Beyond Performance Testing

by:

R. Scott Barber

Part 8: Modifying Tests to Focus on Failure or Bottleneck Resolution

Now that we can conclusively reproduce the bottleneck, slow spot, or failure and the stakeholders agree that it's an issue worth addressing, what next? In this article we're going to explore how to design and build or modify our tests to focus more explicitly on the item of interest. Since we've identified only the functional symptoms of the performance issue so far, this focusing exercise will be critical to the process of determining what's causing those symptoms, thus starting us down the path toward resolving them.

This is the second of four articles on the theme I call "finding bottlenecks to tune," where we're taking a step beyond just performance testing and beginning to explore how to add real value to the development team.

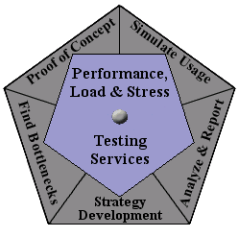
So far, this is what we've covered in this series:

This article is intended for mid- to senior-level performance testers and members of the development team who work closely with performance testers. If you haven't yet read Parts 5, 6, and 7 of this series, I suggest you do so before reading this article.

What the Development Team Needs to Know

"The ability to focus attention on important things is a defining characteristic of intelligence." — *Robert J. Shiller,* Irrational Exuberance

After you report the symptoms of suspected performance issues you've identified, your developers may recognize the symptoms and be able to resolve them in short order. But if not, they're going to need more information, some of it in the form of metrics. If you're a longtime reader of mine, you know that I generally try to avoid talking exclusively about metrics and like to pay at least as much attention to user experience, since metrics aren't the whole story. In the case of chasing performance issues, though, you eventually get to the point where metrics are needed for



evaluation. With that said, I'll list some questions that point to the kinds of information and/or metrics that can help developers identify and/or isolate a performance issue.

Which related activities produce the same symptoms?

The very first thing developers ask once they acknowledge that the symptoms you're reporting are real is "How did you get that to happen?" This is followed closely by "Is there any other way to make that happen?" Sometimes these questions are easy to answer. To repeat an example from Part 7, if you find out that searching for a book and searching for a store near you on a retail site are both slow, this allows the developers to narrow the scope of things they'll need to evaluate.

Sometimes these questions aren't so easy to answer, and you'll need to ask the developers to help you determine which activities are related from their perspective so you can evaluate them. For example, you may learn that the only way those two searches are related is that they both have tables in the database, in which case testing other searches is less likely to produce the same symptoms. You may also learn that they share all the same code, and parameters are just passed into the `generateSQL` class, in which case you'll want to know *all* of the activities that pass parameters into the `generateSQL` class so you can see if they cause the same symptoms.

The point is, the developers know which custom functions, classes, servers, tables, and so forth an activity touches. You generally won't. You'll often need the developers' assistance to determine which activities are related before you start modifying your tests. Working with them, you should be able to identify which related activities demonstrate the same symptoms.

Which other activities are affected by the bottleneck?

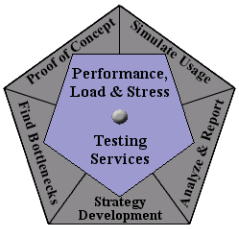
The developers will also want to know which other activities display any symptoms during the test that created the critical symptom. For example, they'll want to know it if the search page is very slow when searching by *t* and other users who are trying to search at the same time receive an internal server error. This is critical as it helps them identify potential causes of the symptoms. It's also not always easy to detect, so ask the developers which related activities they suggest you explore for more information.

What were the load characteristics of the test yielding the symptoms?

The next thing the developers want to know is what the load characteristics of the tests yielding the symptoms were. This isn't just information like "100 users were accessing the system during the test that yielded the symptoms." The developers need to know things like these:

- How many users were performing the activity before and during the appearance of the symptoms?
- What was their distribution in time (arrival rate)?
- What were other users doing before and during the appearance of the symptoms?
- With how few users can you observe the same symptoms?

In case you may have missed it, the answers to those questions are metrics. These metrics are specifically useful in this context. Still, they aren't the whole story you're telling, just some quantifying factors in the story.



What data did you use to create the symptoms?

We know that both the volume and the complexity of data accessed can have a huge effect on performance. For example, searching for all books whose titles contain the letter *t* will certainly put more stress on the system than searching for the title *Lessons Learned in Software Testing*. The developers will want to know what data you used as input values to create the symptoms, and they'll likely have other data they'd like you to try. We'll discuss test variances more below.

What's the configuration of the environment you're testing?

When I start testing, I'm almost always told, "The environment is ready and it mirrors production." As soon as I find a bottleneck suspect, though, someone almost always says, "What environment were you testing against? That doesn't *really* match production. We need to change the settings to . . . " While you may never know the exact configuration of the environment you're testing, your developers will certainly need to know. The best you can do is help them by sharing the information you have.

What other metrics do the developers want you to collect?

On top of all that, the development team will normally ask you for a laundry list of metrics that you may not even understand, let alone know how to collect. It seems that on every project I get asked for at least one metric that I've never heard of. Don't be daunted by this. Simply ask the developers to help you identify and capture the metric they think will aid them in understanding or diagnosing the performance issue. I've never had a developer react poorly to "I'm sorry, I really don't know how to collect that data; could you help me?" If the developers don't give you a list of other things they'd like to know, ask them. I've found that if they're not asking for more information, it's often a sign that they don't have much faith in the results you're presenting.

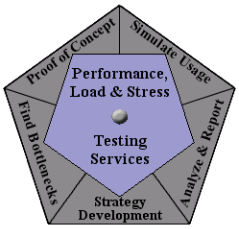
How to Design Tests to Get That Information

Our next step is to design the tests that will get us the information the developers are looking for. This is usually not the difficult part; the difficult part is often the following step, which is creating the tests. When you're designing the tests, don't worry about how you'll develop them with the tools you have available. Thinking about the capabilities of the tools at your disposal while designing will almost always lead you down the road of designing tests that are easy to implement, instead of tests that will provide immediate value. You don't want to go down that road.

Ask Yourself "What If . . . ?" Questions

The first thing to do when trying to come up with tests that focus on a particular set of symptoms is to ask yourself, What would happen to these symptoms if . . .

- I eliminated all other user activity?
- I added more user activity?
- I used different data?



- I changed the load characteristics?
- I changed the delay times?
- I tested from multiple IP addresses?
- I used a different navigation path to get to this activity?

There are probably hundreds more questions you could ask, but this is a good start. Based on your answers, you can decide which tests to design first. Maybe you're more interested in trying out different data than multiple IP addresses based on your symptoms. Some of the "What if . . . ?" questions you can answer right away with a quick manual test, while others will require specific tests built to confirm or deny your suspicions.

Ask Developers to Speculate

I'm always surprised by the number of folks who argue that the tester should know best and thus the developers shouldn't be asked to speculate about which tests will help them identify and diagnose performance issues. I believe that the test engineer should know best how to detect potential issues, but when it comes to exploiting an issue so that it can be diagnosed, experts on that particular system are needed. Ask your developers to speculate or guess what other tests will provide helpful information, and then do everything you can to provide those tests. They're often exactly the right ones. As mentioned earlier, you can also ask the developers what metrics would help them diagnose the suspects, and then ask yourself and/or them what test will provide that metric. These are often the most difficult tests to design and develop.

Evaluate Commands with Slow Responses

Another good source of information about how to design appropriate tests comes right from TestManager after a test execution. We looked at TestManager when we were looking for bottleneck suspects. Now that we have suspects, we should return to TestManager and look at the individual commands that had slow responses. Each one of those commands is related to a specific requested item. Once you identify that item, you can search your log file for other instances of that item and add those to the list of things to test. We've talked about the different components of that process before, but I'll summarize the process here.

Say you look at the performance report output in TestManager and notice a command that had a particularly slow response. For instance, in Figure 1, you see that the response to command `GL_Jour~2.017` took 25.29 seconds, much longer than the responses to other commands.

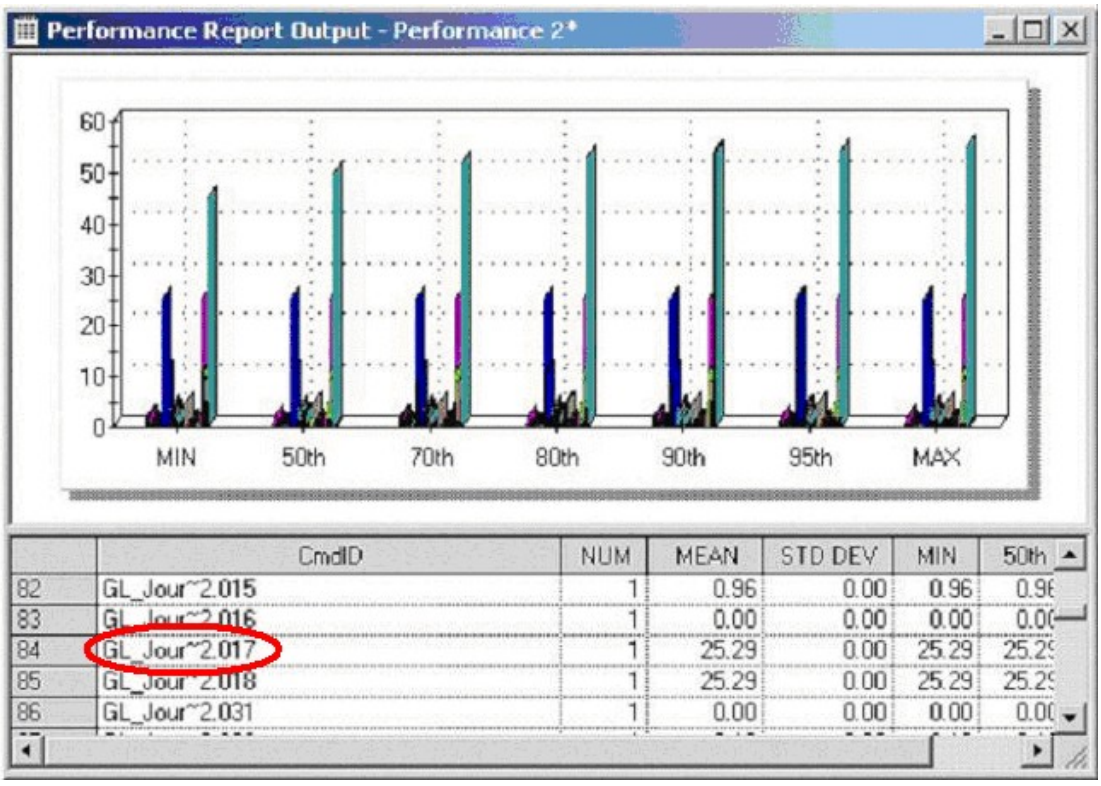


Figure 1: Identifying a slow command response in the report output

You can then find that command ID in the test log and look at the General tab of the Log Event window for more information on that command ID. See Figure 2.

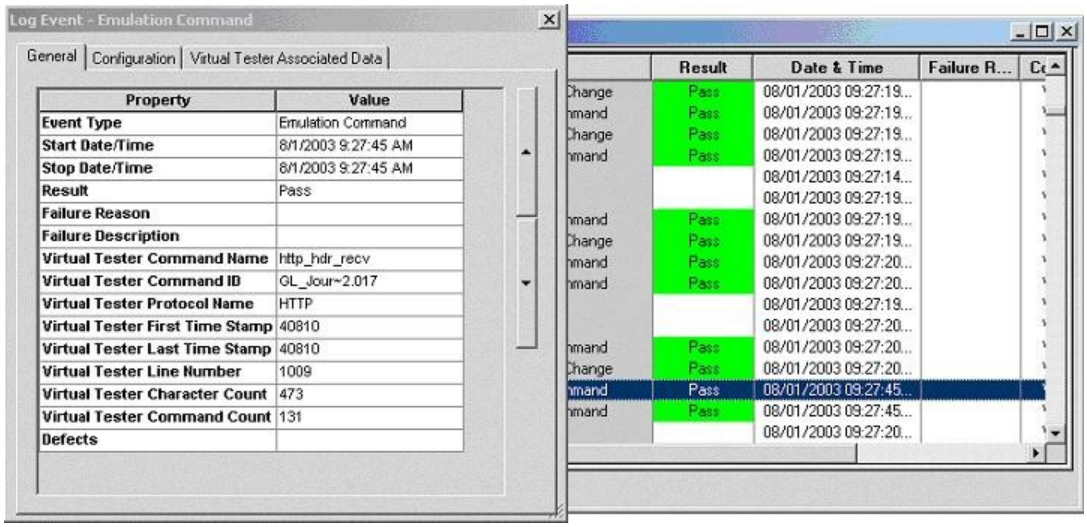


Figure 2: Finding the command ID in the test log

Clicking the Virtual Tester Associated Data tab shows that the item being received was
`POST /psc/GDV01/EMPLOYEE/ERP/c/PROCESS_JOURNALS.JOURNAL_ENTRY_IE.GBL`

See Figure 3.

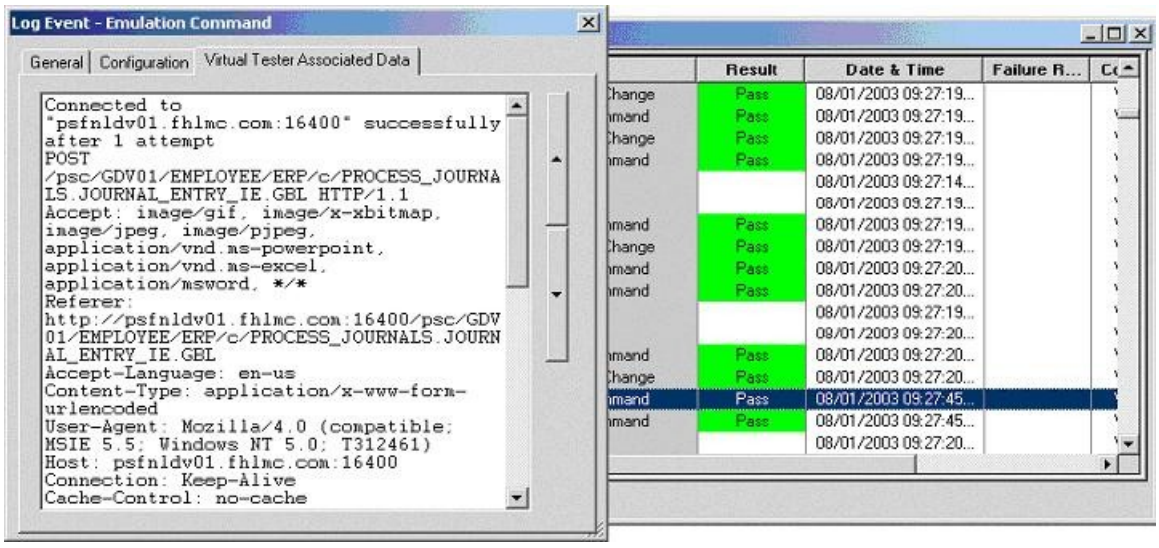


Figure 3: Finding the object of the command ID in the test log

Now that you've identified the object related to the slow response time, you can search the entire log file to see what other activities call that object. You'll remember from Part 5 that the log file is the `d00` file located at

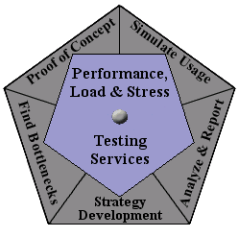
```
[Drive]:\[RepositoryName]\TestDatastore\TMS_Builds\[BuildName]\[SubBuild]\
      [TestRun]\perfdata\
```

and that we open the log file using a text editor such as Notepad. Finding other instances of objects related to the symptoms will certainly provide some insight into the cause of the symptoms, or at least point to some other things to test.

There's a caveat to that, though. Often the problem turns out to be with a previous request/receive pair. If the previous receive returns unexpected data or an unrecognized failure, it may cause subsequent request/receive pairs to fail. You would evaluate this in the same way that you evaluate script failures, as discussed in Part 5. This doesn't mean that the problem is necessarily a script failure — only that the process of finding the offending command is the same. Typically, if it's a previous command that's causing a symptom later on, that turns out to be a failure rather than a slow spot or a bottleneck, but not always.

Think in Terms of Distinguishing Failures, Slow Spots, and Bottlenecks

Another thing to think about when designing tests to focus on suspects and symptoms is how you can design tests to distinguish whether the observed performance issue is actually a failure, a slow spot, or a bottleneck. Many of the considerations for test designs that we've already discussed in this article will help make the distinction, but it's always a good idea when designing a test to ask yourself, Will this test help me determine if this issue is a failure, a slow spot, or a bottleneck? When the answer to that question is no, you should follow it up with, Will another test I've designed help me make this determination, or should I design a new test to do this?



Visualize and Prioritize

Finally, once you've asked yourself and the developers all those questions and done some research on your own, you'll have a whole list in your head of potential tests to create. The thing is, you'll probably be given only a matter of days to track down information about these issues, not weeks. You simply won't have time to develop and execute all of those tests. To pick the right tests to develop, you may want to do what I do, which is to visualize and prioritize. This is actually just a quick-and-dirty way to organize your thoughts about this list of tests you've just come up with.

All I do is fill out a grid like the one in Figure 4 to keep my thoughts straight and help me decide which tests to develop first.

Visualize and Prioritize							
Test Name	Symptom(s) Focused on	Parameters Varied	Metric(s) Collected	Importance of Resulting Data	Other Methods?	Difficulty of Development	Overall Priority
search book title	Slow search	book titles	response vs. title	High	No	Low	High
generateSQL function	Slow search	search types and values	response vs. type vs. value	High	No	Med	High
bigfile.gif	Slow page with graphics	file size	response vs. file size	Med	Yes	Low	Low
report timeout	Report timing out with error	timeout values	does page ever paint?	Med	Yes	Med	Med
25+	Errors after 25th user	load characteristics	error % vs. load characteristics	High	No	Low	High

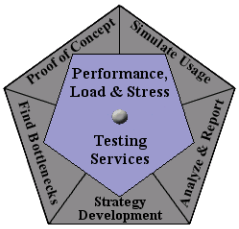
Figure 4: Sample "Visualize and Prioritize" grid

This is normally something I sketch on a whiteboard, and the column heads are different almost every time. It really doesn't matter how you keep track of the tests you come up with, as long as you have a way to remember one idea when the next one hits you and have a list to return to later after you've developed the first several tests and not found what you were hoping to find.

Modify Existing Tests

The quickest and easiest way to gather more information about a particular issue is to use your existing tests. Thinking through the information you'll likely be interested in, the following modifications are ones that can be created quickly and often have large payback — especially in combination with one another — in terms of gathering that information:

- **Eliminate all activities from your test suite that aren't necessary to cause the symptoms and reexecute under various load conditions.** This will help you pinpoint the parameters that lead to the symptoms plus distinguish between a failure, a bottleneck, and a slow spot. One thing to consider is extending the system timeouts to help determine if a symptom is a failure or not.
- **Reexecute using different data.** For instance, if you're doing a search, do a test with a set of data that returns a small number of items, and then another with a dataset that returns a large number of items, or maybe even all of the items. You may also want to execute a test that loops through a large number of potential data items to see if there may be some pattern to the symptoms. It's possible that you could find that only searches for items that begin with, say, the letter *b* are causing



the symptoms (unlikely, but not unheard of).

- **Try various load characteristics.** Don't worry about whether the test reflects reality. Try faster and slower arrival rates, longer and shorter user delays, larger and smaller loads, larger and smaller percentages of users performing the activity displaying the symptoms. These variances will normally help bracket the symptoms. Maybe the symptoms appear only when more than five people do an overlapping search regardless of what other volumes of activities are occurring.

In Part 9 we'll discuss what data other than response times in TestManager to monitor during these tests.

Create New Tests

Sometimes you'll have to record and develop new tests to accomplish the kinds of variance just discussed. There are other situations where you'll want to record and develop new tests as well.

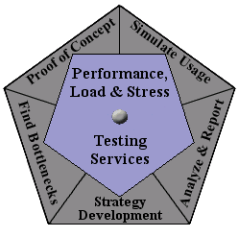
For example, to exercise activities that you and/or the development team have identified as related but that weren't included in your initial test suite, new tests may be needed. This is by far the most common reason to create new tests at this phase in the testing effort. Executing tests on related functionality, both individually and collectively with the tests known to generate symptoms, will generally distinguish between slow spots and bottlenecks.

New tests may also be needed if there's more than one way to accomplish a task that's been identified as a performance issue. For instance, on one application I tested, searching for a particular customer on the "account maintenance" screen took nearly four times as long as searching for the same customer on the "customer maintenance" screen. At first we'd only tested the "account maintenance" screen because the customer-related functionality was intended to be identical. It wasn't until we finally created a new test to evaluate the related functionality on the "customer maintenance" page that we were able to track down the problem by comparing the SQL generated by the two pages.

Sometimes you may want to create a new test to try out a straight-line path to the symptoms even though you have an existing test that's meant to do the same thing. If you're having a particularly difficult time determining the cause of the symptoms, this cause may just be hiding in your script. Rerecord the simplest possible script (no splits, minimal datapools, no abandonment, and so forth) and see if you can recreate the symptoms. If not, do a close comparison of your scripts.

How to Build the Tests

Now we get to the heart of the matter. How do you build tests to collect this next level of information using the Rational® TestStudio software and/or other tools? Unfortunately, there's no cookbook answer. Every piece of information is found in a different way, and even that changes from application to application, platform to platform, and development style to development style. The best I can do is outline some basic techniques and suggest some circumstances where they're most useful. In Part 9 we'll discuss in more detail how to use these tests in combination with other resources at your disposal to conclusively identify the cause of the performance issue.



Use Test Harnesses

I've seen and been part of lots of debates about what a test harness is. Instead of opening up that debate here, let's agree that in this article series, the term *test harness* means any helper application or application modification created for the purpose of making it easier to use Rational TestStudio to collect information about a performance issue.

Test harnesses can be used in many situations. For instance, in the example above with the "account maintenance" and "customer maintenance" screens, we built a test harness to help us evaluate the problem. The test harness was a simple Web page with an input box and a Submit button. We recorded a script that entered various SQL statements into that text box and clicked the Submit button. The Submit button bypassed most of the application we were actually testing and sent the SQL straight to the database. This test harness allowed us to quickly eliminate the database as the cause of the issue without going through the whole battery of tests.

It's unlikely that you'll be the one developing test harnesses. You'll have to work closely with your development staff to create them. You should consider having test harnesses built whenever you can't find another way to isolate a piece of information even though it seems like you should be able to get it using TestStudio. Often, once you start discussing test harnesses with your developers, they'll have ideas for many test harnesses that will provide response time information they wouldn't easily be able to obtain otherwise.

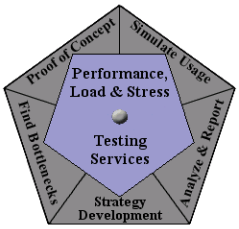
Is It Time to Tune?

Most of the time, after conducting new or modified tests that focus on the symptoms of confirmed performance suspects, the development team will have enough information to start their tuning exercise. If tuning starts at this point, you probably won't be involved again until the developers believe they've solved the problem, at which time you should reexecute all of the tests that previously revealed symptoms.

In general, you should recommend tuning in cases where your tests determine that the suspect was actually a failure or a slow spot rather than a bottleneck. In the case of bottlenecks or inconclusive results, the topics we'll discuss in Part 9 will likely be helpful. Fixing failures or slow spots, or even deciding to accept the performance of a slow spot, may not technically be considered tuning, but they're modifications to either the application or the performance acceptance criteria that affect the performance-testing effort.

Summing It Up

Focusing tests on performance issues is normally a critical step in determining the actual cause of the performance issue and ultimately tuning it. In most cases, modifying or creating focused tests isn't technically difficult but rather is an exercise in determining what tests or modifications will provide information of the highest value in the time you have to collect that information. These tests need to be created and executed quickly and efficiently to be truly useful to the development team. In Part 9 we'll discuss other methods for collecting additional information related to these focused tests.



Acknowledgments

- The original version of this article was written on commission for IBM Rational and can be found on the IBM DeveloperWorks web site

About the Author

Scott Barber is the CTO of PerfTestPlus (www.PerfTestPlus.com) and Co-Founder of the Workshop on Performance and Reliability (WOPR – www.performance-workshop.org). Scott's particular specialties are testing and analyzing performance for complex systems, developing customized testing methodologies, testing embedded systems, testing biometric identification and security systems, group facilitation and authoring instructional or educational materials. In recognition of his standing as a thought leading performance tester, Scott was invited to be a monthly columnist for Software Test and Performance Magazine in addition to his regular contributions to this and other top software testing print and on-line publications, is regularly invited to participate in industry advancing professional workshops and to present at a wide variety of software development and testing venues. His presentations are well received by industry and academic conferences, college classes, local user groups and individual corporations. Scott is active in his personal mission of improving the state of performance testing across the industry by collaborating with other industry authors, thought leaders and expert practitioners as well as volunteering his time to establish and grow industry organizations. His tireless dedication to the advancement of software testing in general and specifically performance testing is often referred to as a hobby in addition to a job due to the enjoyment he gains from his efforts.

About PerfTestPlus

PerfTestPlus was founded on the concept of making software testing industry expertise and thought-leadership available to organizations, large and small, who want to push their testing beyond "state-of-the-practice" to "state-of-the-art." Our founders are dedicated to delivering expert level software-testing-related services in a manner that is both ethical and cost-effective. PerfTestPlus enables individual experts to deliver expert-level services to clients who value true expertise. Rather than trying to find individuals to fit some pre-determined expertise or service offering, PerfTestPlus builds its services around the expertise of its employees. What this means to you is that when you hire an analyst, trainer, mentor or consultant through PerfTestPlus, what you get is someone who is passionate about what you have hired them to do, someone who considers that task to be their specialty, someone who is willing to stake their personal reputation on the quality of their work - not just the reputation of a distant and "faceless" company.