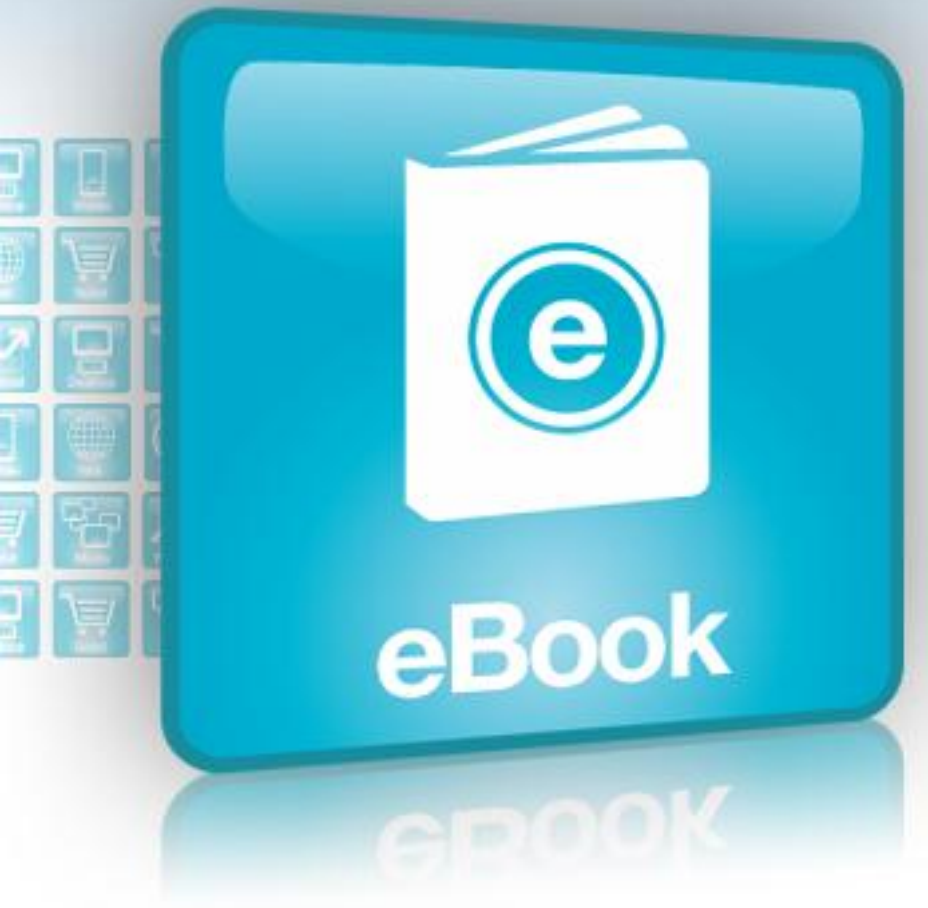
uTest®
Where the best test

Fn Functional Sc Security Ld Load Ln Localization Ux Usability
In-The-Wild Testing for Functional + Security + Load + Localization + Usability



5 Ways To Revolutionize Your QA

By Dr. James Whittaker

A Free Book from uTest

Revolutionize Your QA

It's serious ship mode time for my team here at Microsoft. Everyone is heads-down working our product toward its upcoming beta 1 release. In times like these, people often lose sight of the big picture and concentrate solely on the day-to-day details of finding and resolving bugs. Always being one to buck such trends, I've come up for air and am offering five insights that will revolutionize the way you test and, I hope, make your team more effective:

Insight 1: There are two types of code and they require different types of tests

Insight 2: Take your testing down a level from features to capabilities

Insight 3: Take your testing up a level from test cases to techniques

Insight 4: Improving development is your top priority

Insight 5: Testing without innovation is a great way to lose talent

Insight 1: There are two types of code and they require different types of tests

Screws and nails: two types of fasteners that carpenters use to bond pieces of wood. A carpenter who uses a hammer on a screw wouldn't last long as a professional furniture maker. A hammer drives a nail and a screwdriver a screw. A good carpenter knows when to wield the right tool.

But what is simple for the carpenter is commonly a problem for the software tester. To demonstrate, here's a note I received from a friend (who works for another company):

> *"I was just in test review yesterday and they were seeing diminishing returns on functional automation testing. They were experiencing an inflection point where the bugs are now mostly about behavioral problems and integration with other systems. Their tests were becoming less useful and [it was] harder to investigate the failures. My guess was that the only way to solve this was heavy instrumentation (that could produce an intelligent bug report based on a hell of a lot of state/historical context)."*

Inflection point indeed. I've seen this same inflection point on a number of products here. **When automation stops finding bugs, the tempting conclusion is that quality is high**. Clearly, my friend above didn't make that mistake. Too much faith in automation is one of the problems I believe we suffered from with Windows Vista. The automation stopped finding bugs and we took that as good news.

Even if it is good news, it is only good for part of your code. For the other part, it's useless. This brings me to the point of this section: there are two types of code in every product and the testing concerns, methods and practices are different for each of them.

The first type of code is what I call **experience code**. It's the code that implements the functionality that attracted users to the application in the first place. It provides the user with the *experience**** they are seeking. For a word processor, it is the code that allows a user to type, format and print a document. For an email application, it's the code that allows reading, replying and managing of messages. For a mobile phone, it's the ability to make and receive calls. For my own product in Visual Studio, it's the code that allows testers to write, manipulate and execute test cases.

The second type of code is **infrastructure code**, that is, the code that makes software run on its intended platform (cloud, desktop, phone, etc.) and interoperate with various devices, peripherals, languages and fonts. It is all the input and output code, memory, file system, network, runtime and glue code that makes all the moving parts work together.

For my Visual Studio product, it is the connection to the Team Foundation Server that stores test cases, project data and the various Data Collectors that extract information from the environment where the 'app under' test is executed.

**This is often referred to as end-user behavior, functionality, features and so forth but I think experience is the best encapsulating term. Feel free to think of it in these other terms if you find them more to your liking*.

This is code that a human user generally does not see, as it executes invisibly and only announces its presence when it fails. Infrastructure code can't generally be executed directly by a user. Rather, it responds to changing environmental situations, stress, failure, load and so forth. Thus, its testing is tailor made to be automated.

If my friend above is still trying to find infrastructure bugs, then he may have a point about adding new instrumentation. However, I think he'd be a lot better off with the cheaper solution of just injecting variation in his existing automation. He's run into Beizer's *pesticide paradox,* where testing kills some bugs, but like actual pesticide, other bugs become resistant. One needs to change the chemical formula to kill more insects. For testers, this means injecting variation.

If my friend is trying to find user experience bugs, then I'd suggest ignoring the automation altogether and putting some manual testers on the case. Business logic bugs require putting eyes on screens and fingers on keyboards, and what I like to call a *brain-in-the-loop*.

In short, **automation is simply overmatched by the human tester when it comes to analyzing subtle behaviors and judging user experience**.

THE PESTICIDE PARADOX:

"…where testing kills some bugs, but like actual pesticide, other bugs become resistant. One needs to change the chemical formula to kill more insects. For testers, this means injecting variation."

BOTTOMLINE:

"We must understand the difference between experience code and infrastructure code and use the right technique for the job. Otherwise, the pesticide paradox will bedevil our testing efforts. Like the carpenter, we must use the right tool for the right job."

Insight 2: Take your testing down a level from features to capabilities

Once you pick the right way to test and avoid the pesticide paradox, my second insight will help you get more out of your actual testing. One of my pet peeves is so-called *feature testing*. Features are far too high-level to act as good testing goals, yet so often a testing task is partitioned by feature. Assign a feature to a tester, rinse and repeat.

There's a lot of risk in testing features in isolation. We need to be more proactive about decomposing a feature into something more useful and understanding how outside influences affect that feature.

I push teams to take testing to a lower level and concentrate on what I call *capabilities*. At Microsoft we do an exercise called *Component – Feature – Capability Analysis* to accomplish this. The purpose is to understand more precisely the testable capabilities of a feature and to identify important interfaces where features interact with each other and external components. Once these are understood, then testing becomes the task of selecting environment and input variations that cover the primary cases.

Here's a snapshot outline of what Component – Feature – Capability looks like for the product I am working on for Visual Studio (*continued on next page*):

- Main shell
  - Context (TFS) (red indicates an external dependency)
    - Add a new TFS Server
    - Connect to a TFS Server
    - Delete a TFS Server
  - Open Items
    - Verify that only savable activities show up here
    - Verify that you can navigate to them
- Testing Center
  - Plan
    - Contents
      - Add new static suites
      - Add new query-based suites
      - ...
      - Refresh
    - Properties (current test plan)
      - Modify iteration
      - Modify/create new manual test setting
      - ...
      - Modify start/end dates
    - Plan Manager
      - Select a current test plan
      - Create a new test plan
      - ...
      - Refresh test plan manager
  - Test
    - Run tests
      - Run all tests on a suite (automated and/or manual or mixed)
      - Run tests with a specific environment chosen

Insight 2: Take your testing down a level from features to capabilities

Obviously, there's a lot missing from this, as the use of ellipses shows. This is also only a subset of our feature set, but it gives you an idea of how the analysis is performed:

1. List the components
2. Decompose components into features
3. Decompose the features into capabilities
4. Keep decomposing until the capabilities are simple

Our product is very feature rich (it's a test case management system for manual testing) and this analysis took a total of about two hours during a three-person brainstorm. The entire analysis document serves as a guide to the manual tester or to the automation writer to determine how to test each of the capabilities.

I have some suggestions about how to go about testing these capabilities and that is the subject of the next insight.

Insight 3: Take your testing up a level from test cases to techniques

Test cases are a common unit of measurement in the testing world. We count the number of them that we run, and when we find a bug, we proudly present the test case that we used to find it.

I hate it when we do that.

I think test cases - and most discussions about them - are generally meaningless. I propose we talk in more high-level terms about test techniques instead. For example, when one of my testers finds a bug, they often come to my office to demo it to me (I am a well-known connoisseur of bugs and these demos are often reused in lectures I give around campus).

Most of them have learned not to just repro the bug by re-running the test case. They know I am not interested in the bug per se, but the context of its exposure. These are the type of questions I like to ask during bug demos:

- What made you think of this particular test case?
- What was your goal when you ran that test case?
- Did you notice anything interesting while running the test case that changed what you were doing?
- At what point did you know you had found a bug?

These questions represent a higher level of discussion and one we can learn something from. The test case itself lost its value when it found the bug, but when we merge the test case with its intent, context, strategy and the bug it found, we have something that is far more than a single test case that evokes a single point of failure.

We've created the basis for understanding why the test case was successful and we've implied any number of additional test cases that could be derived from this one successful instance.

At Microsoft, we have begun using the concept of *test tours* as a way to categorize test techniques. This is an even higher level discussion and we are working on documenting the tours and creating experience reports based on their use within the company.

Stay tuned for more information on this effort…

JW ON TEST CASES:

"Test cases are a common unit of measurement in the testing world. We count the number of them that we run, and when we find a bug, we proudly present the test case that we used to find it…I hate it when we do that."

Insight 4: Improving development is your top priority

I had a lunch conversation recently with a test trainer at Microsoft who encourages testers to get more involved in development. He cited cases where testers would recognize weakness in code and suggest new design patterns or code constructs. Both myself and our other lunch partner (also a tester) were skeptical.

Testers are hired to test, not to develop, and most teams I've worked with both inside and especially outside of Microsoft would see such behavior as meddling. So, if testers aren't contributing directly to the code, then are we simply bug finders whose value is determined by what we take *out* of the software?

Well that is certainly one purpose of testing: to find and remove bugs, but there is a broader purpose too. **It is our task to use testing as an instrument of improvement.**

This is where the techniques and tours discussed in the prior section can help. By raising the level of abstraction in our primary test artifact, we can use this to create new avenues of communication with development. In my group we regularly review test strategy, techniques and tours with developers. It's our way of showing them what their code will face once we get a hold of it.

Of course, their natural reaction is to write their code so it won't be susceptible to these attacks. Through such discussions, they often help us think of new ways to test. The discussion becomes one that makes them better developers and us better testers. We have to work harder in the sense that the tests we would have run won't find any bugs (because the developers have anticipated our tests) but this is work I am willing to invest in any day!

This is the true job of a tester: to make developers and development better. **We don't ensure better software - we enable developers to build better software**. It isn't about finding bugs, because the improvement caused is temporal.

The true measure of a great tester is that they find bugs, analyze them thoroughly, report them skillfully and end up creating a development team that *understands the gaps in their own skill and knowledge*.

JW ON THE PERCEIVED ROLE OF TESTERS:

"So, if testers aren't contributing directly to the code, then are we simply bug finders whose value is determined by what we take *out* of the software?"

The end result will be developer improvement and that will reduce the number of bugs and increase their productivity in ways that far exceeds simple bug removal.

This is a key point. It's software developers that build software and if we're just finding bugs and assisting their removal, no real lasting value is created. If we take our job seriously enough we'll ensure the way we go about it creates real and lasting improvement.

Making developers better, helping them understand failures and the factors that cause them will mean fewer bugs to find in the future. **Testers are quality gurus** and that means teaching those responsible for anti-quality what they are doing wrong and where they could improve.

The immortal words of Tony Hoare ring very true:

"The real value of tests is not that they detect bugs in the code, but that they detect inadequacies in the methods, concentration and skill of those who design and produce the code."

He said this in 1996. Clearly, he was way ahead of his time.

JW ON THE REAL ROLE OF TESTERS:

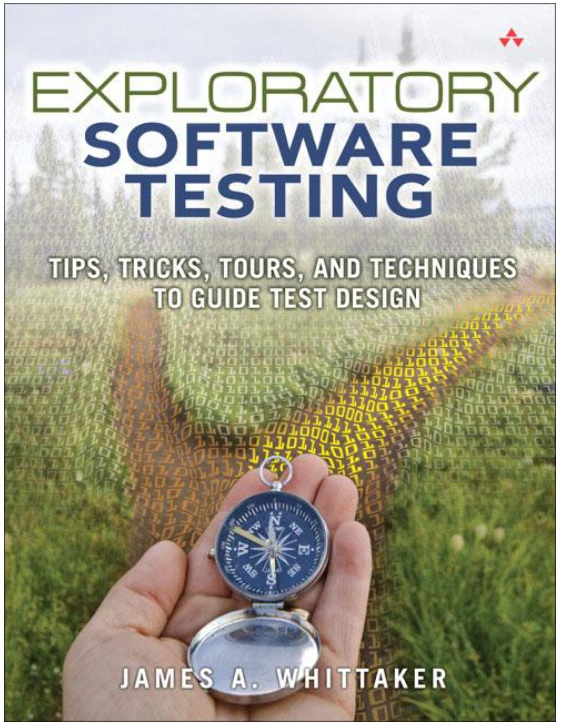
"This is the true job of a tester: to make developers and development better. We don't ensure better software - we enable developers to build better software."

Insight 5: Testing without innovation is a great way to lose talent

Testing sucks. Now let me qualify that statement: Running test cases over and over - in the hope that bugs will manifest - sucks. It's boring, uncreative work. And since half the world thinks that is all testing is about, it is no great wonder few people covet testing positions! Testing is either too tedious and repetitive or its downright too hard. Either way, who would want to stay in such a position?

What is interesting about testing is strategy, deciding what to test and how to combine multiple features and environmental consideration in a single test. The tactical part of testing, actually running test cases and logging bugs, is the least interesting part.

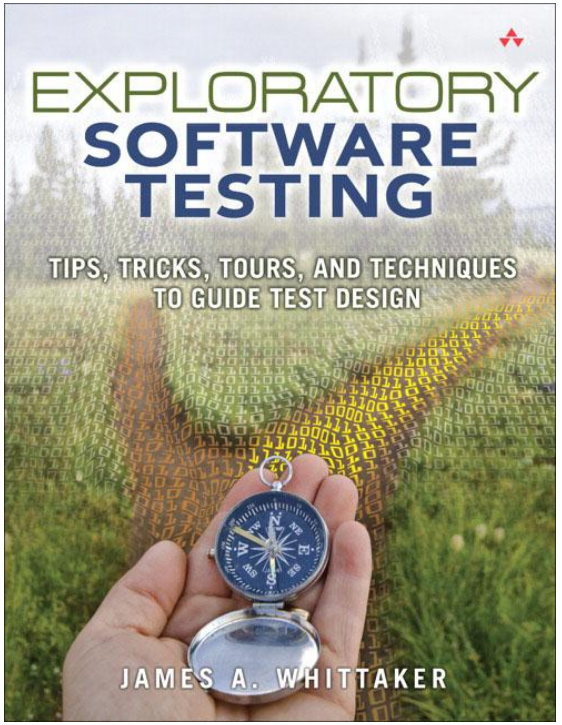
Smart test managers and test directors need to recognize this and ensure that every tester splits their time between strategy and tactics. Take the tedious and repetitive parts of the testing process and automate them. Tool development is a major creative task at Microsoft and is well-rewarded by the corporate culture.



Insight 5: Testing without innovation is a great way to lose talent

For the hard parts of the testing process like deciding what to test and determining test completeness, user scenarios and so forth, we have another creative and interesting task. Testers who spend time categorizing tests and developing strategy (the interesting part) are more focused on testing and thus spend less time running tests (the boring part).

Testing is an immature science. There are a lot of insights that a thinking person can make without inordinate effort. By ensuring that testers have the time to take a step back from their testing effort and find insights that will improve their testing, teams will benefit. Not are such insights liable to improve the overall quality of the test, but the creative time will improve the morale of the testers involved.



About Dr. James Whittaker

James A. Whittaker, now a Software Architect at Microsoft, has spent his career in software testing. He was an early thought leader in *model-based testing,* where his PhD dissertation from the University of Tennessee became a standard reference on the subject. While a professor at Florida Tech, he founded the world's largest academic software testing research center and helped make testing a degree track for undergraduates.

Before he left Florida Tech, his research group had grown to over 60 students and faculty and had secured over $12 million in research awards and contracts. During his tenure at FIT he wrote *How to Break Software* and the series follow-ups *How to Break Software Security* (with Hugh Thompson) and *How to Break Web Software* (with Mike Andrews). His research team also developed the highly acclaimed runtime fault injection tool *Holodeck* and marketed it through their startup Security Innovation, Inc.

Dr. Whittaker currently works at Microsoft as an architect for Visual Studio Team System where he is busy transforming his testing ideas into tools and techniques for developers and testers. He dreams of a future in which software just works.

For more information, visit his blog.

About uTest

uTest provides real-world testing services for web, desktop and mobile applications. By leveraging a community of more than 70,000 professional testers from 190 countries, uTest helps companies test their products under real-world conditions. Thousands of companies – from startups to global enterprises such as Google, Microsoft, AOL and Intuit – turn to uTest to complement the in-the-lab testing they do, and to help them launch better apps. uTest's services span the entire software development lifecycle, including functional, usability, security, localization and load testing.

The company is headquartered near Boston, with offices in Silicon Valley, London and Israel. uTest has raised more than $37MM in funding and consistently generates triple-digit annual revenue growth. uTest won the American Business Association's "Most Innovative Company of 2011" award, and was named a "Best Place to Work" by the Boston Business Journal two years in a row.

More information can be found at www.utest.com