AST UPDATE

Smart Stuff for Career Software Testers

The Process of Exploration
Find the Best Bugs Fast with Charter Based Testing

Experience Report
Achieving Usability Through Testing

Right Click -> View Source and Other Tips for Performance Testing the Front End

Workshop Debrief: WREST
About the first Workshop on Regulated Software Testing

AST Education Update
Black Box Software Testing

Bugs in the Wild
iTunes & More

Testing Tips & Tricks
Creating a Code Boneyard

Volume 1 Issue 2
December 2007

President's Welcome

Welcome to the second issue of the AST newsletter. We are pleased to welcome our new editor, David Christiansen. David is the founder and managing editor of TechDarkSide.com and is a hobbyist writer and speaker. He can be reached by email at dave@techdarkside.com. David is looking for writers interested in columns and features for the AST newsletter.

I'm also pleased to announce that Michael Bolton has agreed to Chair the 2008 Conference for the Association for Software Testing. Michael is planning CAST 08 for Toronto Canada the week of July 14th, 2008. I'm very pleased to announce that our keynote speaker will be Jerry Weinberg. For more information on the conference (call for papers, details, and sponsorship information) check the AssociationForSoftwareTesting.org website for the latest information. If you have any questions or would like to help with the conference, you can contact Michael at mb@developsense.com or conferences@associationforsoftwaretesting.org.

Finally, I wanted to thank the hard work of the students and instructors involved in the AST Black Box Software Testing courses. We've successfully completed two iterations of the BBST – Foundations course, we are planning the first offering of BBST - Bug Advocacy, and we have an instructors program in place to help develop instructors for future offerings. If you want to join a course, you should send an email to member.services@associationforsoftwaretesting.org to be put on the notification list.

Thank you for your membership, volunteerism, and support as we continue to grow the organization. We are proud of what we've done to date, and we look to continue to grow both our membership base and the value we offer our members in the coming year.

Sincerely,

Michael Kelly
President, Association for Software Testing
president@associationforsoftwaretesting.org

AST UPDATE

Managing Editor
David Christiansen
dave@techdarkside.com

Technical Editor
Scott Barber
sbarber@perftestplus.com

A Publication of
The Association for Software Testing

Features

Departments

From the Editor...

Welcome to the second issue of the official newsletter of the Association for Software Testing. I hope you noticed your newsletter's new name - *AST Update, Smart Stuff for Career Software Testers*. Hopefully, this name conveys the intent of the editorial staff of this publication, to give career testers tools and resources that make them better, smarter testers who help promote the craft of testing as one that requires intuition, skill, and a highly engaged brain.

Please send your feedback and comments to me - I'm very interested in hearing how our content is received.

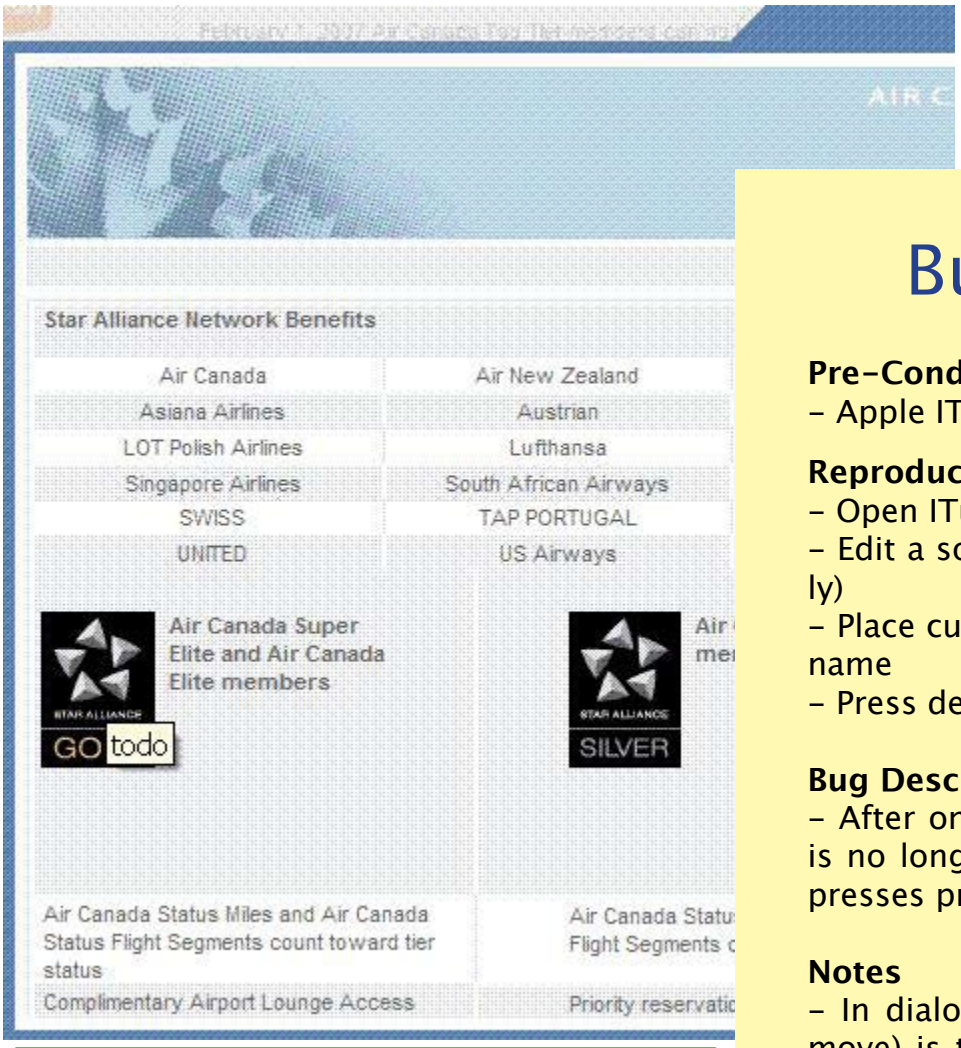
Sincerely,

David Christiansen
Managing Editor

Bugs in the Wild

Clean Up Your Code!

Adam Goucher (www.adam.goucher.ca) posted this bug he found on Air Canada's website to his blog, Note the "TODO" text box on the screen – some developer forgot to do his laundry!



Both Adams received a free copy of Alter Ego, an IT murder mystery, by David Christiansen for their submissions.

Found your own bug in the wild? Send a brief write-up with pictures to dave@techdarkside.com, subject BITW. If we publish your bug, not only will you win the admiration of your peers, but we'll also send you a free AST T-Shirt.

Making Your Music Disappear

Adam White (www.adamk-white.com) sent in this bug report for a problem he found in iTunes, along with a video of the bug as well.

Bug Report

Pre-Conditions
- Apple ITunes Version 7.5.0.20

Reproduction
- Open ITunes
- Edit a song name (F2, or click directly)
- Place cursor at the beginning of song name
- Press delete key multiple times

Bug Description
- After one press of delete song name is no longer in edit mode. Subsequent presses prompts the user to delete file.

Notes
- In dialog pop up default action (remove) is the most devastating one for the end user.
- Making cancel default would be better
- May cause delays when user actually wants to delete file.
- If user edits quickly there is a good chance their files will be deleted

Link to Video

Untangling the Rat's Nest

by Danny R. Faught

I often teach testers to look for multiple bugs in the same input field when they do boundary testing. If a large input triggers a failure, a larger input may trigger a completely different type of failure. A recent testing effort gave me an extreme example of this.

I was testing a numeric input field that represented a length of time in minutes. I fired up the perlclip tool and quickly generated a string of 100 "9"s. I like to start big. I pasted the number into the field. The application seemed okay for a moment, but after moving the focus to a different field and moving the mouse around a bit, I got an error complaining that the number was too large for an "Int32." I clicked OK in the error dialog, and another one popped up just like it. I dismissed that, and a few moments later, I got yet another. I had to kill the application using the Task Manager. Great! That was bug number 1.

When I find a bug like this, there's a good chance that I can try a smaller value, and find a totally different bug, less severe than the first, but more likely for users to encounter because the input is more reasonable. Sure enough, when I tried a smaller number, I got an "Unhandled exception" crash. Even better, when I restarted the application, the crash repeated immediately. I had to edit the Windows registry before I could use the application at all, even after reinstalling it.

Next I wanted to find the boundary between these two bugs. To my surprise, I found some values in between that seemed to work. And then I found yet another error, complaining that I couldn't set the value to a number less than 1, even though the input was actually much larger than 1. As I continued to search for the lower boundary for the "Int32" error, instead I found more instances of the other two errors. It took me a few more hours of testing to discover a pattern in the behavior.

Numbers below a certain threshold (larger than anyone would really need to enter) are ok.

Numbers in a range of about 4,000,000 possibilities caused the unhandled exception. The lower boundary seems to drop by 1 every minute.

The next 35,791,394 numbers trigger the non-fatal error about having a number less than 1.

The next 31,664,026 numbers seem to work, though when the application processes the input, it changes the input to a smaller number using a modulo pattern.

The previous three ranges repeat many times consecutively, until...

Values of 2,147,483,648 and above cause the repeating Int32 error.

This is one of the most complex bug isolation challenges I've ever encountered. It certainly reinforced the need to keep careful notes, and to know exactly what inputs I'm using for each test. Placing a book on the keyboard and going out to lunch doesn't cut it.

Further reading:

- A Bug Begets a Bug
- How to Make your Bugs LonelyTips on Bug Isolation
- Yet another email hyperlink bug
- An Elusive Diagnosis

WREST-ling with Software Testing in a Regulated Industry

WREST, the Workshop on Regulated Software Testing is the newest LAWST-style workshop to be sponsored by AST. The purpose of WREST is to share ideas, generate new techniques, and to provide a forum for people who are interested in improving the testing of regulated systems. We're defining regulated software as software that is subject to review by an internal or external regulatory body.

WREST's primary focus is on better, more efficient ways of testing regulated software systems. This includes considering and discussing any approach or technique to testing regulated software while still ensuring successful completion of audits both internal and external.

We anticipate holding WREST workshops twice a year. Our first workshop was held this past November in Indianapolis. WREST 2 is scheduled for spring, 2008 in Chicago. For more information, see the WREST website at *www.wrestworkshop.com*.

WREST Attendees
Back Row, from left: Mike Goempel, Geordie Keitt, Kelvin Lim, David Christiansen, co-founder John McConda, and Scott Barber
Front Row, from left: Dana Agnew, Crystal Bartlett, co-founder Karen Johnson, Mike Kelly, Cem Kaner, and (not pictured) David Warren

"Our intention is to build a distinct community for software testers working in regulated environments. We've both had experience working in regulated and non-regulated environments and we feel testers working in regulated environments have unique challenges. We want to provide an ongoing forum to exchange and collaborate ideas. WREST is our way of reaching out to testers in regulated environments."

Karen Johnson and John McConda, cofounders of WREST

Right Click -> View Source And Other Tips for Performance Testing the Front End

Scott Barber

I recently read *High Performance Web Sites: Essential Knowledge for Frontend Engineers* by Steve Souders, O'Reilly, 2007. The book is sub-titled "14 Steps to Faster-Loading Web Sites." Before you stop reading because this is a book written for developers, consider the following:

The research Souders presents suggests that approximately 80-90% of a web page's response time results from front end design decisions. My experience suggests numbers more like 50-80%, but most of my experience comes from projects where existing multi-user applications are being retro-fitted with a web-based front end and/or applications with significant back end performance issues that I have been called in to help find.

Virtually all of the tools, training, articles, and conference talks available to individuals who test the performance of software systems are heavily focused on the back-end. So much so, in fact, that most dismiss the front-end aspects of system performance as something not worth worrying about, ostensibly because we can't control the end-user's system.

my experience, the front-end design and development of web sites is conducted with little to no thought to performance, outside of possibly reducing the size of the graphics. Additionally, I've never been made aware of a team conducting peer review on the HTML that generates the web page on the client side, never been made aware of unit tests on such code, nor witnessed a tester deliberately and proactively testing the HTML for possible performance issues. Put these things together, and what you get is no one paying attention to, or checking for, potential performance improvements in the part of the web page most likely to contain the most opportunities for the largest and cheapest performance improvements. Reading this book, I realized that I frequently test for most of these front end performance issues without realizing it, that he mentioned some I likely would have never thought to test for, that there were a few front end performance issues mentioned in the book I wouldn't have even known to test for, that it's been a mistake to not test for these items (or call them out while teaching testers) more deliberately, and that these tests are low-cost, both in terms of time and in terms of the tools and resources required. In fact, I frequently conduct most of these tests during the first 15 minutes of performance testing I conduct on a web site, though I admit that in 15 minutes I can generally only test a couple of pages.

Throughout this article, I describe how to conduct tests manually, using load generation tools, network protocol analyzers, helper websites, and browser plug-ins. I have validated the techniques

Free Tools!

Free or Open Source Load Generators (a.k.a. Performance Testing Tools)

- JMeter (http://jakarta.apache.org/jmeter/)
- WebLoad (http://www.webload.org/)
- OpenSTA (http://www.opensta.org/)

Free or Open Source Network Protocol Analyzer

- Ethereal (http://www.ethereal.com/)
- Fiddler (http://www.fiddlertool.com/)

Free Browser Plug-Ins

- Firebug (http://www.getfirebug.com/) with YSlow (http://developer.yahoo.com/yslow/) for Firefox
- HttpWatch (http://www.httpwatch.com) for IE

Free Helper Websites

- Web Page Analyzer - from Website Optimization: *Free Website Performance Tool and Web Page Speed Analysis* (http://www.websiteoptimization.com/services/analyze/)
- Gomez Instant Site Test (http://www.gomez.com/info_center/instant_test.php)

using the following free tools – not free trials, but free-with-no-strings-attached. If you work for a company that doesn't permit the use of free tools, I'm certain that a quick web search will help you turn up dozens of alternatives that will cost your organization enough money for them to take the tool seriously (for those of you who think this is a joke, it's not. More than 50% of the teams I consult with and individuals I train report not being permitted to use freeware, open source software, shareware, or even time-limited free trials on company machines or networks).

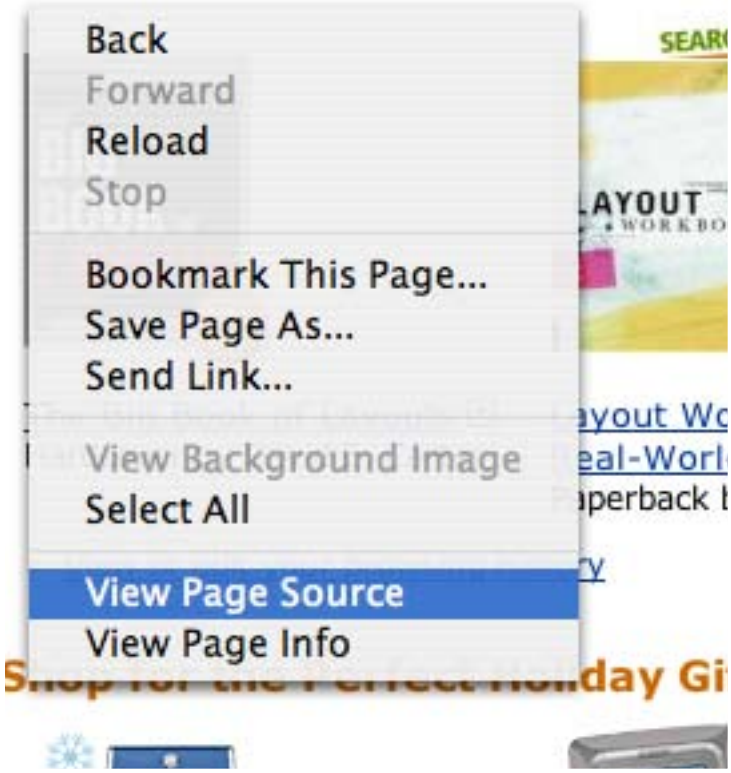
With that, let's take a look at some of the tests you can conduct with no more training than you'll get in this article, tests that could lead to dramatic improvement in end-user response times without requiring that you look any deeper than the web server.

Number of HTTP Requests

Web pages are not retrieved via a single transaction. Pages generally include a single request for the HTML document, one or more requests for stylesheets, one or more requests for external scripts, and multiple requests for graphics, multi-media content, and third party content such as advertisements. Even when many of these objects are stored locally in the browser's cache, a request is still frequently sent to the server to determine if the object in the cache is still "fresh." What this means is that each object used in rendering a web page carries with it significant potential for increased overhead, and thus degraded performance from an end-user perspective, even when the client has a "primed" browser cache containing the object in question. Determining the number of requests a page makes, and what it is requesting, can be done in several ways.

No matter what method you use, you will want to begin by either clearing your browser cache or requesting the page twice (one time using ctrl -> refresh to override the browser cache) to ensure that you can view all of the requests. Since the following methods only collect actual requests, not clearing or overriding the browser's cache could cause an incomplete collection and lead you to think that no stylesheets, scripts, images, or multimedia content is being requested, depending on a variety of settings and conditions you may or may not be aware of and/or have any control over.

1. If you have access to a load generation tool or a network protocol analyzer, you can simply start recording, then navigate to the page or pages of interest. Search your recording for "GET" statements, and make a note of what objects are being requested. Remember that, with some tools, the default view of the recorded script may only contain the base HTML request, not the child requests (i.e. requests for linked stylesheets, external scripts, graphics, etc.), thus requiring an additional step to view all of the requests.
2. If you are testing in an environment where you are permitted to install browser plug-ins, you have several options available to make this task simple. The plug-ins I recommend, depending on which browser you use or wish



to test against, are:

a. Firebug with YSlow for Firefox.

b. HttpWatch for IE.

3. If you have no tools available, you still have several options:

a. Visit one of the helper web sites listed earlier, type the URL for the page you want to test into the text box, and push the "Submit" button.

b. In Firefox, right click on the page, select Page Info, then navigate to the Media tab. Note that this method does not reveal scripts and stylesheets, but will still show you requested graphics and other multi-media content.

c. In IE, right click on the page and select View Source. In this case, you will need to search the code for "link" and "img" tags. Additionally, if you find links to stylesheets (any link to a .css file), you will also need to download each stylesheet by manually requesting it from the navigation bar and searching it for "url" entries which may be used to request scripts, images, or multimedia content. (This is by far the most cumbersome method, but it will still get you the information).

Armed with your list of requests, the first thing you are looking for is volume—excessive requests slow things down. There are several indicators to look out for to decide whether or not the requests you see are excessive.

1. **More than one request for an external stylesheet.** While it is sometimes a good design decision to separate page styles into more than one external stylesheet, this is not common and is certainly not generally helpful in terms of performance. Generally speaking, maintaining more than one external style sheet is a good idea only if there is one base stylesheet that applies to many web pages and a second, large, stylesheet with styles that only apply to a few web pages.

2. **More than one request for scripts from the same domain.** Even though there are more sound reasons for linking to multiple external scripts than linking to multiple external stylesheets, it still not common for links to multiple external scripts to result in better performance than linking to a single, consolidated external script. For example, if the page you are testing is a complex data entry form, it may make sense for the form validation script to be separate from other scripts that are common to all pages. Doing so would reduce the size of all pages except the form, thus improving performance on those pages, even though the extra request would degrade performance slightly on the form page. Regardless, more than one external script is at least worth asking about.

3. **A lot of graphics.** I can't tell you how many is "a lot," but I can tell you that currently, IE7 and FireFox 2.x default to 2 parallel downloads per hostname (for HTTP/1.1 web pages, which most new sites are). This means that, no matter the size of your images, the browser will only download 2 at a time, and that the next two won't start until **both** of the preceding two are complete. This is different from HTTP/1.0, where FireFox defaulted to 8 parallel downloads per hostname and IE varied by version but never to fewer than 2. The impact of this change is that using the fewest graphics of approximately the same size tends to result in

Continued on page 16

Building a Code Boneyard

By Mike Kelly

Great testers suffer from overproduction – they generate more ideas than they could ever reasonably use. It happens so often, we have testing techniques and approaches that are focused on enabling you to make decisions around which tests you should run (because you can't run them all) and which tests you shouldn't. If you are new to testing and don't yet suffer the pain of overproduction, we also have test techniques and approaches that will help you come up with more ideas for testing than you could possibly use.

Why does this dynamic exist? Why would we on one hand have techniques designed to help us generate more ideas than we could possibly use and on the other hand have techniques that help us narrow the scope of our testing?

One problem testers face is that we don't know what we don't know. This causes us to error on the side of producing too much in an effort to compensate for the unknown; and as part of that idea production process we learn about what it is we are testing. Once we sufficiently understand the problem, or think we sufficiently understand the problem because we've produced as many ideas as we possibly can, we can then make informed decisions around which tests might be the right tests to run. This is one of the main reasons overproduction occurs.

People who are experts at producing ideas have a wealth of ideas, data, and experience available to them; more than could possibly be required. Testers who are good at overproduction make idea production cheap, quick, and diversified so they don't worry about getting it right the first time. Any one of their ideas can be a bad idea or could miss the mark and that's ok because they know they will get at least one idea right and will abandon the ones that don't work. This technique is commonly referred to as "shotgunning". Another familiar example of overproduction is the classic brainstorm. Neither of these activities are wasteful if the effort is relatively inexpensive and the ideas sufficiently diversified.

Overproduction often results in growth of the tester; having produced something once you can more easily produce it again, making you more skilled as a result of overproduction. For example, each time I need to produce ideas for performance testing I get a little bit faster and my list grows a little longer. That's not accidental. It's because I'm systematic in how I produce my ideas and I'm systematic about how I abandon and recover them.

Abandonment is another key part of idea generation – without it testers would be completely overwhelmed. A tester who is good at abandonment knows when to stop an activity or leave

data behind and to move on with other activities instead. Ideas that are no longer useful should be abandoned in a way that allows them to be recovered later when a context in which they might be useful emerges. This option for later recovering abandoned ideas produces the ability to refactor an idea, using some or all of the original idea or artifact in a useful way. To be really good at abandonment and recovery, you need to reduce or eliminate the maintenance costs of keeping your ideas at your fingertips.

At this point, you might be asking yourself, "What does all this have to do with code; and where's that cool boneyard I was promised in the title to this article?" Creating a code boneyard is an abandonment and recovery technique. It gives you the ability to freely discard code because you'll know where to look for it later. Overtime, the bits of code (or bones) you discard will start to accumulate and you'll find you have a rich set of examples to draw on when you need to generate

Tactics for managing your ideas

Overproduction, abandonment, and recovery are tactics for how to manage your ideas:

- Overproducing ideas for better selection. Producing many different speculative ideas and making speculative experiments, more than you can elaborate upon in the time you have. Examples are brainstorming, trial and error, genetic algorithms, free market dynamics.
- Abandoning ideas for faster progress. Letting go of some ideas in order to focus and make progress with other ones.
- Recovering or reusing ideas. Revisiting your old ideas, models, questions or conjectures; or discovering ideas already made by someone else.

You can find more skills and tactics critical to the professional exploration of technology here: http://www.satisfice.com/articles/et-dynamics.pdf

code quickly.

I currently have three separate code boneyards that I work with. At work my team has a simple SharePoint list where we can upload scripts (Ruby, SQL, VB, etc...) that we think others might find useful. I also carry a thumb drive with me where I keep a large pile of Ruby scripts and code snippets (the thumb drive keeps my boneyard mobile). And finally, in my webmail account I have a folder for code snippets from mailing lists where I may see something that I can't apply immediately, but I know I might have a use for later.

Here is what's common between those three boneyards:

- Each "pile" has a specific purpose (work code, personal code, community code that might be useful at some point, but I haven't really researched or used yet).
- You have access to it wherever you go. I can access SharePoint anywhere on the office network, the thumb drive is always with me, and I can hit webmail anywhere I have an internet connection.
- You can search it (thumb drive), index it (SharePoint), or sort it (webmail). That is, you can use tools to aid in rapid recovery.
- Here's what you don't want to think about as you build your boneyard:
- Don't think about maintaining the code or worry about compatibility issues.
- Don't limit your boneyard to working code - there can be value in storing ideas that didn't work. Many times your best code can be found in bones created as you struggled with a difficult problem you never managed to completely solve.
- Don't think you always have to go back to the boneyard. I used to keep reusing the same bit of Ruby code that loaded all the filenames in a directory (and all it's subdirectories) into an array. One day I discovered I could actually remember the code enough to write it from scratch.

A code boneyard is more than a dumping ground for physical resources like code that you can abandon and recover. The very act of going through the process of idea generation and pruning changes the tester - even if you abandon an artifact, you still retain in your mind the experiences of creating it. You retain the learning experience, the effort invested in improving your position on the learning curve. The next time you need to create something similar you are better at it and you will be able to achieve the results you want more quickly.

About the Author

Mike Kelly is currently a Software Development Manager for a Fortune 100 company. Mike also writes and speaks about topics in software testing. He is currently the President for the Association for Software Testing and is a co-founder of the Indianapolis Workshops on Software Testing, a series of ongoing meetings on topics in software testing, and a co-host of the Workshop on Open Certification for Software Testers. You can find most of his articles and blog on his website *www.MichaelDKelly.com*.

Achieving Usability Through Testing

by David Rabinek

Effective usability testing can be a key driver in the success or failure of a software product -- where an important element of project success is almost always user satisfaction.

As a card carrying member of the context-driven school of software testing, I strongly believe there is no standard set of best practices for any/all usability testing. Testers that take an intellectual and analytical approach to understanding one's context can create a customized set of practices and artifacts that will be effective in their particular context.

In this article, I describe two software development projects where we took very different routes to deliver effective usability with very different results. It is my hope that the lessons learned from these projects will encourage testers to think carefully and perhaps a little differently about usability before starting their next project.

First, I describe a project that relied heavily on the business user/stakeholder to identify usability requirements and test them with little structure or guidance from either the project manager or the test team. Second, I describe a project that leverages the lessons learned from the first project to carefully plan and manage usability testing as a method to achieve effective usability.

Project #1

The functionality of this software is to store daily business results in a well-controlled database that delivers standard (canned) reports and ad-hoc reports generated by business users.

The approach followed to deliver usability was as follows:

Business requirements were documented and signed off by all stakeholders before development. The document was very detailed with respect to functionality and workflow. The usability requirements – performance, UI intuitiveness and details surrounding the ad-hoc reporting requirement were far less specific.

Project stakeholders included full time involvement of a senior-level business analyst. This individual was very knowledgeable about the business operations, specific workflow, and reports required of this application. He was part of a team using numerous spreadsheets and manual processes to generate what the system would ultimately automate. He was an experienced financial analyst familiar with leveraging software applications, databases and reports to accomplish his day-to-day objectives. He had experience building systems to perform functions similar to the one being built in this project. He had software development experience and understood the typical roles and responsibilities of various stakeholders. He had a vocal, critical style and wouldn't hesitate to actively participate in project work sessions and meetings to share his thoughts and ideas.

The responsibilities of the business user on this project included:

1. Development of detailed business requirements. Generally the requirements were excellent, but certain portions of the specification lacked detail, especially usability attributes. Performance requirements, look and feel, and ad-hoc reporting capability functionality were particularly weak. The development team had already decided on the database reporting tool they would incorporate -- without seeking user or tester involvement in the decision -- so the requirements simply stated that

the tool would be available to the users. The developers were confident it allowed for great amount of reporting flexibility.

2. Participation in all weekly project status meetings during which progress, plans and problems were discussed. The user provided and collected feedback on a weekly basis from his boss. Little was shared with the larger project team about these meetings.

3. Developers generated a design document that was reviewed by all project stakeholders. Several sessions were held to review the proposed GUI, workflow and functionality. All users were involved in these design review sessions which yielded many improvements.

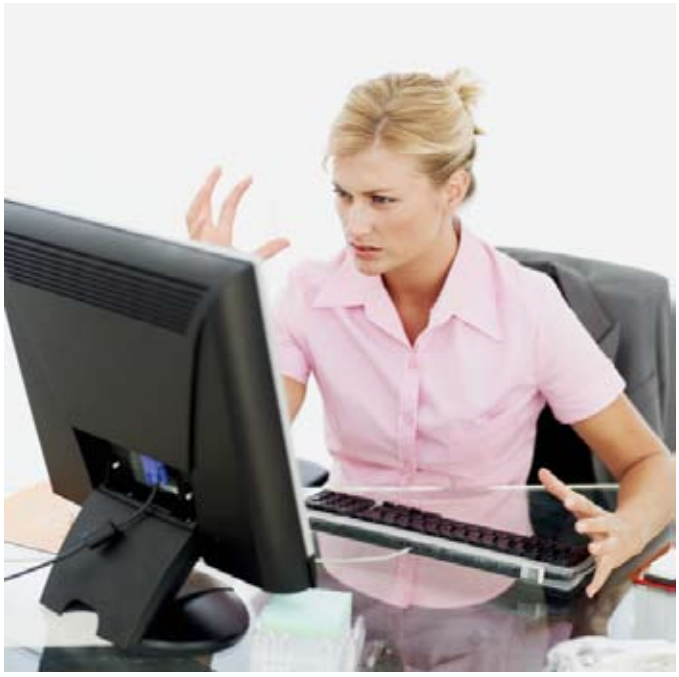
4. The lead user was given access to test versions of the application during the system test phase of the project. There was no plan or structured approach to his use of the system. His feedback was folded into the larger set of defects discovered during testing.

5. The user defined and executed UAT – which included running all production use cases on his hardware. His feedback was given lots of attention and issues were resolved promptly.

6. The user led the rollout of the software product. This included a PowerPoint presentation and demo for all users and management of typical use cases and reports.

Results of the project included:

Numerous defects were discovered when the production version of the software was delivered to the full set of users. Significant performance issues appeared – for example, contention with other applications on user's hardware. Performance issues appeared – for example, database queries and reports were very slow to generate.



The workflow of screens was deemed overly complex – there were too many steps and the order was not intuitive. Calculations built into the system were identified to be more complex than desired – simpler approaches were requested. The GUI text was deemed to be non-intuitive and unclear.

This system recorded key financial risk statistics about our business that were reported to senior management daily. As such, the software had many mechanisms to protect the data, limit update access and generate audit trails of actions. The permissions and levels of approval for various activities were deemed overly complex, cumbersome, and unnecessary.

The workflow for creating ad-hoc reports was deemed to be overly complex and required fairly advanced developer skills -- which most users didn't have.

Most importantly, it was simply not possible to extract results into a spreadsheet. The lead business user was repeatedly heard saying "it's not a database it's a vault". This critical functionality was missing from the product. Interestingly, it was never requested in the business requirements, but simply expected to be delivered.

Lessons Learned:

What is intuitive to one person is not necessarily intuitive to another. In this project we relied on feedback from one experienced user to define intuitiveness for the entire user team that included staff with a wide range of experience and background.

Satisfactory performance (in this case: speed of the software) to one person may not be satisfactory performance to another.

Simple to one user may be complex to other users. The single user defined a series of equations that he

Continued on page 19

Process of Exploration

Exploratory testing can help you find the best bugs in a hurry

by David Christiansen

With only two weeks of user testing left on the project schedule, I was beginning to panic. User testing should have started more than a week earlier, but due to unexpected problems with our software deployment process, we still hadn't delivered a stable build to the test environment.

I had already received one extension to our delivery schedule, and I certainly didn't want to ask for another. With our deployment problems finally fixed, the challenge would be to complete testing in time to save the production release schedule.

Like many project managers in a similar bind, I faced a time crunch. The testing team, operating in three locations across the country, had to execute all the required test scripts before the release could be moved to production. These scripts could be executed quickly -- in about three days -- if there weren't any bugs. Critical bugs would force us to patch the code and start the test scripts all over again. Making matters worse, we had to do all the work manually: Our shop had a plan to deploy automated test capabilities, but it would happen too late to help this release.

Here's what I proposed to my testing team: We would open with three straight days of exploratory testing, find all the bugs we could as rapidly as possible, then start scripted testing while we waited for the new build. Once the new build arrived, we would execute the scripts one more time and pray everything passed. It was a long shot to get it all done in two weeks, but I felt confident that if we could find all the critical bugs in the first three days, we could pull it off.

The team had never tested this way before, but they were eager to try it. They were skilled business users and several of them had complained about the constraints that scripted testing put on them. They would rather "just poke around in the application" than follow written scripts like witless rats in a maze. Unfortunately, our business partners required written testing records, and I knew that "three days poking around" wouldn't satisfy their compliance needs. I felt we could satisfy both groups by providing a little structure to the test effort, and I was right.

Three days into exploratory testing, we had already found several bugs that never would have emerged using the planned scripts, plus we found a few more bugs that would have

Benefits of Exploratory Testing

- Reduced need for detailed test planning
- Increased emphasis on application quality
- More effective use of knowledgeable testers
- Reduced fatigue and boredom of testers
- Increased agility in testing as understanding of the problem space evolved

caused the scripts to fail. The development team set about fixing them and a new build was delivered on day five. Three days later, the scripted testing was successfully completed, and we were authorized to move the application into production.

Everyone was satisfied with the end result, and our test team credited exploratory testing for making the production release date.

Testing by Instinct

How did exploratory testing save my project? By removing unnecessary constraints on the team, it allowed testers to follow their well-honed instincts to find the most critical problems first. The approach reflects the fact that effective testing is more about investigation than it is about simple, scripted procedure.

Removing the constraints of scripted testing had another effect as well — one I'd heard about but had never experienced before — my testers felt more energized and engaged in testing. They enjoyed their work more than ever before, and the resulting energy boost propelled them through testing at great speed.

A highlight of this testing effort was when one of the testers proclaimed that we were progressing at "ludicrous speed," a fantastic reference to the movie "Spaceballs" that characterized perfectly the progress we were making.

Exploratory testing can also benefit projects that aren't behind the eight ball. It's an approach to testing that every project manager should be prepared to use in the right circumstances.

"There are many areas of concern for a tester that are often ignored in business requirements or are simply difficult to execute as scripted tests."

Mike Kelly, President, Association for Software Testing

Mike Kelly, test manager for a Fortune 100 financial services company and president of the Association for Software Testing, says exploratory testing can fill critical gaps in scripted routines.

"There are many areas of concern for a tester that are often ignored in business requirements or are simply difficult to execute as scripted tests. For example, look at the quality criteria outlined in the Satisfice Heuristic Test Strategy Model," Kelly says, rattling off a long list of categories. "Capability, reliability, usability, security, scalability, performance, installability, compatibility, supportability, testability, maintainability, portability, localizability. That is most likely an incomplete list."

"It could be dangerous to assume not only that your requirements contain all of those criteria, but also that all of your requirements are written down and none of them conflict with each other," Kelly warns.

Exploratory testing doesn't have to be paired with a scripted testing program. You can put an extremely high-quality application into production using nothing but exploratory testing, and you wouldn't be the first project manager to do it. A colleague of mine recently did just that, in a corporate IT department with stringent compliance and traceability requirements. How did he demonstrate traceability? By using charters to manage the way they communicated about testing.

Taking Up Charters

One method of managing your exploratory testing efforts is session-based test management. This method makes use of charters to structure the testers' work. A charter, put simply, is an objective for a test period, called a session. Our test charters had several features, listed below:

- A title, briefly describing the objective of the test period
- The feature or general area being tested (this ties the charter back to requirements in a general way)
- Notes on activities performed

and bugs found

- Test coverage, i.e. none, broad, narrow, deep, shallow, complete, etc.
- Status, i.e. not started, critical bugs might exist, critical bugs exist, no critical bugs, non-critical bugs might exist, etc.
- Name of the tester

Exploratory testers create charters as they go -- they don't have to be planned in advance. Instead, they let their testing skills and instincts guide them, testing the areas they think are most likely to lead to bugs. If they're testing accessibility, for instance, and they notice something that might be a security problem, they can use charters to change course without losing track of what they've accomplished.

At the point they observe a new area of interest, exploratory testers simply create a charter for the area they've uncovered, then decide whether to pursue the new charter. If they opt to pursue the new charter, they set aside the previous charter (after updating it with notes of their progress) so it's available to pick up later. On the other hand, if the tester decides to continue on with the first charter, they have the new charter ready once they complete the current work.

Without charters, exploratory testing is essentially just poking around in the code and isn't adequate for most situations. The discipline of creating charters is a lightweight way to plan, execute and measure exploratory testing without having the process get in the way.

In my case, we employed a blend of exploratory and scripted testing to make up for lost time and deliver production code ahead of deadline, but it's clear that this type of testing has real value for many projects, not just the ones that are running late.

About the Author

David Christiansen is a project manager at a financial services company, the author of Technology Dark Side: A Corporate IT Survival Guide and the managing editor of Sapient Testing, the official newsletter of the Association for Software Testing.

Continued from page 9

the best performance. This is counter to the notion that smaller graphics are always better. Combining several small graphics into one or two larger graphics frequently improves performance. There is, of course, a point of diminishing returns. Graphic size vs. number of graphics is something that is worth working closely with the front-end designer, to test various options in search of optimal performance. A well respected "rule-of-thumb" to guide your decision making in these matters urges caution if a page contains more than 12 total requests. This number is used by Souders in his book, Andrew B. King in *Speed Up Your Site: Web Site Optimization*, New Riders, 2003 and most convincingly by Aaron Hopkins in an article published on his site titled "Optimizing Page Load Time" (http://www.die.net/musings/page_load_time/).

Sequence of HTTP Requests

Many of the same methods can be used to determine the sequence of the objects being requested when a page loads. The exceptions are the helper websites we discussed previously and Firefox's "Page Info" screen, as these groups and or sort requests by type and size to highlight volume and size issues rather than sequence issues. The sequences you are most interested in are:

1. **Request stylesheets first.** Web pages will either not display at all until stylesheets (.css) have been downloaded or appear to refresh themselves once the stylesheet is retrieved. For this reason, it is critical that stylesheets are among the first items requested following the base HTML page.
2. **Request scripts last (or at least late).** Once a script is requested, no other objects will be requested until the script has been completely received. Additionally, browsers cease displaying content while scripts are downloading. This means that any objects that complete their download after the script has been requested won't be displayed until the script has been completely downloaded, thus making the rendering of a web page appear to stall. This means that it's highly desirable for scripts to be requested after the objects that are most interesting to the end-user. Remember, most end-users

perceive responsiveness based on the time it takes for the content they are interested in to appear, rather than the time it takes the entire page to load.

Whether you are viewing the HTML source or captured requests, what you are looking for is that stylesheets are requested first and scripts are requested either last or at least very near last. There is a common argument that scripts controlling user interactions such as image maps and roll-over objects should be requested early so that the user will have the "proper" experience, even while the page is downloading. In my experience, however, a user is much more likely to become frustrated or abandon a website if it appears to be stalled while downloading than if a roll-over image or image map isn't enabled until the page downloads completely.

Redirection and/or Hidden Errors

You can use the same methods you used to check for appropriate request sequencing to check for redirection (3xx series response codes), client errors (4xx series response codes), and server errors (5xx series response codes). In this case, the main indicators you are looking for are the following:

1. **Excessive 3xx series response codes.** 3xx series codes indicate that the request was processed, but that the browser must retrieve the object from another location – resulting in additional request/response pairs. While there are plenty of sound reasons for the redirection of some requests, it's worth making sure that redirection is being done intentionally and for good reason. For example, redirecting from a removed web page to its replacement, or redirecting from an obvious misspelling of a web page to the correct page is a good reason. Redirecting requests for images because the image's parent directory has been moved but no one has bothered to update the link is probably not a good reason to accept the inherent performance degradation associated with the additional requests associated with redirection.
2. **Any 4xx series response codes.** A 4xx series code is returned when there is a problem with the client request. The most common is 404, which indicates that the requested object was not found on the server. Generally speaking, if the web page is displaying and functioning properly, but individual requests are returning 4xx codes, that indicates that the page is simply requesting unneeded objects and taking extra time to do so.
3. **Any 5xx series response codes.** 5xx series codes indicate that an error occurred on the web server while trying to fill the request. Any 5xx series code should be of interest to the development team.

I refer to these as hidden errors because when they occur for objects other than the base HTML document they are frequently not obvious or even visible to the end-user. Sometimes seeing these response codes is also indicative of deeper errors, but, they all result in requests that do not contribute to the display or content of the page and are frequently entirely unnecessary.

HTTP Response Headers

To check HTTP response headers, you will need to use a load generation tool, network analyzer, or one of the browser plug-ins. If you don't have any of those tools available, another helper web site might be of value. I suggest Peter Forret's "View and analyze HTTP headers" page (http://web.forret.com/tools/analyze.aspx), where you can enter the URL of a web page and the site will retrieve a list of the HTTP headers sent back by the web server, so you can check page expiration and caching settings. The details about what parts of the response are appropriate vs. unnecessarily performance inefficient are highly dependent on variables such as the frequency with which the site and/or objects change, the frequency with which users of your site visit the site, and the relative risk of those users viewing stale content. Nonetheless, the following items are consistently worth inspecting:

1. **Check for an appropriate Expires: entry.** If the HTTP response for an object does not include an Expires: line, every time a user requests a page containing that object, a request will be sent to the server to determine whether or not the cached version is "fresh." If you have objects that are unlikely to change frequently (for instance, the company logo) you can avoid the "freshness check" request with a date/time in the Expires: line that is far in the future. Expires headers are most often used with images, but they are

often also appropriate for other components including scripts, stylesheets, AJAX, and Flash components. Look for objects with no Expires: line and for Expires: entries that seem inappropriate to you.

2. **Check for appropriate ETags.** Entity tags (ETags) are a method of identification that web servers and browsers use to determine whether or not the object cached on the client's machine matches the one on the server. The challenge with ETags is that they are generally unique to a specific web server, meaning that using them may actually be detrimental if the web site has multiple web servers. If you know that the web site uses a single web server, ETags are probably a good idea. If the web site uses multiple servers, you will want to inquire about whether the multiple servers have been accounted for, or recommend that the ETags be removed.

3. **Check other cache controls.** You may or may not observe other entries following lines such as Cache-Control:, Last-Modified:, Pragma:, Set-Cookie:, and Age:. If you do observe those lines, ensure that the entries make sense to you. If you don't observe those lines and feel like they should be there, bring it up to someone.

The bottom line is that you want to check HTTP response headers to determine whether or not the web site has been configured appropriately to take advantage of browser caching on the client side. Frequently, the only way to determine the appropriateness of these entries is to spend time with administrators and architects discussing both how the site is used and how it has been designed, specifically related to client browser caching.

Source Code and Objects

Finally, if you haven't done so already, you'll need to manually examine the source of the HTML, .css, scripts, graphics, and other remaining objects. To date, I have not found any specific tools that save time over manual inspection in enough situations to recommend for these final front end performance testing tasks, although HTML, script, and graphics editors appropriate to the web site are generally useful. The final front end performance testing tasks that I recommend are:

1. **Ensure that HTML source code does not included embedded scripts and CSS expressions.** It is extremely rare that performance is improved by including scripts and CSS elements or expressions directly in the HTML. The reason for this is simple: the base HTML for a web page is the part of the page that is most frequently updated and therefore least frequently served from cache. Since the HTML is so much more likely to be downloaded every time, it only makes sense to keep it as small as is reasonable. Keeping scripts and CSS elements external to the HTML, and thus cacheable, is virtually certain to improve performance, on average, over time, across the users of the web site.

2. **Ensure that styles and scripts are not duplicated.** In my experience, stylesheets and script files are notorious for containing duplicate or overlapping content. Sometimes content is duplicated across separate files; other times it is duplicated within the same file. While you may not want to spend the time to do a complete review, a quick scan of the source can often reveal whether or not there is significant overlap or duplication.

3. **Check for code minification.** Believe it or not, "minification," at least according to the Random House Unabridged Dictionary, is a valid English word meaning "the act of minimizing." With regard to computer code, it refers to condensing and optimizing the code to perform the desired function using the fewest lines and/or characters of code. While inspecting the HTML source code, external script files, and stylesheets, you want to look for excessive comments, white space, line breaks, variable name length, and other items that increase file size.

4. **Check the appropriateness of graphics' size and compression.** It may seem obvious, but many web designers are still using graphics in formats that have unnecessarily large file sizes, in sizes different than the height/width they are to be displayed in, and of a quality well in excess of what is necessary or reasonable for the purpose of the web site they are being displayed on. In general, .gif formatted images compressed

to 64 or fewer colors are more than adequate for most graphics and thumbnails; .jpg formatted images compressed to 256 or fewer colors are typically adequate for photographs; and it is rarely justifiable to use HTML height/width properties to shrink or stretch an image rather than creating a new image of the correct size.

In each of these cases, use common sense as your guide. For example, some web sites reduce all of their file, directory, and variable names to two or fewer characters each as a matter of policy to minimize file size. From a purely performative perspective, this is excellent; however, the additional work required to document and/ or maintain the code makes this practice completely unreasonable for most web development efforts. You will have to work with your team to find the proper balance between duplication/ minification/ compression and practicality.

Summary

This article describes several tests that can be used to determine if a web site is likely to exhibit poor front end performance. Identifying these areas of potential performance improvement could result in a 50% or greater reduction in the user-perceived response time of the web site. I am confident that once you get your tool box of applications, plug-ins, and helper web sites in place, and practice these tests just a few times, you will be able to scan a website for significant offenders of each of these items in less time than you just spent reading this article. With such a significant potential for dramatic performance improvement, and such a small investment in time and tools required, I see absolutely no reason why any web site should go live without these tests being conducted.

About the Author

Scott Barber is the Chief Technologist of PerfTestPlus, Vice President and Executive Director of the Association for Software Testing, Co-founder of the Workshop on Performance and Reliability, and lead contributing author of Performance Testing Guidance for Web Applications. Scott specializes in testing and analyzing performance for complex software systems, training performance testers, and Context-Driven testing. Scott is an international keynote speaker and contributor to various software testing publications.

Continued from page 13

felt were a simple, intuitive way to generate the set of required results. A much simpler set was defined by the larger set of users and implemented.

"Easy to use" to one user may not be easy to use to another. Relying on a single user to define easy to use is risky.

Don't assume users want a tool to automate the current business workflow – analyzing the flow and suggesting improvements as a standard part of the project may not only be appreciated, but expected. Workflow is often defined by the tools available to projects that implement new tools need to take advantage of the opportunity to simplify workflow.

Don't assume users' lack of engagement in requirements development implies a lack of interest in usability. Find a mechanism to extract those requirements that fits your context.

In the world of financial systems, it is safe to assume users will always want the capability to download all details to EXCEL for further analysis and reformatting of reports.

Project #2

An application that loads portfolio data and model parameters and initiates batch runs that measure risk. This system generated the risk statistics stored in the database descriped in project #1.

For this project, two approaches were followed to achieve the desired level of usability:

1. Developers worked directly with the user group, holding numerous work sessions to discuss the work flow and functionality the system would provide, developing "prototypes" on a white board in the Director's office. This information was recorded and fleshed out in the business requirements and software.

2. The test team and development team structured their plans so that usability was tested first – with a planned new build to incorporate feedback – before moving forward to functionality and user acceptance testing. In this manner, the initial test release was treated as a prototype, and the usability test as a hands-on mechanism to collect the usability requirements.

The usability testing was staffed

with business and test team participants. The testers defined specific step-by-step instructions for every button and field on the UI as well as every standard use case so that usability testing would be comprehensive and therefore minimize the likelihood that usability defects would be identified later in the project. The testers prepared a template for users to record specific feedback on every test.

The test team Director and business team Director spent hours defining what usability meant for this software and the type of feedback required to be provided by business-testers. Usability was defined as intuitive GUI text, easy to use GUI layout and workflow between screens. Usability also included UI response time as well as batch run performance – for all standard and several non-standard use cases.

A diverse team of business users was engaged to conduct usability tests. The business users included users ranging from the experienced to the inexperienced with this set of tools, workflow and risk statistics. It also included management-level business users.

The usability tests were conducted on each of the user's computers using production data at the time of day when production activities are conducted.

Usability tests were also conducted for several non standard uses of the system with wider tolerances for usability such as performance and ease.

A session was held to consolidate feedback from all users, define the solution, and set priorities for development. Project teams included in this session were the entire set of business users as well as management-level staff. Developers and testers participated to get clarification and express issues.

Results of this usability test included:

Lots of feedback was collected during usability testing. It was prioritized and addressed in a new build that was retested by users to confirm usability quality met their expectations for the first release.

The users were more engaged in the entire project, and felt stronger ownership of the final product and more invested in the success of it. The users gained significantly more appreciation for the benefits of careful planning in the phases of projects in general, and the testing phase in particular. The system delivered satisfactory performance, defined as UI response time and batch run time for all standard use cases.

The system delivered satisfactory intuitiveness in terms of the UI screen content, work flow and report content and labeling. The UAT phase was completed faster than planned due to user familiarity achieved during the usability test phase, reduced user questions, and fewer user errors. Far fewer user/operator errors occurred in the early months of production because of that same familiarity and the quality of software's usability.

Development and test teams received rich feedback during usability testing. Numerous early insights into future release usability requirements were revealed. As a result of all stakeholders sitting and discussing user's feedback the team came to understand the user's vision for the tool

Lessons learned from this approach to usability testing included:

Hands-on user testing of a prototype can be a very effective way to collect requirements for usability. Provide the user some tools, such as a plan (to ensure completeness and timeliness) and template for collecting feedback.

A carefully selected mix of users (inexperienced, experienced, and management-level) results in richer feedback and reduces risks inherent in taking feedback from a limited set of users. A process for consolidating and prioritizing this large volume of feedback will be required. It is useful for the process to include all project stakeholders.

The initial project plan should include a new build after usability testing, and time for users to test their usability defect fixes.

Heavy involvement by a wide range of users early in the project accelerated later phases, improved their overall project engagement, and provided useful vision to the project team.

About the Author

Dave Rabinek has been involved in software development, testing, business analysis and project management for 25 years, primarily in Financial Service and Consulting firms (Booz Allen Hamilton). His education includes a BS Computer Science and an MBA Finance.

Starting a New Educational Program in the Association for Software Testing

by Cem Kaner, J.D., Ph.D.

A year ago, AST joined Rebecca Fiedler's and my research proposal to the National Science Foundation (NSF) to experiment with a new model for online course design and an open source vision for creating a sustainable instructional community. The project was funded last fall, and so we (AST, Rebecca and I) have started developing and offering free courses to our members.

The project starts with a black box software testing (BBST) course that Hung Quoc Nguyen, James Bach and I developed for commercial instruction, which I then adapted for academic use at Florida Tech. In essence, AST is bringing a grown up version of that course home, to its commercial roots.

The AST BBST series will be a set of about 20 fully online short courses. So far, we have developed one course, BBST Foundations, taught it twice, and had excellent results.

The Courses

There are plenty of commercial courses on software testing, available from talented teachers (including several of the most active members of AST). As we discussed this in AST Executive meetings, it was clear that our goal had to be to create a compellingly better learning experience—for students who had the time to devote to it.

The challenge of time forced our first design question: ***How much time should our students have to devote to the course?***

Online professional-development courses have stunning dropout rates—often 90%. Even online courses people have paid a lot of money to take have huge dropout rates. One factor that influences dropout rate is the quality of engagement between the instructor and the students. Another factor is perceived value of the course, relative to the workload. A third factor is length—how long will this course last? A course that runs too long has a high dropout rate.

Based on our reading of the literature and a bit of pilot work (an online course for potential instructors), Dr. Fiedler and I decided that the course should require about 6-14 hours of work per week. Less than that and students would not be engaged enough with the course to stay motivated. More would conflict with the intense on-the-job workload of most testers. We also decided that any course longer than a month was probably too long for most of AST's students.

Given the one-month time frame, our next question was: ***What is the optimal amount of course content for a one-month course?***

The tradeoff is between coverage (how much is supposedly taught) and depth of learning. Gerald Weinberg once said that you can make a program meet any other requirement—as long as it doesn't have to work. The same thing is true for teaching. You can cover any amount of material in a course—as long as the student doesn't have to walk out knowing it.

- For us, teaching isn't just giving a lecture and hoping that interested students will learn something. Improving our teaching isn't just making our lectures better. Teaching happens when the students actually engage with the course and learn from it.
- For us, teaching isn't just helping people memorize things or apply them to examples simpler than almost anything they'll see in the real world. Teaching is about helping people learn things that will be useful or interesting in their real (post-course) life. That requires a much deeper level of knowledge.
- To help people achieve our instructional objectives, we include work beyond the video-based lectures. Coursework includes homework, discussions, group projects, quizzes, and exams. Students submit their own work, review the work of at least two other students, commenting on assignments and grading each other's exams. We expect detailed comments (and provide some training) because people learn more if they try a problem, see and critically think about other ways to do it, then go back to their own work and reflect on what they did.

In the Florida Tech courses, we taught roughly one instructional unit per week. That is, one week might cover domain testing, the next week was scenario

testing, and so forth. We decided these were probably the right size chunks for the AST courses.

Here's the list of courses we plan to offer to AST members for the AST BBST series, in the approximate order we hope to create the classes:

- Fundamental issues in software testing
- Bug advocacy
- Domain testing
- Function testing
- Scenario testing
- Overall perspective on test design
- Risk-based testing
- Specification-based testing
- Testing variables together (combination testing)
- Introduction to regression testing
- Analyzing requirements for test documentation
- Scripted testing
- Exploratory testing
- Measurement in software testing
- Quality cost analysis
- Introduction to GUI level regression testing: Analyzing requirements for successful GUI test automation
- Introduction to high volume test automation

In addition, we hope to add courses from some other instructors. Scott Barber, for example, is thinking about how to fit courses on performance testing into our structure.

We cannot promise that we will offer all of these courses. This is a volunteer effort that will progress as we find volunteers to develop and teach the courses.

The Initial Course: BBST Foundations

The core learning objective for BBST Foundations is to prepare students for success in the later, increasingly difficult, courses in the BBST series.

Preparation includes foundational content:

- Overview of testing terminology, including demonstrating the extent of honest, rational controversy over many of the key terms and concepts in the field;
- introduction to the objectives of testing, including tailoring objectives to the context of a specific project;
- introduction to test oracles and oracle heuristics;
- introduction to test effort estimation and the myth of "complete" testing;
- introduction to the concepts of misleading metrics and measurement dysfunction, in the context of assessing how close to complete your particular testing effort has been.

Preparation also includes work on a variety of learning skills that are important for success in online courses.

We've offered BBST Foundations twice. So far, we've had students from Africa, Asia, Europe, New Zealand, North America, and South America at all levels of experience. We expect this diversity to be typical.

Students often work in groups, created by the instructors. Our heuristic is to maximize diversity (geography, industry, culture, gender, expertise), making everyone work with people in significantly different time zones, who spoke with significantly different accents, and who approached problems in interestingly different ways. Some of our students haven't yet faced projects like this in their jobs. Our courses will help them prepare for those situations (and give them good stories for their next job interview.)

In terms of workload, we expect the typical student to work 8 hours per week on a course. In practice, there was a lot of variance. A few students reported spending less than 4 hours per week, some spent 15. On average, the typical student worked about 12 hours per week on the BBST Foundations course. The workload will (we hope) stay about the same in later courses, though the content and expectations will get harder.

For Further Reference

The full set of Florida Tech course materials is at www.testingeducation.org/BBST. You are welcome to use these materials as you develop your own courses.

As we wrote the instructional objectives for each course, we had two classes of questions. (1) What do we want students to learn? And (2) How well do we want students to learn it? I wrote about the application of instructional ideas on depth of learning to testing courses on my blog, at:

- Assessment Objectives. Part 1–Bloom's Taxonomy
- Assessment Objectives. Part 2–Anderson &

Student Reactions to the BBST Foundations Course

Jeff Fry, Pradeep Soundararajan and Louise Perold have posted blog entries describing their experiences in the course.

Most students who stayed to the end passed the course, but it was a challenging one. In the post-course survey (students spent about an hour giving us detailed, anonymous feedback), students rated the difficulty and more importantly, the perceived value of the course as follows (data combined across courses):

Compared to the commercial courses that you have taken, this course was:

- Much more difficult: 31%
- More difficult: 31%
- As difficult: 13%
- Less difficult: 0%
- Much less difficult: 0%
- Not applicable: 25%

Compared to the university courses that you have taken, this course was:

- Much more difficult: 19%
- More difficult: 25%
- As difficult: 31%
- Less difficult: 6%
- Much less difficult: 0%
- Not applicable: 19%

Compared to the commercial courses that you have taken, this course was:

- Much more valuable: 62%
- More valuable: 13%
- As valuable: 6%
- Less valuable: 0%
- Much less valuable: 0%
- Not applicable: 19%

Compared to the university courses that you have taken, this course was:

- Much more valuable: 33%
- More valuable: 27%
- As valuable: 27%
- Less valuable: 0%
- Much less valuable: 0%
- Not applicable: 13%

These results are better than I had hoped for and I don't have confidence that we'll sustain them. Early adopters might to be more enthusiastic than later ones–only time will tell.

Krathwohl's (2001) update to Bloom's taxonomy

- Assessment Objectives. Part 3–Adapting the Anderson & Krathwohl's taxonomy for software testing

Acknowledgements

This article is partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The NSF project includes several overlapping sub-projects that use the BBST materials to reach several different academic, commercial, and in-house professional development audiences. If you or your organization is interested in collaborating in this project, please contact Cem Kaner.

Rebecca Fiedler designed the student survey and leads the sub-project to create an Instructor's Course and an Instructor's Manual for the BBST family of courses. Dr. Fiedler is Assistant Professor of Education at St. Mary-of-the-Woods College in Terre Haute, Indiana.

About the Author

Cem Kaner is Professor of Software Engineering at Florida Institute of Technology and head of its Center for Software Testing Education & Research. You can reach him at kaner@kaner.com.

CAST 2008: Beyond the Boundaries

CALL FOR PAPERS
The 3rd Annual Conference of the Association of Software Testing (CAST) 2008
http://www.associationforsoftwaretesting.org/CAST2008
Toronto, Ontario, Canada, July 14-16, 2008
Beyond the Boundaries: Interdisciplinary Approaches to Software Testing
Keynote Presentation by Gerald M. Weinberg

The Association for Software Testing is pleased to announce its third annual conference (CAST 2008), to be held July 14-16. The meeting will be held in Toronto, Canada, a city which features enormous diversity in culture, businesses, educational institutions, and the arts. Toronto is the perfect location for a conference on this year's theme: "*Beyond the Boundaries: Interdisciplinary Approaches to Software Testing*".

Interdisciplinary approaches draw from diversified branches of learning or practice, such that insights can be drawn upon and synthesized to influence a particular craft. The CAST 2008 Program Committee is now seeking papers that explain how one, two or more disciplines might assist with software testing.

Examples could include ways in which statistics and metrics combined with critical thinking can help software testers interpret performance test results; ways in which logical thinking combined with document design and modeling help testers better understand business requirements and execute functional tests; or ways in which research in human/computer interaction might influence usability testing.

Apropos of this theme, the Association is delighted to announce that the first of our keynote speakers will be Gerald M. Weinberg, presenting a talk entitled ***Lessons from Past to Carry into the Future.*** Fifty years ago, in 1958, Jerry established the very first separate software testing group, to aid in producing life-critical software for Project Mercury. Jerry will speak of many steps, done and not yet done, needed to complete the task of creating a true software testing profession.

Both academic research papers and industrial experience reports are welcome. The following (non-exclusive) list suggests topics of interest that the Committee would consider highly suitable for submission:

- General systems (e.g. modeling, non-linearity, complexity)
- Mathematics (e.g. probability, statistics, combinatorics / permutations, graphing, metrics, equivalence partitioning)
- Epistemology (e.g. logic, lateral thinking, critical thinking, experiment design, decision

making)

- Cognitive science (e.g. biases, perception, descriptive decision making, human factors, dynamics of heuristics, learning)
- Communication (e.g. rhetoric, document design, writing)
- Visualization (e.g. graphical presentation of test results, display and presentation of test data)
- Interdisciplinary approaches to teaching software testing

In addition to looking for papers that demonstrate an interdisciplinary approach to software testing, we're looking for personal experience reports that clearly demonstrate skills and practices of seasoned software testing professionals. We'll be looking for rich, diverse experiences and intriguing papers that illuminate the theme. If you have hands-on experience and a fascinating story to tell, contact us and we will assist you in evolving your tale so it will be ready to present at CAST.

CAST 2008 Important Dates

- Monday Feb 4, 2008 : Deadline for paper submission
- Monday February 25, 2008 : Notification of acceptance/ rejection to authors
- Monday March 17, 2008 : Submission of revised paper integrating the reviewers' comments
- Friday April 4:, 2008 : End of the second period of reviewing
- Monday April 28, 2008 : Final camera-ready papers due
- July 14-16, 2008: Conference

CONFERENCE FORMAT

CAST is designed as a forum to stimulate discussions leading to innovation in software testing, and so is distinguished by significant interaction among presenters and attendees. Papers and experience reports accepted by the program committee are challenged, debated, and discussed by the conference attendees. We encourage and facilitate conversation by building flexibility into the schedule so that topics generating high energy can be explored more deeply without adversely disrupting the course of conference events. Trained facilitators will ensure that discussion sessions are appropriately structured and productive. Discussion sessions will have a recorder, and transcripts or summaries of the discussions will be made available to participants after the conference.

SUBMISSIONS

CAST 2008, although not associate with ACM, encourages authors to follow the ACM SIG Proceedings style, freely available at http://www.acm.org/sigs/publications/proceedings-templates.

We expect a typical submission to be between 4 to 6 pages long. All papers should be submitted electronically in PDF format via email to: CFP@associationforsoftwaretesting.org

Authors of accepted papers will receive complimentary registration to CAST 2008. Papers will be published in the conference proceedings. Authors will also be invited to submit their papers for inclusion in a future edition of the Journal of the Association of Software Testers (JAST).

CONFERENCE CONTACT

For further information about CAST 2008, please contact a member of the conference committee as listed below:

Sponsorship: Scott Barber, executive.director@associationforsoftwaretesting.org
General Conference Information: Michael Bolton, cast2008("at" symbol)michaelbolton.net
Program: CFP@associationforsoftwaretesting.org

AST UPDATE

eVoting SIG

The eVoting SIG was chartered to comment on the current eVoting certification guidelines (Voluntary Voting System Guidelines, VVSG), the first U.S. government regulations to deal so extensively with software testing. Our goal is to make it clear that the heavyweight, suffocating processes required by the VVSG will make it impossible for testers to find software bugs. We expect to catalog patterns of failures that will evade VVSG-compliant testing, and report our findings and recommendations to the Election Assistance Commission's Technical Guidelines Development Committee and the IEEE SCC38 standards group for voting systems.

So far the SIG has attracted several members, a few of whom are actively building the wiki infrastructure to collect and refine our comments on the VVSG. We have also started creating our failure mode taxonomy, based on the model of an eVoting system that we have in-house. If you would like to join the group, please see the eVoting SIG web page (http://www.associationforsoftwaretesting.org/drupal/sigs/evoting) for instructions. For more information on the VVSG: http://www.eac.gov/vvsg/

New AST Web Site in Beta

AST is scheduled to promote a brand new web site shortly before January 7, 2008, but you can get a sneak peek at http://www.associationforsoftwaretesting.org/drupal/. Contact Scott Barber(sbarber@perftestplus.com) with comments or to volunteer to assist with porting content to the new site.

AST Gear Available Now

Get your AST shirts, coffee mugs, notebooks and hoodies at *AST's CafePress Store*.

Recent Peer Workshops

WREST

AST sponsored the first meeting of the Workshop on Regulated Software Testing. WREST1 was hosted by Karen Johnson and John McConda, facilitated by Mike Goempel and held in Indianapolis on Nov. 16 and 17. See Page 6 of this issue for more information.

STiFS

AST sponsored the 5th meeting of the Software Testing in Financial Services Workshop (STiFS), which took place on December 2nd and 3rd at Liquidnet Holdings, Inc. (www.liquidnet.com) in New York. The theme was "Getting Business Knowledge into the Heads of Testers". Facilitated by Scott Barber, STiFS5 explored the methods in which various kinds of "business knowledge" are obtained and used by testers at financial firms. For more information about future STiFS meetings, please see www.stifs.org or contact Bernie Berger (bernie@associationforsoftwaretesting.org) or Scott Barber (sbarber@perftestplus.com).

Upcoming Peer Workshops

The Workshop for Teaching Software Testing(WTST) will be held in Melbourne, Florida this January. Look for a debrief in the next issue.

Scott Barber and Cem Kaner are considering hosting either the next installment of the Software Test Managers Roundtable(STMR), or possibly a new workshop, in Melbourne, FL just before STAR East or just after. If you are passionate about a topic for this meeting, please contact them.

Doug Hoffman is considering reopening the original LAWST workshops. The Los Altos Workshops on Software Testing created the model on which the AST-supported peer workshops are based. The original founders were Cem Kaner and Drew Pritsker, who were soon joined by Brian Lawrence III, then by Elisabeth Hendrickson. LAWST will consider any topic of broad interest to the testing community, adding its best value when the topic is (a) controversial or subject to a lot of misinterpretation, and (b) focused into one or two tight questions for purposes of discussion. The heuristic for the makeup of LAWST is that 1/3 of the attendees should be test managers, 1/3 test-related consultants, and 1/3 be highly skilled individual contributors (people who do testing for a living). It is traditional in LAWST to include 1-2 people who are relatively inexperienced (a year or two, or students, but exceptionally promising). If you live in the Bay Area and are interested in helping Doug organize and host the next LAWST, please contact Doug or Cem Kaner. Note: LAWSTs are funded primarily by the hosts. It is commonplace for a LAWST organizer to spend $500 or more in unreimbursed expenses.