

REDWOOD
Collaborative Media

Software Test & Performance MAGAZINE

VOLUME 6 • ISSUE 11 • NOV/DEC 2009 • \$8.95 • www.stpcollaborative.com

GET
FLEXIBLE!

Apply Agile
Development
Techniques to
Performance Tests

Mitigate Risk
With Data-Driven
Scenario Modeling

Automated Testing

***How Do the Metrics
Measure Up?*** page 14



Software Test & Performance

COLLABORATIVE

Join STP Collaborative for Community, Resources & Knowledge Sharing for the Test & QA Profession.

STP Collaborative serves a global community that provides more than 50,000 software professionals with information, education, training, & professional networking opportunities.



Join by December 31st 2009 to take advantage of charter membership rates.

MEMBERSHIP BENEFITS INCLUDE:

- Software Test & Performance magazine online
- Test & QA Report
- STP Insider eNewsletter
- Magazine and eNewsletter archives
- Resource directory with online demos and special discounts for members
- Member profiles
- Reader commentary
- Online education: Webinars and eSeminars
- Whitepapers
- Case studies
- Special industry reports
- Conference proceedings
- RSS feeds for news and blogs
- Member surveys
- Searchable content
- ...and more

www.STPCollaborative.com

www.seapine.com/stpswift
Satisfy your quality obsession.

Save time while protecting software quality.



© 2009 Seapine Software, Inc. All rights reserved.

Swiftcover.com cut their testing time in half with TestTrack Studio and QA Wizard Pro, while still providing the quality their customers expect.

Seapine's end-to-end Software Quality Assurance (SQA) solutions help you deliver quality products faster. Start with **QA Wizard Pro** for automated testing and add **TestTrack Studio** for issue tracking and test case management—integrated quality assurance solutions that together reduce testing time, saving you money and improving customer satisfaction.

- Reduce quality assurance costs with automated functional and regression testing.
- Manage test case development, defect assignments, and QA verification with one application.
- Track which test cases have been automated, schedule script runs, and link test results with defects.

“So much of success boils down to time. QA Wizard Pro and TestTrack Studio allow us to be more profitable because we do more in less time.” —Test Manager, Swiftcover

Learn how to test faster while protecting quality. Visit www.seapine.com/stpswift

 **Seapine Software™**

QA Wizard® Pro
Automated Testing



Seapine CM®
Change Management



Surround SCM®
Configuration Management



TestTrack® Studio
Test Planning & Tracking



TestTrack® TCM
Test Case Management



TestTrack® Pro
Issue Management



DEPARTMENTS

6 Editorial

At the member-based STP Collaborative, we can hear you now.

8 Conference News

NASA Colonel Mike Mullane shared a lesson in leadership at the October *STP* Conference. Check out coverage of this event and more.

9 ST&Pedia

When it comes to code coverage, what exactly are we measuring when we refer to "percentage of the application"?

By Matt Heusser & Chris McMahon

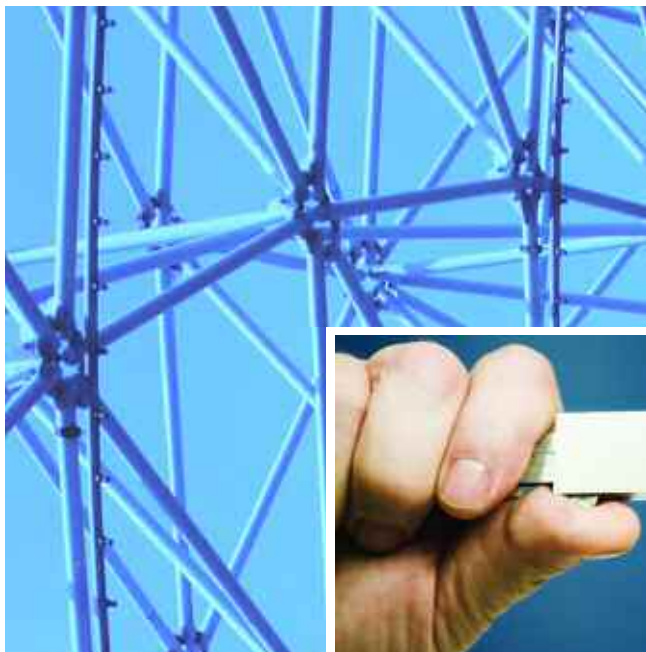
11 CEO Views

Relinquishing control is critical to maximize the benefits of outsourced testing, says Qualitest US CEO Yaron Kottler, who shares global insights into the state of software testing and QA.

38 Case Study

Want reliable runtime intelligence from the field about end user feature usage? Post-build injection modeling, which is being integrated into .Net, Java and more, allows just that.

By Joel Shore



14 COVER STORY

Automated Testing: How Do Your Metrics Measure Up?

Without the proper metrics, all the automation in the world won't guarantee useful software test results. Here's how to ensure measurements that matter.

By Thom Garrett

20 Agile Performance Testing

Flexibility and iterative processes are key to keeping performance tests on track. Follow these guidelines.

By Alexander Podelko

27 Data-Driven Software Modeling Scenarios

To get a clear picture of how a system will perform, create a realistic framework built on real-world data.

By Fiona Charles

34 Six Sigma Part II

How to apply Six Sigma measurement methods to software testing and performance.

By Jon Quigley and Kim Pries



CONTRIBUTORS



THOM GARRETT

has 20 years' experience in planning, development, testing and deployment of complex processing systems for U.S. Navy and commercial applications. He currently works for IDT.



ALEXANDER PODELKO

is a Consulting Member of Technical Staff at Oracle, responsible for performance testing and tuning of the Hyperion product family.



FIONA CHARLES

has 30 years' experience in software development and integration. She has managed and consulted on testing in retail, banking, financial services, health care and more.



JON QUIGLEY

is a test group manager at Volvo Truck and a private product test and development consultant.



*At Stoneridge, **KIM PRIES** is responsible for hardware-in-the-loop software testing and automated test equipment.*

GET MORE ONLINE AT

Software Test & Performance COLLABORATIVE

FORUM

Have you ever tried to automate a test and had it backfire? Tell us your story. Visit our community forum to take part in the discussion.
bit.ly/3Hlapt

STP BLOGS

Visit STP Contributing Editor Matt Heusser's blog, Testing at the Edge of Chaos, at blogs/stpcollaborative.com/matt

NEWSLETTER

The Test & QA Report offers commentary on the latest topics in testing. In our current issue, Karen Johnson shares insights into how to motivate test teams, at stpcollaborative.com/knowledge/519-motivating-test-teams

WHITEPAPER

"Transforming the Load Test Process" discusses the shortcomings of traditional load testing and problem diagnostic approaches with today's complex applications. Download at stpcollaborative.com/system/DynaTraceWhitepaper.pdf
(Sponsored by Dynatrace)

RESOURCES

Research, source, comment and rate the latest testing tools, technologies and services, at stpcollaborative.com/resources

A Legacy Worth Imitating

► THERE'S SOMETHING REFLECTIVE THAT occurs after a big event, whether it's the launch of a new piece of software or, in my case, a conference for the people who test software. Happily, STPCon 2009 was a great success (see page 8), and while we received some constructive criticism, we heard overwhelmingly positive feedback.

Two "deep" thoughts occurred to me in this pensive state. The first is that STP Collaborative is gathering some real momentum. The second is that life takes us down some unexpected paths. One tester at STPCon told me he feels he's rare in that he sought out a career in testing, whereas many tend to fall into the job by circumstance. As someone who planned a career in corporate finance and now finds himself happily serving as CEO of a community-focused media company, I can relate to his peers.

In my case, I credit genetics. My father, Ron Muns, was a social media pioneer before social media technology existed. He founded the Help Desk Institute (now HDI) in 1989—not only before Twitter, but before AOL launched a DOS-based interface. His business succeeded by focusing on its members and working diligently to raise the state of the technical support profession. He would cringe to hear his business called a media company, because to him a membership organization was something different. HDI was an *institute*—an association in the business of serving its members. And being a member was far better than being a mere subscriber because every member had a voice, and HDI magnified that voice.

The company had a conference and publications just like a media business, but it put members first and let advertisers follow. It communicated with members through snail mail (not much choice then!) and enabled networking among members through local chapter meetings in addition to conferences. Before online community building became the rage, HDI built a social media company offline.

The critical parts of this legacy are those we at STP Collaborative strive to imitate: to

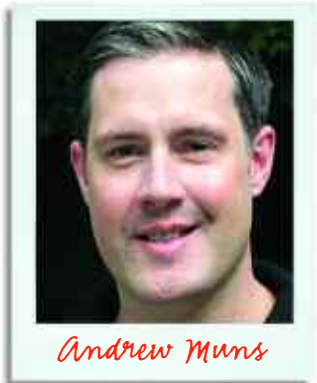
be a member-centric, listening organization that helps individual testers do their jobs better, and to advance the craft and business of software testing.

That may sound like PR-speak, but we're putting our money where our mouths are and bringing in several new team members, including three seasoned HDI alumni. We're excited to welcome Peggy Libbey, who will serve as our new president and COO. Peggy is a business professional with nearly 30 years' experience in finance and executive management, including nine years with HDI. Peggy's background managing a large professional community is marked by her ability to build a corporate infrastructure highly focused on customer service.

We're also pleased to welcome Abbie Caracostas, handling professional training and development; Janette Rovansek, focusing on IT, Web and data management services; Joe Altieri, in sales; and last but not least, veteran business-tech journalist Amy Lipton as our new editor. They all join our existing marketing team, including Jen McClure, CMO and director of community development, and Teresa Cantwell, marketing coordinator. The new team spent their first days on the job together at STPCon, where they talked face to face with testers and STP Collaborative advisory board members, including Matt Heusser, James Bach, Scott Barber, BJ Rollison, Ross Collard and Rex Black. I only wish I'd have had the same opportunity my first week on the job!

So what does all this mean to you? As our operational team grows, our ability to deliver on our vision for STP Collaborative increases. We're both excited and humbled by the opportunity ahead, and we're confident with the participation of a diverse member base we can build a community resource that delivers best-in-class content, training, professional networking and knowledge sharing. We hope you'll join us in pursuing this mission, and making ours a community in which every voice is valued.

Thanks for listening! ☒



Andrew Muns

Editor
Amy Lipton
alipton@stpcollaborative.com

Contributing Editors
Joel Shore
Matt Heusser
Chris McMahon

Copy Editor
Michele Pepe
mpepe@stpcollaborative.com

Art Director
LuAnn T. Palazzo
lpalazzo@stpcollaborative.com

Publisher
Andrew Muns
amuns@redwoodcollaborative.com

Media Sales
Joe Altieri
jaltieri@redwoodcollaborative.com

Chief Operating Officer
Peggy Libbey
plibbey@redwoodcollaborative.com

Chief Marketing Officer
Jennifer McClure
jmcclure@redwoodcollaborative.com

Marketing Coordinator
Teresa Cantwell
tcantwell@redwoodcollaborative.com

Training & Professional Development
Abbie Caracostas
acaracostas@redwoodcollaborative.com

IT Director
Janette Rovansek
jrovansek@redwoodcollaborative.com

Reprints
Lisa Abelson
abelson@stpcollaborative.com
516-379-7097

Membership/Customer Service
membership-services
@stpcollaborative.com

Circulation & List Services
Lisa Fiske
lfiske@stpcollaborative.com

Cover Art by The Design Diva

REDWOOD
Collaborative Media

Founder & CEO
Andrew Muns

Chairman & President
Peggy Libbey

105 Maxess Road, Suite 207
Melville, NY 11747
+ 1-631-393-6051
+ 1-631-393-6057 fax
www.stpcollaborative.com

Next Generation Automated GUI Testing





Object-based Capture & Replay Editor

- ✓ Maintainable recordings via the actions table editor
- ✓ Integration of Ranorex repositories



Automated Testing of Web & Windows Applications

- ✓ Winforms / C# / VB.NET
- ✓ WPF / Silverlight / Win32 / MFC
- ✓ Flash / Flex / Web 2.0 / AJAX /  



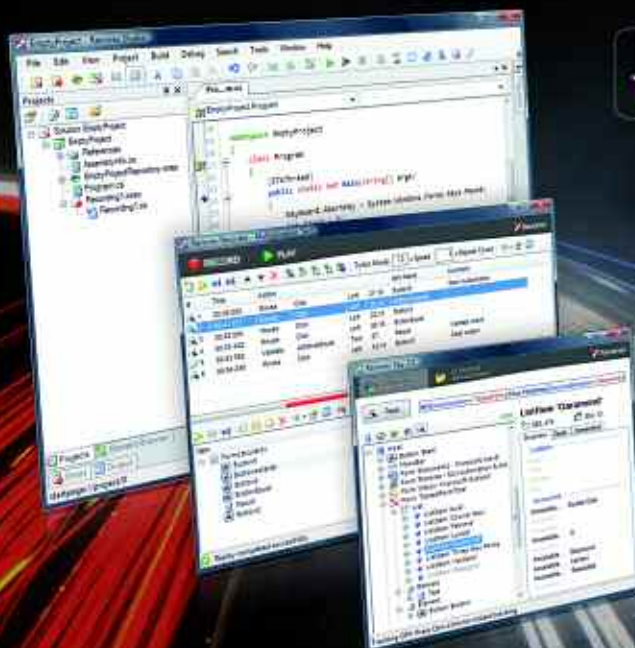
Maintainable GUI Object Repositories

- ✓ Easy to maintain all types of GUI objects
- ✓ Separate test automation from GUI identification



Write tests in C#, VB.NET and IronPython

- ✓ Automatic and flexible code generation in C#, VB.NET and IronPython
- ✓ All-on-one test environment with code editor, code completion and debugging



Get your 30-day Trial
www.ranorex.com

STPCon 2009 Takeaways

► "THE CHALLENGER WAS NO accident—it was an example of a predictable surprise," said Colonel Mike Mullane, a retired NASA astronaut, in the opening keynote at the Software Test & Performance Conference (STPCon 2009), which kicked off Oct. 19 at the Hyatt Regency in Cambridge, Mass. Mullane, who flew in three space shuttle missions, shared insights into the dangers of the "normalization of deviance" as it relates to the 1986 Challenger disaster, as well as to testing in general.

What is "normalization of deviance?" "When there is no negative repercussion for taking a shortcut or deviating from a standard, you come to think it's OK to do it again," he explained, "and that almost always leads to a 'predictable surprise.' "

How to avoid this type of outcome? Mullane recommended the following:

- Realize you are vulnerable to accepting a normalization of deviance. "We all are. We're human."
- Plan the work and work the plan, and be aware of the context. "When you're testing software, use it as the customers will use it, not as the developers designed it."
- Be a team member, not a "passenger." If you spot a problem, point it out—don't assume others know more than you do.
- Lead others by empowering them to contribute to the team.

Making the Leap

"NASA is just like us," and deals with many of the same challenges, including multiple and conflicting vendors, changing vendors midstream, and changing rules and priorities," said DevelopSense's Michael Bolton, who presented the afternoon keynote, "Testing Lessons Learned From the Astronauts."

Bolton discussed two dominant myths: Scientists are absolutely special people, and scientists are simply sophisticated chefs. NASA is an example of these

myths in progress, he said.

Bolton introduced the idea of heuristics, which he defined as a fallible method for solving problems or making decisions. "It's part of the world of being a tester to find inconsistencies," he explained. "Heuristics are valuable in testing because they help accomplish this."

"Testing is about exploration, discovery, investigation and learning," he added. "Testing can be assisted by machines, but it can't be done by machines alone. Humans are crucial."

Speed Geeking, Anyone?

In addition to more than 30 breakout sessions held during the conference, covering topics from agile testing to test automation, performance testing, test management and the latest test tools, technologies and trends, STPCon hosted its first (and very lively) "speed geeking" session, led by *ST&P Magazine* contributor editor Matt Heusser. Participants included Scott Barber, executive director of the Association for Software Testing; exploratory testing experts James Bach and Jonathan Bach; David Gilbert, president and CEO of Sirius SQA; Justin Hunter, president, Hexawise; and Michael Bolton. Each discussion leader had five minutes to cover one aspect of software testing, followed by two minutes of Q&A.

More than 50 test and QA professionals also took part in two new STP training courses at the weeklong conference: Agile Testing with Bob Galen, director of R&D and Scrum Coach at iContact and principal consultant for RGalen Consulting Group, and Test Automation with Rob Walsh, president, EnvisionWare. Galen covered just-in-time test ideas, exploratory testing, all-pairs testing, and ways to drive collaborative agile testing. Walsh addressed how to make the business case for automated testing and introduced participants to a variety of test automation tools. Open source expert Eric Pugh delivered a half-day session on the automated continuous integration process using the open-source Hudson system as an example.



Be a team member, not a "passenger," advised NASA's Colonel Mike Mullane.

New one-day STP workshops included one on test management, led by consultant Rex Black with CA's Bob Carpenter, and another on performance testing, led by Scott Barber, joined by a panel of performance testing experts including James Bach, Dan Bartow, Ross Collard, and Dan Downing.

The Value of Exploratory Testing

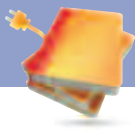
Brothers James and Jonathan Bach presented the closing keynote, sharing a newly revised model of the structure of exploratory testing. "It's wrong to call exploratory testing unstructured testing," James Bach said. "The structure of the test becomes evident through stories."

Exploratory testing involves learning, test design and execution as activities that run in a parallel process, he explained. It emphasizes "the personal freedom and responsibility of testers to optimize the value of their work."

More Online

Check out blog posts and tweets from the conference at www.stpcollaborative.com/community and photos by *ST&P Magazine* contributor Joel Shore at <http://www.stpcon.com>. ☒

Jen McClure is CMO and director of community development for Redwood Collaborative Media.



Considering Code Coverage

▶ **AUTOMATED TESTS ALLOW US** to run a test suite and compare it with an entire application to see what percentage of the application is covered by the tests. The question then becomes: "What exactly are we measuring when we say "percentage of the application"?"

Here are some definitions of various forms of software test coverage, courtesy of Wikipedia:

Function coverage: Number of functions tested/number of functions in the application

Statement coverage: Number of lines of code in the application/total source lines of code

Condition coverage (or predicate coverage): In addition to lines of code, measure statements that might not execute

Decision (or branch) coverage: Every conditional (such as the "if" statement) has been exercised/total possible number of branches

Entry/exit coverage: Has every possible call and return of the function been executed?

Path coverage: Every combination of conditionals/total number of combinations ...And the list goes on.

To further explain the concept, let's look at code coverage for a simple function—a subroutine that validates a student ID at a local college.

According to the requirements, "student IDs are seven-digit numbers between one million and six million, inclusive."

Matt Heusser and Chris McMahon are career software developers, testers and bloggers. Matt also works at Socialtext, where he performs testing and quality assurance for the company's Web-based collaboration software.

Pseudocode for this function is:

```
function validate_studentid(string sid) return
TRUEFALSE
BEGIN
    STATIC TRUEFALSE isOk;
    isOk:= true;

    if (length(sid) is not 7) then
        isOk = False;

    if (number(sid)<=1000000
    or number(sid)>6000000) then
        isOk = False;

    return isOk;
END
```

Now let's see what it would take to get to 100 percent coverage of this subroutine.

Function coverage: Call the function once with the value "999," return with "That is not a valid student ID," and we have 100 percent function coverage.

Statement coverage: Likewise, an invalid value is likely to hit all the lines of the application. (If we did this, would you feel "covered?")

were called from four or five different places, we'd need to test it four or five different times. But since the logic doesn't change, those tests wouldn't tell us anything new.

Path coverage: Of the coverage metrics commonly used, path testing is among the most rigorous. A method containing two binary conditions has a total of four possible execution paths, and all must be tested for full path coverage. Table 1 illustrates a way to accomplish this.

But which are not covered?

- The software has a bug at the bounds. When this comparison is made:

`number(sid)<=1000000`

It will consider all numbers at or below one million invalid and set isOk to false. Yet one million is a valid student ID according to the requirements.



Matt Heusser and Chris McMahon

TABLE 1: TESTING FOR FULL PATH COVERAGE

Input	length(sid) is not seven	number(sid)<=1000000 or number(sid)>6000000	Expected Result
"foo"	True	True	invalid id
7000000	False	True	invalid id
01000001	False	True	invalid id
5000000	False	False	valid id

Decision coverage: This requires us to execute the "if" statements in both true and not true conditions. We can do this with two tests: The text string "999" along with the text string "1500000," or "one point five million." The first value will execute both conditions; the second hits neither.

Entry/exit coverage: If the function

- There's a bug in the *requirements*, which read "one million through six million, inclusive," but should have read "...through 6,999,999, inclusive." (Apparently, the analyst writing the requirements had a different definition of "inclusive" from the rest of the world.)

- What if the student enters an ID of the form 123 456 789 or includes commas? These entries might be limited

by the GUI form itself, which is usually not counted in code coverage metrics.

- Likewise, what if the input window or screen is resized? Will it be repainted? How? Behavior in these situations is often hidden or inherited code that coverage analyzers might not find.

- The subroutine calls `number()` and `length()`, which are external libraries. What if those libraries have hidden bugs in areas we don't know about? (Say, a boundary at 5.8 million causes a "sign bit" rollover, or "if" `number()` throws an exception when words or a decimal point is passed into it?)

- What if the user enters a very long number in an attempt to cause a stack overflow? If we don't test for it, we won't know.

- `isOk` is a static variable. This means only one variable will exist, even if multiple people use the application at the same time. If we're testing a Web app, maybe simultaneous use can cause users to step on each other's results. What happens if

we have simultaneous users in the system? *We don't know. We didn't test for it!*

Suddenly, path testing doesn't look like a magic potion after all. What might work better is input combinatorics coverage. Unfortunately, though, the number of *possible* test combinations is usually infinite.

The Place for Coverage Metrics

The total behavior of an application is more than the "new" code written by developers; it can include the compiler, operating system, browser, third-party libraries, and any special user operations and boundary conditions—and yes, it *is* those special user operations and boundary conditions that get us into trouble.


What coverage metrics tell us is generally how well the developers have tested their code, to make sure it's *possible* the code can work under certain conditions. A "heat map" that shows untested code can be very helpful, for

both the developers and the testers.

Now, when the technical staff crows that test coverage is a high number, consider that the developers have done a good job ensuring the code did as they asked it to do. Often, however, that's a different question from, "Is the software fit for use by a customer?"

When programmer test coverage is high, we've reached the difference between "as programmed" and "fit for use." That's not when we're done. Instead, it's time for testers to step in and do their thing.

Recommended Reading

For more information on this subject, we suggest a paper published in 2004 by Walter Bonds and Cem Kaner called "Software Engineering Metrics: What Do They Measure and How Do We Know?" (<http://www.kaner.com/pdfs/metrics2004.pdf>) and Brian Marick's "How To Misuse Code Coverage," from 1997 (<http://www.exampler.com/testing.com/writings/coverage.pdf>). 

STOP OVERPAYING for HP/Mercury Test Automation SAVE UP TO 50% YOUR FIRST YEAR. SIGN UP NOW!

Join the **Mission Off Mercury** and bring your costs down to earth with the award-winning TestComplete software.

Sign up now and receive a free analysis of your HP/Mercury test automation projects. Learn how you can enjoy **better features**, more **responsive support** and **save up to 80%** over HP in just 3 years. Join thousands of other **smart** businesses and **switch to TestComplete**.



TestComplete™

Sign up now: www.MissionOffMercury.com
or call us directly at (978) 236-7900



Outsourced Testing Demands Trust and Transparency

► YARON KOTTLER IS IN GOOD company. As someone who started at an entry-level position and worked his way to CEO, he shares an experience with top executives from Best Buy, GE, McDonald's, Morgan Stanley, Nortel and Seagate.

Today, Kottler runs QualiTest US, the North American branch of the QualiTest Group. QualiTest is a global software testing and quality assurance consultancy and service provider focused exclusively on onshore QA and testing. The company employs more than 800 testing and QA professionals in 10 locations throughout the United States, Europe and the Middle East.

ANDREW MUNS: What was the first job you ever had?

YARON KOTTLER: The very first job was working at a bike shop in Tel Aviv, but my first real job—believe it or not—was at QualiTest, where I was hired as a junior testing engineer.

How did you get started in the field of software testing and what was the career path that led you to be CEO of QualiTest US?

Originally, I wanted to be a software developer, and when I joined QualiTest [10 years ago], I thought of software testing as a path toward a career as a developer. Nonetheless, I ended up staying in software testing and moved up through the ranks from senior test engineer to test lead, test manager, business manager, load testing manager, and was eventually sent to Turkey to build out the effort there. I ended up traveling back and forth between there and Israel to build that business.

I became CEO after the company acquired IBase Consulting [in 2006]. This was primarily a strategic acquisition that QualiTest used to launch its business in the US market.

What are the key differentiators of QualiTest's testing services that allow you to compete effectively in the US market?



Yaron Kottler

Second is our strength in the onshore market. We believe high-quality onshore teams in many cases outperform their offshore counterparts.

Having a large international practice gives us advantages as well. We have the ability to take the new technologies and practices from Europe or North America and apply them in other markets.

Lastly, we're very good at hiring good people and developing them into world-class testers. We focus more on smarts and potential than resumes, and have a mentorship program and internal training resources that allow people to grow within the organization. Consider my career path!

You interviewed attendees of our conference last year about onshore vs. offshore testing services and found that—contrary to popular belief—onshore testing is often more cost-effective. Have trends since then shown increased interest in

onshore vs. offshore testing?

Absolutely. For many reasons, a test team located onshore outperforms and is more efficient than an offshore team. Testers sharing a similar culture, language and time zone end up having a huge cost advantage that may not be apparent from a glance at the hourly rate.

There has been growth in onshore outsourcing, but I believe this is also due to the overall growth in outsourcing and not necessarily only to taking share from the offshore side.

In your opinion, what is the key to successful outsourcing of test services?

I would say the most important things are transparency and having the appropriate test management tools. Also, being able to trust your service provider and feel comfortable that they can deliver. This is key: If you are going to outsource, you have to learn to let go of some control.

Lastly, I'd say internal organization buy-in. If the operations team or the developers disagree with the outsourcing decision, it will be hard for them to work with the external test team.

What challenges arise for distributed teams using agile methodologies? Does this make outsourcing more or less appropriate for agile practitioners?

I might be old-fashioned, but in my opinion agile and distributed teams don't work together. I see a lot of teams who describe their methodology as agile or Scrum, but who in reality have created a hybrid workflow that's adapted to the realities of a distributed team.

As for the main challenges of agile, these aren't usually related to applying the methodology. Barriers to successful

RESUMÉ

JOB: CEO, QualiTest US
EDU: MBA, Hebrew University;
BA, Interdisciplinary Center
PRIORS: QA specialist, international business development, business manager, testing engineer, testing team leader, all with QualiTest



implementation usually come from the organizational culture. After all, agile theory is simple to understand. Getting teams to embrace the necessary culture for the method to work is the hard part.

How is software testing conducted differently in the US vs. overseas? Are there cultural or regional differences in the definition of "quality"?

Differences in the definition of quality and the importance of quality seem to vary more by organization and industry than by country. Corporate culture varies widely even within a given country, and that's a bigger driver of how much weight is put on shipping a quality product.

There are regional differences in how testing is done, however. US-based organizations are generally much further along in adopting agile and are usually more willing to try new things. The UK is more or less like the US.

In the Netherlands and Germany, testing is much more scripted and there is much more emphasis placed on metrics and measurement.

The use of the words "testing" and "QA" also varies, with many in the US using the term QA to describe what I think is really testing.

Companies overseas tend to have a better understanding of the value of testing and how testing is an activity that saves money when done correctly.

You've spoken a lot about "virtuology." What does this mean and why are testers interested?

The idea is to use virtualization techniques to record a testing session in its entirety. This is analogous to a DVR for a computer where you can rewind, fast forward, pause and go live at any moment. That is basically a time machine, if you think about it. You can now record everything occurring at the time a test is conducted, including network conditions, RAM, ROM, OS conditions as well as any server conditions—not just a series of clicks and keystrokes as is done by GUI automation tools.

This is an amazing tool for debugging and resolving defects that are difficult or impossible to reproduce. A recent study of the defect life cycle in 28 testing projects, conducted by QualiTest, showed that about 9 percent of the defects receive the "can't reproduce"

status at some point of the defect life cycle. This can be attributed to either inadequate reporting or nondeterministic application behavior.

So virtuology can simplify test result documentation. It can also contribute greatly to the efficiency of the testing process, as testers can now spend more of their time actually testing and less on activities such as preparation, environment setup, running pre-steps and gathering test results. I like to summarize the benefits as the three "E's" of virtuology—exploration, efficiency and evidence.

What are some of the most useful virtuology tools currently available?

In my opinion VMware's products offer the most value for the money. We've rolled it out to clients with good success.

In addition to virtuology, how will the next generation of testers make use of cloud computing technologies?

The benefits provided in environment setup are becoming invaluable to functional and compatibility testing and I am sure load testing will continue to be an area to benefit, as well as application environments central to this process.

Virtuology tools aren't currently hosted in the cloud, but I hope we'll see them start to take advantage of cloud computing environments. VMware is great, but is still in the early stages of what these tools can become as they mature.

Call me crazy, but my expectation for the future is that the world is going back to the days of the mainframe in which the cloud is replacing your two-tone computer.

How is the role of the tester within the organization changing and what will test teams look like in 10 years?

I hope testing teams will become more of an extension of operations and will work more closely with, and as advocates for, the users rather than report to development or IT as they usually do.

I also think we'll see agile being used more widely and that we'll see agile and Scrum teams mature.

What is the most important skill an individual tester can possess?

Good analytical skills, understanding the big picture and most importantly understanding what the end user needs and wants.

Testers must understand the business they're working in, and for that to happen you need a dialog with operations or with end-user proxies.

Of course, testing requires a highly specialized skill set that operational people usually don't possess, so we are talking about a multidisciplinary approach to both testing and operations.

Are there any companies that are doing this well?

I'm not sure, but this is essentially about training—not training on testing methodologies, but training that helps testers understand the business context.

What advice would you give to someone beginning a career in testing?

Try to get a job at a place that will let you experiment with lots of different technologies, tools and methodologies. Most importantly, look for a company that will give you some freedom to grow in the direction that suits you best.

I know this is talking my own book, but working with a consulting firm is a great way to start out. You get a lot of experience on a diverse range of projects. This allows you to find out what you like, and limits the time you spend on projects that aren't a good fit. ☒

“My expectation is that the world is going back to the days of the mainframe in which the cloud is replacing your two-tone computer.”

Is your build and test process a little outdated?



The weak link in most software development infrastructures is easy to spot: it's that bottleneck known as the build and test process.

It's not just a technical issue; it's a business issue, because every hour spent manually scripting, dealing with homegrown tools, and waiting for "clean" builds to test is an hour that can't be spent on software innovation.

Find out how Electric Cloud has streamlined and automated build and test processes for Qualcomm, Caterpillar, Intuit, and other leading enterprises—reducing annual development costs by as much as \$2 million and enabling QA and development teams to focus on other things...like producing great software.

Visit www.electric-cloud.com to learn more.



Without the Proper Metrics, All the Automation
In the World Won't Yield Useful Results

Automated Software Testing: Do Your Metrics Measure Up?

By Thom Garrett

“When you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.”

—Lord Kelvin, Physicist

Metrics are an essential gauge of the health, quality, and progress of any automated software testing program. They can be used to evaluate past performance, current status and future trends. Good metrics are objective, measurable, sharply focused, meaningful to the project and simple to understand, providing easily obtainable data. And while many metrics used in conventional software quality engineering tests can be adapted to automated software tests, some metrics—such as percent automatable, automation progress and percent of automated test coverage—are specific to automated testing.

Before we discuss those and other metrics in depth, however, note that not every test should be automated just because it *could* be automated. Perhaps the most

critical decision in the test case requirements-gathering phase of your automation effort is whether it makes sense to automate or not. And that decision itself is typically based on a metric: return on investment. Indeed, the test cases for which you can show that automation will deliver a swift and favorable ROI compared with that of general testing are the cases for which automated testing is justified.

START WITH THE END IN MIND

An integral part of any successful automated testing program is the definition and implementation of specific goals and strategies. During implementation, progress against these goals and strategies must be continuously tracked and measured using various types of auto-

Thom Garrett has 20 years' experience in planning, development, testing and deployment of complex processing systems for U.S. Navy and commercial applications. Specific experience includes rapid introduction and implementation of new technologies for highly sophisticated architectures that support users worldwide. In addition, he has managed and tested all

aspects of large-scale complex networks used in 24/7 environments. He currently works for Innovative Defense Technologies (IDT), and previously worked for America Online, Digital System Resources and other companies, supporting system engineering solutions from requirements gathering to production rollout.

ated and manual testing metrics.

Based on the outcome of these metrics, we can assess the software defects that need to be fixed during a test cycle, and adjust schedules or goals accordingly. For example, if we find that a feature still has too many high-priority defects, the ship date may be delayed or the system may go live without that feature.

Success is measured based on the goals we set out to accomplish relative to the expectations of our stakeholders and customers. That's where metrics come in.

As Lord Nelson implied in his famous quote, if you can measure something, you can quantify it; if you can quantify it, you can explain it; if you can explain it, you have a better chance to improve on it.

We know firsthand that metrics and methods can improve an organization's automated testing process and tracking of its status—our software test teams have used them successfully. But software projects are becoming increasingly complex, thanks to added code for new features, bug fixes and so on. Also, market pressures and corporate belt-tightening mean testers must complete more tasks in less time. This will lead to decreased test coverage and product quality, as well as higher product cost and longer time to deliver.

When implemented properly, however, with the right metrics providing insight into test status, automated software testing can reverse this negative trend. Automation often provides a larger test coverage area and increases overall product quality; it can also reduce test time and delivery cost. This benefit is typically realized over multiple test and project cycles. Automated testing metrics can help assess whether progress, productivity and quality goals are being met.

It serves no purpose to measure for the sake of measuring, of course, so before you determine which automated testing metrics to use, you must set clearly defined goals related directly to the performance of the effort. Here are some metrics-setting fundamentals you may want to consider:

- How much time does it take to run the test plan?
- How is test coverage defined (KLOC, FP, etc.)?
- How much time does it take to do data analysis?
- How long does it take to build a scenario/driver?
- How often do we run the test(s) selected?
- How many permutations of the test(s) selected do we run?

- How many people do we require to run the test(s) selected?
- How much system time/lab time is required to run the test(s) selected?

It is important that the metric you decide on calculate the value of automation, especially if this is the first time automated testing has been used on a particular project. The test team will need to measure the time spent on developing and executing test scripts against the results the scripts produced. For example, the testers could compare the number of hours to develop and execute test procedures with the number of defects documented that probably would not have been revealed during manual testing.

Sometimes it is hard to quantify or measure the automation benefits. For instance, automated testing tools often discover defects that manual tests could not have discovered. During stress testing, for example, 1,000 virtual users execute a specific functionality and the system crashes. It would be very difficult to discover this problem manually—for starters, you'd need 1,000 test engineers!

Automated test tools for data entry or record setup are another way to minimize test time and effort; here, you'd measure the time required to set up the records by hand vs. using an automated tool. Imagine having to manually enter 10,000 accounts to test a system requirement that reads: "The system shall allow the addition of 10,000 new accounts"! An automated test script could easily save many hours of manual data entry by reading account information from a data file through the use of a looping construct, with the data file provided by a data generator.

You can also use automated software testing metrics to determine additional test data combinations. Where manual testing might have allowed you to test "x" number of test data combinations, for example, automated testing might let you test "x+y" combinations. Defects uncovered in the "y" combinations might never have been uncovered in manual tests.

Let's move on to some metrics specific to automated testing.

PERCENT AUTOMATABLE

At the beginning of every automated testing project, you're automating existing manual test procedures, starting a new automation effort from scratch, or some

**Not every
test should be
automated
just because
it could be
automated.**

FIG. 1: PERCENT AUTOMATABLE

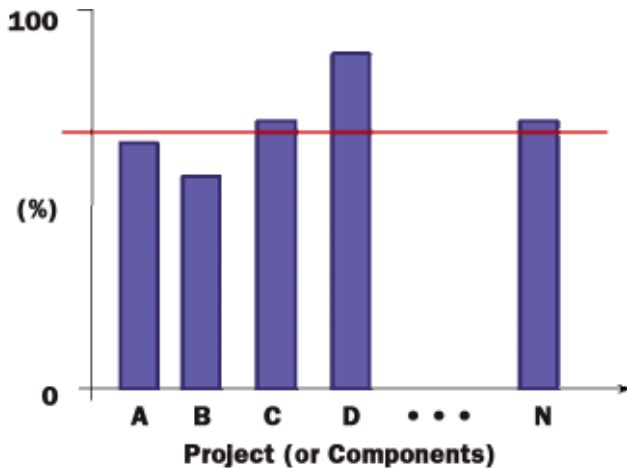


FIG. 2: AUTOMATION PROGRESS

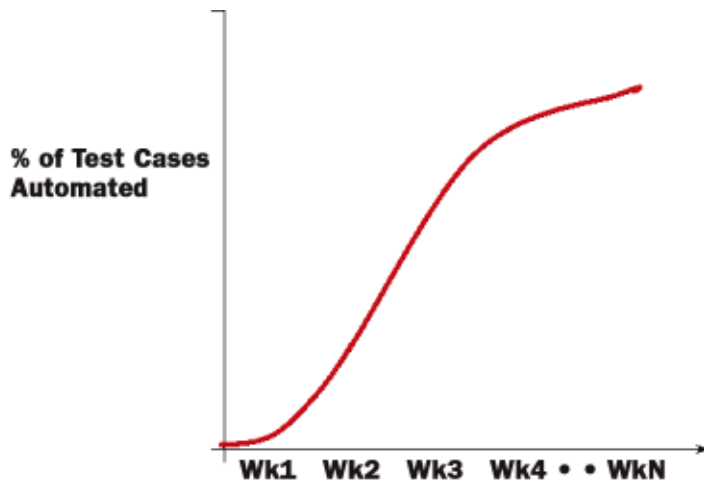
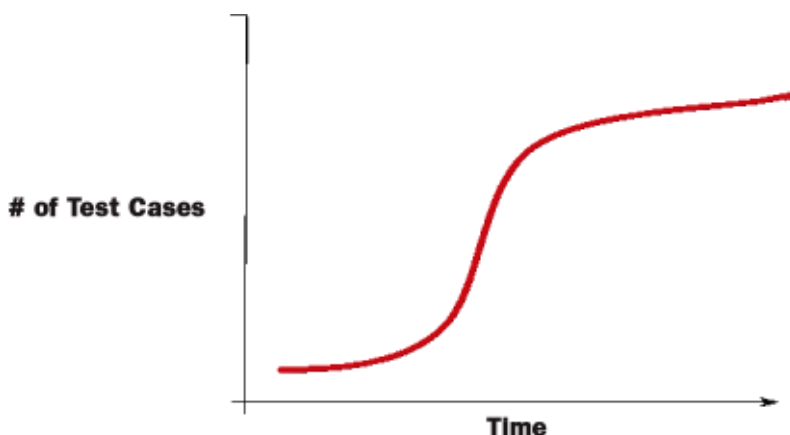


FIG. 3: PROGRESS OVER TIME



combination of the two. Whichever the case, a percent automatable metric can be determined.

Percent automatable can be defined as this: Of a set of given test cases, how many are automatable? This can be represented by the following equation:

$$PA (\%) = \frac{ATC}{TC} = \left(\frac{\# \text{ of test cases automatable}}{\# \text{ of total test cases}} \right)$$

PA = Percent automatable
ATC = # of test cases automatable
TC = # of total test cases

In evaluating test cases to be developed, what should—and should-n't—be considered automatable? One could argue that, given enough ingenuity and resources, almost any software test could be automated. So where do you draw the line? An application area still under design and not yet stable might be considered “not automatable,” for example.

In such cases we must evaluate whether it makes sense to automate, based on which of the set of automatable test cases would provide the biggest return on investment.: Again, when going through the test case development process, determine which tests can and *should* be automated. Prioritize your automation effort based on the outcome. You can use the metric shown in Figure 1 to summarize, for example, the percent automatable of various projects or of a project's components, and set the automation goal.

Automation progress: Of the percent automatable test cases, how many have been automated at a given time? In other words, how far have you gotten toward reaching your goal of automated testing? The goal is to automate 100% of the “automatable” test cases. It's useful to track this metric during the various stages of automated testing development.

$$AP (\%) = \frac{AA}{ATC} = \left(\frac{\# \text{ of actual test cases automated}}{\# \text{ of test cases automatable}} \right)$$

AP = Automation progress
AA = # of actual test cases automated
ATC = # of test cases automatable

The automation progress metric is typically tracked over time. In this case (see Figure 2), time is measured in weeks.

A common metric closely associated with progress of automation, yet not

exclusive to automation, is test progress. Test progress can be defined simply as the number of test cases attempted (or completed) over time.

$$TP = \frac{TC}{T} = \left(\frac{\# \text{ of test cases (attempted or completed)}}{\text{time (days, weeks, months, etc.)}} \right)$$

TP = Test progress

TC = # of test cases (either attempted or completed)

T = some unit of time (days/weeks/months, etc)

The purpose of this metric is to track test progress and compare it with the project plan. Test progress over the period of time of a project usually follows an "S" shape, which typically mirrors the testing activity during the project life cycle: little initial testing, followed by increased testing through the various development phases, into quality assurance, prior to release or delivery.

The metric you see in Figure 3 shows progress over time. A more detailed analysis is needed to determine pass/fail, which can be represented in other metrics.

PERCENT OF AUTOMATED TESTING COVERAGE

Another automated software metric is percent of automated testing coverage: What test coverage is the automated testing actually achieving? This metric indicates the completeness of the testing. It doesn't so much measure how much automation is being executed, but how much of the product's functionality is being covered. For example, 2,000 test cases executing the same or similar data paths may take a lot of time and effort to execute, but this does not equate to a large percentage of test coverage. Percent of automatable test coverage does not specify anything about the effectiveness of the testing; it measures only the testing's dimension.

$$PTC(\%) = \frac{AC}{C} = \left(\frac{\text{automation coverage}}{\text{total coverage}} \right)$$

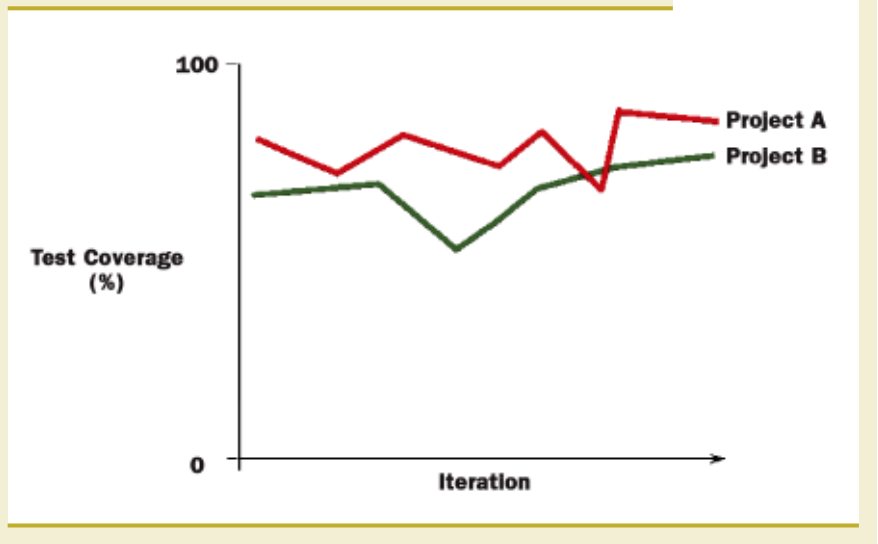
PTC = Percent of automatable testing coverage

AC = Automation coverage

C = Total coverage (KLOC, FP, etc.)

Size of system is usually counted as lines of code (KLOC) or function points (FP). KLOC is a common method of sizing a system, but FP has also gained acceptance. Some argue that FPs can be used to size software

FIG. 4: TEST COVERAGE



applications more accurately. Function point analysis was developed in an attempt to overcome difficulties associated with KLOC (or just LOC) sizing. Function points measure software size by quantifying the functionality provided to the user based on logical design and functional specifications. (There is a wealth of material available regarding the sizing or coverage of systems. A useful resource is Stephen H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. (Addison-Wesley, 2003).

The percent automated test coverage metric can be used in conjunction with the standard software testing metric called test coverage.

$$TC(\%) = \frac{TTP}{TTR} = \left(\frac{\text{total \# of TP}}{\text{total \# of test requirements}} \right)$$

TC = Percent of testing coverage

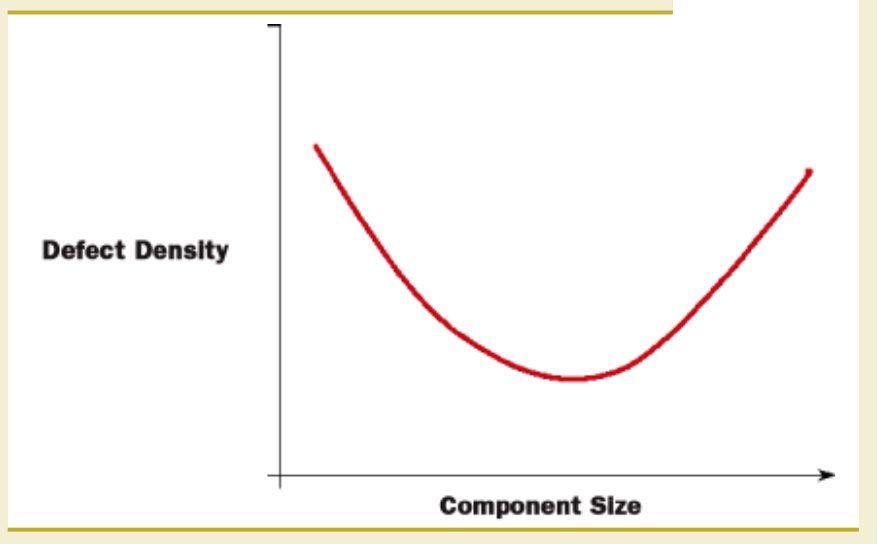
TTP = Total # of test procedures developed

TTR = Total # of defined test requirements

This metric of test coverage (see Figure 4) divides the total number of test procedures developed by the total number of defined test requirements. It provides the test team with a barometer to gauge the depth of test coverage, which is usually based on the defined acceptance criteria.

When testing a mission-critical system, such as operational medical systems, the test coverage indicator must be high relative to the depth of test coverage for non-mission-critical systems. The depth of test coverage for a commercial software product that will be used by millions of end users may also be high relative to a govern-

FIG. 5: COMMON DEFECT DENSITY CURVE



ment information system with a couple of hundred end users.

DEFECT DENSITY

Defect density is another well-known metric not specific to automation. It is a measure of the total known defects divided by the size of the software entity being measured. For example, if there is a high defect density in a specific functionality, it is important to conduct a causal analysis. Is this functionality very complex, so the defect density is expected to be high? Is there a problem with the design/implementation of the functionality? Were insufficient or wrong resources assigned to the functionality because an inaccurate risk had been assigned to it? Could it be inferred that the developer, responsible for this specific functionality needs more training?

$$DD = \frac{D}{SS} = \left(\frac{\text{\# of known defects}}{\text{total size of system}} \right)$$

DD = Defect density
D = # of known defects
SS = Total size of system

One use of defect density is to map it against software component size. Figure 5 illustrates a typical defect density curve we've experienced, where small and larger components have a higher defect density ratio.

Additionally, when evaluating defect density, the priority of the defect should be considered. For example, one application requirement may have as many as 50 low-priority defects and still pass because the acceptance criteria have been satis-



SOFTWARE TEST METRICS: THE ACRONYMS

AA	# of actual test cases automated
AC	Automation coverage
AP	Automation progress
ATC	# of test cases automatable
D	# of known defects
DA	# of acceptance defects found after delivery
DD	Defect density
DRE	Defect removal efficiency
DT	# of defects found during testing
DTA	Defect trend analysis
FP	Function point
KLOC	Lines of code (thousands)
LOC	Lines of code
PR	Percent automatable
PTC	Percent of automatable testing coverage
ROI	Return on investment
SPR	Software problem report
SS	Total size of system to be automated
T	Time (some unit of time—days, weeks, months, etc.)
TC	# of total test cases
TP	Test progress
TPE	# of test procedures executed over time

fied. Still, another requirement might only have one open defect that prevents the acceptance criteria from being satisfied because it is a high priority. Higher-priority requirements are generally weighted more heavily.

Figure 6 shows one approach to using the defect density metric. Projects can be tracked over time (for example, stages of the development cycle).

Defect trend analysis is another closely related metric to defect density.

Defect trend analysis is calculated as:

$$DTA = \frac{D}{TPE} = \left(\frac{\text{\# of known defects}}{\text{\# of test procedures executed}} \right)$$

DTA = Defect trend analysis
D = # of known defects
TPE = # of test procedures executed over time

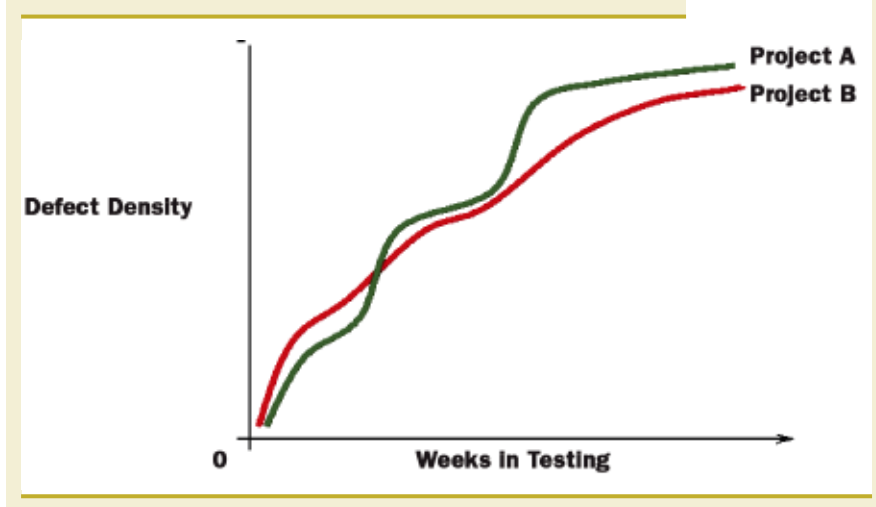
Defect trend analysis can help determine the trend of defects found. Is the trend improving as the testing phase is winding down, or is the trend worsening? Showing defects the test automation uncovered that manual testing didn't or couldn't have is an additional way to demonstrate ROI. During the testing process, we have found defect trend analysis one of the more useful metrics to show the health of a project. One approach to showing a trend is to plot total number of defects along with number of open software problem reports (see Figure 7).

Effective defect tracking analysis can present a clear view of testing status throughout the project. A few more common metrics related to defects are:

Cost to locate defect = Cost of testing/number of defects located

Defects detected in testing = Defects detected in testing/total

FIG. 6: USING DD TO TRACK OVER TIME



system defects

Defects detected in production = Defects detected in production/system size

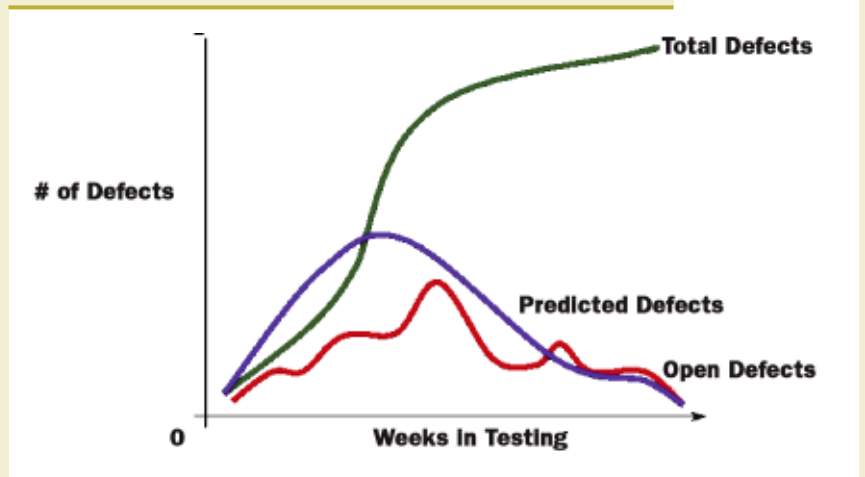
Some of these metrics can be combined and used to enhance quality measurements, as shown in the next section.

IMPACT ON QUALITY

One of the more popular metrics for tracking quality (if defect count is used as a measure of quality) through testing is defect removal efficiency. DRE, while not specific to automation, is very useful in conjunction with automation efforts. It is used to determine the effectiveness of defect removal efforts and is also an indirect measurement of the quality of the product. The value of DRE is calculated as a percentage—the higher the percentage, the higher the positive impact on product quality because it represents the timely identification and removal of defects at any particular phase.

$$DRE(\%) = \frac{DT}{DT+DA} = \left(\frac{\text{\# of defects found during testing}}{\text{\# of defects found during testing} + \text{\# of defects found after delivery}} \right)$$

FIG. 7: DEFECT TREND ANALYSIS



Courtesy of <http://www.teknologika.com/blog/SoftwareDevelopmentMetricsDefectTracking.aspx>

DRE = Defect removal efficiency
DT = # of defects found during testing
DA = # of acceptance defects found after delivery

The highest attainable value of DRE is "1," which equates to "100%." But we have found that, in practice, an efficiency rating of 100% is not likely.

DRE should be measured during the different development phases. If DRE is low during the analysis and design stages, for instance, it may indicate that more time should be

spent improving the way formal technical reviews are conducted, and so on.

This calculation can be extended for released products as a measure of the number of defects in the product that were not caught during product development or testing. ☒



SEE MORE ON SOFTWARE TESTING METRICS at www.stpcollaborative.com

LOAD TEST COMPLEX WEBSITES FOR A FRACTION OF THE COST

INTRODUCING BROWSERMOB. Reduce your costs with a unique pay-as-you-go model. Control of thousands of common web browsers in minutes. FREE consultation by industry experts during your trial. Come see how we are redefining the world of load testing. Sign up today!

RECEIVE **100 FREE CREDITS**


- \$300 VALUE, ENOUGH TO TEST 10K USERS
- CONSULTATION WITH EXPERT TESTERS
- LOG ON TODAY AND START TESTING!

The screenshot shows the BrowserMob website interface. At the top, there's a navigation bar with links like 'Home', 'About Us', 'My Account', and 'Logout'. Below the navigation bar, there's a main content area with several articles. One article is titled 'The Four Pillars of Testing in a DevOps Environment'. Another article is titled 'The Agile Process Creates Testing Phases - Which Are Solvable?'. There's also a sidebar on the right with a section titled 'Categories' and a list of categories like 'All', 'Agile', 'DevOps', 'Performance', 'Security', 'Testing', and 'Tools'. At the bottom of the page, there's a footer with the BrowserMob logo and the website URL www.browsermob.com/stp.

Agile Performance Testing

Flexibility and
Iterative Processes
Are Key to Keeping
Tests on Track





Agile software development involves iterations, open collaboration and process adaptability throughout a project's life cycle. The same approaches are fully applicable to performance testing projects. So you need a plan, but it has to be malleable. It becomes an iterative process involving tuning and troubleshooting in close cooperation with developers, system administrators, database administrators and other experts.

By Alexander Podelko

You run a test and get a lot of information about the system. To be efficient you need to analyze the feedback you get, make modifications and adjust your plans if necessary. Let's say, for example, you plan to run 20 different tests, but after executing the first test you discover a bottleneck (for instance, the number of Web server threads). Unless you eliminate the bottleneck, there's no point running the other 19 tests if they all use the Web server. To identify the bottleneck, you may need to change the test scenario.

Even if the project scope is limited to pre-production performance testing, an agile, iterative approach helps you meet your goals faster and more efficiently, and learn more about the system along the way. After we prepare a test script (or generate workload some other way), we can run single- or multi-user tests, analyze results and sort out errors. The source of errors can vary—you can experience script errors, functional errors and errors caused directly by performance bottlenecks—and it doesn't make sense to add load until you determine the specifics. Even a single script allows you to locate many problems and tune the system at least partially. Running scripts separately also lets you see the amount of resources used by each type of load so you can build a system "model" accordingly (more on that later).

The word "agile" in this article doesn't refer to any specific development process or methodology; performance testing for agile development projects is a separate topic not covered in this paper. Rather, it's used as an application of the agile principles to performance engineering.

WHY THE 'WATERFALL' APPROACH DOESN'T WORK

The "waterfall" approach to software development is a sequential process in which development flows steadily downward (hence the name) through the stages of requirements analysis, design, implementation, testing, integration and maintenance. Performance testing typically includes these steps:

- Prepare the system.
- Develop requested scripts.
- Run scripts in the requested combinations.
- Compare results with the requirements provided.
- Allow some percentage of errors according

to the requirements.

- Involve the development team if requirements are missed.

At first glance, the waterfall approach to performance testing appears to be a well-established, mature process. But there are many potential—and serious—problems. For example:

- The waterfall approach assumes that the entire system—or at least all the functional components involved—is ready for the performance test. This means the testing can't be done until very late in the development cycle, at which point even small fixes would be expensive. It's not feasible to perform such full-scope testing early in the development life cycle. Earlier performance testing requires a more agile, explorative process.

- The scripts used to create the system load for performance tests are themselves software. Record/playback load testing tools may give the tester the false impression that creating scripts is quick and easy, but correlation, parameterization, debugging and verification can be extremely challenging. Running a script for a single user that doesn't yield any errors doesn't prove much. I've seen large-scale corporate performance testing where none of the script executions made it through logon (single sign-on token wasn't correlated), yet performance testing was declared successful and the results were reported to management.

- Running all scripts simultaneously makes it difficult to tune and troubleshoot. It usually becomes a good illustration of the shot-in-the-dark antipattern—"the best efforts of a team attempting to correct a poorly performing application without the benefit of truly understanding why things are as they are" (http://www.kirk.blogcity.com/proposed_antipattern_shot_in_the_dark.htm). Or you need to go back and deconstruct tests to find exactly which part is causing the problems. Moreover, tuning and performance troubleshooting are iterative processes, which are difficult to place inside the "waterfall." And in most cases, you can't do them offline—you need to tune the system and fix the major problems before the results make sense.

- Running a single large test, or even sev-

Alexander Podelko has specialized in performance engineering for 12 years. Currently he is a Consulting Member of Technical Staff at Oracle, responsible for performance testing and tuning of the Hyperion product family.

eral large tests, provides minimal information about system behavior. It doesn't allow you to build any kind of model, formal or informal, or to identify any relationship between workload and system behavior. In most cases, the workload used in performance tests is only an educated guess, so you need to know how stable the system would be and how consistent the results would be if real workload varied.

Using the waterfall approach doesn't change the nature of performance testing; it just means you'll probably do a lot of extra work and end up back at the same point, performance tuning and troubleshooting, much later in the cycle. Not to mention that large tests involving multiple-use cases are usually a bad point to start performance tuning and troubleshooting, because symptoms you see may be a cumulative effect of multiple issues.

Using an agile, iterative approach doesn't mean redefining the software development process; rather, it means finding new opportunities inside existing processes to increase efficiency overall. In fact, most good performance engineers are already doing performance testing in an agile way but just presenting it as "waterfall" to management. In most cases, once you present and get management approval on a waterfall-like plan, you're free to do whatever's necessary to test the system properly inside the scheduled time frame and scope. If opportunities exist, performance engineering may be extended further, for example, to early performance checkpoints or even full software performance engineering.

TEST EARLY

Although I've never read or heard of anybody arguing against testing early, it rarely happens in practice. Usually there are some project-specific reasons—tight schedules or budgets, for instance—preventing such activities (if somebody thought about them at all).

Dr. Neil Gunther, in his book *Guerrilla Capacity Planning* (Springer, 2007), describes the reasons management (consciously or unconsciously)

resists testing early. While Gunther's book presents a broader perspective on capacity planning, the methodology discussed, including the guerrilla approach, is highly applicable to performance engineering.

Gunther says there's a set of unspoken assumptions behind the resistance to performance-related activities. Some that are particularly relevant to performance engineering:

- The schedule is the main measure of success.
- Product production is more important than product performance.
- We build product first and then tune performance.
- Hardware is not expensive; we can just add more of it if necessary.
- There are plenty of commercial tools that can do it.

It may be best to accept that many project schedules don't allow sufficient time and resources for performance engineering activities and proceed in "guerrilla" fashion: Conduct performance tests that are less resource-intensive, even starting by asking just a few key questions and expanding as time and money permit.

The software performance engineering approach to development of software systems to meet performance requirements has long been advocated by Dr. Connie Smith and Dr. Lloyd Williams (see, for example, their book *Performance Solutions*, Addison-Wesley, 2001). While their methodology doesn't focus on testing initiatives, it can't be successfully implemented without some preliminary testing and data collection to determine both model inputs and parameters, and to validate model results.

Whether you're considering a full-blown performance engineering or guerrilla-style

approach, you still need to obtain baseline measurements on which to build your calculations. Early performance testing at any level of detail can be very valuable at this point.

A rarely discussed aspect of early performance testing is unit performance testing. The unit here may be any part of the system—a component, service or

device. This is not a standard practice, but it should be. The later in the development cycle, the more costly and difficult it becomes to make changes, so why wait until the entire system is assembled to start performance testing? We don't wait in functional testing. The predeployment performance test is an analog of system or integration tests, but it's usually conducted without any "unit testing" of performance.

The main obstacle is that many systems are somewhat monolithic; the parts, or components, don't make much sense by themselves. But there may be significant advantages to test-driven development. If you can decompose the system into components in such a way that you may test them separately for performance, you'll only need to fix integration problems when you put the system together. Another problem is that many large corporations use a lot of third-party products in which the system appears as a "black box" that's not easily understood, making it tougher to test effectively.

During unit testing, variables such as load, security configuration and amount of data can be reviewed to determine their impact on performance. Most test cases are simpler and tests are shorter in unit performance testing. There are typically fewer tests with limited scope—for example, fewer variable combinations than in a full stress or performance test.

We shouldn't underestimate the power of the single-user performance test: If the system doesn't perform well for a single user, it certainly won't perform well for multiple users. Single-user testing is conducted throughout the application development life cycle, during functional testing and user acceptance testing, and gathering performance data can be extremely helpful during these stages. In fact, single-user performance tests may facilitate earlier detection of performance problems and indicate which business functions and application code need to be investigated further.

So while early performance engineering is definitely the best approach (at least for product development) and has long been advocated, it's still far from commonplace. The main problem here is that the mindset should change from a simplistic "record/playback" performance testing occurring late in the product life cycle to a more robust, true perform-

**Why wait
until the entire
system is
assembled
to start
performance
testing?**

ance engineering approach starting early in the product life cycle. You need to translate “business functions” performed by the end user into component/unit-level usage, end-user requirements into component/unit-level requirements and so on. You need to go from the record/playback approach to using programming skills to generate the workload and create stubs to isolate the component from other parts of the system. You need to go from “black box” performance testing to “gray box.”

If you’re involved from the beginning of the project, a few guerrilla-style actions early on can save you (and the project) a lot of time and resources later. But if you’re called in later, as is so often the case, you’ll still need to do the best performance testing possible before the product goes live. The following sections discuss how to make the most of limited test time.

THE IMPORTANCE OF WORKLOAD GENERATION

The title of Andy Grove’s book *Only the Paranoid Survive* may relate even better to performance engineers than to executives! It’s hard to imagine a good performance engineer without this trait. When it comes to performance testing, it pays to be concerned about every part of the process, from test design to results reporting.

Be a performance test architect. The sets of issues discussed below require architect-level expertise.

1) Gathering and validating all requirements (workload definition, first and foremost), and projecting them onto the system architecture:

Too many testers consider all information they obtain from the business side (workload descriptions, scenarios, use cases, etc.) as the “holy scripture.” But while businesspeople know the business, they rarely know much about performance engineering. So obtaining requirements is an iterative process, and every requirement submitted should be evaluated and, if possible, validated. Sometimes performance requirements are based on solid data; sometimes they’re just a guess. It’s important to know how reliable they are.

Scrutinize system load carefully as well. Workload is an input to testing, while response times are output. You may decide if response times are acceptable even after the test, but you must define workload beforehand.



The gathered requirements should be projected onto the system architecture because it’s important to know if included test cases add value by testing different sets of functionality or different components of the system. It’s also important to make sure we have test cases for every component (or, if we don’t, to know why).

2) Making sure the system under test is configured properly and the results obtained may be used (or at least projected) for the production system:

Environment and setup considerations can have a dramatic effect. For instance:

- What data is used? Is it real production data, artificially generated data or just a few random records? Does the volume of data match the volume forecasted for production? If not, what’s the difference?
- How are users defined? Do you have an account set with the proper security rights for each virtual user or do you plan to re-use a single administrator ID?
- What are the differences between the production and test environments? If your test system is just a subset of your production

system, can you simulate the entire load or just a portion of that load? Is the hardware the same?

It’s essential to get the test environment as close as possible to the production environment, but performance testing workload will never match production workload exactly. In “real life,” the workload changes constantly, including user actions nobody could ever anticipate.

Indeed, performance testing isn’t an exact science. It’s a way to decrease risk, not to eliminate it. Results are only as meaningful as the test and environment you created. Performance testing typically involves limited functional coverage, and no emulation of unexpected events. Both the environment and the data are often scaled down. All of these factors confound the straightforward approach to performance testing, which states that we simply test X users simulating test cases A and B. This way, we leave aside a lot of questions: How many users can the system handle? What happens if we add other test cases? Do ratios of use cases matter? What if some administrative activities happen in parallel? All of these questions and more require some investigation.

Perhaps you even need to investigate the system before you start creating performance test plans. Performance engineers sometimes have system insights nobody else has; for example:

- Internal communication between client and server if recording used
- Timing of every transaction (which may be detailed to the point of specific requests and sets of parameters if needed)
- Resource consumption used by a specific transaction or a set of transactions

This information is additional input—often the original test design is based on incorrect assumptions and must be corrected based on the first results.

Be a script writer. Very few systems today are stateless systems with static content using plain HTML—the kind of systems that lend themselves to a simple “record/playback” approach. In most cases there are many obstacles to creating a proper workload. If it’s the first time you see the system, there’s absolutely no guarantee you can quickly record and play back scripts to create the workload, if at all.

Creating performance testing scripts and other objects is, in essence,

a software development project. Sometimes automatic script generation from recording is mistakenly interpreted as the entire process of script creation, but it's only the beginning. Automatic generation provides ready scripts in very simple cases, but in most nontrivial cases it's just a first step. You need to correlate and parametrize scripts (i.e., get dynamic variables from the server and use different data for different users).

After the script is created, it should be evaluated for a single user, multiple users and with different data. Don't assume the system works correctly just because the script was executed without errors. Workload validation is critical: We have to be sure the applied workload

is doing what it's supposed to do and that all errors are caught and logged. This can be done directly, by analyzing server responses or, in cases where that's impossible, indirectly—for example, by analyzing the application log or database for the existence of particular entries.

Many tools provide some way to verify workload and check errors, but a complete understanding of what exactly is happening is necessary. For example, HP LoadRunner reports only HTTP errors for Web scripts by default (500 "Internal Server Error," for example). If we rely on the default diagnostics, we might still believe that everything is going well when we get "out of memory" errors instead of the requested reports. To catch such errors, we should add special commands to our script to check the content of HTML pages returned by the server.

When a script is parameterized, it's good to test it with all possible data. For example, if we use different users, a few of them might not be set up properly. If we use different departments, some could be mistyped or contain special symbols that must be properly encoded. These problems are easy to spot early on, when you're just debugging a particular script. But if you wait until the final, all-script tests, they muddy the entire picture and make it difficult to see the real problems.

My group specializes in performance testing of the Hyperion line of Oracle products, and we've found that a few scripting challenges exist for almost every product. Nothing exceptional—they're usually easily identified and resolved—but time after time we're called on to save problematic performance testing projects only to discover serious problems with scripts and scenarios that make test results meaningless. Even experienced testers stumble, but many problems could be avoided if more time were spent analyzing the situation.

Consider the following examples, which are typical challenges you can face with modern Web-based applications:

1) Some operations, like financial consolidation, can take a long time. The client starts the operation on the server, then waits for it to finish, as a progress bar shows on screen. When recorded, the script looks like (in LoadRunner pseudocode):

```
web_custom_request("XMLDataGrid.asp_7",
"URL={URL}/Data/XMLDataGrid.asp?Action=
EXECUTE&TaskID=1024&RowStart=1&ColStart=
2&RowEnd=1&ColEnd=2&SelType=0&Format=
JavaScript", LAST);
```

```
web_custom_request("XMLDataGrid.asp_8",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS",
LAST);
```

```
web_custom_request("XMLDataGrid.asp_9",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS",
LAST);
```

```
web_custom_request("XMLDataGrid.asp_9",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS",
LAST);
```

Each request's activity is defined by the *?Action=* part. The number of *GETCONSOLSTATUS* requests recorded depends on the processing time.

In the example above, the request was recorded three times, which means the consolidation was done by the moment the third *GETCONSOLSTATUS* request was sent to the server. If you play back this script, it will work this way: The script submits the consolidation in the *EXECUTE* request and then calls *GET-*

CONSOLSTATUS three times. If we have a timer around these requests, the response time will be almost instantaneous, while in reality the consolidation may take many minutes or even hours. If we have several iterations in the script, we'll submit several consolidations, which continue to work in the background, competing for the same data, while we report subsecond response times.

Consolidation scripts require creation of an explicit loop around *GETCONSOLSTATUS* to catch the end of the consolidation:

```
web_custom_request("XMLDataGrid.asp_7",
"URL={URL}/Data/XMLDataGrid.asp?Action=
EXECUTE&TaskID=1024&RowStart=1&ColStart=
2&RowEnd=1&ColEnd=2&SelType=0&Format=
JavaScript", LAST);
```

```
do {
```

```
sleep(3000);
```

```
web_reg_find("Text=1","SaveCount=abc_count",
LAST);
```

```
web_custom_request("XMLDataGrid.asp_8",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS", LAST);
```

```
} while (str-
cmp(lr_eval_string("{abc_count}"),"1")==0);
```

Here, the loop simulates the internal logic of the system, sending *GETCONSOLSTATUS* requests every three seconds until the consolidation is complete. Without such a loop, the script just checks the status and finishes the iteration while the consolidation continues long after that.

2) Web forms are used to enter and save data. For example, one form can be used to submit all income-related data for a department for a month. Such a form would probably be a Web page with two drop-down lists (one for departments and one for months) on the top and a table to enter data underneath them. You choose a department and a month on the top of the form, then enter data for the specified department and month. If you leave the department and month in the script hardcoded as recorded, the script would be formally correct, but the test won't make sense at all—each virtual user will try to overwrite exactly the same data for the same department and the same month. To make it meaningful, the script should be parameterized to save data in different data intersections. For example, different departments may be used by each

user. To parameterize the script, we need not only department names but also department IDs (which are internal representations not visible to users that should be extracted from the metadata repository). Below is a sample of correct LoadRunner pseudocode (where values between { and } are parameters that may be generated from a file):

```
web_submit_data("WebFormGenerated.asp",
"Action=http://hfmtest.us.schp.com/HFM/data/WebFormGenerated.asp?FormName=Tax+QFP&caller=GlobalNav&iscontained=Yes",
ITEMDATA,
"Name=SubmitType", "Value=1",
ENDITEM,
"Name=FormPOV", "Value=TaxQFP",
ENDITEM,
"Name=FormPOV", "Value=2007",
ENDITEM,
"Name=FormPOV", "Value=[Year]",
ENDITEM,
"Name=FormPOV", "Value=Periodic",
ENDITEM,
"Name=FormPOV", "Value=
{department_name}", ENDITEM,
"Name=FormPOV", "Value=<Entity
Currency>", ENDITEM,
"Name=FormPOV",
"Value=NET_INCOME_LEGAL", ENDITEM,
"Name=FormPOV", "Value=[ICP Top]",
ENDITEM,

"Name=MODVAL_19.2007.50331648.1.
{department_id}.14.407.2130706432.4.1.90.0.345
", "Value=<1.7e+3>;", ENDITEM,

"Name=MODVAL_19.2007.50331648.1.
{department_id}.14.409.2130706432.4.1.90.0.345
", "Value=<1.7e+2>;", ENDITEM, LAST);
```

If department name is parameterized but department ID isn't, the script won't work properly. You won't get an error, but the information won't be saved. This is an example of a situation that never can happen in real life—users working through GUIs would choose department name from a drop-down box (so it always would be correct) and matching ID would be found automatically. Incorrect parameterization leads to sending impossible combinations of data to the server with unpredictable results. To validate this information, we would check what's saved after the test—if you see your data there, you know the script works.

TUNE AND TROUBLESHOOT

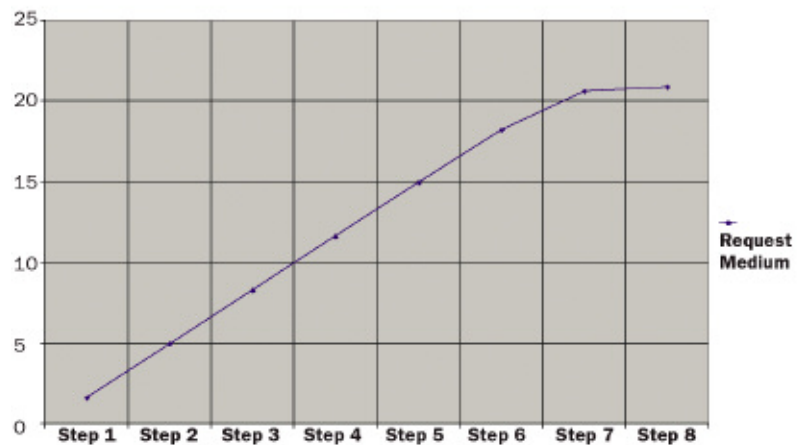
Usually, when people talk about performance testing, they don't separate it from tuning, diagnostics or capacity planning. "Pure" performance testing is possible only in rare cases when the system and all optimal settings are well-known. Some tuning activities are typically necessary at the beginning of testing to be sure the system is tuned prop-

erly and the results will be meaningful. In most cases, if a performance problem is found, it should be diagnosed further, up to the point when it's clear how to handle it. Generally speaking, performance testing, tuning, diagnostics and capacity planning are quite different processes, and excluding any one of them from the test plan (if they're assumed) will make the test unrealistic from the beginning.

Both performance tuning and troubleshooting are iterative processes where you make the change, run the test, analyze the results and repeat the process based on the findings. The advantage of performance testing is that you apply the same synthetic load, so you can accurately quantify the impact of the change that was made. That makes it much simpler to find problems during performance testing than to wait until they happen in production, when workload is changing all the time. Still, even in the test environment, tuning and performance troubleshooting are quite

ing—testers look for bugs and log them into a defect tracking system, then the defects are prioritized and fixed independently by the developers—doesn't work well for performance testing. First, a reliability or performance problem often blocks further performance testing until the problem is fixed or a workaround is found. Second, usually the full setup, which tends to be very sophisticated, should be used to reproduce the problem. Keeping the full setup for a long time can be expensive or even impossible. Third, debugging performance problems is a sophisticated diagnostic process usually requiring close collaboration between a performance engineer running tests and analyzing the results and a developer profiling and altering code. Special tools may be necessary; many tools, such as debuggers, work fine in a single-user environment but do not work in the multi-user environment, due to huge performance overheads. What's usually required is the

FIG. 1: THROUGHPUT



sophisticated diagnostic processes usually requiring close collaboration between a performance engineer running tests and developers and/or system administrators making changes. In most cases, it's impossible to predict how many test iterations will be necessary. Sometimes it makes sense to create a shorter, simpler test still exposing the problem under investigation. Running a complex, "real-life" test on each tuning or troubleshooting iteration can make the whole process very long and the problem less evident because of different effects the problem may have on different workloads.

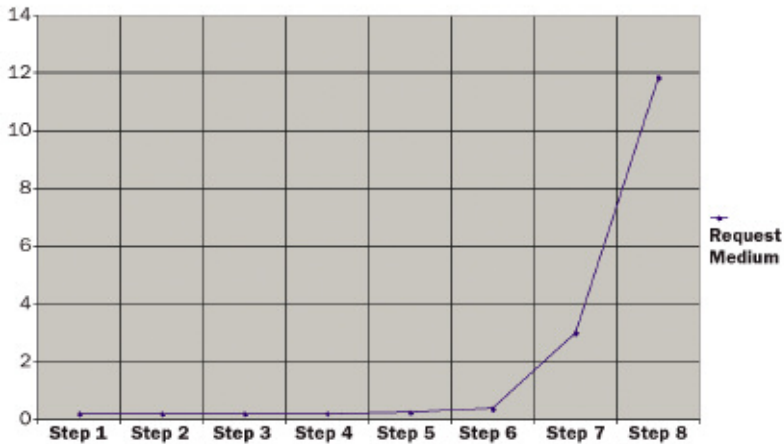
An asynchronous process to fixing defects, often used in functional test-

synchronized work of performance engineering and development to fix the problems and complete performance testing.

BUILD A MODEL

Creating a model of the system under test significantly increases the value of performance testing. First, it's one more way to validate test correctness and help to identify system problems—deviations from expected behavior might signal issues with the system or with the way you create workload. Second, it allows you to answer questions about the sizing and capacity planning of the system.

Most performance testing doesn't require a formal model created by a sophisticated modeling tool—it may

FIG. 2: RESPONSE TIME

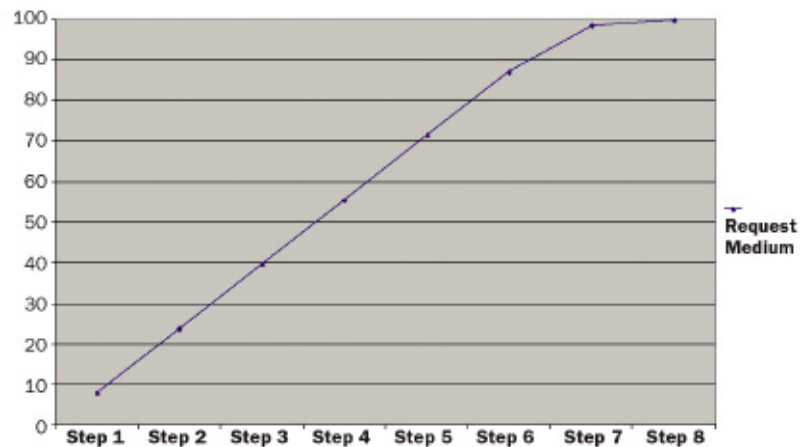
involve just simple observations of the amount of resources used by each system component for the specific workload. For example, workload A creates significant CPU usage on server X while server Y is hardly touched. This means that if you increase workload A, the lack of CPU resources on server X will create a bottleneck. As you run increasingly complex tests, you verify results you get against your “model”—your understanding of how the system behaves. If they don’t match, you need to figure out what’s wrong.

Modeling often is associated with queuing theory and other sophisticated mathematical constructs. While queuing theory is a great mechanism to build sophisticated computer system models, it’s not required in simple cases. Most good performance engineers and analysts build their models subconsciously, without even using such words or any formal efforts. While they don’t describe or document their models in any way, they take note of unusual system behavior—*i.e.*, when system behavior doesn’t match the model—and can make some simple predictions (“It looks like we’ll need X additional resources to handle X users,” for example).

The best way to understand the system is to run independent tests for each business function to generate a workload resource usage profile. The load should not be too light (so resource usage will be steady and won’t be distorted by noise) or too heavy (so it won’t be distorted by nonlinear effects).

Considering the working range of processor usage, linear models often can be used instead of queuing models for the modern multiprocessor machines

(less so for single-processor machines). If there are no bottlenecks, throughput (the number of requests per unit of time), as well as processor usage, should increase proportionally to the workload (for example, the number of users) while

FIG. 3: CPU USAGE

response time should grow insignificant. If we don’t see this, it means there’s a bottleneck somewhere in the system and we need to discover where it is.

For example, let’s look at a simple queuing model I built using TeamQuest’s modeling tool for a specific workload executing on a four-way server. It was simulated at eight different load levels (step 1 through step 8, where step 1 represents a 100-user workload and 200 users are added for each step thereafter, so that step 8 represents 1,500 users). Figures 1 through 3 show throughput, response time and CPU usage from the modeling effort.

An analysis of the queuing model results shows that the linear model

accurately matches the queuing model through step 6, where the system CPU usage is 87 percent. Most IT shops don’t want systems loaded more than 70 percent to 80 percent.

That doesn’t mean we need to discard queuing theory and sophisticated modeling tools; we need them when systems are more complex or where more detailed analysis is required. But in the middle of a short-term performance engineering project, it may be better to build a simple, back-of-the-envelope type of model to see if the system behaves as expected.

Running all scripts simultaneously makes it difficult to build a model. While you still can make some predictions for scaling the overall workload proportionally, it won’t be easy to find out where the problem is if something doesn’t behave as expected. The value of modeling increases drastically when your test environment differs from the production environment. In that case, it’s important to

document how the model projects test results onto the production system.

THE PAYOFF

The ultimate goal of applying agile principles to software performance engineering is to improve efficiency, ensuring better results under tight deadlines and budgets. And indeed, one of the tenets of the “Manifesto for Agile Software Development” (<http://agilemanifesto.org/>) is that responsiveness to change should take priority over following a plan. With this in mind, we can take performance testing of today’s increasingly complex software to new levels that will pay off not just for testers and engineers but for all stakeholders. ☐

Data-Driven Modeling Scenarios

To Get a Clear Picture of How
A System Will Perform, Create
A Realistic Test Framework

By Fiona Charles

Many software test efforts depend on scenarios that represent real sequences of transactions and events. Scenarios are important tools for finding problems

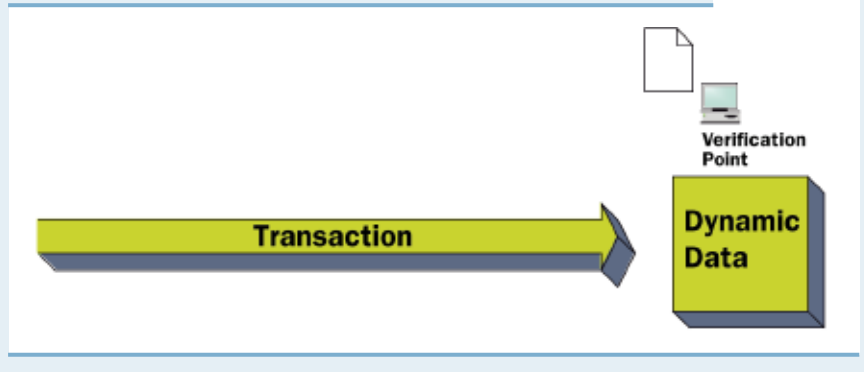
that matter to stakeholders in business applications and integrated solutions, giving us tests of functionality from end to end. Often, scenarios are essential for business acceptance, because they encapsulate test ideas in a format that is meaningful for business users and easy for them to understand and review.

User stories, use cases and other business requirements can be good sources of scenario test ideas. But testers know that these are rarely comprehensive or detailed enough to encompass a thorough test without additional analysis. And if we base our test model entirely on the same sources used by the programmers, our test will reflect the assumptions they made when building the system. There is a risk that we will miss bugs that arise from misinterpreted or incomplete

Fiona Charles (www.quality-intelligence.com) teaches organizations to match their software testing to their business risks and opportunities. With 30 years' experience in software development and integration projects, she has managed and consulted on testing for clients in retail, banking, financial services, health care and telecommunications.

Photograph by Tom O'Grady

FIG. 1: TRANSACTION BASICS



requirements or user stories.

One way to mitigate this risk is to build a scenario model whose foundation is a conceptual framework based on the data flows. We can then build scenarios by doing structured analysis of the data. This method helps to ensure adequate coverage and testing rigor, and provides a cross-check for our other test ideas. Because it employs a structure, it also facilitates building scenarios from reusable components.

Below, I'll share an example of a data-driven scenario test my team and I designed and ran for a client's point-of-sale project. But before we go any further, let's define our terms:

Scenario: The *American Heritage Dictionary* defines "scenario" as "an outline or model of an expected or supposed sequence of events." In "An Introduction to Scenario Testing," Cem Kaner extends that definition to testing: "A scenario is a hypothetical story, used to help a person think through a complex problem or system....A scenario test is a test based on a scenario." (<http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf>)

Test model: Every software test is based on a model of some kind, primarily because we can never test everything. We always make choices about what to include in a test and what to leave out. Like a model of anything—an airplane, a housing development—a test model is a simplified reduction of the system or solution we are testing. As software pioneer Jerry Weinberg explains in *An Introduction to General Systems Thinking*, "Every model is ultimately the expression of one thing...we hope to understand in terms of another that we do understand."

The test model we employ embod-

ies our strategic choices, and then serves as a conceptual construct within which we make tactical choices. This is true whether or not we are aware that we are employing models. Every test approach is a model—even if we take the "don't think about test design, just bang out test cases to match the use cases" approach. But if we're not consciously modeling, we're probably not doing it very well.

In these definitions, we see the inextricable connection between scenarios and models. For our purposes, then,

*The test model
we employ embodies our
strategic choices.*

a **test model** is any reduction, mapping or symbolic representation of a system, or integrated group of systems, for the purpose of defining or structuring a test approach. A **scenario** is one kind of **model**.

We don't have to use scenarios to model a software test, but we do have to model to use scenarios.

DESIGNING A MODEL FOR A SCENARIO-BASED TEST

It is useful to model at two levels when we are testing with scenarios: The **overall execution model** for testing the solu-

tion and the **scenarios** that encapsulate the tests.

There are many different ways to model both levels. The retail point-of-sale (POS) system example discussed below uses a hybrid model based on **business operations** combined with **system data** as the basis for the overall test execution model, and **system data** for the scenario model.

Business operations—e.g., day (week, month, mock-year) in the life or model office or business. A business operations model is the easiest to communicate to all the stakeholders, who will immediately understand what we are trying to achieve and why. Handily, it is also the most obvious to construct. But it is not always the most rigorous, and we may miss testing something important if we do not also look at other kinds or levels of models.

System data—categorized within each flow by how it is used in the solution and by how frequently we intend to change it during testing: static, semistatic or dynamic.

Combining different model bases in one test helps testers avoid the kinds of unconscious constraints or biases that can arise when we look at a system in only one way. We could also have based our models on such foundations as:

The **entity life cycle** of entities important to the system—e.g., product life cycle (or account, in a banking system) or customer experience.

Personae defined by testers for various people who have some interaction with the system, or who depend on it for data to make business decisions. These could include a merchandising clerk, store manager, sales associate or a customer and category manager. (See books by Alan Cooper for an explanation of persona design.)

High-level **functional decomposition** of the system or organization, to ensure that we have included all major areas, including those that may not have changed but could be impacted by changes elsewhere.

In a point-of-sale application, this might include functional areas within the system or business (e.g., ordering, inventory management, billing), processes in each area (e.g., order capture, provisioning) and functions within each process (e.g., enter, edit, cancel order).

Already defined **user stories** or **use cases**.

Stories created by testers, using their imaginations and business domain knowledge, including stories based on particular metaphors, such as “soap opera tests” (described by Hans Buwalda, in “Soap Opera Testing,” *Better Software*, February 2004).

Although none of these was central in our test model, we used each to some degree in our test modeling for the POS system (except for predefined user stories and use cases, which didn’t exist for this system). Using a range of modeling techniques stimulated our thinking and helped us avoid becoming blinkered by our test model.

MODELING BASED ON DATA

We constructed our primary central model using a conceptual framework based on the system data. This is a good fit for modeling the scenario test of a transactional system.

A data-driven model focuses on the data and data interfaces:

- Inputs, outputs and reference data
- Points of entry to the system or integrated solution
- Frequency of changes to data
- Who or what initiates or changes data (actors)
- Data variations (including actors)

A model based on data represents a systems view that can then be rounded out using models based on business views. As well as providing a base for scenarios, focusing on the data helps us identify the principal components we will need for the overall execution model for the test:

- Setup data
- Entry points
- Verification points
- Events to schedule

It is also easy to structure and analyze.

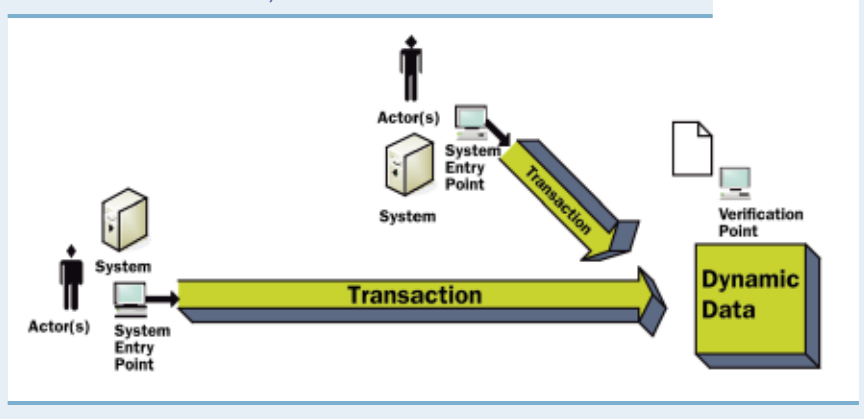
CONCEPTUAL FRAMEWORK FOR A DATA-DRIVEN MODEL

Here is an example of a conceptual framework.

At the most basic level, we want scenarios to test the outcome of system transactions.

Transactions drive dynamic data—i.e., data we expect to change in the course of system operation (see Figure 1). Transactions represent the principal business functions the system is designed to support. Some examples

FIG. 2: ADD ACTORS, MORE TRANSACTIONS



for a POS system are sales, returns, frequent shopper point redemptions and customer service adjustments.

A transaction is initiated by an **actor**, which could be human or the system (see Figure 2).

There may be more than one actor involved in a transaction—e.g., a sales associate and a customer

Many scenarios will have **multiple transactions** (see Figure 2).

Subsequent transactions can affect the outcome by acting on the dynamic data created by earlier related transactions.

Transactions operate in a context partly determined by **test bed data** (see Figure 3).

Reference test bed data is static (it doesn’t change in the course of the test)—e.g., user privilege profiles, frequent shopper (loyalty) points awarded per dollar spent, sales tax rates. Deciding which data should be static is a test strategy decision.

The test bed also contains **semistatic data**, which can change the context and affect the outcomes of transactions.

Semistatic data changes occasionally during testing, as the result of an event. Examples of semistatic data include items currently for sale, prices and promotions.

Determining which data will be semistatic and how frequently it will change is also a test strategy decision.

Events affect transaction outcomes—by changing the system or test bed data context, or by acting on the dynamic data. (see Figure 4).

Events can represent:

- periodic or occasional **business processes**, such as rate changes, price changes, weekly promotions (deals) and month-end aggregations
- **system happenings**, such as system interface failures
- “external” **business exceptions**, such as shipments arriving damaged or trucks getting lost

Scenarios also operate within a context of **date and time**. The date and time a transaction or event occurs can have a significant impact on a scenario

FIG. 3: FACTORING IN TEST BED DATA

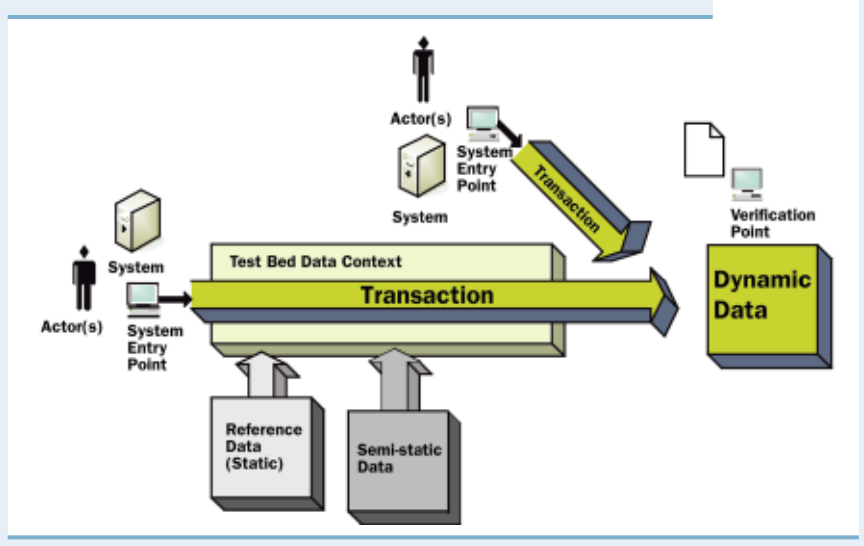
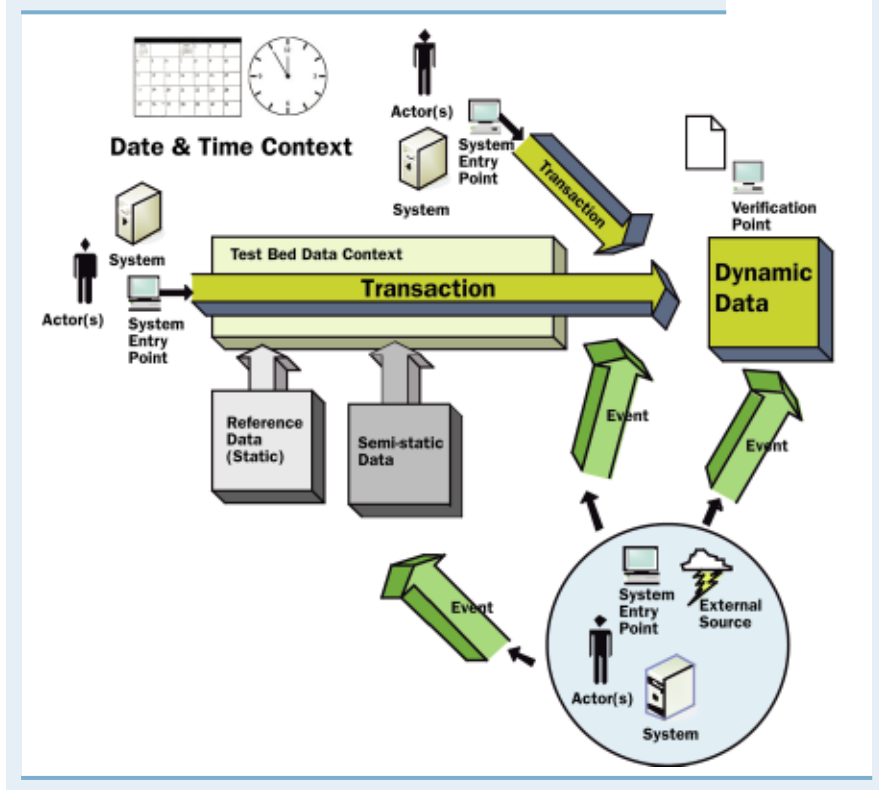


FIG. 4: ADD EVENTS, PLUS DATE & TIME



outcome (see Figure 4).

Categorizing the data determines each type's role in the test and gives us a conceptual framework for the scenario model:

- Reference data sets the context for scenarios and their component transactions.
- A scenario begins with an event or a transaction.
- Transactions have expected results.
- Events operate on transactions and affect their results: A prior event changes a transaction context (e.g., an overnight price change), and a following event changes the scenario result and potentially affects a transaction (e.g., product not found in warehouse).
- Actors influence expected results (e.g., through user privileges, or customer discount or tax status).

We can apply this framework to the design of both levels of test model: the overall execution model (the central idea or metaphor for the test approach) and the scenario model.

TEST DESIGN FOR A POS SYSTEM

The example that follows shows how, together with a small test team, I applied the conceptual framework described above to testing a POS sys-

tem on a client project. David Wright, a senior tester with whom I have worked on many systems integration test projects, and who has contributed many

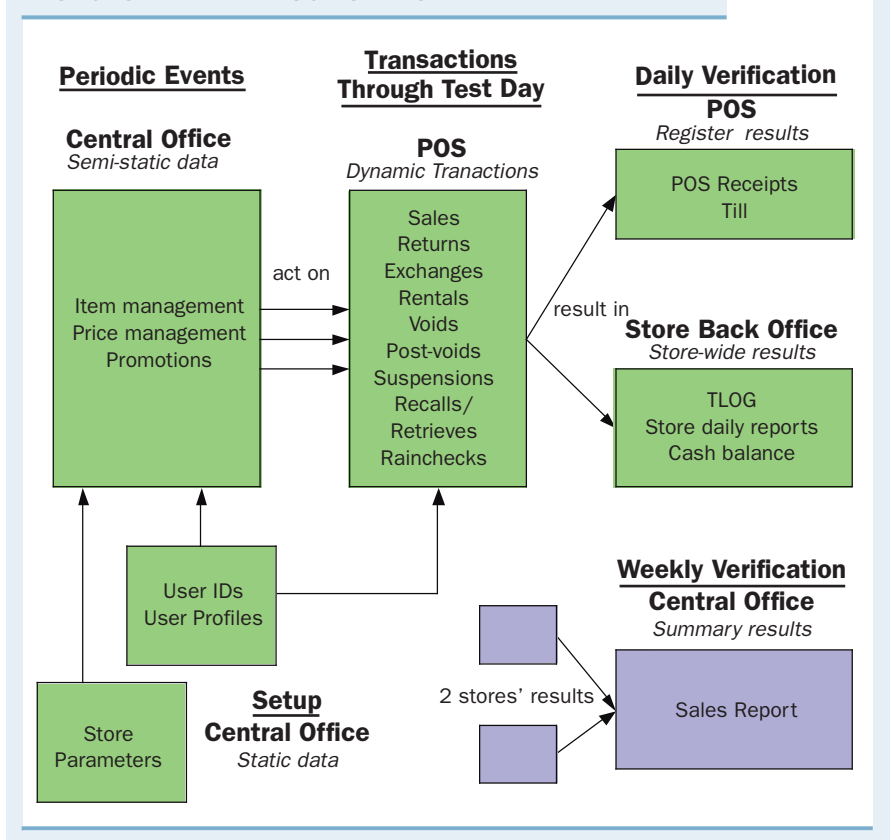
ideas and practical details to development of this method, was a core member of my team. In addition to experience with end-to-end scenario testing, David and I both have extensive retail system domain knowledge, which was essential to successful testing on this project.

THE BACKSTORY

The client operates a chain of more than 1,000 retail drugstores across Canada. This project was for the company's secondary line of business, a home health care company, which sells and rents health care aids and equipment to retail customers through 51 stores in different regions of Canada. The plan was to implement the POS system standalone for the home health care company, and then, following successful implementation, to implement it in the main drugstores, fully integrated with an extensive suite of store and corporate systems.

The POS system is a standard product, in production with several major retailers internationally, but it was being heavily customized by the vendor for this client. The client had contracted with an integrator to manage the implementation and mitigate

FIG. 5: OVERALL EXECUTION MODEL



its risks. I reported to the integrator with my test team.

The vendor was contractually obligated to do system testing. The integrator had a strictly limited testing budget, and there were several other constraints on my test team's ability to develop sufficient detailed knowledge of POS to perform an in-depth system test within the project's timelines. I therefore decided that our strategy should be to perform an end-to-end acceptance test, focusing on financial integrity (*i.e.*, consistency and accuracy end-to-end). Because we expected the client would eventually proceed with the follow-on major integration project in the drugstores, I built the test strategy and the artifact structure to apply to both projects, although an estimated 40 percent of the detailed scenario design would need to be redone for the primary line of business.

EXECUTION MODEL

Having analyzed and categorized the data for the POS system, I designed this overall model for executing the test (see Figure 5).

The POS system had four principal modules (Item and Price Management, Promotions Management, Store Operations, and Central Reporting), operating in three location types. The POS client would operate in each store and Store Back Office. The Central Office modules would be the same for all stores and all would offer the same items, but actual prices and promotions would differ by region and be fed overnight to each store. We decided to run two test stores, covering two regions in different time zones and with different pricing and provincial tax structures. One of our test stores could be switched to a different region if we identified problems requiring more in-depth testing of location variations.

We combined a business operations view with our data-driven overall model to create a test cycle model (see Figure 6).

SCENARIO DESIGN

We decided to build our scenarios from an element we defined as an item transaction. The team began by drilling down into each framework element and defining it from a test point of view: *i.e.*, important attributes for testing and the possible variations for each. Item transactions had some other elements listed

(actors, items) with them, giving us a list that looked like this:

Transaction

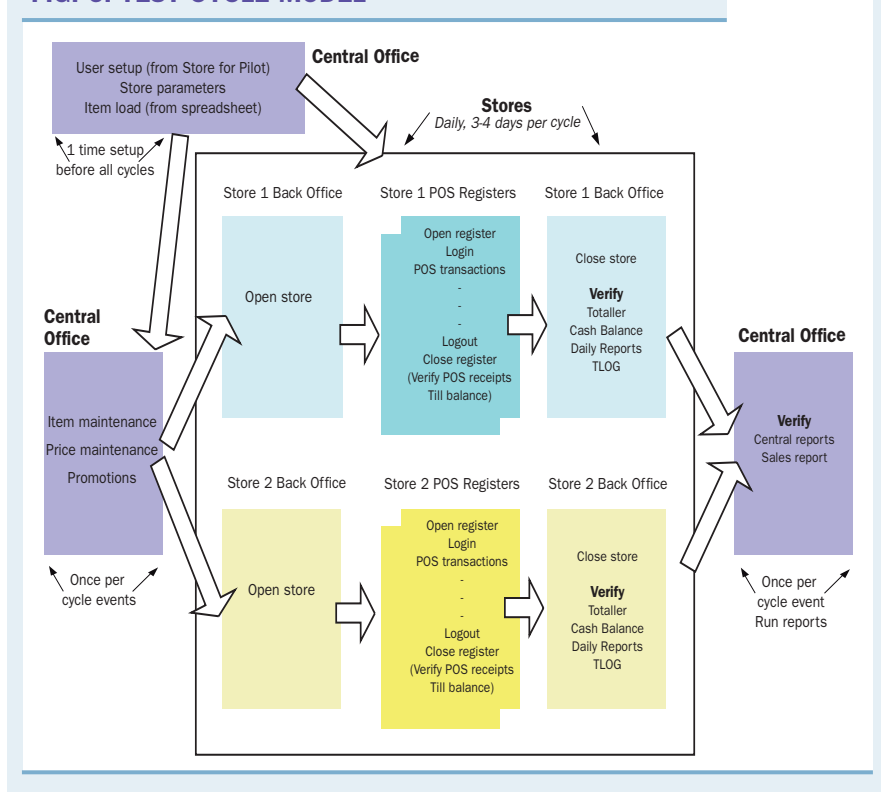
- Type (sale, return, post-void, rental, rental-purchase...)
- Timeframe (sale/rental + 3 days...)
- Completion status (completed, void, suspended)
- Store
- Register
- User (cashier, manager, supervisor, store super-user)
- Customer (walk-in, preferred, loy-

We designed events in a similar fashion, including price changes and promotions. We also constructed an item table with all their attributes and variations.

We then designed scenarios, using the item transactions we had defined as the lowest-level building blocks. This allowed us to combine item transactions and also to use them at different points in scenarios. (Figure 7 shows our scenario model.)

The scenario spreadsheets refer-

FIG. 6: TEST CYCLE MODEL



- alty, status native, employee...)
- Item(s) (multiple attributes, each with its own variations)
- Tender (cash, check, credit card, debit, coupon, loyalty redemption...)
- Loyalty points (y/n)
- Delivery/pickup (cash and carry, home delivery...)
- Promotions in effect (weekly flyer, store clearance...)
- Other discount (damage...)

Using spreadsheets to structure our analysis, we designed a full range of item transactions, working through the variations and combining them to make core test cases. Wherever possible, we built in shortcuts—*e.g.*, Excel lists for variations of most attributes. (Table 1 provides a simplified example of the transaction definitions.)

enced the item transaction and item sheets, so we could make changes easily. Our spreadsheet structures also allowed us to calculate expected results for each test day and test cycle. This was essential for a system with so many date dependencies, where we had to evaluate the cumulative results of each store sales day daily, and at the end of each cycle evaluate the corporate week for all stores. (Table 2 provides a simplified version of the scenario design.)

In addition, we wrote Excel macros to automate the creation of some test artifacts from others. The testers' worksheets for test execution, for example, were generated automatically.

RESULTS OF THE POS TEST

The vendor POS system, and in particular

TABLE 1: TRANSACTION DEFINITIONS (SIMPLIFIED)

Txn-ID	Store	Type	Timeframe	Completion	Register	User Profile	Customer	Item(s)	Tender	Points	Pickup/Deliver	Promos	Discounts
POS-1	1	sale	n/a	C	4	cashier	walk-in	45270	Cash-CAD	n	y	n/a	n/a
POS-2	1	return	sale+1	C	2	manager	walk-in	45270	Cash-CAD		C&C	n/a	n/a
POS-3	2	sale	n/a	c	1	cashier	employee	98651	Visa	y	HD	sidewalk	raincheck
								54945					
								21498					

the customizations, turned out to be very poor quality. As a result, a test that had been planned to fit our budget, with four cycles (including a regression test) over four weeks, actually ran through 28 cycles over 14 weeks—and then continued after pilot implementation. My strategy, and our test scenarios, proved effective. We logged 478 bugs, all but 20 of which our client considered important enough to insist on having fixed (see Table 3).

It is important to ask whether this was the most cost-effective way to find those bugs. Probably it was not, because many of them should have been found and fixed by the vendor before we got the system. But we found many other bugs, primarily integration issues, which would probably not have been found by any other kind of test. And—given the total picture of the project—it was critical that the integrator conduct an independent integrated test. This was the most efficient way for our team to do that within the given constraints.

There were several business benefits from our test and test strategy. The most important was buy-in from the client's finance department. Because our test verified financial integrity across the system, it provided the finance people with evidence of the state of the solution from their point of view.

Our scenario test facilitated acceptance for implementation in the home health care stores and provided information on which to base the decision of whether or not to proceed with integration and implementation in the 1,000-plus drugstores.

Because we tested the end-to-end function of complex data interactions that are business norms for this client—such as sales, returns and rentals of items with date-dependent prices and promotions, and returns when prices have changed and new promotions are in effect—we were able to provide the business with an end-to-end view of system function that crossed modules and business departments. Our varied scenarios provided sufficiently realistic data to verify critical business reports.

Finally, the spreadsheet artifacts we used throughout the test and gave to the client supplied solid evidence of testing performed, in the event it should ever be required for audits.

BIGGEST POS PROJECT BENEFITS

The principal benefit of my strategy from a testing point of view was that my team supplemented the vendor's testing rather than attempting to duplicate it. The vendor's testing did not include an integrated view of POS function. Ours did, and that allowed us to focus primarily on system

outcomes, and only secondarily on the users' immediate experiences.

By adopting a building-block approach, we incorporated efficiency in test preparation and execution. This gave us flexibility to reschedule our testing according to the state of the system on any given day. When we encountered bugs, we could work with the vendor and drill down into the components of a scenario, such as item setup, item transaction and promotion setup, to find the problem.

Our robust and structured transaction-scenario artifacts provided the client with a reusable regression test for future releases, upgrades to infrastructure and so on. We were able to layer operability tests on top of our scenario testing, simulating real outage conditions and verifying the outcomes.

WHEN TO CONSIDER SCENARIOS

Scenario testing as I have described it here is not always the best test method. It does not fit well with exploratory testing, and it is not the most effective way to test deep function. It is, however, a very useful method for testing widely across a system. It is better for testing system outcomes than for evaluating a user's immediate experience.

Scenario testing should, therefore, be considered as part of an overall test strategy that includes different kinds of

TABLE 2: SCENARIO DESIGN (SIMPLIFIED)

Scenario ID	Description	Prior Txns	PRIOR EVENTS			Main Txn	Follow-on Txns	Post-Txn Events
			Central Office	Store Back Office	Store			
POS-S-S1	Senior phones in to buy 3 items on Super Senior's Day of which 1 is a charge sale item and 1 is a govt funded item, delivery and service charges applied, pre-paid delivery	n/a	PROMO-45 PROMO-7	Open store	Open Register Login	POS-S133		Logout Close Register Close Store RPT-16
POS-S-S19	Loyalty customer pays for 5 items, of which 4 are points-eligible; customer changes mind before leaving register and 1 point-eligible and 1 non-eligible are post-voided	n/a	n/a	Open store	Open Register Login	POS-S17	POS-PV17	
POS-S-S65	Senior phones in to buy 3 items on Super Senior's Day of which 1 is a charge sale item and 1 is a govt funded item, delivery and service charges applied, pre-paid delivery	n/a	PROMO-12 PROMO-7 ITEM-96	Open store	Open Register Login	POS-S133	n/a	

tests. Some situations where it could be appropriate include:

- Acceptance tests of business systems—e.g., UAT or vendor acceptance.
- End-to-end systems integration tests of multisystem solutions or enterprise integrated systems.
- Situations where a test team lacks sufficient detailed system knowledge to do under-the-covers or deep function testing and has no way to get it, and there is insufficient time to explore. When there is a combination of inadequate documentation, restricted or zero access to people who wrote the software, and critical time pressures, scenario testing can be the best solution to the testing problem. (All of these constraints applied on the POS project, and we overcame them with scenario tests informed by business domain knowledge.)

CRITICAL SUCCESS FACTORS

The single most important requirement for designing good scenario tests is business domain knowledge, in the form of one or both of the following:

- Testers who have experience in

the domain

- Input from, and reviews of scenarios by, business representatives

Where neither of these is available, it is at least possible to resort to industry books, as Cem Kaner suggests in his article.

To use the approach I've described here, you need:

- A model with a framework that fits the type of application. A data-driven model works well for transactional systems.

For other types of applications—e.g., a desktop publishing system—you would need to create a different model and framework, such as one based on usage patterns.

- Testers skilled in structured analysis. This is not the no-brainer you would think. Not all testers—not even all *good* testers—have this skill.
- A building-block approach, so you can design and build varied and complex tests from simple elements. Among other advantages,

TABLE 3: BUGS

Severity	Count	%
1	8	2%
2	331	68%
3	116	25%
4	23	5%
Total	478	

this allows you to begin testing with the simplest conditions before adding complexity to your scenarios.

It also makes it possible to automate some of the test artifacts and build in calculated expected results. This becomes essential in large-scale systems integration tests, where you have to communicate accurate expected results to multiple teams downstream in the integration.

THE RISKS

If you adopt scenario testing, these are the critical things to watch out for:

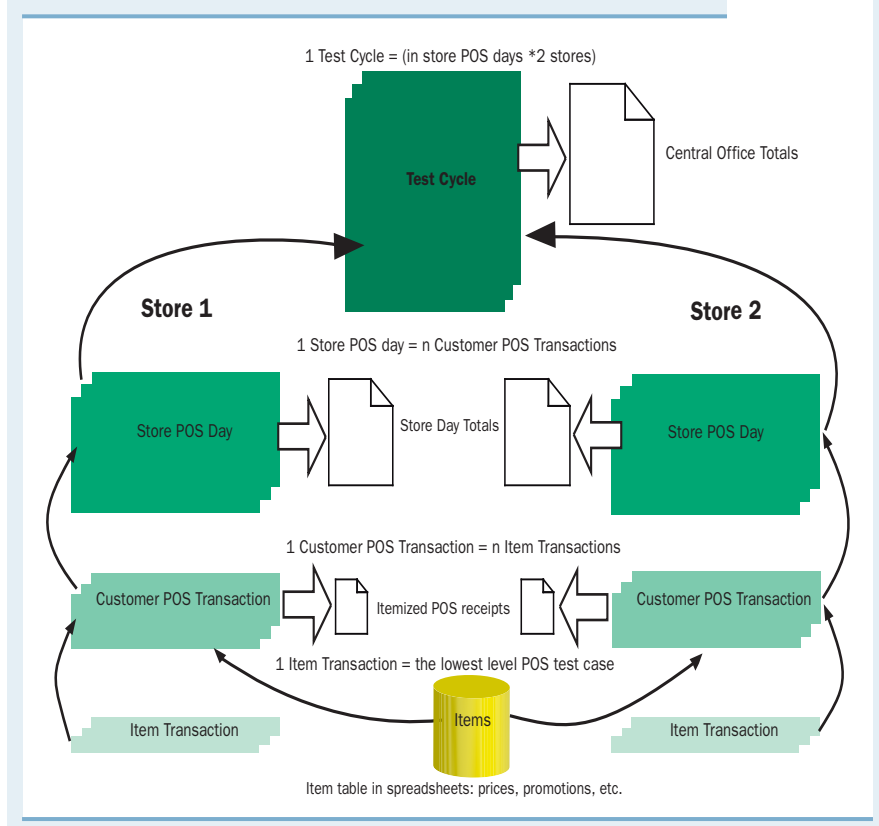
- Scenario testing can miss important bugs you could find with under-the-covers or deep function testing. Remember that scenario testing is better for testing end-to-end than it is for testing deeply, and build your strategy accordingly.
- Bugs found in scenario outcomes can be difficult to diagnose. Especially if system quality is suspect, it is essential to begin with simple scenarios that test basic end-to-end function, only proceeding to complex scenarios when you have established that fundamental quality is present.
- In choosing a model, there is always a risk of fixing on one that is too restrictive. Applying two or more model types will help prevent this.
- Equally, there is a risk of choosing a model that is too expensive (and expensive).

FINAL ANALYSIS

Finally, never fall in love with one model type. This applies to your models for a single test as much as it applies to your model choices for different tests.

Every test is different, and every model type can bring benefits that others lack. ☒

FIG. 7: SCENARIO MODEL



Six Sigma, Part II: Time to Measure

SIX SIGMA

I. Define

II. Measure

III. Analyze

IV. Improve

V. Control



By Jon Quigley and Kim Pries

In the first article in this five-part series (see “Tracking Quality Variations With Six Sigma Methods” at stpcollaborative.com), we discussed the “Define” phase of the Six Sigma quality management process as it relates to software testing and performance. We explained how clearly defining the problem and explicitly describing what needs to be accomplished allows us to set the scope for project activities so we’ll

know how and when we’ve hit the target—that is, how we’ve made the correction appropriately.

Here, we’ll assume we’ve already defined a problem issue, whether code-specific or process-specific (process-specific issues are often related to software productivity and failure to meet

deadlines or budget targets), and we’ll delve into the second piece of the Six Sigma puzzle: “Measure.” (In upcoming issues, we’ll address the next three aspects: “Analyze,” “Improve” and “Control.”)

Before doing any measuring, you must:

- Identify key quality attributes to be measured
- Devise a measurement plan
- Understand the limitations of the measurement system

We’ll break our discussion into two compo-

Jon Quigley is the manager of the verification and test group at Volvo Truck. Kim Pries is director of product integrity and reliability for Stoneridge, which designs and manufactures electronic automotive components.

nents: measuring the software itself and measuring the software process.

MEASURING THE SOFTWARE

Some of the common metrics used with software are:

- Lines-of-code
- Function points
- Defect counts
- Defect types
- Complexity

Lines-of-code is an objective measure of software size. It's important for developers to understand that higher-level languages like C++ don't translate one-for-one into the executable code the machine will use. This is also the case when we use dynamic languages such as Perl, Python and Ruby. Java is converted into byte-code, which, in turn, is converted into executable code. For these reasons, lines-of-code can be a problematic metric. On the positive side, though, if all development work is being done in one language, the metric is simple to generate and provides a basic level of measurement.

Function points are an alternative to lines-of-code. When using the function point approach, we work with the software specification and attempt to estimate the complexity of the product using a special set of rules. Function points present problems because the estimation process itself is highly subjective. Sometimes an organization will use some well-known numbers and "backfire" the function point value from the lines-of-code. So why not use the lines-of-code in the first place? We might use function points when we're comparing the potential complexity of a product to some standard and we have no existing code.

Defect counts are easy to accumulate. They comprise categorical data, which Six Sigma practitioners have to use to make any meaningful statistical analysis. Luckily, this kind of analysis doesn't have to be that difficult: We can use a **Pareto chart** to display defect counts by function. (See Figure 1 for an example.)

Using our count data and the Pareto chart—which is organized by count from left to right, high to low—we can easily see that the "read_analog" function is a leading source of software defects. By measuring our work using this approach, we can focus our attention on functions that show frequent problems. This

doesn't mean that we'll ignore the other functions, just that we know we have serious issues with the functions on the left side of the graph.

Using **defect types** as a metric allows us to check our software to see if we're repeating certain kinds of defects regularly and may provide some direction to our testing group during the "Analyze" phase of our Six Sigma project. A few examples of defect types are:

- Boundary issues
- Byte boundaries (with embedded code)
- Array/matrix out-of-bounds issues (with non-embedded code)
- Logic issues
- Odd expressions, especially with "negative" tests (those using "not")
- Unusual case structures
- Syntax checking
- Functioning of compiler
- Data issues
- Illegal inputs
- Incorrect casting to convert from one data type to another

One benefit of measuring the quantity of each data type is that we can again apply Pareto analysis to the results and use the information during our Six Sigma "Analyze" phase. Also, we can either construct a taxonomy of types or we can use a technical standard such as IEEE 1044-1993. A taxonomy will represent the types in a hierarchical tree format.

Software complexity metrics attempt to represent the intrinsic complexity of the software by making calculations. Probably the most well-known approach is the cyclomatic complexity calculation of Thomas McCabe. McCabe initially

used a graph theoretical approach—a flow graph—to estimate complexity. The formula for such an approach is:

Cyclomatic Complexity = $E - N + P$, where

E = the count of edges of the graph

N = the count of nodes of the graph

P = the count of connected components

Luckily, we don't have to develop the graphs or calculate the formula by ourselves. We can use tools such as JavaNCSS for Java or Saikuro for Ruby. These tools will go through the code and make the calculations based on the number of "if" and "case" statements, as well as other flow-altering code, and present the developer with the value. The cyclomatic complexity is a completely objective value. While some people may question the meaning of cyclomatic complexity, the value generated is, at a minimum, suggestive of areas of code that may merit closer attention.

MEASURING THE SOFTWARE PROCESS

We can also use our measurement tools to take a look at the development process itself. For example:

- Project status
- Quality (addressed in the first section of this article)
- Delivery
- Compliance with international standards
- Coding standards
- Factor/response analysis

When looking at a portfolio of software projects and assessing them over time, we can put the **status** of the software projects into the following categories:

- Ahead of budget/schedule

FIG. 1: DEFECT COUNT BY FUNCTION

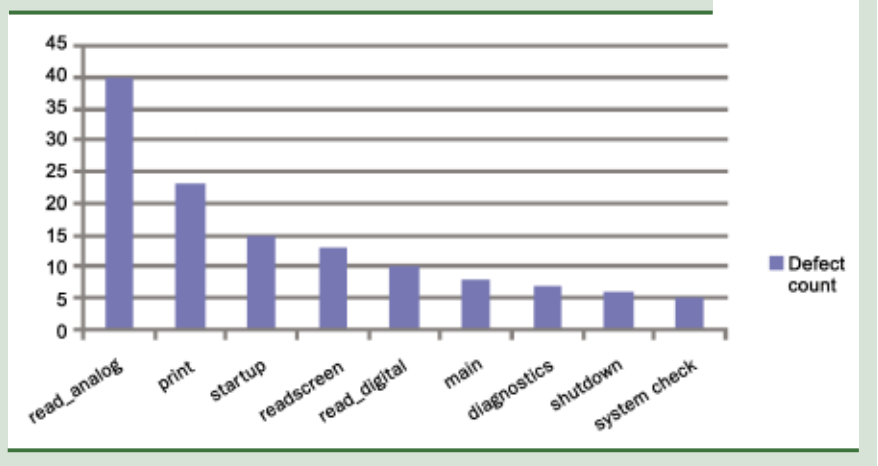
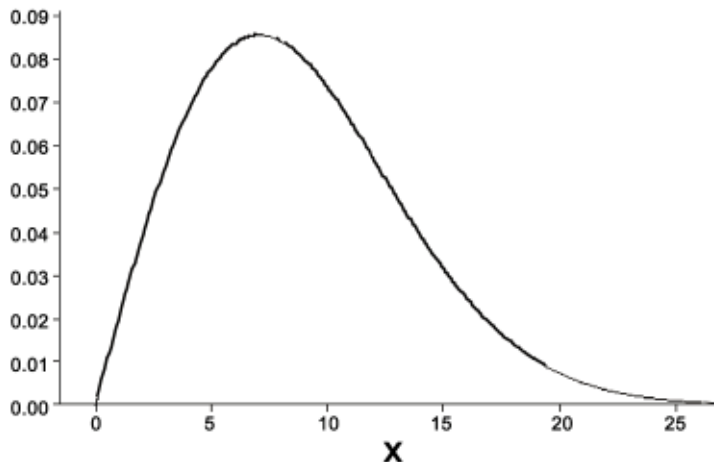


FIG. 2: FRONTLOADED EFFORT



- On budget/schedule
- Behind budget/schedule
- Suspended (may resume)
- Stopped (candidate for aborted)
- Unstarted
- Completed
- Unknown
- Aborted (completely dead)
- Zombie (similar to unknown and still consuming resources)

Some of these—"unknown," for instance—represent serious problems with the process. Basically, the simplest approach would be to build a table that lists the development projects in the first column and the status of each project in the second.

Project **budgets** are an obvious metric for the development process and fit into a few of the categories we just discussed. The highest cost frequently comes from the payroll for the developers. During a development we might wish to estimate our level of effort so we can put a value on the software engineering we're proposing to do. One way to do this would be to model our efforts using a Rayleigh distribution.

Figure 2 shows how we can measure our actual effort (e.g., hours of labor) against the probability distribution function. Front-loading is desirable in order to avoid the "student" effect (named for the tendency of students to put in all their effort the night before a test).

Another interesting feature of this Rayleigh distribution is that we can also measure the *arrival* of defects through the duration of the project and compare our results against the model.

Usually real data is noisy (some elements of randomness), but we can still use this approach to potentially provide a basis for releasing the software product as our defect arrival rate slows down and we move into the right tail of the distribution.

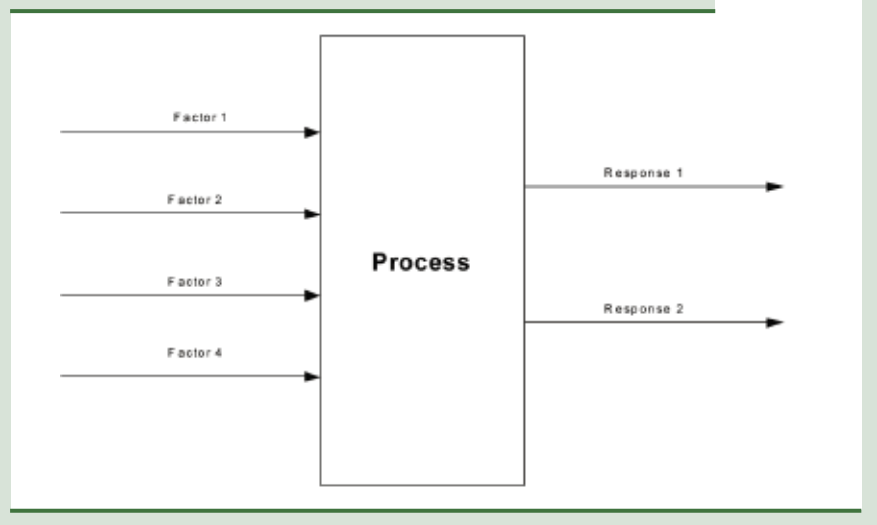
During our software process, we want to measure delivery of functions, modules and other configuration items to see if we're meeting our schedules. Instead of using the states we defined previously, we measure our historical progress against development time lines. Tools such as Microsoft Project allow the project manager to baseline the initial plan and then produce a secondary plot of the actual dates superimposed on the first timeline. We can assess the distribution type, the means, the variance and any other statistical values that would allow us to

model the software development project. For example, using a tool like @Risk from Palisade Corp. in concert with Microsoft Project, we can run Monte Carlo simulations of the project. The Monte Carlo approach generates random values from a given probability distribution; that is, we're using the probability distribution we measured to generate a model for each delivery. In so doing, we can provide our software project managers with a method for simulating the delivery times of an entire project or portions of a project.

Another method of process improvement is to drive the development team to compliance with an international standard. One such standard is ISO/IEC 15504 in nine parts. ISO/IEC 15504 is also known as the Software Process Improvement and Capability dEtermination (SPICE), which is a variation on the maturity models developed by the Software Engineering Institute in the 1980s—the Capability Maturity Model (CMM) and the Capability Maturity Model Integration (CMMI). These approaches specify a stepwise approach to increasing the capability of software organizations. All of these organizations define best practices and publish documents for use by auditors, but no software organization has to wait for a formal audit to establish the practices contained in these standards.

We can also measure compliance with **coding standards** such as MISRA C. MISRA, the Motor Industry Software Reliability Association, is one group that specified best practices in the C language

FIG. 3: CORRELATING INPUTS AND OUTPUTS



for embedded systems in the automotive industry. An organization doesn't have to purchase a coding standard—with enough product history, it can develop its own. The standard developers can refer to the software metrics, especially the taxonomy of failure types and the typical functions that become "problem children."

One of the most valued tools in the Six Sigma toolbox is the IPO diagram. (See Figure 3.)

This diagram offers a model for measuring and correlating factors (inputs) against their effects (outputs). For example, we might use this approach to discern which factors are providing our developers with fewer defects: They could be using static analyzers, dynamic analyzers, debuggers, oscilloscopes, logic analyzers and other tools. We'd then conduct experiments to provide ourselves with a meaningful basis for recommending one tool or family of tools over another. Some readers may recognize this approach as the Design of Experiments, or DoE. The downside to this approach is that we usually need special software to perform the analysis.

Some choices of support software are:

- Minitab
- Design Expert
- DOE++
- Nutek-4

In many cases, Pareto charts and other basic tools are enough to provide direction for future analysis and improvement.

For something as complex as regression analysis, we can use a free statistical language called "R" to perform our calculations. R is command-line driven, although a graphical user interface called "Rcmdr" helps ease the

shock of using an old-fashioned command line. R even has an interface to Microsoft Excel that allows R to be used as a server to Excel as a client. Designed experiments and regression are powerful tools for analyzing processes.

WHAT'S NEXT?

We can apply the Six Sigma "Measure" approach to software directly, and we can also use it for the software development process. When Six Sigma tools are applied to a process, we usually call it transactional Six Sigma because we're dealing with people. From "Measure," we'll move to "Analyze" in our next article and show you what we do with the factors we've already measured.

Don't get frustrated, thinking you have to use statistics as part of the project. In many cases, using Pareto charts and other basic tools is more than enough to provide direction for future analysis and improvement. And sometimes, simply measuring a previously unknown factor reveals the solution to your problem. ☒

Statement of Ownership, Management, and Circulation for Software Test & Performance as required by PS Form 3526-R. Publication title: Software Test & Performance. Publication number: 023-771. Filing date: September 25, 2009. Issue frequency: Monthly. Number of issues published annually: 12. Annual subscription price: \$69.00. Complete mailing address of known office of publication: 105 Maxess Road, Suite 207, Melville, NY 11747. Publisher: Andrew Muns; Editor/Managing Editor: Edward J. Correia; Owner: Redwood Collaborative Media, Inc., all located at 105 Maxess Road, Melville, NY 11747. Shareholders holding 1% or more of the total amount of stock: Ronald Muns, 1640 Little Raven, Denver, CO 80202; Robert Andrew Muns, 20 McKesson Hill Rd, Chappaqua, NY 10514; Katherine Muns, 1106 E. Moyamensing, Philadelphia, PA 19147; Pamela Kay Garrett, 3358 Turnberry Circle, Charlottesville, VA 22911; Erik Leslie, 2485 S. Williams St, Denver, CO 80210; Cole Leslie, 16420 Little Raven, Denver, CO 80202. The tax status has not changed during preceding 12 months. Issue date for circulation data below: October 2009. Extent and nature of circulation: Requested US distribution. The average number of copies of each issue published during the 12 months preceding the filing date include: total number of copies (net press run): 12,267; paid and/or requested outside-county mail subscriptions: 11,496; in-county paid/requested subscriptions: 0; sales through dealers, carriers, and other paid or requested distribution outside USPS: 0; requested copies distributed by other mail classes through the USPS: 0; total paid and/or requested circulation: 11,496; non-requested copies by mail outside-county: 413; in-county non-requested copies by mail: 0; non-requested copies distributed by other mail classes through the USPS: 0; total non-requested distribution: 413; total distribution: 11,909; copies not distributed: 358; for a total of 12,267 copies. The percent of paid and/or requested circulation is 93.7%. The actual number of copies of the October, 2008 issue include: total number of copies (net press run): 5,148; paid and/or requested outside-county mail subscriptions: 4,408; in-county paid/requested subscriptions: 0; sales through dealers, carriers, and other paid or requested distribution outside USPS: 0; requested copies distributed by other mail classes through the USPS: 0; total paid and/or requested circulation: 4,408; non-requested copies by mail outside-county: 326; in-county non-requested copies by mail: 0; non-requested copies distributed by other mail classes through the USPS: 0; total non-requested distribution: 326; total distribution: 4,734; copies not distributed: 414; for a total of 5,148 copies. The percent of paid and/or requested circulation is 85.6%. Publication of the Statement of Ownership for a requestor publication is required and will be printed in the November/December 2009 issue of this publication. I certify that all information furnished on this form is true and complete. Signed: Andrew Muns, President and CEO, September 21, 2009.

Index to Advertisers

Software Test
& Performance
MAGAZINE

Advertiser	URL	Page
Automated QA	www.MissionOffMercury.com	10
Browsermob	www.browsermob.com/stp	19
Checkpoint	www.checkpointtech.com	39
Electric Cloud	www.electric-cloud.com	13
Hewlett-Packard	www.hp.com/go/alm	40
Ranorex	www.ranorex.com	7
STP Collaborative	www.stpcollaborative.com	2
Seapine	www.seapine.com/stpswift	3



Post-Build Injection Pros

► THIS SHOULD BE ENGRAVED ON a plaque in every development lab: Modeling application usage in the wild is no longer a black art.

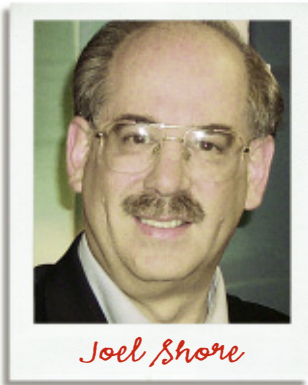
That's certainly good to know, but why was application usage profiling so labyrinthine in the first place? It's self-evident, says PreEmptive Solutions' Sebastian Holst. "A runtime application must serve many masters with markedly diverse profiling requirements, priorities and expectations."

Development organizations view things from a softwarecentric perspective while IT organizations, of necessity, adopt an operational worldview, according to Holst. As a result, developers' priorities are more granular than management's, and are organized differently from those of their QA peers. "IT auditors have a different charter than the operations managers and security officers with whom they collaborate," Holst says. And, he adds, let's not forget the end user and line-of-business owner who ultimately fund this diverse community of application stakeholders.

Profiling and testing must serve the varied and often mutually exclusive interests of these stakeholders. Performance, accuracy, ease of use, security, bandwidth efficiency and a dozen other features and functions are critical, but only to specific stakeholders in specific ways.

The mix changes depending on the application and the business mission. Holst cites four examples: An automated teller machine manufacturer wants to measure implementation efficiency, monitor usage for refactoring Java and ensure end-of-life policy enforcement; a trans-

portation services ISV strives to improve support response efficiency for its end-user audience of nontech-savvy trucking industry professionals; an ISV aims to automate its customer-experience improvement process; and a manufacturer of hearing aids plans to improve the process its dealers use to fit and tune devices to users' ears.



Joel Shore

Regardless of the metrics development and test teams track for these diverse markets and goals, a common core of questions remains constant, says Holst: What's running, how well does it perform, how does it compare (with a prior version or as a hosted vs. nonhosted model), how does it affect business outcome or ROI, and can it be managed from cradle to grave?

How much more effective and efficient could testing be if you had reliable and accurate runtime intelligence from the field detailing which features were being used, by whom, where, when and how? It's not as far away as it may seem: Post-build injection is emerging as a de facto .NET standard and is or will be available for both .NET and Java. Support for Azure and Silverlight is provided.

Application runtime intelligence is emerging as a great way to measure how enterprises not only use but experience software, Holst says. PreEmptive's Runtime Intelligence and Instrumentation Platform technology enables executables to be injected—post-build—with application instrumentation logic that streams runtime data to one or more cloud-based repositories. The resulting

intelligence can be integrated into CRM, ALM and ERP platforms.

How likely is it to become standard in every professional tester's arsenal? For starters, Microsoft feels strongly enough about PreEmptive's Dotfuscator Community Edition to integrate its functionality into Visual Studio 2010.

To realize this ideal, Holst identifies four requirements for the technology: Runtime data collection, transport, analytics and reporting; developer instrumentation platform and tools; security and privacy policy management services; and role-based access for both the software supplier and end-user organizations.

Injection makes it easier to understand various end-user hardware configurations, for example. It's virtually impossible for any company to test all the configurations its customers use, but a post-build injection lets you break out usage

data by nearly any criteria. A company may find it has only one customer still using Windows 2000 and decide to drop support, but the analytics could grab the customer ID, query the CRM system and return the contract value. "If it shows that this one customer constitutes millions of dollars in revenue, the business may continue support," Holst says.

Attempting to model the real-world environment doesn't necessarily yield accurate results

because the "squeaky wheels" who speak up may not represent users as a whole, while conversely, satisfied customers tend to stay silent. But through injection of instrumentation, it's now possible to understand the environment and see which features are used most and least often, and which can have significant ramifications on systems design and sup-

“
Post-build
injection is or will
be available for
both .NET and
Java.”

Joel Shore is a 20-year industry veteran and has authored numerous books on personal computing. He owns and operates Reference Guide, a technical product reviewing and documentation consultancy in Southborough, Mass.

On-Point Project-Based Testing Services

A new service offering from Checkpoint Technologies and HP



New On-Point Project-Based Testing Services from Checkpoint Technologies brings you the power of HP Software and the knowledge of our consulting services on an engagement-by-engagement basis.

BENEFITS HELP YOU DRIVE RESULTS

- Increase budget flexibility by shifting software expenses from CAPEX to OPEX
- Access to HP Software based only on your consumption needs
- Checkpoint Technologies experienced engineers working in your test environment
- Customized testing solution designed to meet your needs
- Rapid deployment for time-sensitive situations

THE POWER OF HP SOFTWARE

- HP Quality Center
- HP QuickTest Professional
- HP Business Process Testing
- HP LoadRunner

Checkpoint Technologies, Inc.
13650 W. Hillsborough Ave.
Tampa, FL 33635

Phone (813) 818-TECH (8324)
Fax (813) 818-8302

Business Partner



CHECKPOINT
Technologies



ALTERNATIVE THINKING ABOUT APPLICATION LIFECYCLE MANAGEMENT:

Computers Don't Run Your Apps. People Do.

Alternative thinking is looking beyond the development cycle and focusing on customer satisfaction. Because the real application lifecycle involves real people – and the customer's perception is all that matters in the end.

HP helps you see the big picture and manage the application lifecycle. From the moment it starts – from a business goal, to requirements, to development and quality management – (and here is the difference) – all the way through to operations where the application touches your customers.

HP ALM offerings help you ensure that your applications not only function properly, but perform under heavy load and are secure from hackers. (Can't you just hear your customers cheer now?)

Technology for better business outcomes. hp.com/go/alm

