Specialists and other myths: because you aren't a specialists doesn’t mean you can't do it

Michael Kelly
www.MichaelDKelly.com
Indianapolis IN
011-1-317-709-2419
mike@michaeldkelly.com

ABSTRACT

This paper explores how to broadly recognize, develop, and apply different testing skills in many areas of expertise, without becoming a superstar or specialist in each domain.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking and Validation

General Terms

Performance, Design, Reliability, Security, Human Factors, Verification.

Keywords

specialist, testing, coverage, model, oracles, risk, performance, security, usability, automation, exploratory

1. INTRODUCTION

At conferences and workshops, and at client sites when consulting, I talk with many testers who feel like certain types of testing are out of their reach. Specifically, many testers are not comfortable when asked to do automation, performance, security, exploratory, or usability testing. Many testers see these as specialty skill sets that only the elite can do. (*Elite meaning anyone who has experience doing it; who sounds like they know what they are talking about.*)

Now don't get me wrong. There *are* specialists in each of these fields. I'm not as good a performance tester as Scott Barber or Rolland Stens. I'm probably not as good in test automation as Chris McMahon or Bret Pettichord. I'm certainly no James Whittaker or Herbert Thompson when it comes to security. Moreover, I don't think I'll ever be mistaken for a Bach or Bolton when it comes to exploratory testing. However, I can do all of them, to some degree or another, and have performed all of them while consulting.

More importantly, I know that with a little bit of time I can learn how to do each of them better. I know who to talk to if I have questions about them, who to ask about which books to read, and where to find the blogs, articles, and tools I’ll need. I know my limitations (an important skill picked up as a consultant) and when to walk away and recommend someone else. Nevertheless, I certainly don't feel that there is any area of testing that I can't get involved in at some level and be a productive contributor to a project.

Maybe I'm not the lead. Maybe I'm not the expert. Maybe I'm not even the best looking. But as a tester who understands the basic principles of testing, I should be able to contribute productively. This paper is intended to be an informal look at what I actually do when I test; written in a way that I hope allows you to relate to the material so you can see that what I do, is not so different from what you do. I believe that most “testing specialists” aren’t so special. While there are certainly situations that call for specialists, I think that most of us are better served to train as generalists. That way we can dynamically contribute our talents to our project teams in the best way possible. This year’s CAST is on testing techniques. This paper is a collection of the techniques that I use every day.

2. PERFORMING SPECIALTY TESTING

This paper will use the Universal Testing Method [1] as defined by James Bach and Michael Bolton in their class on Rapid Software Testing course. I find it to be a useful way to show how I approach various testing problems. The Universal Testing Method consists of the following:

1. Model the test space
2. Determine coverage
3. Determine oracles
4. Determine test procedures
5. Configure the test system
6. Operate the test system
7. Observe the test system
8. Evaluate the test results
9. Report test results

2.1 I should be able to model the testing space

Modeling is how we build cohesive ideas. It’s how we take the random bits of information we gain about something and put them together to build something useful for our exploration. Our models grow and contract over time. As we learn new information, we test the model with it. If we need to, we incorporate the new information into the model.

Formal models are simply models that have been explicitly specified in some format. That format can be text, a diagram, or a mix of the two. There are a lot of formal modeling languages [2] that get used every day to help people communicate using a consistent set of rules and common structures. Don't think you need to use an existing modeling language to have a formal model. When you sketch on a whiteboard or scribble a specification, you've just created a formal model.

Formal models are very common and tangible. Most of us are comfortable working with formal models (maps, state diagrams, data models, etc...).

As a tester, I feel like I need to be able to work with mental models as well. Mental models are the pictures you form in your head when you think about a problem or an object. It's an internal representation of the problem you are trying to solve or the object with which you are interacting.

When I'm modeling the testing space, I'm trying to understand the application I'm testing. If I'm completely unfamiliar with the application, or applications like it, I start by touring [3]. This tells me what's in the application. As I'm touring, I make notes, sketch pictures (if it's a large application), and write down all my questions. I might do some research (for example, if I were touring BookPool.com, I might go look at NerdBooks.com to see how they solved a similar problem).

If there's nothing to tour (for example, no code has been delivered or the software is a web service without a user interface), I do a mental tour based on the documents I have and the questions I ask. I normally ask to speak to the requirements analysts and the developers. If a customer is available, so much the better.

Once I'm done touring, which could take anywhere from five minutes to five days, I collect what I know into some sort of formal model. When I say formal model, all I mean is a model that I've written down. I like formal models - they help me. I'll draw on a whiteboard, pop open Visio, or I'll look for formal models that others on the project team have already produced and I'll draw on top of those with a pen.

Once I have my formal model (which is just a visual representation of the informal model in my head), I start poking at it. I commonly do this using the Satisfice Heuristic Test Strategy Model [4], focusing on the product elements and quality criteria. I want to make sure I understand all the factors that make up what I'm looking at, as well as all the factors that I can't see that might affect its quality.

At this point, I think I might understand the testing space. Notice, no specialization has come into play yet. All I've done is develop an understanding of the problem space. For most projects that I can imagine, I would still try to model the testing space in this way. While the focus of my touring and model may change based on the type of testing I'm doing, my actions remain the same.

2.2 I should be able to help determine coverage

Determining the testing coverage is how I understand what I'm testing in the application. To develop a coverage outline, I rely on my understanding of my testing mission as well as my model I developed earlier. If my mission specifically involves something like automation, performance, security, or usability, then I might collaborate with someone who *is* an expert if I have one available. However, assuming one is not, this is how I determine coverage.

I like to develop coverage outlines in Excel. I think I might have a bias towards developing matrices, but they tend to work well in my context (which is typically financial service applications). I'll often start by developing a generic list based on my model. I typically do this by working through the elements of the SFPDO heuristic [5] to get things started.

If you are not familiar with the SFDPO heuristic, it addresses the following:

- Structure (what the product is)
- Function (what the product does)
- Data (what it processes)
- Platform (what it depends upon)
- Operations (how it will be used)

Once I have my initial list, I put it down and walk away from it. I do this for a couple of reasons. Normally, it's because I'm tired, but also to give myself time away from the list to see if anything new occurs to me while I keep it in the back of my thoughts.

Next, I'll see if there is any data I currently have access to that's ready to use (or could be ready to use with very little work). Is there test data lying around from past projects or production that I can use? What coverage does that give me? Is there test data I can create easily with tools or automation? What coverage does that give me? If I find anything interesting, or if the data I find sparks any ideas, I'll go back and add that to the coverage outline. In financial services testing, data is a big part of coverage.

After that, I'll think about specific risks that I can identify based on the project environment, the quality criteria that I most care about, and based on the implementation technology and strategy for the application. Sometimes I'll use bug taxonomies to spark my thinking if I have a hard time getting started. These normally help me with generic risks. The one I reference most is the appendix to Testing Computer Software [6]. Once the taxonomy gets me going, I can normally think of some additional risks that are more specific to my application.

It is here, in the thinking of specific risks, that I think the specialist has the advantage over the novice or general tester. Roland Stens will "out-risk" me when it comes to performance testing. He's done it for more years, in more contexts, with more tools, on more platforms, with more hardware configurations, and he has more training. I don't stand a chance!

But that doesn't mean I can't be a productive member of a performance testing effort, and certainly doesn't mean I can't work, or even lead, simpler performance testing projects. If I understand risk, understand how to recognize when I may be in over my head, and when I may need help, then I have everything I need to act with confidence as a tester.

Once I have a coverage outline, I work to get it reviewed with project stakeholders. That typically involves dialog and tradeoffs. I cut out a bunch of the stuff I wanted to test and add a bunch of stuff I didn't think of. Over time, this outline evolves as my understanding of the application and the risks to the project evolve.

2.3 I should be able to help identify oracles

Once I have a coverage outline, I need oracles. An oracle is the mechanism for determining whether the application has passed or failed a test. I feel like I use a lot of heuristic oracles [7] when testing, but I also do a lot with parallel testing (using another application as an oracle) and specification-based testing. When looking for oracles for some of the specialty testing topics, I've noticed some patterns to my oracle selection.

When I start looking for oracles for automation, I first look for applications for parallel testing. Running parallel tests with large amounts of data is highly desirable for financial service applications. For regression, I find that I look for consistency with the history of the product, and much of my regression automation is focused on hard coded requirements verification. When I can, I leverage model based testing using automated oracles such as calculated values or even something a simple as just a simple check for spectacular failures of the system (like Java exceptions or other program crashes).

When I start looking for oracles for performance testing, I normally have to start with a focus on regression. That's because many of the applications I test have poorly defined performance requirements, so consistency with the history of the product is the low hanging fruit. After that, service-level agreements (SLAs) or requirements for specification-based testing are next on the list. These can lead me to better defined model based tests or tests that look at specific thresholds in the system.

If I don't have an idea of past performance (perhaps it is a new system or no one has collected historic performance data) or I don't have some form of formal requirement, then many times I'll default to talking with an expert user or customer who can provide some insight into acceptable performance. I can ask clarifying questions such as, "It performs this fast under these conditions, is that good enough?" If all else fails, I can again just start designing tests based on my model of the system and try to simply provide baseline data to the customer in an effort to get them thinking about performance.

When I start looking for oracles for security testing, I look for any information that I absolutely should not be able to find or any area of the application I shouldn't be able to access (given certain user privileges). Then, I use an "inverse oracle" that says, "If I can find it, I've found a failure." I don't get much opportunity to do penetration testing, but the couple times that I have, I develop some simple threat models based on my understanding of the system, its users, and the types of information a malicious user might want.

If all else fails, for most technologies that I interact with, there are already databases full of known exploits. These known exploits become useful oracles in and of themselves, and often, they will lead me to think of new ways in which the application might fail as I start to test each exploit. As I test an exploit, I make notes about what new exploits I can imagine based on my understanding of the specific system I'm testing.

When I start looking for oracles for usability testing, I begin with any formal usability requirements that might have been specified. Given that I've only worked on one project where that's been the case, I don't often get much in the way of requirements based oracles for usability. Normally, I invoke the HICCUPP heuristic [7] for usability testing.

I stole the HICCUPP heuristic from James Bach, and use it liberally in all my testing. Here's what the letters for the mnemonic stand for:

- History
- Image
- Comparable Product
- Claims
- User Expectation
- Product
- Purpose

Inconsistent with History: A product should be consistent with past versions (or history). History can include previous versions, patches, claims, etc. If something has changed, and no one told you it was supposed to change, then you might have found a problem.

Inconsistent with Image: Most companies want to have a good image in the marketplace. Therefore, their software needs to look professional and be consistent with accepted standards. If a product is inconsistent with the desired image, what you're saying is this: "We'll look silly (or unprofessional) if we release this software to market."

Inconsistent with Comparable Product: You're letting another product serve as your oracle for this test. As long as the comparable product really is comparable, and you want your product to be an alternative to that product, or you want to get the users that that product has, then this oracle can be very compelling.

Inconsistent with Claims: A "claim" can be anything that someone in your company says about the product. If something is inconsistent with claims, it's inconsistent with the product's stated requirements, help, marketing material, or just something that a project stakeholder said in the hallway.

Inconsistent with User Expectations: This product doesn't do something that a reasonable user of this product would expect it to do, or doesn't perform a task in a way that the user would expect. Using this oracle means that you have some idea of who the user is and some indication of what he or she expects.

Inconsistent Within the Product: Something behaves in one way in one part of the product, but in a different way in another part of the product. The change could be related to terminology, look and feel, functionality, or feature set. All you're doing is pointing out where the product is inconsistent with itself. These are often compelling bugs.

Inconsistent with Purpose: In my mind, this is the most compelling of the oracles. It states that the behavior you found is contradictory to what a user would want to do with this software. You might be talking about the purpose of a feature, for example: "Look, I can enter a negative value for headers in this Microsoft Word dialog box." That wouldn't really be consistent with the purpose of headers and footers, would it? This oracle is often used in conjunction with Inconsistent with Claims or Inconsistent with User Expectations because those oracles also tend to address the purpose of the software.

While those build the foundation for my usability testing, I also keep a keen eye out for any other user expectation (formal or informal) I can find while working on a project.

When I start looking for oracles for exploratory testing, everything is fair game. Requirements documents, design documents, functional specifications (or any of the other 101 different ways we document explicit or implicit requirements on software projects), or any of the oracles mentioned above serve as my foundation. From there, I tend to create test charters [9], where each charter has an associated oracle of some sort.

The interesting thing about oracles for exploratory testing is that that finding them is no different then finding oracles for scripted testing. It’s the same process. So anyone skilled at finding traditional functional test oracles should have no problem finding oracles for exploratory testing.

I find most people I mentor and work with feel the oracles for exploratory testing are more abstract then with scripted testing. It’s only once they’ve done exploratory testing for a period of time that they discover they are really using all the same oracles they were using before, they are just using them real-time instead of documenting them ahead of time.

2.4 I should be able to give input related to test procedures

What really makes a specialist special? I like to think it’s their advanced understanding the methods and tools of the type of testing in which they’ve specialized. Figuring out the test procedures that will be used while testing is, for me, the most difficult part of testing in the murky waters of specialized fields of testing.

A test procedure is simply the instructions for setting up a test, how to execute the test, and for how to evaluate the results. Like a model, a procedure can be informal or formal. Test procedures for functional testing tend to be fairly straightforward. But imagine the test procedure for browser compatibility testing for Amazon.com, or performance testing for TurboTax Online, or security testing for Fifth Third Bank’s public facing sites. Where do you start with each of those? What needs to be set up? How do you execute the tests? Using which tools? How do you know if a test passes or fails?

As a non-specialist, I don’t need to be able to define any of the test procedures above on my own. In fact, I would be surprised if for any of the above, there was only one person who defined the test procedures used. All I need to do is provide input. That’s the value I should always be able to add. If I’m working on a performance testing project with Scott Barber, he should know that I know enough that he can bounce ideas off of me and I can provide feedback. I can review his procedures with a critical eye.

As my skill in each type of testing increases, and as I gain experience, my ability to provide valuable input increases. If I want to develop a test procedure for a type of testing I’m not an expert in, I find that I start with what I know and make small adaptations. For example, I once developed a test procedure for a large website migration project. At the time I knew nothing about web propagation and replication, and to be honest, I’m still a bit fuzzy on some aspects.

I started by outlining what I would do if I were doing basic functional testing for the website. Then I asked myself what would change in the way I setup my test if instead of looking just at functionality, I was also concerned with a new operating environment, new host servers, etc…? This added some coordination points to my setup. I would need to be in constant contact with someone at the new and old hosting companies when I started my testing.

Then I looked at how executing my tests changed. What tools could I use to enable me to go faster? I found that a simple link and content scanner like Rational SiteCheck (a tool packaged with IBM Rational Robot [10]) could save me a lot of time and energy. I was more concerned with source file versions, link locations, and basic content checking then with the actual functionality of the deployed files and software.

Finally, my results verification became a process of log comparisons between execution runs, diffs between domain listings and directory structures, and a process of again coordinating multiple checkpoints with the vendors. My test procedure had changed, but not in a way that was completely foreign to me. I was boldly going where I had never gone before, but I understood how I got there. As my test procedure underwent review by the project team and the vendors, everyone felt it was the right approach.

While I can’t share the details of the test procedure example, there is a great example I can point you to. The single best example I know of for a publicly available example of a test procedure is the *General Functionality and Stability Test Procedure for Microsoft Windows 2000 Application Certification* [11] that James Bach developed for Microsoft to help them test Windows 2000 compatibly.

2.5 I should be able to perform the detailed steps of software testing

All the steps in the Universal Test Method up to this point have focused on understanding the testing problem. The next four steps in the Universal Test Method look at the details of software testing: setup, interact, observe, and evaluate. Regardless of my level of specialization, as a tester I should be able to:

- help configure the system or at least understand the current configuration
- operate the test system or interact with it in some way
- observe the test system
- evaluate the results of my testing

It’s in this section that I think specialization falls away and the generalist can be just as effective as the specialist. With the few exceptions of using highly specialized tools or programming languages, most testers can probably effectively setup, interact, observe, and evaluate software tests if they have a model, a concept of test coverage, an oracle, and a reasonable test procedure.

I once tested a self-service ticket machine at a movie theatre [12]. It looked and functioned similar to an ATM: select your movie, slide your credit card, and print your tickets. No one was paying me to test the kiosk, I was just killing time. What was great about the opportunity is that it allowed me to practice exploratory testing, usability testing, performance testing, and security testing all at once.

The system would allow you to select up to ten tickets for each type of ticket you could purchase (adult, child, senior). While testing the limits of ticket selection and the proper calculation of the total amount, I noticed that if you max out the number of tickets for senior and child priced tickets, the system beeps at you each time you try to select more then ten tickets. However, when trying to select more then ten tickets priced for adults, there is no beep. It made me wonder about the beep. Was it a usability feature?

After I was done doing my functional analysis of the system I had a chance to do some usability testing by watching people interact with the system. I noticed one case in particular that showed what I consider to be a serious defect. A lady using the system selected her movie, entered her debit card information, and started waiting as the screen displayed “Please wait while processing your transaction.” I assume that at this point the system was attempting to connect to whatever service it uses to process credit cards.

As luck would have it, at that moment credit card processing for the theater went down. I know this due to the very vocal population of customers at the ticket counter. Unfortunately for the lady making her self-service purchase, the ticket machine seemed to have hung as well. It just sat there saying “Please wait while processing your transaction.” No message saying “Timed out while connecting to service. Please try again.” No message saying, “Trying your transaction again, please wait.” Nothing. It just sat there.

After about five minutes, the lady finally lost her patience and started pushing the cancel button. She pushed it once. She pushed it a second time - harder. She then pushed it five times in rapid succession. She then put all of her weight into the pushing of the button and kept the button down for several seconds. This processed continued for some time. I counted her push the button over 40 times. Still the screen read, “Please wait while processing your transaction.” So much for cancel… She then left the machine and went to the ticket counter for help.

I found other issues while testing, but what stands out for me when I look back on this experience is not the issues I found, but that the process of finding issues “in the wild” is the same that we use “in the lab.” There was setup and configuration for my testing (show times, my credit card, connectivity to the bank, real users I could observe, my watch to time transaction response times).

There was interaction with the system (myself and other uses pushing buttons, the system with the bank, the system with the system at the counter that the clerks used, customers swiping cards, the system printing tickets and receipts). There was observation of results (me noticing beeps and information on the screen, me looking at my receipt and tickets, me looking at the time on my watch, me listening to customer reactions and the conversations at the counter, the actions the user took under stress). And I was able to draw conclusions based on those observations (there could have been better error messaging in the system, there might be a bug around the beeping for adults, the fact that the cancel key sticks could be due to people applying fifty pounds of pressure for extended periods of time).

Usability specialists probably could have noticed more then me, would have used a more controlled population sample for their test, would have done better recording results (I used my notepad, they would have most likely used a camera), and would have been able to come up with more researched recommendations for improvement. A performance specialist could have looked at the different results different connection speeds had on the user wait time. A security specialist may have been successful at figuring out what the secret sequence of keys is that brings up the admin interface (I couldn’t do it in the few minutes I tried, but it’s my suspicion that there is one).

As a generalist, I’m ok with all that. It’s not my goal to be able to do all of those. I want to be able to offer a fist line of defense, and if I find or suspect a larger problem I can then recommend a specialist. I want to be able to help configure the system for testing. I want to be able to interact with the system in a way consistent with the type of testing I’m doing. And when I observe the system, I want to know what types of information to look for. When I evaluate my results, I want to know what I’m looking at and have a good set of heuristics that tell me what each result might mean.

2.6 I should be able to report test results

I have a framework for thinking about my testing that lets me report results with confidence and to tailor my test report appropriately for the audience [13]. Regardless of the type of testing, for me a test report should address mission, coverage, risk, techniques, environment, status, and obstacles. This holds true for both written reports as well as verbal reports.

I want my test reports to cover what I’m attempting to accomplish with my testing. This is my mission. Am I trying to find important problems? Assess product quality or risk? Or am I trying to audit a specific aspect of the application (such as security or compliance)? Having a clear mission makes it much easier for me to know what my status is. If I know my mission, I have a precise idea of what I’m supposed to be doing.

For specialty testing, I try to be as specific in stating my mission as possible. I don’t want to get in trouble because someone thinks I’m doing something I’m not. It has happened to me before with both performance and automation. I often state my mission by listing the questions I’m trying to answer.

I include my coverage in my reports; the same coverage outlined above. Depending on my audience, I may just do a high level outline, or I may go into the details. If I’m ever uncertain about what coverage information to include, I go back to my mission and ask myself what areas I needed to cover to answer the question. It’s not enough to say what you covered; you should also indicate why you covered it. That’s where risk comes in

If I think the audience for my report will be interested, the report will also include what techniques I used while testing. In addition, I might also list the environment and configuration where the testing took place. These details can again be high-level, or very specific.

For me, the most important aspect of a test report is my status. I try to answer the following questions:

- How far along am I?
- How far did I plan to be?
- What have I found so far?
- How much more do I have to do?

I follow-up my status with a list of any obstacles I might have. This can be as simple as "I didn’t do X because of Y" or as detailed as "If we had X we could do 20% more of Y and 10% more of Z, but that might also mean that we don’t get around to testing W until Friday." It can be helpful to think of questions like these:

- Do I have any issues I need help with?
- Is there anything I can’t work around?
- Are there any tools that would allow me to test something that I can’t test right now?

3. LEARNING SPECIALTY TESTING

Other then the Universal Testing Method, what are the common themes running through this paper? I like to think they are fearlessly attacking problems, rapid learning, leveraging your community, and practice. I think those are the keys to developing and applying different testing skills and techniques in many areas of expertise, without becoming a superstar or specialist in each domain.

There are a lot of ways to practice rapid learning. In his CAST 2006 tutorial on Self-Education for Testers [13], James Bach offered the following:

- ***Touring****: I read a survey piece.*
- ***Experiencing****: I build an example; or do the activity.*
- ***Serendipity****: I learn from unexpected events.*
- ***Teacher****: I go see someone.*
- ***Reading****: I find famous books and papers.*
- ***Global Supermind****: I tour Google.*
- ***Standards****: I discover what is considered "correct."*
- ***Communities****: I find a forum or professional association.*
- ***Conferences/Classes****: I attend with a critical attitude.*
- ***Browsing****: I skim and riffle.*
- ***Acquisition****: I gather a library.*
- ***Testing****: I contrast alternatives, critique, or consider extremes.*
- ***Teaching****: I try to explain it.*

I find that list inspiring. For me, many of those components are a part of my daily diet of information. I'm constantly browsing, touring, reading, and teaching. I learn from others, call my friends, email colleagues, and eat lunch with software professionals I don't see very often. Constant learning is the key to training your mind to be able to quickly learn the new aspects of specialty testing. As you encounter the various contexts where you need to apply these skills, rapid cognition enables you to know when and where to apply them.

Following developing your rapid learning abilities, the second most important thing to do is practice [14]. Each time you practice testing, you should be interested in doing some specific thing better. By improving one specific technique at a time, you gradually improve your overall ability over time. The focus of your practice should not be repetition, it should be on improving a specific aspect of what you are practicing (speed, technique, tools, and so on).

Practice can build new thought patterns—and can also reinforce existing thought patterns. By doing something over and over or repeatedly thinking about something in a specific way, you actually change the way your mind works. Remember that the goal of practice is to stretch yourself and to increase your control over your performance. Identify where your testing may be weak and think of a series of practice sessions that might help improve that aspect. Do you need to become more technical? Do you need to brush up on your black box testing techniques? Or do you simply need to step back from test management and actually get your hands dirty again? Identify what you want to improve and focus on doing that better.

4. REFERENCES

[1] Bach, James and Michael Bolton. Rapid Software Testing version 2.1.2. Satisfice, Inc. 1995-2007.

[2] "Diagram." Wikipedia 9 Apr 2007. <http://en.wikipedia.org/wiki/Diagramming_technique>.

[3] Kelly, Michael. "Taking a Tour Through Test Country: A Guide to Tours to Take on Your Next Test Project." Software Test and Performance Magazine. Feb 2006: 20-25.

[4] Bach, James. Heuristic Test Strategy Model version 4.8. Satisfice, Inc. 1996-2006.

[5] Bach, James. "How Do You Spell Testing? A Mnemonic to Jump-Start Exploratory Testing." StickyMinds.com 14 May 2001.

[6] Kaner, Cem, Jack Falk, and Hung Q. Nquyen. Testing Computer Software. 2nd Ed. New York: Wiley, 1999.

[7] Kelly, Michael. "Using Heuristic Test Oracles." InformIT.com 28 Apr 2006.

[8] Bach, Jonathan. "Session-Based Test Management." Software Test and Quality Engineering. Nov 2000.

[9] IBM Software – Rational Robot – Product Overview: Rational Robot. 2007. 9 April 2007 <http://www-306.ibm.com/software/awdtools/tester/robot/index.html>.

[10] Bach, James. General Functionality and Stability Test Procedure version 1.0. Satisfice, Inc. 1999.

[11] Kelly, Michael. "Testing at the movies." testingReflections.com 31 Jan 2001.

[12] Kelly, Michael. "Dimensions of a Good Test Report." InformIT.com 24 Mar 2006.

[13] Bach, James. "Self-Education for Testers." Satisfice, Inc. 2006. <http://www.associationforsoftwaretesting.org/conference/cast2006/James_Bach_Self-Education_for_Testers.pdf>.

[14] Kelly, Michael. "How Do You Practice Software Testing?" InformIT.com 12 Aug 2005.