Domain Expert
Quest Software, Inc.



World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
e-mail: info@quest.com
U.S. and Canada: 949.754.8000

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

Updated—May, 2007

CONTENTS

ABSTRACT 4

INTRODUCTION 5

WHY IS THE CURRENT STATE OF SOFTWARE TESTING FAILING? 7

INTRODUCING TEST-DRIVEN DEVELOPMENT AND CONTINUOUS INTEGRATION 9

- TEST-DRIVEN DEVELOPMENT LIFECYCLE 9
- BENEFITS 10
- CONTINUOUS INTEGRATION 11
- CONTINUOUS INTEGRATION DEFINED 12
- PRACTICES OF CONTINUOUS INTEGRATION 12
- CONTINUOUS PERFORMANCE MANAGEMENT 16

ACHIEVING CONTINUOUS PERFORMANCE MANAGEMENT 19

- CONTINUOUS PERFORMANCE MANAGEMENT PREREQUISITES 19
- CONTINUOUS PERFORMANCE MANAGEMENT IN UNIT TESTS 21
- UNIT TEST PERFORMANCE ANALYSIS REPORT 24
- CONTINUOUS INTEGRATION SERVER 26
- PERFORMANCE UNIT TESTS IN A CONTINUOUS INTEGRATION SERVER 29
- PROBLEM DIAGNOSIS SCENARIO – SLOW RUNNING CODE 31
- PROBLEM DIAGNOSIS SCENARIO – MEMORY LEAK 33
- CONTINUOUS PERFORMANCE MANAGEMENT IN INTEGRATION AND LOAD TESTS 35
- PROBLEM DIAGNOSIS SCENARIO – SLOW RUNNING CODE 38
- SUMMARY 41

ABOUT THE AUTHOR 42

ABSTRACT

When poorly performing code slips into production and a customer-facing application goes down or responds slowly, the organization will ultimately lose revenue, customers and credibility. Code problems in internal business applications also hamper productivity and increase operating costs.

Organizations can avoid many code problems in production through continuous performance management in development. By proactively implementing best practices, applications will operate seamlessly. Daily, repeated unit testing and integration analysis can ensure stability when the application is in staging. The result is dramatic cost-efficiency and reliability in developing applications.

INTRODUCTION

The quality and performance of enterprise applications in our industry are astonishingly poor. Consider the following:

- According to Forrester Research, nearly 85 percent of companies with revenue of more than $1 billion reported incidents of significant application performance problems. Survey respondents identified the architecture and deployment as the primary causes of these problems. Based on this survey, it seems that no matter how much you tune your hardware and environment, applications problems will persist.
- Infonetics Research found that medium-sized businesses (101 to 1,000 employees) are losing an average of 1 percent of their annual revenue, or $867,000, to downtime. Application outages and degradations are the biggest sources of downtime, costing these companies $213,000 annually.

The cost of outages and performance degradation can be calculated in several ways, based on the type of application.

- Business-to-consumer: direct loss of sales revenue because of web site abandonment and no confidence
- Business-to-business: loss of credibility, which can eventually lead to a lost business partnerships
- Internal applications: loss of employee productivity

With respect to business-to-consumer applications, consider the typical shopping pattern. For example, I usually find an item at the major retailers and then search for the best price online. In the end, the prices do not usually vary more than 5 percent, so I choose a vendor I trust within that range. However, if the vendor's site is running slow or makes finalizing the purchase too difficult, I simply move down my list to my next preferred vendor. By and large, retailer loyalty has been replaced by price competition and convenience. So even a minor site outage means lost business.

In a business-to-business relationship, the stakes are much higher. If I am one of several companies that sells widgets and I have established a relationship with a major retailer, the reliability and performance of my B2B application is my livelihood. If the major retailer submits an order for additional widgets and my application in unavailable or the order is lost, I run the risk of losing the partnership – and the revenue. Continual application problems mean lost accounts.

Slow or poorly performing internal applications can also impact the bottom line. Employee productivity suffers when an application goes down or responds slowly. More significantly, lost time can delay product delivery. When that happens, a company can either delay the product release or reduce the scope of the release to meet deadlines. Either option can lead to a disaster.

Let's use a software company as an example. Presenting the perfect software to a customer *after* it purchases from a competitor is a lost sale. And presenting software that is missing key features can result in the loss of competitive edge and hence the loss of the sale.

WHY IS THE CURRENT STATE OF SOFTWARE TESTING FAILING?

Current software testing is inadequate as it doesn't address functional and performance issues early enough in the software development lifecycle. In extreme cases, formal testing is just an afterthought. And even formal testing plans often lack adequate time allotment and detail to identify core problems. It is not sufficient for a development organization to simply package an application's subcomponents and hand them to quality assurance with a set of use cases. Instead, proactive attention is required to test an application from the inception of a software development project.

There are several critical problems with the state of test suites that development and quality assurance teams are producing today:

- Test suites are based on out-of-date artifacts
- Test suites written by someone other than the code author may not be comprehensive enough
- Unless test suites are automated, tests will not be performed regularly enough to be effective
- Unless test suites are automated, fixing one part of code may inadvertently break another part of the code and go unnoticed

One of the core benefits in implementing an iterative development methodology is that information gained later in the process can affect artifacts constructed earlier. For example, the available data in a monitoring product may limit its capabilities and require a design change. In the next iteration, the design is changed to reflect this new information. Unfortunately, architecture documents are not always updated completely to reflect changes. If test suites are developed during the quality assurance phase, they may be based on old artifacts and fail to accurately test significant portions of the application.

Following this same line of reasoning, if test suites are not written by the code's author they might only cover one facet of the code, neglecting other important functionality. Only the code author understands the code's behavior at a low level and what potential abnormalities may affect that behavior. For example, if the code in question is a search function, it may follow a different path if wild cards are used, or even fail if no input is provided. If the architecture artifacts only specify that a call into the search functionality accepts multiple search strings, then the test suites may not exercise the code thoroughly enough.

The key to effective testing is automation. And the key to successfully and rapidly developing software is automating tests regularly. An automated test suite running at the conclusion of an integration phase will not uncover errors soon enough to avoid impacting the iteration schedule. Consider that simple

errors may be compounded as additional code is written on top of those errors. If the errors are uncovered as soon as they are created (or realistically, within a couple days of coding) their impact is minimized.

Finally, automated testing is imperative to enable regression testing. It is the nature of programming that if code is changed, the behavior of other code that interacts with the changed code may break. If automated testing is performed regularly, however, then as one bug is fixed, code that breaks because of that fix is identified during the next test.

INTRODUCING TEST-DRIVEN DEVELOPMENT AND CONTINUOUS INTEGRATION

Test-driven development was created to address the limitations of our current testing practices. In short, test-driven development promotes the practice of software developers writing test cases for their code *prior* to implementing the code. Those test cases are then coalesced from all developers to build an exhaustive test suite. Specifically, test-driven development defines the following requirements:

- Test cases must be written by developers
- Each code component must be delivered with a corresponding test suite
- Test cases must be written prior to writing code

Requiring developers to write their own test cases helps ensure that all facets of their code are properly tested and written to the current implementation of the code, not out-of-date artifacts. Additionally, requiring developers to write test cases *prior* to writing code helps ensure that the code implements exactly what is required and nothing more. This helps reduce the side effects that sometimes plague code. Finally, by requiring that all code be delivered with accompanying test suites, the combined test suite is robust and accurate.

As this process is followed meticulously throughout the development lifecycle, the result is a thorough, accurate, and comprehensive test suite that will uncover functional errors when they occur and quickly pinpoint, or triage, the root cause of the failure.

Note: *While this strategy is effective for new projects, it is also an option for existing projects. I worked with a large company in the financial industry to slowly roll out such a strategy: we could not require that developers go back and write test cases for hundreds of thousands of lines of code, but we could require that all new code include accompanying test cases. Following the code evolution patterns of this company, we estimated about 80 percent test coverage over a two-year period. So if you have an existing project, do not tune out just yet!*

Test-Driven Development Lifecycle

Test-driven development does not limit itself to the three aforementioned requirements, but rather it includes a reproducible lifecycle to model the process. That lifecycle includes the following steps:

- Add a test to an existing test suite prior to implementing the underlying code
- Exercise the test suite to prove that the new test fails
- Implement the functionality exercised by the new test
- Exercise the test suite and ensure that the new test passes
- Refactor the code

The first step in building new code for your application is to construct a test case which exercises code functionality and integrates the code into your master test suite. Because the code has not yet been implemented, this test case should fail. Verify that the test does, in fact, fail – this ensures that the test case is properly written. If the test case passes, then you cannot be confident that it is properly testing your code, and hence you cannot have confidence in the code itself.

After implementing the new functionality, re-run the test suite to verify the previously failing test now passes. At this point you have satisfied the test case and can now re-factor the code to implement a more elegant solution. Because you have verified that the test case works and have successfully satisfied it, then you will discover that failure immediately if re-factoring breaks the code.

Benefits

As with any development methodology, there are pros and cons to test-driven development. But consider the following benefits of adopting this approach:

- Shorter development cycles
- Limited debugging
- Faster triaging
- Clearly solved business problems
- Confidence in deployment

Perhaps one of the biggest objections that I hear when proposing test-driven development is that it requires advance coding that could impact release schedules. After all, it takes time to write good test cases that properly test your code, both for success and failure cases. But how do you test your code now? Most often code is tested using debug log messages or `System.out.println()` messages. It takes a lot of time to prove that code works through this method. So if it takes the same amount of time to write test cases, then you might as well take this approach because of its other benefits.

Building granular test cases up front streamlines debugging efforts since the test cases identify exactly what code paths the tests are exercising. For

example, if a test case should fail for a search function's behavior when no input is supplied, you know exactly what to look for when examining your code. Further, if you walk through the code in a debugger, you know exactly what input is causing the failure.

A huge benefit of having an elaborate test suite is that when a failure occurs, triaging the root cause is straightforward. This is because a subset of test cases will fail and describe precisely the malfunction – including the specific input criteria. Without this test harness in place, problems will most likely be discovered by quality assurance in business test cases. A business test case, which usually represents a scenario in a use case, does not provide the granularity required to identify the root cause of the problem, but rather identifies the problem itself. For example, if the business test case is exercising the search functionality of the application and it fails, there is no way to know whether the failure is in the search algorithm, in a database query, or even in a database connection pool mis-configuration. The business test case is more like *black-box* testing: when I provide input X to functionality Y, I expect the result of Z; if I do not receive the result Z then the business test case fails.

By disassembling business cases to create test cases and developing them, you gain a deeper understanding of the business cases. And satisfying those test cases ensures that the code does, in fact, satisfy the business case in a clear and concise manner.

Finally, test-driven development provides you with confidence that when your application is deployed to a production environment that it will be functionally correct. Of course, your test cases must be robust. The percentage of code that your tests cover is directly proportional to your level of confidence. Weak test cases that only exercise a small percentage of your application code will result in a low level of confidence while strong test cases that exercise a high percentage of your application code will result in a high level of confidence.

Note: *This reminds me of a project for which we outsourced the development of a major component. The outsourcing company delivered the component without any logging, which would make production troubleshooting nearly impossible, so we asked them to add it. What we got was a single log message stating, "Logging initialized". The point is that even if you integrate good processes into your development lifecycle, they will not save you if they are not properly applied.*

Continuous Integration

I am indebted to Martin Fowler for publishing an outstanding article on continuous integration at www.martinfowler.com:

`http://www.martinfowler.com/articles/continuousIntegration.html`

In his article, he shares his observations from working as a summer intern at a large English electronics company. During this internship, he learned that the company had already spent several months integrating the code from developers into a working product. This story rang a bell because in the late '90s I was working for a factory automation company that did the same thing. It divided the application development into 12 iterations. At the completion of the first iteration, it took us nearly three months to integrate our parts into a product that could be passed to quality assurance. Martin's experience, as well as my own, illustrates that unless integration is a properly planned process, it can represent a bottleneck in an application's development lifecycle. Further, due to all of the variables that can occur during integration, it is very difficult to estimate how long the process will take. In extreme cases, integration may take longer than the development itself!

A facet of extreme programming (XP) promotes the notion of integrating your applications continually, or at least several times a day. When new code is committed to a central code repository, it should be compiled, then the entire project should be tested against an exhaustive unit test suite, as well as tested as an integrated whole. By doing this, we make integration a non-issue in the development lifecycle; if your code breaks the build (or causes the test suite to fail) then you know about the failure within a couple hours and can resolve it. When the development iteration is complete, you simply label a build and pass it to quality assurance to test it against business cases.

Continuous Integration Defined

Wikipedia defines continuous integration as:

> *A software engineering term describing a process that completely rebuilds and tests an application frequently…continuous integration is accomplished via a serialized build process. At the completion of a task, the developer (or development pair) takes a build token and runs the build process, including tests. If the tests pass, the task can be committed to the source code repository and release the token.*

Practices of Continuous Integration

To succeed at continuous integration, you must meet several specific requirements and follow a process. I have paraphrased the requirements that Martin Fowler presents in his article:

- Single code repository
- Build automation
- Self-testing
- Daily commits
- Build on integration machine

- Fast builds
- Test in a cloned production environment
- Public availability of the latest build
- Status broadcasting
- Automated deployment

It is important for all code and build artifacts to be checked into one central code repository. An automated process for building and testing the application needs access to everything. Further, this requirement makes it simple for anyone in the development team to successfully run local builds: check out the project and execute a single command to build it. This offers a huge benefit to new developers, quickly making them more productive.

Your choices of source code repositories are plentiful and include the following more popular ones:

- Concurrent Versions System (CVS)
- Subversion
- Perforce
- Microsoft Visual Source Safe
- Rational ClearCase

In my career, I have used all of these products and each one has its pros and cons. The most popular product is CVS. It is open source and freely available, but because of its age and exposure, its weaknesses are well known. Subversion is an open source product created to address CVS's weaknesses and I use it to maintain my own code. Perforce, Visual SourceSafe, and ClearCase are all commercial products that you might want to evaluate.

Regardless of your selection, the first step is to get all of your source code and build artifacts into the source code repository and then create a *mainline* branch that everyone works against. You may have multiple branches (or concurrent versions of the source code) in the future to signify the completion of iterations, bug fixes, product releases, and so forth, but you need a consistent branch that everyone is working from to ensure that all integrations are in synch with one another.

A build script that can produce the application with a single command needs to be included in those artifacts. There are various build tools, but for the purposes of this white paper we will use Apache Ant, an XML-based build tool that has gained wide acceptance throughout the software industry. It has its competitors, the main one being Maven. But if you familiarize yourself with Ant, not only will you have a powerful tool for building your applications, but you will also be able to build most of the open source projects on the web today. Ant can build your applications, execute test scripts, and report failures when test scripts fail.

When implementing continuous integration, we are going to require all test cases to succeed before declaring a build to be successful, even if the code compiles properly. Continuous integration works hand in hand with test-driven development, so all test-driven development requirements are still applicable. All components of the application must be delivered with a thorough test suite and the test suite will be compiled into an exhaustive test harness that will be executed during every build.

The process that we require developers to implement is:

- Check out the mainline branch of code for the application
- Write a test case for the functionality being developed
- Execute a local build of the project with the integrated test case and verify that it fails
- Build the code functionality
- Execute another local build of the project and verify that this test case and others in the test harness succeed
- Re-factor the code if necessary
- Verify that the re-factoring did not break the test case with another local build
- Check in and commit the changes to the source code repository; if there is a conflict with other developers, check out the latest mainline branch, re-test with the new code and test cases, and then check in and commit the changes

A core tenet of continuous integration is that each developer needs to commit code frequently, at least daily. If the functionality is too coarse to be completed in a few hours or a day, then it should be broken down into a form that's granular enough for completion in at least a day. The purpose of this is to ensure that developers do not check out code from the mainline branch and work against it for two weeks only to learn that they or someone else inadvertently broke the project. Fixing a problem after two weeks of independent developer work is a time-consuming task, but fixing a problem after two hours of developer work is much more manageable.

Once a developer successfully commits code to the source code repository, the continuous integration system (which will be an open source product called CruiseControl in our examples) detects that code has changed, checks out the code, then builds and tests the entire product on an integration machine. The integration machine is a "clean" environment, meaning that it does not have any developer tools. It has everything on it to run the application, such as a database, application server, a Java Virtual Machine, and so forth. Other than the base set up, it only has access to the artifacts in the source code repository, which is why it is so important to put everything required to build and run the application there.

CruiseControl was originally developed by ThoughtWorks – Martin Fowler's company – and released as an open source project on SourceForge.net. At the time of this writing, several of the ThoughtWorks developers are active committers to the project and Martin reports that they use CruiseControl on "nearly every project we do and have been very happy with the results."

It is important to build the project on the integration machine. Developers, myself included, build code that works seamlessly on a local machine, but it may crash in a production or production-like environment. This happens for many reasons, most often because a dependency, environmental configuration, or library exists on the developer's local machine that did not make its way to the production environment. These problems can be time-consuming to track down in production after months of development, but are much more manageable if discovered within a couple hours.

Because we are requiring builds to run several times a day, build speed is a concern. For example, if a build takes three hours to run and developers are doing it 10 times a day, then there are literally not enough hours in the day to test each change. For extreme programming, a good rule of thumb is to complete builds within 10 minutes. Even in sophisticated projects, testing time (not compilation time) is often the problem. The more robust our test harness is, the longer it will take to run a build. The solution to reducing build time is to set up a *staged build*. Also known as a build pipeline, this calls for performing multiple builds in a specific sequence:

1. The first build is a *commit build*; this is a fast build that may allow for a number of shortcuts, including stubbing out calls to external dependencies, such as services and databases, or using *Mock Objects*
2. The *secondary build* performs a more robust end-to-end test suite that includes interactions with external dependencies

The commit build should be able to run within 10 minutes, enabling developers to continue their work against an active commit build. The secondary build or builds may take substantially longer. A failed secondary build does not stop the development team, but the problem should be diagnosed and resolved. Then an additional test case should be added to the commit build to identify this problem, and ensure quicker discovery, in the future.

The test environment in which a secondary test runs should duplicate the production environment – if financially feasible. This increases confidence in the tests. There are various environmental factors that can affect the behavior of your application and it is best to flush those out earlier in the development cycle. For example, if your application will be deployed to WebLogic on Solaris, then do not test it using JBoss on Linux. Try to mimic

your production hardware as closely as possible, but if that's impossible, then first scale down the number of machines and secondly the class of machines.

After the builds are successfully completed, make them readily available to anyone involved in the project for demonstrations or testing. Be sure there is a moving collection of milestone builds (such as those at the completion of an iteration) posted in a common repository.

Another benefit in adopting continuous integration is that it promotes communication among all team members. When performing multiple daily builds, your developers will probably interact with other team members frequently. It's important for everyone to know the system's current state and most importantly the state of the mainline build. In my company, we handle this in two ways: automated emails are sent at the completion of each build, and a web page is maintained that anyone can visit. In Martin Fowler's article, he describes fanciful approaches, including lava lamps to represent status and a rubber chicken to identify the individual who broke the build.

Finally, because of the different test suites that you will be executing, you need an automated process and/or tool to deploy the application to the production-like environments. This process may need to create and populate databases, deploy enterprise archives, and so forth. Being able to do this automatically not only reduces time, but also allows you to reproduce deployments, which is essential in troubleshooting problems.

In short, continuous integration minimizes the integration phase of your development lifecycle and increases your confidence in your applications. It has greatly helped the companies I've assisted.

Continuous Performance Management

Performance management has become a popular and possibly overused term. Some vendors in this space use it to describe a variety of product functions, but they miss key principles essential to the integrity of performance in software engineering. As a practice, performance management involves planning and accounting for the performance of your applications throughout the development, deployment, and maintenance lifecycles.

Performance management includes the following activities:

- Integrating performance criteria into use cases
- Integrating performance tests into unit tests, including profiling of code, memory and coverage
- Running performance tests as part of tests for integration and production staging
- Performing capacity assessments

- Post-production analysis of usage patterns and validation of test criteria
- Trending, forecasting, and capacity planning

Note: *A thorough discussion of performance management is beyond the scope of this white paper, but I have published an entire book on the subject, Pro Java EE 5 Performance Management and Optimization. In this white paper, I discuss specifically how performance management practices can be integrated into test-driven development and continuous integration.*

Continuous performance management applies the principles of performance testing to the practice of continuous integration. Specifically, we configure various secondary continuous integration builds to execute performance unit tests, integration tests and load (or stress) tests. Because a secondary build is in place, failed performance will not break the commit build. However, you will quickly learn about performance issues so that they never get out of hand. Additionally these tests also allow you to construct a performance profile of your application and trace the impact of code changes to the behavior of your application.

With respect to performance unit testing, we are interested in profiling our unit tests in the areas of code, memory and coverage.

First, we want to know if there are any egregiously slow algorithms or portions of code that we can improve. We also want to compare the profiles of our unit tests across builds to identify performance issues as they arise.

Next, we want to profile our application's memory behavior. Memory problems can come in two flavors: loitering objects and object cycling. Loitering objects, sometimes referred to as lingering object references, represent Java memory leaks. These objects continue to exist in memory after the code completes its function. We need to examine them and determine whether they are expected to remain in memory or not. Object cycling results from rapidly creating and destroying objects. Because these objects are destroyed, they are not memory leaks, but they do tend to fragment the heap and increase the frequency of garbage collections. Both represent serious performance problems in your applications

Finally, when running unit tests, we need to know how much of the code is being executed. If all of your tests are functionally passing, your code is performing quickly, you are not leaking memory or cycling objects, and you are only testing five percent of your code – then your test suite results do not provide you with an accurate representation of your application's performance. But if you are seeing the same results as you are testing 95 percent of your code, then you can be confident in your results.

After you have a set of solid components tested at the unit level, the next step is to assemble those components into a working solution. This is a core

step in continuous integration and again we will extend it to perform larger and more robust code profiling. A single request may pass between JVMs and touch multiple technologies. So we would like to trace a request across those technologies and construct a performance blueprint of the application. The blueprint enables us to diagnose performance problems that only occur in the context of the integrated application.

Finally, we want to subject the integrated application to a load test. Many application problems only appear when the application is subject to load. For example, a single request may add a seemingly benign object to the heap without effect, but 100 users executing that request 50 times an hour may cause your application to run out of memory. The only way to realistically discover this problem is to execute a load test against your application. And if we automate the load test, capture a performance profile of the application during the load test, and execute it several times a day in continuous integration secondary builds, then problem discovery and diagnosis is simple.

ACHIEVING CONTINUOUS PERFORMANCE MANAGEMENT

Achieving continuous performance management (CPM) is not as arduous as it may initially appear. And while an organization may not be able to fully carry out all facets of test-driven development and continuous integration, continuous performance management is attainable. In this section, I present the prerequisites for embracing CPM as well as the steps to implement it in unit testing and in integration (cross-JVM) and load testing.

Continuous Performance Management Prerequisites

The principles of CPM are based on the implementation of performance tests in a continuous integration environment. Therefore, the prerequisite of a CPM strategy is the development of unit tests. Without unit tests, there is nothing to run performance tests against. If your organization has implemented some level of unit testing in a scriptable framework (such as JUnit), then you are positioned to implement CPM in your environment. If your organization has not adopted the practice of building unit tests, then now is the time to do so!

While unit tests are required for CPM in performance unit test analysis, integration and load testing environments require a different test bed. This test bed is more akin to exercising business test cases than unit test cases. The latter ensures that the code is functionally correct while the former ensures that the code components all work together to satisfy business requirements. Business test cases can usually be derived from architectural use cases, including all use case scenarios.

In a simple functional implementation, business test cases can be exercised as an extension to your existing testing framework, such as the JUnit extension HttpUnit. HttpUnit works with JUnit to execute HTTP requests against a specified server and then test for the presence of various components in the resulting document. For example, HttpUnit may verify the title of the resulting document and the presence of a specific form or table. If the criteria are not satisfied then the functional aspect of the HttpUnit test fails and thus the integration is unsuccessful.

From a performance testing perspective, there are two options for profiling these functional tests:

1. Profile the execution of the HttpUnit tests themselves
2. Skip HttpUnit performance tests altogether and profile these tests at the load test level (initially with a single user)

Unless the HttpUnit tests are already written, I typically opt for the latter. After all, we need to profile the code at the load test level anyway, so why write the tests twice (once for HttpUnit and once for our load generator)?

This brings us to the next major phase in CPM: the load test. While load testing may seem like a simple matter of subjecting your application to an increasing number of virtual users executing your business test cases, it is a little more complicated than that. To effectively load test an application, you first need to understand your existing user behavior (in the case of an existing application) or your projected user behavior (in the case of a new application.)

In other words, load tests must subject your application to balanced and representative service requests. A service request is simply a business interaction with your application, which might be an HTTP request or a message placed in a queue for processing. These service requests must represent how your users will use your application and the appropriate balance with which they will execute each service request. A good place to start is analyzing use case scenarios to determine the business functionality of your application. Then, work in conjunction with the application business owner to determine how frequently each request will be executed by an individual user; it helps to storyboard user-application interactions.

For example, in an internal customer service application, a user arrives at work and logs in at 8 a.m. Throughout the morning, the user generates customer reports and processes incoming customer claims. At lunch, the user's session times out, requiring another log in and then customer claim processing continues. At the end of the day, the user enters summary reports for the cases opened and closed, including root cause analysis, and then logs out before going home. On average, a user is expected to process 50 claims a day, generate 100 reports and enter 10 summary reports. For this scenario, the representative service requests are:

1. Log in
2. Generate report
3. Process customer claim
4. Enter summary report
5. Log out

And the balance of the service requests is:

1. Log in: 2
2. Generate report: 100
3. Process customer claim: 50
4. Enter summary report: 10
5. Log out: 1

If the generated load does not mimic the balance of service requests, then tuning efforts may be misguided. Consider balancing each of the aforementioned requests evenly: log-in and log-out requests would receive the same load as generating a report. In this scenario, you might learn that

such excessive log-in and log-outs were causing problems in the LDAP server that validates user credentials, causing you to spend too much time upgrading and tuning that infrastructure. When your application is pushed into a production environment, your LDAP infrastructure can support a load that it will never receive, but the report generation functionality may fail because it cannot support near the load that it does receive.

Business test cases can be built while developing a CPM strategy, so with unit test cases in hand, you are ready to begin!

Continuous Performance Management in Unit Tests

As previously mentioned, the prerequisite for CPM in development is a set of unit test cases; the more robust your unit testing framework, the better your results will be. In this example, I demonstrate how to integrate unit test performance profiling using Apache Ant as the build tool and Quest Software's JProbe as the performance profiler.

The heart of Apache Ant is the build XML file, typically named `build.xml`. This file contains a set of *targets* that, when invoked by Ant, perform a specified set of actions. Common targets include the following:

- *init*: initializes the build environment by performing activities such as creating build and distribution directories
- *clean*: cleans up all work from a previous build by performing activities such as deleting and removing existing build and distribution directories
- *compile*: compiles source code to a build directory
- *dist*: creates distribution files, such as a JAR containing the compiled source code as well as any WAR or EAR files for an enterprise deployment
- *docs*: build JavaDoc documentation for your code
- *test or tests*: executes unit tests, typically using the `junit` ant task

In the case of CPM, the build script needs to be modified to include a new target that profiles the unit tests by executing them while running the JProbe profilers (code, coverage, and memory). JProbe provides out-of-the-box functionality for Ant (and Maven) integration with individual and batch test execution as well as report-generation from profiles. But for the purposes of CPM, the granularity is too coarse. Therefore, I built a set of enterprise tools that extend the JProbe/Ant integration to dynamically generate and import a profiling build script on-the-fly and create a single report summarizing the results of all test cases.

The workflow for integrating JProbe code, coverage, and memory profiling into is outlined as follows:

1. Install the JProbe enterprise tools

2. Modify the build script to execute the JProbe pre-processor
3. Import the generated script
4. Call the performance profiling Ant target

Assuming that the JProbe enterprise tools are installed in `c:\jprobe-ent`, the following addition is required to be in the `build.xml` file:

```
<!-- Set location of the JProbe Enterprise Tools -->
<property name="jprobe.enterprisetools.home"
          location="c:\jprobe-ent" />

<!-- Define the Pre-processor CLASSPATH -->
<path id="classpath.preprocessor">
   <fileset dir="${jprobe.enterprisetools.home}" includes="lib/*.jar"/>
</path>

<!-- Execute the preprocessor -->
<java classname="com.javasrc.anttools.JProbePreprocessor">
   <sysproperty key="src.dirs" value="${src};${src.test}" />
   <sysproperty key="jprobe.home"
                value="C:\Program Files\JProbe 7.0" />
   <sysproperty key="jprobe.build.dest"
                value="${jprobe.enterprisetools.home}\jprobe.xml" />
   <classpath refid="classpath.preprocessor"/>
</java>

<!-- Import the Generated Script -->
<import file="${jprobe.enterprisetools.home}\jprobe.xml" />
```

Because this is a top-level task, it will be executed before all targets are computed. The resultant `jprobe.xml` file contains the following new tasks:

- *performance-unit-tests.init*: initializes the performance unit testing environment; it relies on the following three tasks that should exist in your build file: *clean*, *init*, and *compile.debug*
- *performance-unit-tests.execute*: executes all performance unit tests by calling the following additionally generated Ant targets: *coverage-unit-tests*, *memory-unit-tests*, and *performance-unit-tests*; additionally it executes the *generate-performance-report* task to generate an XML performance profile and an HTML report
- *generate-performance-report*: called internally by the *performance-unit-tests.execute* target; serves to parse all individual reports that JProbe generates and construct a single aggregate report

In order to obtain line-level profiling information, your code must be compiled in debug mode. The following demonstrates how to modify your Ant compile target to do so, assuming that your compile target reads as follows:

```
<target name="compile" depends="init" description="compile the source">
<!-- Compile the java code from ${src} and ${src.test} into ${build}-->
  <javac destdir="${build}" >
    <src path="${src}" />
    <src path="${src.test}" />
    <classpath refid="classpath"/>
  </javac>
 </target>
```

Create the debug compile target with the following modification (in bold):

```
<target name="compile.debug" depends="init"
        description="compile the source " >
<!-- Compile the java code from ${src} and ${src.test} into ${build}-->
  <javac debug="on" destdir="${build}" >
    <src path="${src}" />
    <src path="${src.test}" />
    <classpath refid="classpath"/>
  </javac>
 </target>
```

The additions made to your `build.xml` file invoke the JProbe preprocessor. The JProbe preprocessor scans through your source code (defined in the semicolon delimited list of source code directories in the `src.dirs` property) looking for classes that follow the JUnit naming convention, namely ending in the word "`Test`". For example: `MetricTest` is a test class name in my project. For each test class, the JProbe preprocessor identifies all test methods, defined by the JUnit naming convention of prefixing all test case methods with the name "`test`" and returning `void`. For example, `testRange()` is one of my test case methods. The JProbe preprocessor adds those test methods to the profiling suite. If you follow the standard naming convention then the JProbe preprocessor generates JProbe performance profiles for all of your test cases.

Figure 1 illustrates the role of the preprocessor in the CPM workflow.

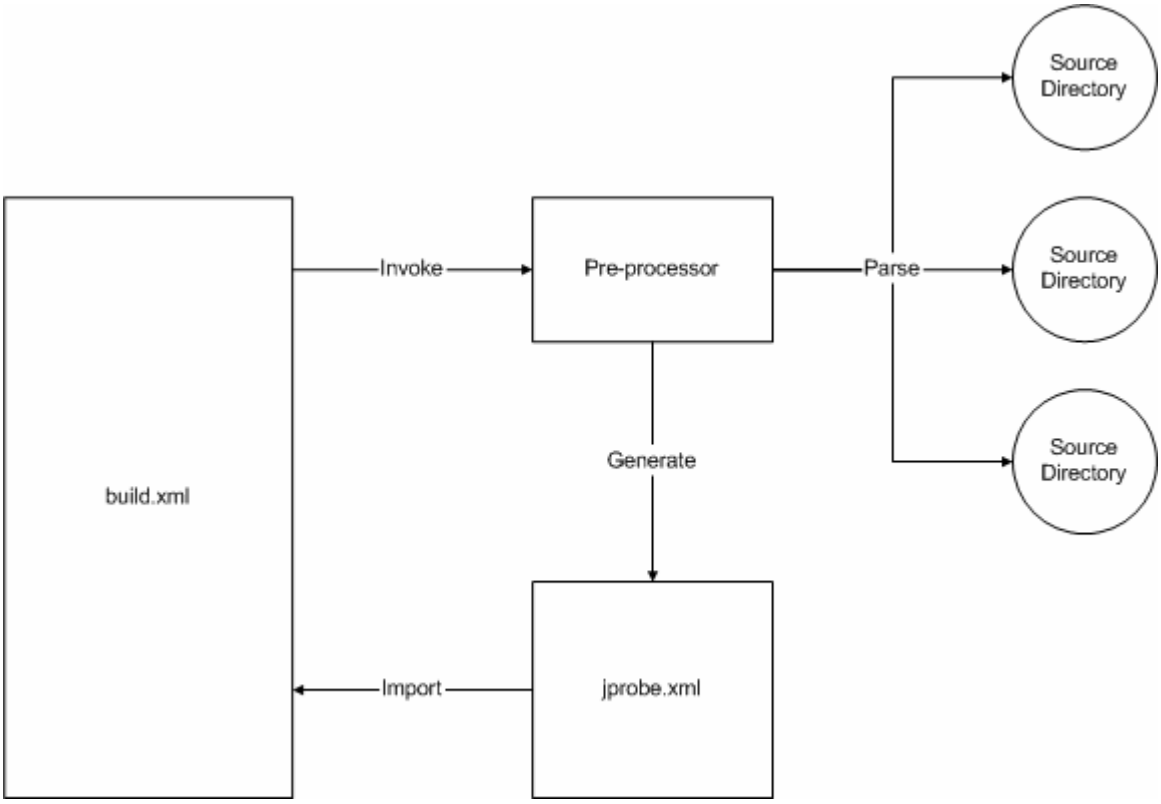


Figure 1. JProbe Preprocessor Workflow

Unit Test Performance Analysis Report

The resultant HTML report provides an executive summary defining the following metrics for the entire test:

- Unit test count
- Total coverage (with a breakdown of classes, methods, lines, and conditions missed)
- Total memory change
- Total object change
- Total test time

The detailed coverage section of the report breaks the coverage down further to individual packages, classes, and methods. If you find a class or a method of interest and you seek additional coverage information, the JProbe interactive console can be invoked against the recorded snapshot and you can browse the source code directly, with coverage indication on each line of source code. There is no need to rerun any tests as all artifacts for interactive browsing are stored and can be easily transported to a developer machine for deeper diagnostics.

The detailed memory and performance sections of the report break down their respective reports by individual test case. The memory report displays test cases (sorted by the number of bytes of memory that each leaves in the heap after completion) and includes the top object types, object counts, and individual object type byte contribution. Similar to the coverage report, all memory artifacts are saved for interactive browsing by the JProbe console. In this case, there is one set of before and after heap snapshots for each test case, which allows you to interactively track down the line of code that created each object left in the heap.

Finally, the detailed performance section of the report displays each test case, sorted by total test case time, and includes each method that measurably contributed to the response time of the test case. It includes individual method time, cumulative method time, and the number of executions that ended in an exception.

Figure 2 shows a sample screenshot of the Executive Summary of the Unit Test Performance Analysis Report.

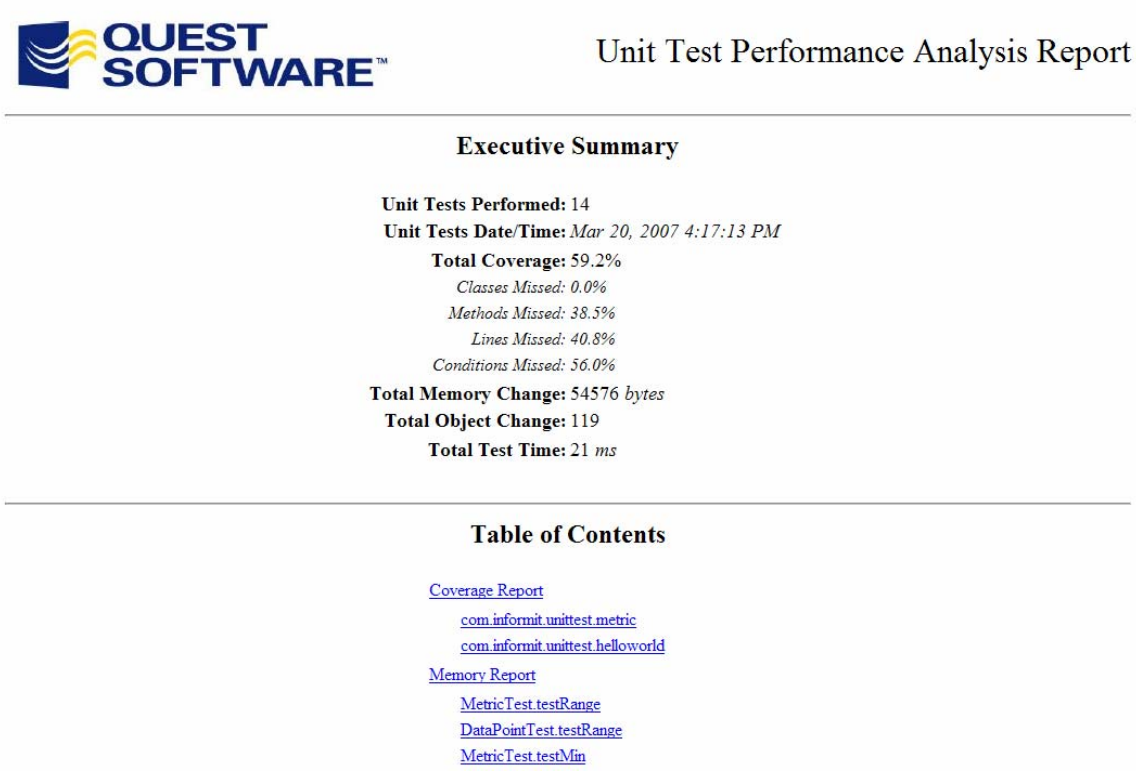


Figure 2. Unit Test Performance Analysis Report Sample

The purpose of this report is to present a quick application performance reference to development managers, team leads, CIOs, or any other interested parties. It provides a business-centric overview of application performance, and allows the user to drill down into individual application

components for deeper diagnosis. Ideally, a development manager or team lead would review this report, looking for performance anomalies, and direct performance problems to the appropriate developer. The developer would receive the details of the offending component(s) and then obtain the appropriate JProbe snapshots to diagnose the problem.

By centralizing and maintaining these reports, development managers and team leads can quickly identify trends in application performance. Then, they can rapidly triage offending components to curb performance problems before they ever appear in the application itself. For the brave, each report is also stored in XML with its HTML equivalent, allowing automatic trending and impact analysis between builds.

Continuous Integration Server

With this build script in hand, the next step is to integrate it into a continuous integration server. This ensures that performance tests are conducted when code is checked in to the source code repository. There are various continuous integration servers on the market, both commercial and open source, and for this exercise I chose CruiseControl. As mentioned earlier, CruiseControl was developed by Martin Fowler and the folks at ThoughtWorks. With Fowler being one of the fathers of continuous integration, I hold CruiseControl in very high regard. You can download it from `http://cruisecontrol.sourceforge.net/`.

CruiseControl is available as a Windows executable or as a compressed file containing source code and/or binary files. Follow the CruiseControl installation instructions or simply download and decompress the binary distribution to a directory on your continuous integration server.

The first step in the integration process is checking out a copy of your source code to CruiseControl's project directory. For example, the following demonstrates how to check out the "trunk" branch (main branch) of the `tddci` repository from Subversion to the CruiseControl projects directory. From your CruiseControl project directory, for example: `c:\lib\cruisecontrol-bin-2.6\projects`

Execute the following command:

```
svn checkout file:///c:/lib/svn-win32-1.4.3/repositories/tddci/trunk
```

The checkout process creates a new subdirectory in the CruiseControl `project` directory that contains a checked out version of the projects stored in Subversion's `tddci` repository. This specific example includes a single project named `ant-junit`.

CruiseControl is driven by a central XML configuration file named `config.xml` that can be found in the root of the installation directory. This configuration file tells CruiseControl about the projects it is responsible for building, where they are located, how to build them, and who to notify of events such as

build successes and failures. Before adding performance testing to CruiseControl, let's first add the project itself for standard builds and functional unit tests. The following project node adds support for building the `ant-junit` project to CruiseControl's `config.xml` file:

```
<project name="ant-junit">
  <listeners>
    <currentbuildstatuslistener
        file="logs/${project.name}/status.txt"/>
  </listeners>

  <bootstrappers>
    <svnbootstrapper localWorkingCopy="projects/${project.name}" />
  </bootstrappers>

  <modificationset quietperiod="30">
    <svn localWorkingCopy="projects/${project.name}"/>
  </modificationset>

  <schedule interval="60">
    <ant anthome="apache-ant-1.6.5"
         buildfile="projects/${project.name}/build.xml"
         target="dist-clean" />
  </schedule>

  <log>
    <merge dir="projects/${project.name}/test-results"/>
  </log>

  <publishers>
    <onsuccess>
      <artifactspublisher dest="artifacts/${project.name}"
         file="projects/${project.name}/dist/lib/AntJUnitExample.jar"/>
    </onsuccess>
    <htmlemail mailhost="mail.mymailserver.com"
               mailport="25"
               username="steve@javasrc.com"
               password="secret"
               returnaddress="steve@mymailserver.com"
               defaultsuffix="@mymailserver.com"
               logdir="logs/${project.name}"
               css="c:\lib\cruisecontrol-bin-2.6\
                         webapps\cruisecontrol\css\cruisecontrol.css"
               xsldir="c:\lib\cruisecontrol-bin-2.6\
                         webapps\cruisecontrol\xsl"
               buildresultsurl="http://localhost:8080/cruisecontrol/
                   buildresults/${project.name}">
      <map alias="steve" address="steve@mymailserver.com" />
      <always address="steve" />
    </htmlemail>
  </publishers>

</project>
```

The important components of this CruiseControl project are outlined as follows:

- *bootstrappers*: these are plug-ins that are executed before a build is performed. In this example, the svnbootstrapper is used to check for new code in the Subversion project repository. If new code has been checked-in then this bootstrapper checks out the modified code and prepares the project to be built. CruiseControl supports a host of bootstrappers, for CVS, ClearCase, Perforce, VSS, and others.
- *modificationset*: this performs the actual comparison of the project and source code repository and tells CruiseControl about the change. This is the mechanism through which the schedule task (see below) determines whether or not to perform a build. In this example I used the `svn` task to check Subversion, but CruiseControl provides support for multiple version control systems modification checks.
- *schedule*: this controls how often the source code repository is checked for modifications. In this example I check the source code repository every 60 seconds, but for a large project it might be more reasonable to check every 30 minutes. If the code has been modified then the schedule's subtask is executed, which in my case is the execution of an Ant build. Notice that the Ant task is provided with the exact build script and target within that build script to execute. And while I chose Ant, CruiseControl supports other build technologies, including Maven, Maven 2, and NAnt.
- *publishers*: these are run after a build has completed its run, regardless of whether the build is successful. There are two specific subtasks of the publishers task that control execution based upon of the success of a build: `onfailure` and `onsuccess`. In this example, I configured an `onsuccess` task to copy the JAR file that my build created to an artifacts directory on success (the purpose of the `artifactspublisher` task is to copy build products to a specified directory.) The other action that the *publishers* task does in this example is send me an email after every build completes with that build's results. Be aware that CruiseControl can even send instant messages or control an X10 device, which could turn on a green light in the office for a success or a red light for failure. If it's the latter, then through X10, CruiseControl can start your coffee maker and lock the doors because it's going to be a long night!
- *log*: this specifies the location of the log file as well as supports the merging of additional log files with the CruiseControl build log. In this example, the JUnit log files, which are contained in my project's `test-results` directory, are merged into the CruiseControl build log. This allows CruiseControl to display unit test results in its web reports as well as in email notifications.

With this project added to the CruiseControl `config.xml` file, you can start CruiseControl by executing its `cruisecontrol.bat` or `cruisecontrol.sh` file,

and watch it run. CruiseControl provides an embedded Jetty Servlet container and web engine that you can access at: `http://servername:8080`

The CruiseControl Web user interface allows you to see the state of the build for all projects as well as the results of unit tests. Figure 3 shows a sample screenshot of CruiseControl's Web interface displaying the unit test results for my project.

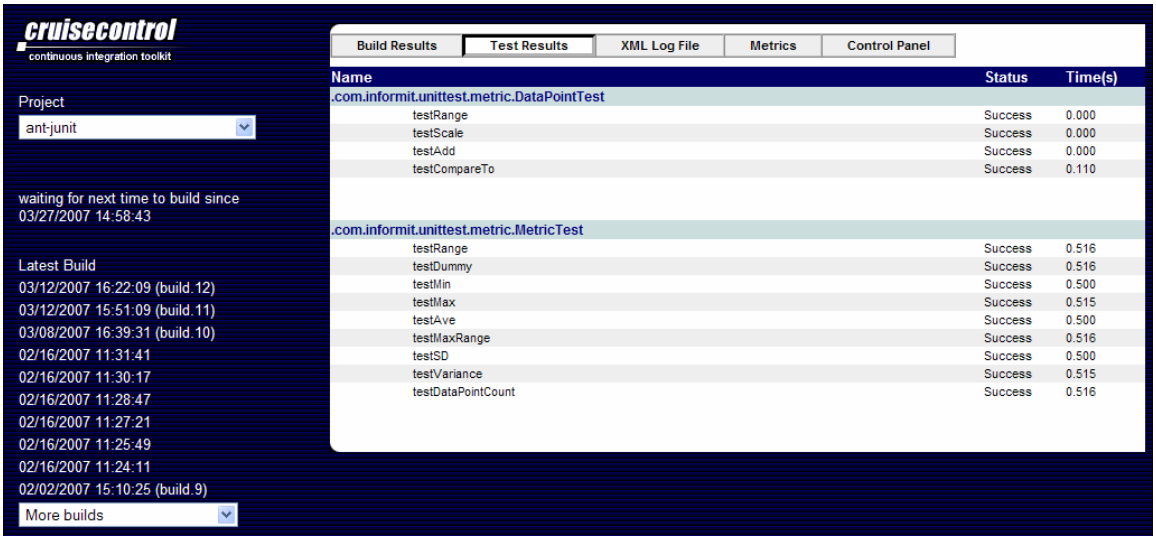


Figure 3. CruiseControl Web console - test results page

Performance Unit Tests in a Continuous Integration Server

The final step in this process is to integrate performance unit test execution into CruiseControl. You can try different strategies, but one that I found very convenient was to create a new virtual project in CruiseControl that points to the source code in my main project. This virtual project uses the same build script, but executes the dynamically generated `performance-unit-tests.execute` target. Further, it executes at a less frequent interval than the commit build does. The `ant-junit-performance-unit-tests` project is shown below.

```
<project name="ant-junit-performance-unit-tests">
  <listeners>
    <currentbuildstatuslistener
       file="logs/${project.name}/status.txt"/>
  </listeners>

  <bootstrappers>
    <svnbootstrapper localWorkingCopy="projects/ant-junit" />
  </bootstrappers>
```

```
  <modificationset quietperiod="30">
    <svn localWorkingCopy="projects/ant-junit"/>
  </modificationset>

  <schedule interval="600">
    <ant anthome="apache-ant-1.6.5"
         buildfile="projects/ant-junit/build.xml"
         target="performance-unit-tests.execute" />
  </schedule>

  <publishers>
    <onsuccess>
      <artifactspublisher dest="artifacts/${project.name}"
                          dir="projects/ant-junit/profiling-reports"/>
    </onsuccess>

    <htmlemail mailhost="mail.mymailserver.com"
               mailport="25"
               username="steve@mymailserver.com"
               password="secret"
               returnaddress="steve@mymailserver.com"
               defaultsuffix="@mymailserver.com"
               logdir="logs/${project.name}"
               css="c:\lib\cruisecontrol-bin-2.6\
                          webapps\cruisecontrol\css\cruisecontrol.css"
               xsldir="c:\lib\cruisecontrol-bin-2.6\
                          webapps\cruisecontrol\xsl"
               buildresultsurl="http://localhost:8080/cruisecontrol/
                   buildresults/${project.name}">
      <map alias="steve" address="steve@javasrc.com" />
      <always address="steve" />
    </htmlemail>
  </publishers>

</project>
```

The main differences between the initial project and the performance testing project are:

1. The Ant task executes the "*performance-unit-tests.execute*" target instead of the "*clean-dist*" target
2. The scheduling interval is configured to check for source code changes every five minutes, as opposed to every minute; in an enterprise application, you might want to configure this check every hour.
3. If the build is successful then the profiling report is the published artifact. The `artifactpublisher` task can publish a directory (instead of just a single file) to the artifact directory. In this case, the CruiseControl's artifact directory will have a subdirectory named "ant-junit-performance-unit-tests" (or whatever you decide to call your project) and that subdirectory will store each performance

report in its own directory, named by the date and time that the build was completed. For example:

```
20070328105328
```

In this case, the performance report was created on March 28, 2007 at 10:53:28 (military time).

It is worth noting that each report directory contains a resultant aggregate HTML report as well as an XML report that can be programmatically data-mined and compared with other reports to identify and display trends.

Let's review some common scenarios to diagnose performance problems that are identified by these automated tests.

Problem Diagnosis Scenario – Slow Running Code

Consider a scenario where a particular test method is running slower than usual. The report displays the response times of each method, but in order to resolve the problem, line-level information is needed. Because the automated CPM process saves all performance snapshots with quickly identifiable names of the form `TestClassName_TestMethodName_....jpp`, you can simply launch the JProbe console and open the snapshot of the offending code.

The first step is to browse through the calling methods and find the slow points. Figure 4 shows a sample screenshot of the JProbe Call Graph view.

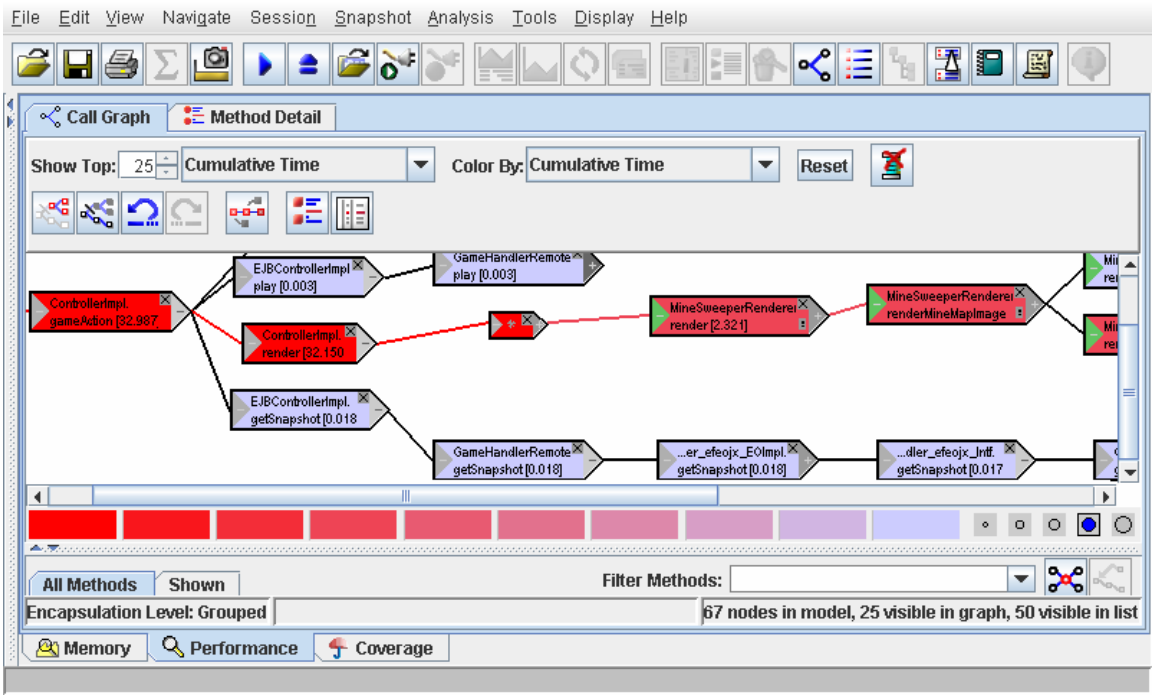


Figure 4. JProbe Call Graph

From this JProbe Call Graph view, you can quickly identify slow methods (they are the brightest red). By default, the methods are color-coded by cumulative time, which is the time spent in a method plus the time spent in all methods that it calls. For quick diagnosis, you can change the color-coding scheme to make the slowest individual method times the brightest red. Once you have identified the offending method, you can then drill down into it and look at the source code, which is accented by a list of call counts and line execution times. Figure 5 shows a sample screenshot of the JProbe Source Code view.

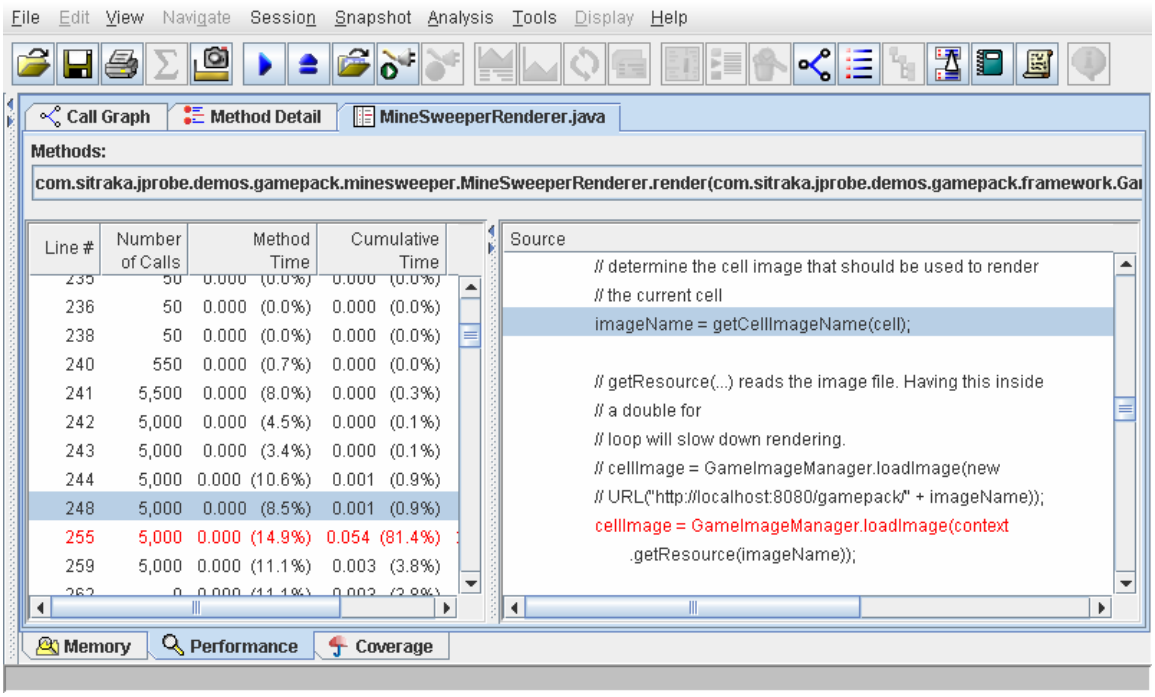


Figure 5. JProbe Source Code View

In this example, line 255 is the slowest line in the method source code, accounting for 81.4 percent of the method response time. From this analysis you might determine that this might be a fast line of code (in this example it was not even measurable at a granularity of 1ms), but because it is called 5,000 times, it is impacting performance.

The automated invocation and profiling of test cases helps identify performance problems and can be used to identify trends in performance over time. Once a problem is identified, the interactive JProbe console makes it easy for a developer to quickly identify the root cause of that performance problem.

Problem Diagnosis Scenario – Memory Leak

Consider a scenario where a memory leak is suspected. The CPM report identified a significant amount of memory left in the heap as the result of a test case. In this scenario we launch the JProbe console and then open two heap snapshots:

- The snapshot captured before the test case was executed
- The snapshot captured after the test case was executed

Then, we compare the two snapshots to see the difference. Figure 6 shows a screenshot of the difference between two heaps, displaying all objects created by classes in our application packages (in this example, our classes are in the package `com.bea.medrec` and its sub-packages.)

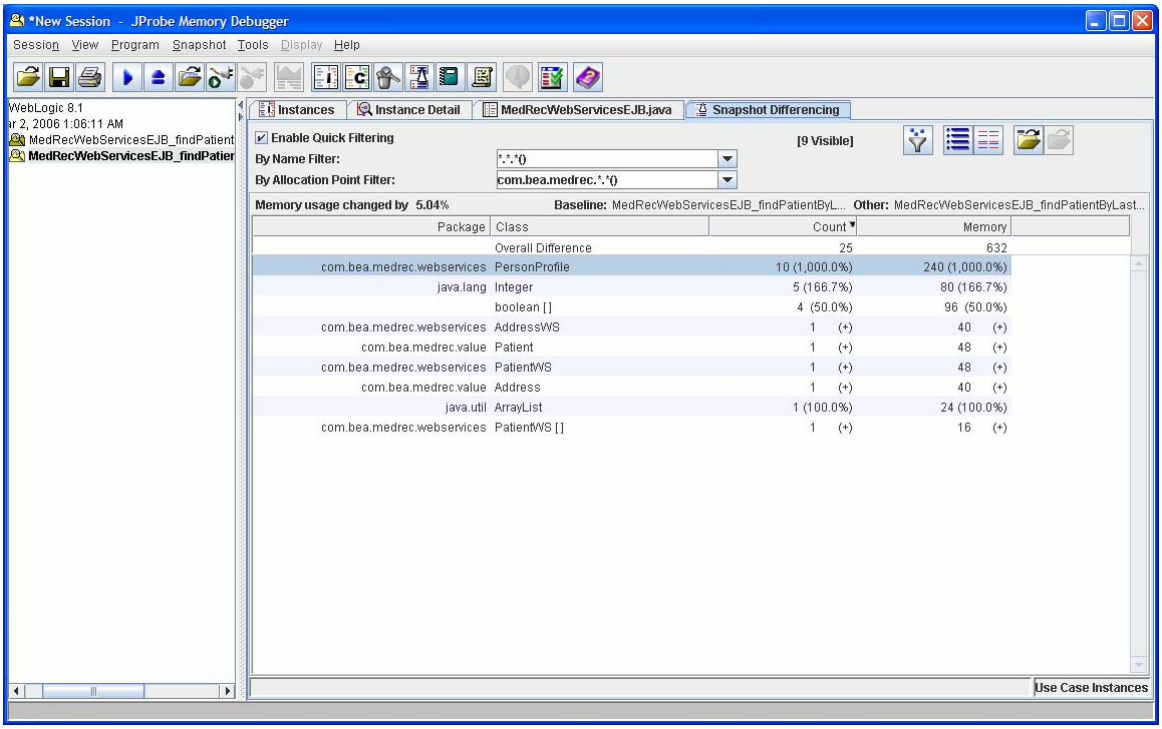


Figure 6. JProbe Heap Differencing

We see a 1000 percent increase in the number of `com.bea.medrec.webservices.PersonProfile` instances (from one instance to 11 instances, which is an increase of 10 instances). There are other differences, but this appears to be the most suspect object (or at least a good place to start.)

The next step is to open the post-test case heap snapshot, find the `PersonProfile` object, and view its instance details. Figure 7 shows a screenshot of the details for one of the 11 instances of the `PersonProfile` in the heap.

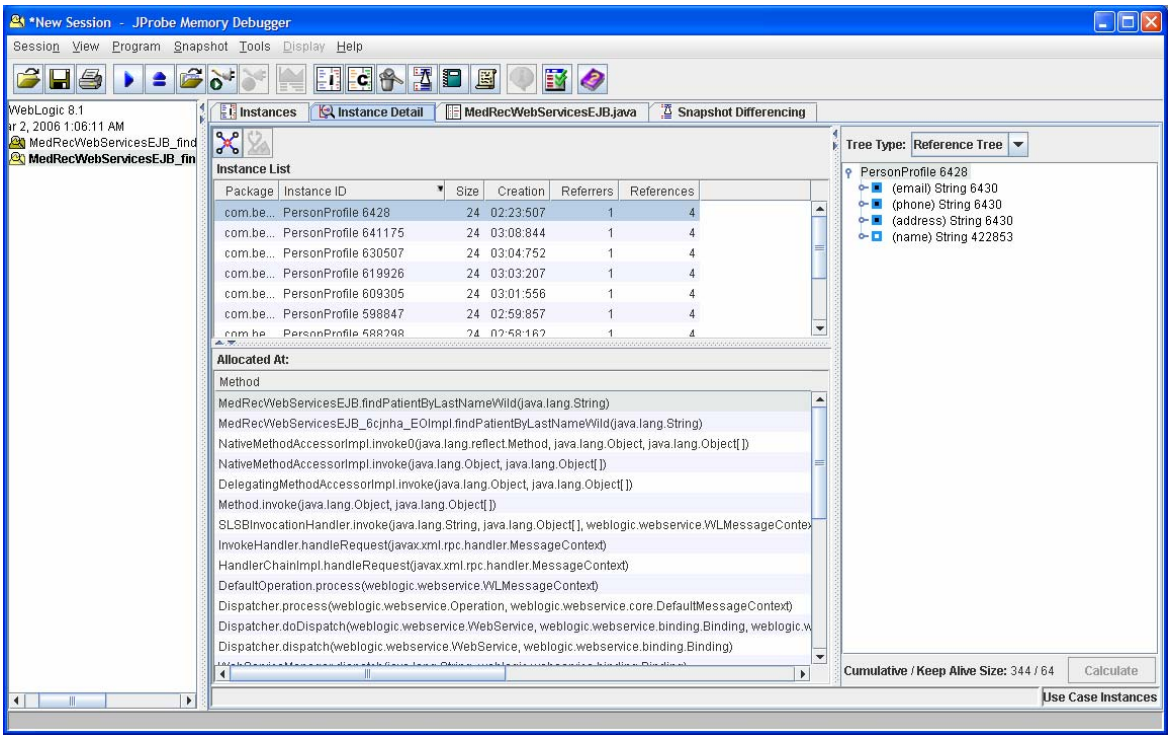


Figure 7. JProbe Object Instance Details

We can see each PersonProfile instance in the instance list. By clicking on one instance, JProbe displays all objects that it references (email, phone, address, and name strings) on the right and the number of total bytes that it and its dependencies are occupying (344 bytes in this example) on the bottom. We can also see the method that allocated it (`findPatientByLastName(java.lang.String)`). To make the final diagnosis easier, we right-click on an instance from the instance list and find the line of code where the instance was allocated. Figure 8 shows a screenshot of this view.

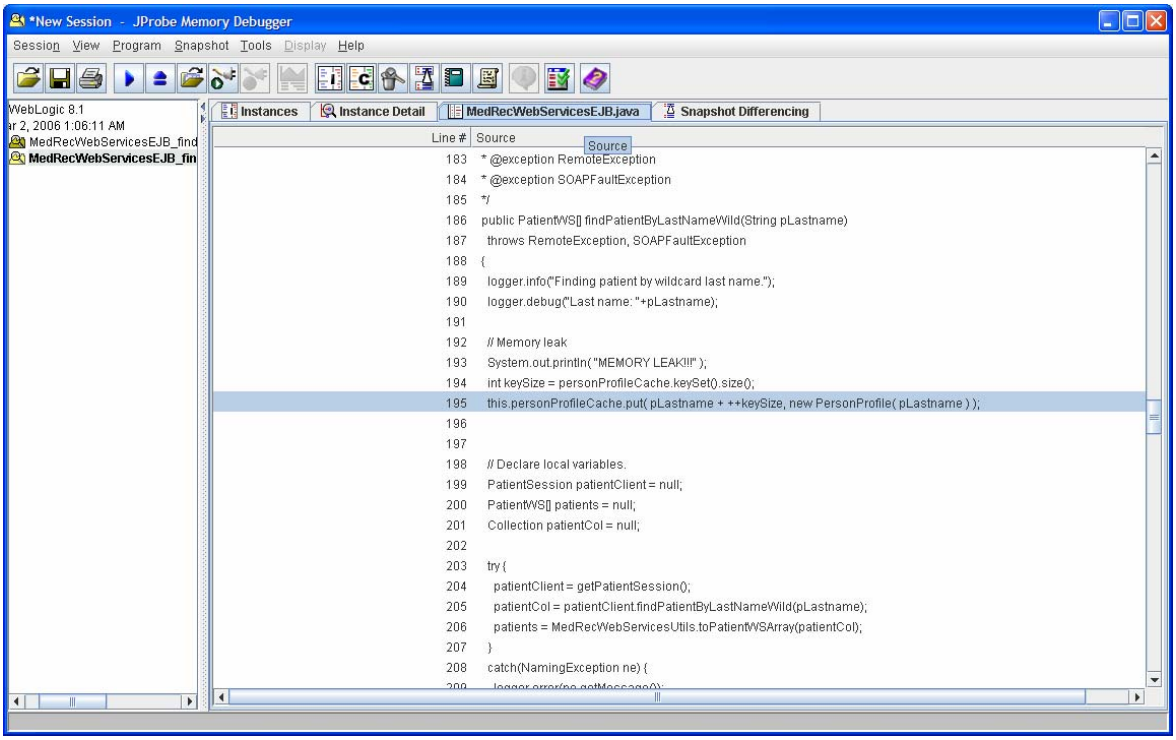


Figure 8. Source code where the object was allocated

JProbe opened the source code viewer and highlighted the line of code that allocated this object. In this example, the `PersonProfile` was created and stored in a cache, but the key to the cache was always unique, so the cache would grow indefinitely.

Tip: *If you carefully review Figure 8, you'll notice that I comment on my memory leak. It is best to identify memory leaks as you make them because diagnosis will be much easier with explicit comments. Or consider not making them at all.*

The automated invocation and memory profiling of test cases helps identify memory leaks and can be used to identify trends in memory usage over time. Once a problem is identified, the interactive JProbe console makes it easy for a developer to quickly identify the root cause of that memory leak, all the way down to the actual line of code that leaked the memory.

Continuous Performance Management in Integration and Load Tests

Much of your integration and load-testing effort is dependent upon your load generator. In this example, I demonstrate how to configure Apache JMeter to generate load on Web-based applications and then integrate JMeter into an Ant build script. Regardless of your choice of load generator, the CPM

requirement is that your load generator can be controlled by an automated build process and can generate a report that can be programmatically analyzed.

Note: *The complete configuration and usage discussion of JMeter is beyond the scope of this white paper, but I have published a series of articles on using JMeter on Informit.com. For this discussion, I summarize the features of JMeter and demonstrate how to integrate it into your build process.*

JMeter is a fully featured open source load generator. It may not be as sophisticated as some of its commercial rivals, but with its feature set and price, it is hard to beat. To its credit, it provides a local proxy server that can be used to record interactions with a Web application. This comes with filters to exclude unwanted artifacts, such as JavaScript files and images, and allows the rapid generation of load scripts. In the end, it provides you with ultimate control over the services requested as well as the think times between requests. I have experienced great success in building Web-based load tests using JMeter.

With respect to CPM, the important aspect of JMeter is that it can be easily integrated with Ant. Programmer Planet has created a JMeter Ant Task that is available for free and easy to use. The JMeter Ant task can be downloaded at:

```
http://www.programmerplanet.org/media/ant-jmeter/ant-jmeter.jar
```

After downloading the `ant-jmeter.jar` file, add it to your Ant's CLASSPATH, either explicitly or by copying it to your Ant's `lib` directory. Then, define the `jmeter` Ant task in your Ant script by adding the following:

```
<taskdef name="jmeter"
classname="org.programmerplanet.ant.taskdefs.jmeter.JMeterTask"/>
```

JMeter is driven by one or more test plans, which are XML files (with a .jmx extension) that specify the details of the load test. The `jmeter` Ant task can execute a single or multiple JMeter test plans and then summarize load test results in a log file (with a .jtl extension.) The JMeter test plan defines such metrics as the number of virtual users (threads) and test iterations (loops), but the `jmeter` task allows these values to be overloaded in the context of your Ant script. Thus, we can differentiate between integration performance testing and load performance testing. In the former we can execute each request type once, and in the latter we can perform a full load test.

The JMeter generated log file is a comma-separated value (CSV) file by default, but if we overload that through the `jmeter.save.saveservice.output_format=xml` argument, then it generates an XML report. An XML report provides the following two benefits: (1) we can programmatically parse this XML file to validate successes and failures and check response times against our service-level agreements; and (2) we can

transform the XML file through one of the available XML Style sheets (XSL) using the Ant `xslt` task to generate a human readable HTML report. Programmer Planet provides two XSL style sheets, depending on how much information you would like to review:

- JMeter Results Report
- JMeter Results Detailed Report

While JMeter XML reports can be parsed and evaluated against service level agreements, diagnosing why a service request is failing to meet its SLA is the job of another profiler. In this example, I describe how to integrate Quest Software's PerformaSure to capture a performance session using Ant.

PerformaSure generates a performance profile, broken down by individual service requests. It traces method calls from their inception (such as an HTTP request) through all application layers (such as your Web tier and business tier) and back into a database. PerformaSure provides an intuitive interface that allows rapid performance diagnosis at the application and application server levels. As such, it is a great tool for diagnosing performance problems that the JMeter tests identify.

PerformaSure provides a command line script that allows detailed control over recording performance sessions invoked by Ant. The command line script is *nexusctl.cmd* and is located in PerformaSure's scripts directory. Ant can run scripts by using the `exec` task to execute a command shell. The following demonstrates how to execute the PerformaSure *nexusctl.cmd* script:

```
<property name="pas.home" location="j:\PerformaSure5.0" />
<target name="pas" description="Execute PerformaSure to start a
recording" >
   <!-- Execute nexusctl -->
   <exec executable="cmd">
     <arg value="/c" />
     <arg value="${pas.home}/scripts/nexusctl.cmd" />
     <arg value="start-recording" />
     <arg value="-user" />
     <arg value="user" />
     <arg value="-l" />
     <arg value="30m" />
     <arg value="-t" />
     <arg value="10s" />
     <arg value="-f" />
   </exec>
 </target>
```

Because the `exec` task cannot directly execute command scripts, we must invoke the command shell (`cmd`) and pass it the "`/C`" option to tell it to

execute the arguments passed to it. In this case, we specify the *nexusctl.cmd* script and the following parameters:

- Command to execute – such as `start|stop` to start or stop the nexus server, `start-recording|stop-recording` to start or stop recordings, `show-connections|show-connections-xml` to display active connections, `reset-agent-list` to reestablish the list of connected agents, and `version` to display the PerformaSure Nexus version information
- Arguments to pass to the specified command – in this example, the command is `start-recording`, so the arguments are:
    - `–user`: username that starts the recording
    - `–pwd`: password for this user, in this case there is none
    - `–l`: the length of the session to record, in this case 30 minutes
    - `–t`: the time slice in which to aggregate request and application server metrics, in this case 10 seconds
    - `-f|-c`: the detail level to record the session at: full detail (method level) or component detail recording, in this case we chose full detail

You have two options when integrating PeformaSure with JMeter in Ant:

- Start a PerformaSure recording for a specified time period that will cover the entire JMeter session and then start the JMeter load test
- Start a PerformaSure recording without specifying a time period, launch JMeter, and after JMeter completes, then manually stop the PerformaSure recording (through another automated execution of *nexusctl.cmd* passing it the `stop-recording` command)

Regardless of your preferred approach, in the end you will have JMeter HTML and XML reports that describe the behavior of your application as well as a PerformaSure session file for interactively diagnosing the root cause of performance issues. All of this can be controlled by an Ant script and executed on a regular basis by CruiseControl.

Problem Diagnosis Scenario – Slow Running Code

Consider a scenario where a particular service request is running so slowly under load that it violates the service level agreement. Because the automated CPM process captures a PerformaSure session while the JMeter load test is being performed, we can log into the PerformaSure Nexus (server) using the PerformaSure Workstation (client) and open that session. The first view displays all service requests, which can be sorted by average execution time (for a single request), maximum execution time (how bad the response time was at its worst), and the total cumulative time for the entire

session. Figure 9 shows a screenshot of the PerformaSure Service Requests view.

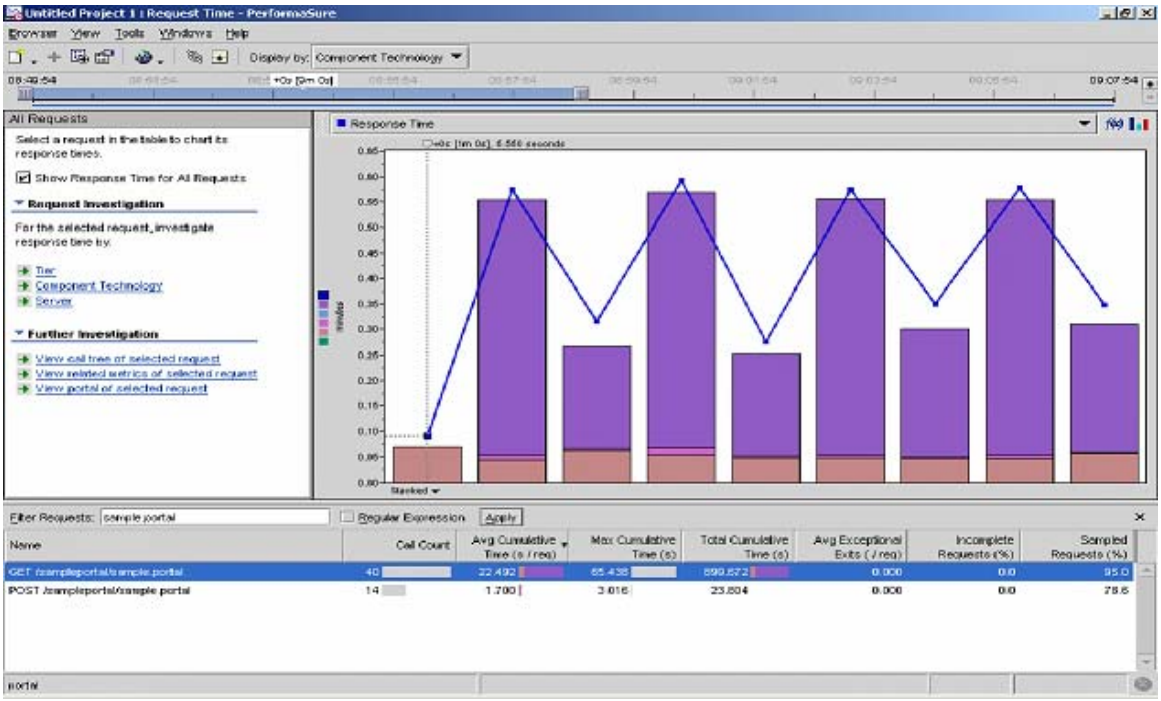


Figure 9. PerformaSure service requests

In this example, we see that a GET against a portal application was executed 40 times and, on average, took over 22 seconds to run, with a maximum time of over 65 seconds. The histogram displays a breakdown of the response time for the selected service request for each sampling interval. It allows you to drill down into a single sampling interval if there's an apparent response time anomaly (it might have performed well on average, but had a single request that was extremely slow.) Further, the individual histogram bars can be color-coded by application tier, including web, business, and database, as well as by technology, including Servlets, JDBC calls, EJBs, JNDI, and so forth.

Next we can pick a service request and view the call tree for that request, shown in Figure 10.

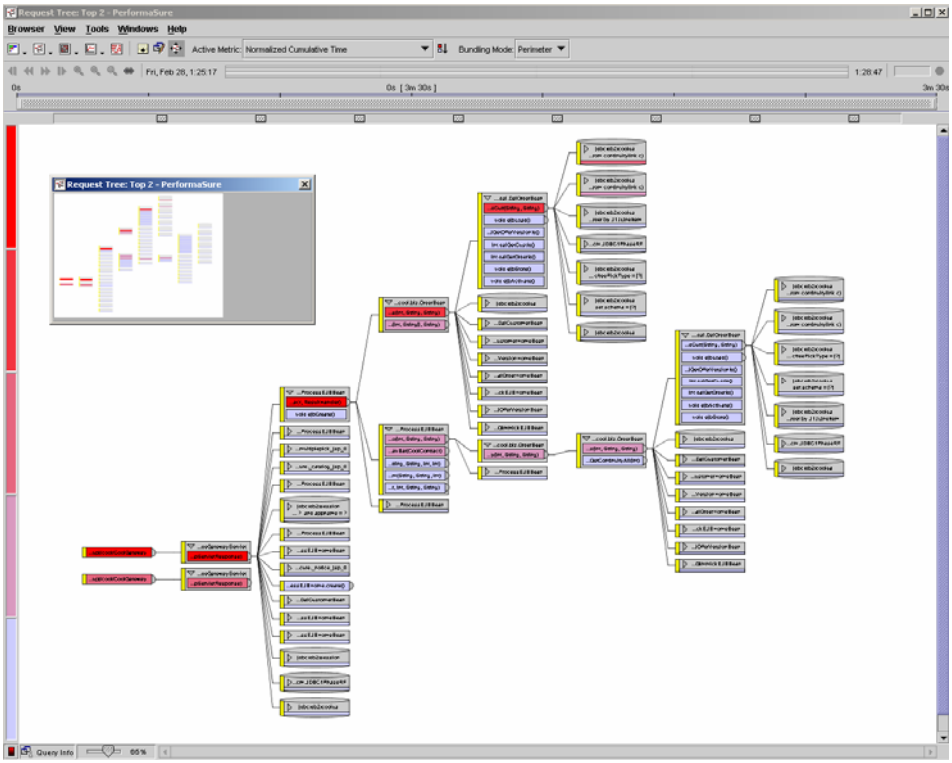


Figure 10. PerformaSure call tree

The PerformaSure call tree shows all methods that were invoked to satisfy the service request and color codes the response time of each method. By default, the color coding displays the brightest red for the slowest cumulative time of each method (the time spent in a method and all methods that it invokes.) This allows you to easily see the “hot path” through your request. By changing the color coding to display the brightest red for the slowest exclusive method time, you can quickly identify the “hot point”, or slowest method, for that request.

The automated invocation of JMeter against your fully integrated and deployed application helps identify service requests that violate their service level agreements. Once you identify a problem, you can use the PerformaSure Workstation to identify the root causes of that performance problem from the session that was captured during the JMeter load test. The PerformaSure Workstation helps you find the method, or methods, that caused your service request to violate its service level agreement. It also allows you to correlate slow application performance with a robust set of application server metrics (such as thread pool utilization and JDBC connection pool utilization). This correlation helps you to determine if the performance issue is an application problem or a configuration problem.

Summary

The majority of enterprise applications fail to meet their production performance requirements. A change is needed across the development, testing, and deployment operations within organizations to resolve these failures. The most constructive and effective change that an organization can make is to address performance at the application level as early in the development lifecycle as possible. This includes defining service level agreements in architecture artifacts such as use cases, as well as a proactively monitoring performance during the development of the application itself.

This paper proposes a revolutionary change in performance management that, if properly implemented, may eliminate the challenges of performance testing. The RIM BlackBerry eliminated hours of nightly email communication for the business person by facilitating this communication throughout the day. For large application development, continuous integration can eliminate integration issues (that can persist longer than the development effort itself) by integrating an application several times a day.

Likewise, continuous performance management aims to eliminate application performance issues by automating application performance tests several times a day in a continuous integration environment. By ensuring application performance at every step of the development process, countless days, weeks, and months can be saved in diagnosing production performance issues. The process has been defined and the tools seamlessly enable the process. The choice is yours. For more information about the Quest Software's Application Assurance product suite, please visit *www.quest.com*.

About the Author

Steven Haines is currently the Java EE Domain Expert at Quest Software, defining the expert rules and views used to monitor the performance of enterprise Java applications and application servers. Previously at Quest Software he created a Java EE performance tuning professional services organization where he improved the performance of and resolved critical production issues for many Fortune 500 companies. He is the author of *Pro Java EE 5 Performance Management and Optimization* (Apress), *Java 2 Primer Plus* (Sams), and *Java 2 From Scratch* (Que), and shares author credits on *Java Web Services Unleashed*. Steven is the Java host on Pearson Education's online IT resource, Informit.com, where he posts weekly articles and blogs. Steven has taught all facets of Java programming at the University of California, Irvine and Learning Tree University. Look for his latest digital shortcut book, *Agile Java Development with Test-Driven Development and Continuous Integration* this summer on Informit.com.