Revolution or Realization?


Scott Barber

Before I dig into this month's topic, I owe you a book review. While I was writing the August column on Linux performance tuning, I received a request to review a new book titled "Performance Tuning for Linux Servers," by Sandra Johnson, Gerrit Huizenga and Badari Pulavarty (IBM Press, 2005). After reading it, I offer the following as a summary:

This book is a must-have for a novice-to-midlevel *nix tuner/administrator, in terms of technical content and information, and a "nice-to-have" reference book for more senior-level folks. It has a higher density of information that is directly applicable to tuning Linux servers for optimal multiuser application performance than any other book I have come across that presents material in ways a non-expert can directly apply.

Another positive but not-so-common thing this book does is provide the author's reference material at the end of each chapter, making it easy for readers who want to know more to get the next level of detail.

On the other hand, editorially this book leaves something to be desired. The chapter introductions are often repetitive, and it's obvious that the contributing authors were solicited to write their own sections independently of the others. I draw this conclusion from the way the presentation style changes from section to section, and how I quickly came to recognize the writing style of certain contributors as I made my way through the book.

To be honest, it reads more like a well-organized collection of articles than a book. Additionally, there are enough editorial oversights and errors (misspellings, etc.) to be periodically annoying if you are reading the book from cover to cover—which I don't recommend.

However, none of these minor criticisms keeps this from being simply fantastic reference material for someone with at least some basic knowledge of Linux who has specific tasks to accomplish.

* * *

Now back to our regularly scheduled column. Recently I've read quite a number of articles, blogs, forum posts and e-mails that discuss various topics that the authors refer to as "agile performance testing." My first thought was, "Cool! It's about time!" But then I started to wonder if this is really a revolution, or just a realization of what has always been.

As I thought more about it, I first tried to visualize what non-agile performance testing looks like. For illustration purposes, let's consider the V-model. With this model, one would establish performance requirements and then create performance test cases early in the project. Then, during the main development effort, scripts would be created, data and integration issues would be dealt with, and then, during the functional testing effort, all the scripts would be finalized and issues resolved so that performance testing could launch smoothly when the application and environment were deemed functionally stable. Once functional stability is achieved, all of those performance test cases would be executed, requirements would be listed as pass or fail—with some supporting data in defect reports—and the race would be on to resolve those issues before the looming go-live date. Sure, that's a simplified model, but I think it's a reasonable representation.

Remember, the V-model assumes the following: Requirements are static (and are actually *required* to pass prior to go-live); testers detect and report defects in writing; and then developers research and resolve them. Testing practices such as exploratory testing (ET) are not generally used, and interdepartmental communication is typically limited to documents, reports and status meetings. Fundamentally, it's assumed that the testing phase will reveal defects that are basically minor and/or relatively easy to resolve.

The benefits of this non-agile model, if done well, are well-defined requirements, well-thought-out test cases and well-developed scripts. This front-end planning and preparation leads to tests that demonstrate compliance or non-compliance with the predefined requirements under the conditions defined by the test cases. Now, while I am an avid supporter of both agile development and context-driven testing, I do support models such as this in some situations. For instance, I've seen this type of model work exceptionally well in situations where the goal of the testing was to determine service-level agreement (SLA) compliance, or when the goal is independent validation and verification (IV&V) of an application publisher's or vendor's performance claims.

So how often *are* the goals of our performance testing to determine SLA compliance or to perform IV&V against a publisher's or vendor's claims? My experience says that an answer of less than 10 percent of the time is probably a safe enough bet to be comical, but let's use that figure for now. What model, then, do the remaining 90 percent of the performance testing projects out there follow? I submit that the following example is at least fairly common.

It is a fairly well-known, yet under-

Scott Barber is the CTO at PerfTestPlus, Inc. His specialty is context-driven performance testing and analysis for distributed multiuser systems. Contact him at sbarber@perftestplus.com.

documented, reality that application performance requirements are rarely well defined, testable and/or actually required for an application to go live. It is also commonly known among performance testers that those performance requirements that were stated at the beginning of the project quickly become goals, with new requirements forming as soon as failing performance test results start coming in—which inevitably changes the performance scenarios to be executed.

To illustrate, consider this example from a project I worked on some years ago. When I reported back to the project stakeholders that our tests were showing 8-second response times rather than the 5 seconds dictated by project requirements, the response I received was something along the lines of, "We're getting 8 seconds, not 5? Yeah, we can live with that, I just pulled that number out of an article anyway. I just need to be convinced that the application will support 1,000 transactions during the first hour of production. We've already paid for the Super Bowl commercial time slot, you know!"

Now let's think for a moment about what qualifies a process as "agile." Signatories of the Manifesto for Agile Software Development (www.agilemanifesto.org) profess the following:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

"That is, while there is value in the items on the right, we value the items on the left more."

To my way of thinking, the scenario I mentioned above easily fits into the "Responding to change over following a plan" category. The initial plan had been to validate both the response time and the application capacity requirements, but as the Super Bowl commercial time slot drew closer, capacity became the priority. One could argue that what really happened wasn't intentionally agile, but rather represents poor requirements collection, or possibly poor schedule management, and both are probably right—but does that change the fact that a tester in a scenario such as this will end up tossing the plan over his or her shoulder to respond to the change?

Or how about another well-known reality of performance testing: that virtually all performance test plans get scrapped after the first significant performance defect is uncovered? Unlike functional testing that, in theory, is "simply" revealing bugs that unit and integration testing missed, virtually everything that is revealed during performance testing is a surprise and could affect the entire application all the way back to fundamental architectural and technology decisions.

All the successful performance testers I have known have used agile methods. They don't say it that way, but they all do it.

Additionally, in the vast majority of cases, finding a functional defect does not preclude a continued search for functional defects—but oftentimes, the first significant performance defect halts additional performance testing until it can be resolved.

So what happens when there is a crisis of this magnitude on software development projects? The plan goes out the window, and everyone is pulled onto this new and frightening issue to resolve it as quickly as possible—and the performance tester is tasked with something to the effect of "Find every other scenario you can that demonstrates this issue. Find out what are the maximum number of users that can use the system before this happens, find out if it recovers, find out what other activities are affected...."

Is it just me, or does that sound like exploratory testing to you? Remembering that ET is defined as simultaneous learning, test design and test execution, I don't see how you could define this situation otherwise. Yes, one could argue that if everything else were done correctly, this situation wouldn't occur, but again, does that really matter when just a few paragraphs ago we agreed that these types of situations occur on 90 percent (conservatively) of all performance testing projects?

What I am saying comes down to this: All of the successful performance testers I have interacted with over the years have used agile methods as regular and expected components of their performance testing efforts. Many don't say it that way, but they all do it. No matter whether the development effort follows the V-model, XP or RUP, the performance testing effort almost always "goes agile" as soon as the first negative results are reported.

Maybe this buzz about agile performance testing is really an indication that performance testers are accepting that their work is often inherently agile by virtue of their exposure to agile testing methods in general. Maybe performance testers are finding that words and concepts that are common in the agile community better reflect what they do.

And maybe, if that is the case, there really is a growing awareness of what performance testers actually do—and maybe, just maybe, this is one small reason why the industry as a whole suddenly seems ready to accept that performance testing is a unique task, requiring unique skills, and not just "one more project management box to check" before go-live, or as "an additional duty for the top automated functional tester."

So is it revolution or realization? You decide. As for me, I don't care one way or the other. I'm just glad that it is happening—whatever "it" is called.