



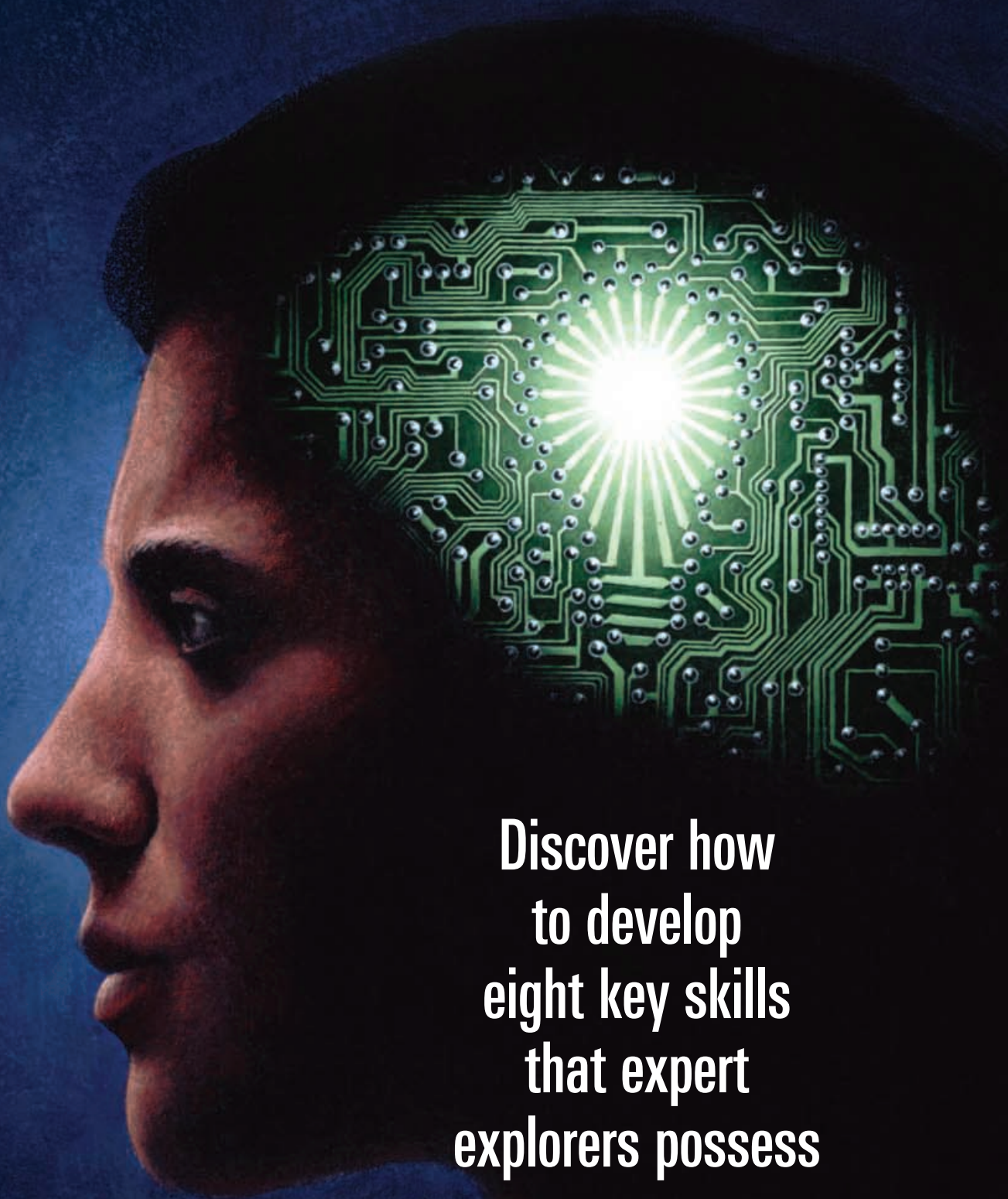
# INSIDE THE MIND OF AN EXPLORATORY TESTER

AMONG THE HARDEST THINGS TO EXPLAIN IS SOMETHING THAT EVERYONE ALREADY KNOWS. We all know how to listen, how to read, how to think, and how to tell anecdotes about the events in our lives. As adults, we do these things every day. Yet the *level* possessed by the average person of any of these skills may not be adequate for certain special situations. Psychotherapists must be *expert* listeners, and lawyers *expert* readers; research scientists must scour their thinking for errors, and journalists report stories that transcend parlor anecdote. ■ So it is with exploratory testing (ET). It's a simple concept, but to do it well

## **INFO TO GO**

- Exploratory testing is a simple concept, but to do it well requires skill and practice.
- There are systematic, specific exercises you can do to become a better exploratory tester.

requires substantial skill and practice. The outer trappings, inputs, and outputs of exploratory testing are worth looking at, but it is the inner structure of ET that matters most—the part that occurs inside the mind of the tester. That's where ET succeeds or fails; where the excellent explorer is distinguished from the amateur.



**Discover how  
to develop  
eight key skills  
that expert  
explorers possess**

**BY JAMES BACH**

## DISCIPLINED, PURPOSEFUL TESTING

To give you an example of what an exploratory tester might be thinking about as he tests, let me share one of my ET experiences.

I once had the mission of testing a popular photo editing program in four hours. My mission was to assess it against the specific standards of the Microsoft Windows Compatibility Certification Program. The procedure for performing such a test is laid out as a formalized exploratory testing process. My goal was to find any violations of the compatibility requirements, all of which were clearly documented for me.

With my charter in mind, I set myself

to test. Applying one of the simplest heuristics of exploring, I chose to begin by walking through the menus of the application, trying each one. While doing so, I began creating an outline of the primary functions of the product. This would become the basis for reporting what I did and did not test, later on.

I noticed that the Save As... function led to a sophisticated set of controls that allowed the user to set various attributes of image quality. Since I knew nothing about the technical aspects of image quality, I felt unprepared to test those functions. Instead, I started an issues list and made a note to ask if my client was willing to extend the time allowed for testing so that I could study documentation

about image quality and form a strategy for testing it. Having made my note, I continued walking through the menus.

A basic strategy of ET is to have a general plan of attack, then allow yourself to deviate from it for short periods of time. It's like being on a tour bus. Even though the tour bus takes you to certain places at certain times, you can still step off occasionally and wander around. The same is true with exploratory testing. There's value in seeing what you can see on the planned tour, but it's also important to occasionally look at something more closely or to investigate something that might not have been on the itinerary. Whatever you do, don't fall asleep on the tour bus—this

## EXPLORATORY TESTING DEFINED

I've tried various definitions of exploratory testing. The one that has emerged as the all-around favorite among my colleagues is this:

*Exploratory testing is simultaneous learning, test design, and test execution.*

In other words, exploratory testing is any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

Have you ever solved a jigsaw puzzle? If so, you have practiced exploratory testing. Consider what happens in the process. You pick up a piece and scan the jumble of unconnected pieces for one that goes with it. Each glance at a new piece is a test case ("Does this piece connect to that piece? No? How about if I turn it around? Well, it almost fits but now the picture doesn't match..."). Instead of looking for problems, you're looking for connections, but otherwise it's the same sort of thinking. It is pretty straightforward to test if two pieces fit together, but the choice and progression of which pieces to pick up in the first place is not so simple. There is no sure way to know when you start the puzzle which "tests" to run and in what order.

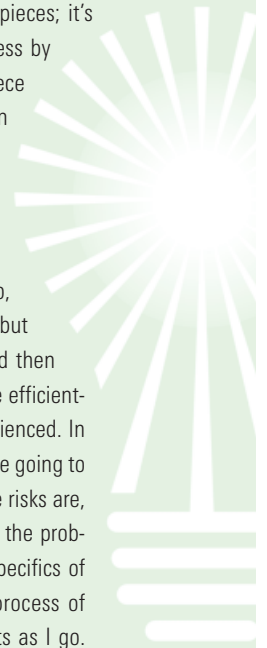
But it isn't a random selection, either. This is where skill comes in. You home in on shape, color, or pattern. You might sort the pieces into little piles, first. You might flip them all face up. If you find you've got a big enough block of pieces assembled, you might move it into the frame of the puzzle to find where it connects with everything else. You may feel disorganized from time to time, and when that happens, you can step back, analyze the situation, and adopt a more specific plan. If you work on one kind of "testing" for a while (attempting to fit border pieces together, for instance), you might switch to another kind just to keep your mind fresh. Even though you aren't handed a step-by-step

procedure for doing it, solving a jigsaw puzzle well is a systematic process—just as is the design of software tests.

Notice how the process *flows*, and how it remains continuously, *each moment*, under the control of the practitioner. New ideas and strategies occur to you as the pattern of the puzzle emerges. Each puzzle piece placed is one less in the pile, making the rest of the search process a little easier. The puzzle changes as you solve it.

If you solve the same puzzle several times, your familiarity with the picture and the pieces will allow you to do the job more quickly. You will have a better idea of how to sort the pieces; it's even possible that you may script the process by putting an ID code on the back of each piece (there is at least one jigsaw puzzle I've seen where the pieces come with numbers on the back). But the first time through, you learn as you go. You can't avoid the learning.

If you happened to know in advance exactly where each piece was supposed to go, then it would no longer be a jigsaw *puzzle*, but rather a jigsaw assembly process. You could then have a method to put it together much more efficiently. But that's not like most testing I've experienced. In most testing, I don't know where the bugs are going to be when I start. I'm not even sure where the risks are, most of the time. When I test, I puzzle over the problem of what the next test should be. The specifics of that puzzle, as they emerge through the process of solving that puzzle, affect my choice of tests as I go. This feedback dynamic—think a little, test a little, think a little more, test a little more—is at the heart of any exploratory investigation, be it for testing, development, or even scientific research or detective work.



# Excellent exploratory testers are more careful observers than novices, or for that matter, experienced scripted testers.

happens to you when you adopt a plan to visit various parts of the product, then visit them without really thinking about how well the product is working. A successful exploratory tester is always questioning what happens.

My first urge to leave the tour of the menus was when I found a dialog box in the program that allowed me to control the amount of memory used by the application. This immediately gave me an idea (sudden ideas are valued in exploratory testing). Since stability is one of the requirements of the Windows Compatibility program, I thought it would be interesting to try to destabilize the product by setting it to use the minimum amount of memory, then ask it to perform memory-intensive functions. So I set the slider bar to use 5% of system memory, then visited the image properties settings and set the image size to 100 inches square. That's a big canvas. I filled the canvas with purple dots and went to the effects menu to try activating some special graphical effects.

Okay, here comes an important part: I chose a "ripple" effect from the menu and *bam*, the product immediately displayed an error message informing me that there was not enough memory for that operation. This is very interesting behavior because it establishes a standard. I have a new expectation from this point forward: A function should be able to prevent itself from executing if there is not enough memory to perform the operation. This is a perfect example of how, in exploratory testing, the result of one test influences the next, because I then proceeded to try other effects to see if the rest of them behaved in the same way. What did I find? None of the others I tried behaved that way. Instead, they would crank away for five minutes, doing nothing I could see other than driving the hard disk into fits. Eventually an error popped up ("Error -32: Sorry, this error is fatal.") and the application crashed.

This is a nice result, but I felt that the test wouldn't be complete (exploratory testers strive to anticipate questions that their clients will ask later on) unless I set that memory usage lever all the way up, to use the most memory possible. To my

surprise, instead of yielding the Error -32, the entire operating system froze. Windows 2000 is not supposed to do that. This was a far more serious problem than a mere crash.

At this point in the process, I had spent about thirty minutes of a four-hour process, and already found a problem that disqualified the application from compatibility certification. That was the good news. The bad news is that I lost my test notes when the system froze. After rebooting, I decided I had learned enough from stress testing and returned to the menu tour.

I submit that this test story is an example of disciplined, purposeful testing. I can report what I covered and what I found. I can relate the testing to the mission I was given. It was also quite repeatable, or at least as repeatable as most scripted tests, because at all times I followed a coherent idea of what I was trying to test and how I was trying to test it. The fact that those ideas occurred to me on the fly, rather than being fed to me from a document, is immaterial. I hope you see that this is a far cry from unsystematic testing.

## IMPROVING YOUR EXPLORATORY TESTING SKILLS

Let's look at eight key elements that distinguish an expert exploratory tester from an amateur, and some things you can do to get better at those skills.

1

**Test Design:** An exploratory tester is first and foremost a test designer. Anyone can design a test accidentally. The excellent exploratory tester is able to craft tests that systematically explore the product. Test design is a big subject, of course, but one way to approach it is to consider it a questioning process. To design a test is to craft a question that will reveal vital information about a product.

**To get better at this:** Go to a feature (something reasonably complex, like the table formatting feature of your favorite word processor) and ask thirty questions about it that you can answer, in whole or

part, by performing some test activity, by which I mean some test, set of tests, or task that creates tests. Identify that activity along with each question. If you can't find thirty questions that are substantially different from each other, then perform a few tests and try again. Notice how what you experience with the product gives you more questions.

Another aspect of test design is making models. Each model suggests different tests. There are lots of books on modeling (you might try a book on UML, for instance). Pick a kind of model, such as a flowchart, data flow diagram, truth table, or state diagram, and create that kind of model representing a feature you are testing. When you can make such models on napkins or whiteboards in two minutes or less, confidently and without hesitation, you will find that you also are more confident at designing tests without hesitation.

2

**Careful Observation:** Excellent exploratory testers are more careful observers than novices, or for that matter, experienced scripted testers. The scripted tester need observe only what the script tells him to observe. The exploratory tester must watch for *anything* unusual, mysterious, or otherwise relevant to the testing. Exploratory testers must also be careful to distinguish observation from inference, even under pressure, lest they allow preconceived assumptions to blind them to important tests or product behavior.

**To get better at this:** Try watching another tester test something you've already tested, and notice what they see that you didn't see first. Notice how they see things that you don't and vice versa. Ask yourself why you didn't see everything. Another thing you can do is to videotape the screen while you test, or use a product like Spector that takes screen shots every second. Periodically review the last fifteen minutes of your testing, and see if you notice anything new.

Or try this: describe a screen in writing to someone else and have them draw

## Excellent exploratory testers are able to review and explain their logic, looking for errors in their own thinking.

the screen from your description. Continue until you can draw each other's screens. Ideally, do this with multiple people, so that you aren't merely getting better at speaking to one person.

To distinguish observation from inference, make some observations about a product, write them down, and then ask yourself, for each one, did you actually see that, or are you merely inferring it? For instance, when I load a file in Microsoft Word, I might be tempted to say that I witnessed the file loading, but I didn't really. The truth is I saw certain things, such as the appearance of words on the screen that I recall being in that file, and I take those things to be evidence that the file was properly loaded. In fact, the file may not have loaded correctly at all. It might be corrupted in some way I have not yet detected.

Another way to explore observation and inference is to watch stage magic. Even better, learn to perform stage mag-

ic. Every magic trick works in part by exploiting mistakes we make when we draw inferences from observations. By being fooled by a magic trick, then learning how it works, I get insight into how I might be fooled by software.

### 3

**Critical Thinking:** Excellent exploratory testers are able to review and explain their logic, looking for errors in their own thinking. This is especially important when reporting the status of a session of exploratory tests, or investigating a defect.

**To get better at this:** Pick a test that you recently performed. Ask what question was at the root of that test. What was it really trying to discover? Then think of a way that you could get a test result that pointed you in one direction

(e.g., program broken in a certain way) when reality is in the opposite direction (e.g., program not broken, what you're seeing is the side effect of an option setting elsewhere in the program, or a configuration problem). Is it possible for the test to appear to fail even though the product works perfectly? Is it possible for the product to be deeply broken even though the test appeared to pass? I can think of three major ways this could happen: inadequate coverage, inadequate oracle, or tester error.

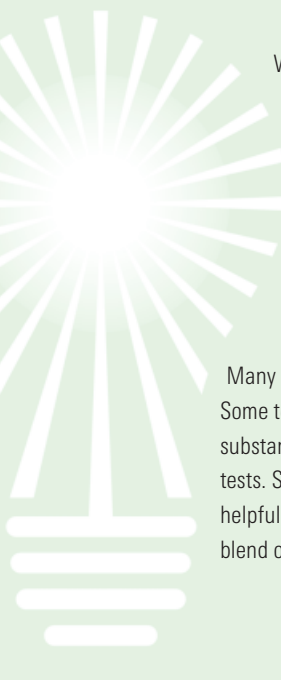
**Inadequate coverage** means that your test doesn't touch enough of the product to fulfill its goal. (Maybe you have tested printing, but not enough different printing situations to justify confidence that the print function works.) Oracles are mechanisms or principles for recognizing a problem if it occurred. **Inadequate oracle**, then, means you used a weak method of determining whether a bug is present, and that led either to reporting something that isn't a problem or failing to notice something that is a problem. (Maybe you printed something to a file, and you verified that the file was created, but you didn't check the contents of the file.) **Tester error** means that your test design was fine, but you simply did not notice something that happened, or used the wrong data, failed to set up the system properly for testing, etc. (Maybe you saw that the print-out looked correct, but it later turned out that you were looking at the results of a different test.)

Since testing is basically an infinite process, all real-life testing involves compromises. Thus, you should be able to find many ways your tests could be fooled. The idea is to maintain awareness about the limitations of your testing. For a typical complex product, it takes lots of different tests to answer any given question with high confidence.

### 4

**Diverse Ideas:** Excellent exploratory testers produce more and better ideas than novices. They may make use of heuristics to accomplish this. Heuristics are mental devices such as guidelines, generic checklists, mnemonics, or rules of

## SCRIPTED TESTING IS THE OPPOSITE OF EXPLORATORY TESTING



When I say "scripted test," I mean a set of instructions for executing a test. In scripted testing, tests are specified in advance of test execution. Tests are not changed while execution is underway, nor are they routinely changed based on any learning that might happen while performing the test. A fully scripted test informs the tester what to do and what to look for. If a test tells you exactly what to do, but does not tell you how specifically to know if a bug has occurred, then that test is to some extent relying on your ability to think for yourself; to explore.

Actually, pure scripted testing, executed by humans, is not very common. Many scripts intended to be performed by hand are not specified in great detail. Some test cases are specified in just a few words, such as "try long inputs." It requires substantial judgment and background knowledge to turn those test cases into real tests. So, even though exploratory testing and scripted testing are opposites, it's more helpful to see them as opposite ends of a spectrum. Most real-life tests are some blend of exploratory and scripted behavior.

thumb. The Satisfice Heuristic Test Strategy Model (<http://www.satisfice.com/tools/satisfice-tsm-4p.pdf>) is an example of a set of heuristics for rapid generation of diverse ideas. James Whittaker and Alan Jorgensen's "17 attacks" is another (see *How to Break Software*).

**To get better at this:** Practice using the Heuristic Test Strategy Model. Try it out on a feature of some product you want to test. Go down the lists of ideas in the model, and for each one think of a way to test that feature in some way related to that idea. Novices often have a lot of trouble doing this. I think that's because the lists work mainly by pattern matching on past experience. Expert testers see something in the strategy model that triggers the memory of a kind of testing or a kind of bug, and then they apply that memory to the thing they are testing today. The ideas in the model overlap, but they each bring something unique, too.

Another exercise I recommend is to write down, off the top of your head, twenty different ways to test a product. You must be able to say how each idea is unique among the other ideas. Because I have memorized the heuristic test strategy model, when I am asked this question, I can list thirty-three different ways to test. I say to myself "CITESTDSFDPOCRUSPICSTMPLESDFSFCURR" and then expand each letter. For instance, the second letter stands for information, which represents the idea "find every source of information I can about this feature and compare them to each other and to the product, looking for inconsistencies." The "O" stands for operations, which represents the idea "discover the environment in which the product will be used, and reproduce that environment as close as I can for testing."

## 5

**Rich Resources:** Excellent exploratory testers build a deep inventory of tools, information sources, test data, and friends to draw upon. While testing, they remain alert for opportunities to apply those resources to the testing at hand.

**To get better at this:** Go to a shareware site, such as Download.com, and review the utilities section. Think about how you might use each utility as a test tool. Visit the Web sites related to each technology you are testing and look for tutorials or

white papers. Make lots of friends, so you can call upon them to help you when you need a skill they have.

## 6

**Self-Management:** Excellent exploratory testers manage the value of their own time. They must be able to tell the difference between a dead end and a promising lead. They must be able to relate their work to their mission and choose among the many possible tasks to be done.

**To get better at this:** Set yourself a charter to test something for an hour. The charter could be a single phrase like "test error handling in the report generator." Set an alarm to go off every fifteen minutes. Each time the alarm goes off, say out loud why you are doing whatever you are doing at that exact moment. Justify it. Say specifically how it relates to your charter. If it is off-charter, say why you broke away from the charter and whether that was a well-made decision.

## 7

**Rapid Learning:** Excellent exploratory testers climb learning curves more quickly than most. Intelligence helps, of course, but this, too, is a matter of skill and practice. It's also a matter of confidence—having faith that no matter how complex and difficult a technology looks at first, you will be able to learn what you need to know to test it.

**To get better at this:** Go to a bookstore. Pick a computer book at random. Flip through it in five minutes or less, then close the book and answer these questions: what does this technology do, why would anyone care, how does it work, and what's an example of it in action? If you can't answer any of those questions, then open the book again and find the answer.

## 8

**Status Reporting:** Tap an excellent exploratory tester on the shoulder at any time and ask, "What is your status?" The tester will be able to tell you what was tested, what test techniques and data were used, what mechanisms were used to detect problems if they occurred, what risks

the tests were intended to explore, and how that related to the mission of testing.

**To get better at this:** Do a thirty-minute testing drill. Pick a feature and test it. At the end of exactly thirty minutes, stop. Then without the use of notes, say out loud what you tested, what oracles you used, what problems you found, and what obstacles you faced. In other words, make a test report. As a variation, give yourself ten minutes to write down the report.

## WHAT EXPLORATORY EXPERTISE FEELS LIKE, INSIDE YOUR HEAD

In *Operating Manual for Spaceship Earth*, Buckminster Fuller wrote, "Real wealth only increases." He was speaking of knowledge. I often think about that line when I'm testing, because when I sit down to explore, I don't know what's going to happen. But I do know that I will definitely learn something interesting and probably important. Good things will happen. My wealth will only increase.

At a certain point in your practice, you will begin to feel a calm confidence that the process works. Your mind *will* respond. You *will* notice things. Potent tests *will* occur to you and you will catch many important bugs. You may not always be able to articulate what you're doing and how it works, but you will be able to demonstrate it, and suggest specific exercises that help other people learn how to do it, too. **STQE**

*James Bach owns Satisfice, Inc. (www.satisfice.com), a consulting and training company specializing in rapid software testing techniques. He is the author, with Cem Kaner and Bret Pettichord, of Lessons Learned in Software Testing: A Context-Driven Approach.*

**AUTHOR'S NOTE:** *Learning and improvement is a lifelong process. I am grateful to my "rich resources" Cem Kaner and Brian Marick for their criticisms and suggested fixes for this article.*

## STICKY NOTES

For more on the following topics go to [www.stqemagazine.com](http://www.stqemagazine.com)

- Satisfice Heuristic Test Strategy Model
- James Whittaker and Alan Jorgensen's "17 attacks"