User Acceptance Testing – A Context-Driven Perspective

Michael Bolton, DevelopSense
mb@developsense.com

Biography

Michael Bolton is the co-author (with senior author James Bach) of Rapid Software Testing, a course that presents a methodology and mindset for testing software expertly in uncertain conditions and under extreme time pressure.

A testing trainer and consultant, Michael has over 17 years of experience in the computer industry testing, developing, managing, and writing about software. He is the founder of DevelopSense, a Toronto-based consultancy. He was with Quarterdeck Corporation for eight years, during which he delivered the company's flagship products and directed project and testing teams both in-house and around the world.

Michael has been teaching software testing around the world for eight years. He was an invited participant at the 2003, 2005, 2006, and 2007 Workshops on Teaching Software Testing in Melbourne and Palm Bay, Florida; was a member of the first Exploratory Testing Research Summit in 2006. He is also the Program Chair for TASSQ, the Toronto Association of System and Software Quality, and a co-founder of the Toronto Workshops on Software Testing. He has a regular column in Better Software Magazine, writes for Quality Software (the magazine published by TASSQ), and sporadically produces his own newsletter.

Michael lives in Toronto, Canada, with his wife and two children.

Michael can be reached at mb@developsense.com, or through his Web site, http://www.developsense.com

Abstract: Hang around a software development project for long enough and you'll hear two sentences: "We need to keep the customer satisfied," and "The Customer doesn't know what he wants." A more thoughtful approach might be to begin by asking a question: "Who IS the customer of the testing effort?"

The idiom *user acceptance testing* appears in many test plans, yet few outline what it means and what it requires. Is this because it's to everyone obvious what "user acceptance testing" means? Is because there is no effective difference between user acceptance testing and other testing activities? Or might it be that there are so many possible interpretations of what might constitute "user acceptance testing" that the term is effectively meaningless?

In this one-hour presentation, Michael Bolton will establish that there is far more to those questions than many testing groups consider. He doesn't think that "user acceptance testing" is meaningless if people using the words establish a contextual framework and understand what they mean by "user", by "acceptance", and by "testing". Michael will discuss the challenges of user acceptance testing, and propose some remedies that testers can use to help to clarify user requirements--and meet them successfully.

User Acceptance Testing in Context

A couple of years ago, I worked with an organization that produces software that provides services to a number of large banks. The services were developed under an agile model. On the face of it, the applications did not seem terribly complicated, but in operation they involved thousands of transactions of some quantity of money each, they bridged custom software at each bank that was fundamentally different, and they needed to defend against fraud and privacy theft. The team created user stories to describe functionality, and user acceptance tests—fairly straightforward and easy to pass—as examples of that functionality. These user acceptance tests were not merely examples; they were also milestones. When all of the user acceptance tests passed, a unit of work was deemed to be done. When all of the user stories associated with the current project were finished development and testing, the project entered a different phase called "user acceptance testing". This phase took a month of work in-house, and was characterized by a change of focus, in which the testing group performed harsh, complex, aggressive tests and the developers worked primarily in support of the testing group (rather than the other way around) by writing new test code and fixing newly found problems. Then there was a month or so of testing at the banks that used the software, performed by the banks' testers; *that* phase was called "user acceptance testing."

So what is User Acceptance Testing anyway? To paraphrase Gertrude Stein, is there any *there* there?

The answer is that there are many potential definitions of user acceptance testing. Here are just a few, culled from articles, conversation with clients and other testers, and mailing list and forum conversations.

- the last stage of testing before shipping
- tests to a standard of compliance with requirements, based on specific examples
- a set of tests that are run, for a customer, to demonstrate functionality
- a set of tests that are run, *by* a customer, to demonstrate functionality
- not tests at all, but a slam-dunk demo
- outside beta testing
- prescribed tests that absolutely must pass before the user will take the product happily
- prescribed tests that absolutely must pass as a stipulation in a contract
- any testing that is not done by a developer
- tests that are done by real users
- tests that are done by stand-ins or surrogates for real users
- in Agile projects, prescribed tests (often automated) that mark a code-complete milestone
- in Agile projects, tests (often automated) that act as examples of intended functionality; that is, tests as requirements documentation

Words (like "user", "acceptance", and "testing") are fundamentally ambiguous, especially when they are combined into idioms (like "user acceptance testing"). People all have different points of view that are rooted in their own cultures, circumstances and experiences. If we are to do any kind of testing well, it is vital to begin by gaining understanding of the ways in which other people, even though they sound alike, might be saying and thinking profoundly different things.

Resolving the possible conflicts requires critical thinking, context-driven thinking, and general semantics: we must ask the questions “what do we mean” and “how do we know?” By doing this kind of analysis, we adapt usefully to the changing contexts in which we work; we defend ourselves from being fooled; we help to prevent certain kinds of disasters, both for our organizations and for ourselves. These disasters include everything from loss of life due to inadequate or inappropriate testing, or merely being thought a fool for using approaches that aren’t appropriate to the context. The alternative—understanding the importance of recognizing and applying context-driven thinking—is to have credibility, capability and confidence to apply skills and tools that will help us solve real problems for our managers and our customers.

In 2002, with the publication of Lessons Learned in Software Testing, the authors (Kaner, Bach, and Pettichord) declared a testing community called the Context-Driven School, with these principles:

The Basic Principles of the Context-Driven School

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project's context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn't solved, the product doesn't work.
6. Good software testing is a challenging intellectual process.
7. Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products

For context-driven testers, a discussion of user acceptance testing hinges on identifying aspects of the context: the problem to be solved; the people who are involved; the practices, techniques, and approaches that we might choose.

In any testing project, there are many members of the project community who might be customers of the testing mission[1]. Some of these people include:

- The contracting authority
- The holder of the purse strings
- The legal or regulatory authority
- The development manager
- The test manager
- The test lead
- Technical Support
- Sales people
- Sales support
- Marketing people
- The shareholders of the company
- The CEO

1 Here’s a useful way to think of this, by the way: in your head, walk through your company’s offices and buildings. Think of everyone who works in each one of those rooms—have you identified a different role?

- Testers
- Developers
- The department manager for the people who are using the software
- Documenters
- The end-user's line manager
- The end-user
- The end-user's customers[2]
- Business analysts
- Architects
- Content providers
- The CFO
- The IT manager
- Network administrators and internal support
- Security personnel
- Production
- Graphic designers
- Development managers for other projects
- Designers
- Release control
- Strategic partners

Any one of these could be the user in a user acceptance test; several of these could be providing the item to be tested; several could be mandating the testing; and several could be performing the testing. The next piece of the puzzle is to ask the relevant questions:

- Which people are offering the item to be tested?
- Who are the people accepting it?
- Who are the people who have mandated the testing?
- Who is doing the testing?

With thirty possible project roles (there may be more), times four possible roles within the acceptance test (into each of which multiple groups may fall), we have a huge number of potential interaction models for a UAT project. Moreover, some of these roles have different (and sometimes competing) motivations. Just in terms of who's doing what, there are too many possible models of user acceptance testing to hold in your mind without asking some important context-driven questions for each project that you're on.

What is Testing?

I'd like to continue our thinking about UAT by considering what testing itself is. James Bach and I say that testing is:

- Questioning the product in order to evaluate it.[3]

Cem Kaner says

- Testing is an empirical, technical investigation of a product, done on behalf of stakeholders, with the intention of revealing quality-related information of the kind that they seek. (citation to QAI November 2006)

---

[2] The end-user of the application might be a bank teller; problems in a teller application have an impact on the bank's customers in addition to the impact on the teller.

[3] James Bach and Michael Bolton, *Rapid Software Testing*, available at http://www.satisfice.com/rst.pdf

Kaner also says something that I believe is so important that I should quote it at length. He takes issue with the notion of testing as confirmation over the vision of testing as investigation, when he says:

> The confirmatory tester knows what the "good" result is and is trying to find proof that the product conforms to that result. The investigator wants to see what will happen and is expecting to learn something new from the test. The investigator doesn't necessarily know how a test will come out, how a line of tests will come out or even whether the line is worth spending much time on. It's a different mindset.[4]

I think this distinction is crucial as we consider some of the different interpretations of user acceptance testing, because some in some cases, UAT follows an investigative path, and other cases it takes a more confirmatory path.

What are the motivations for testing?

Kaner's list of possible motivations for testing includes

- Finding defects
- Maximizing bug count
- Blocking premature product releases
- Helping managers make ship / no-ship decisions
- Minimizing technical support costs
- Assessing conformance to specification
- Assessing conformance to regulations
- Minimizing safety-related lawsuit risk
- Finding safe scenarios for use of the product (workarounds that make the product potentially tolerable, in spite of the bugs)
- Assessing quality
- Verifying the correctness of the product

I would add

- assessing compatibility with other products or systems
- assessing readiness for internal deployment
- ensuring that that which used to work still works, and
- design-oriented testing, such as review or test-driven development.

Finally, I would add the idea of "tests" that are not really tests at all, such as a demonstration of a bug for a developer, a ceremonial demonstration for a customer, or executing a set of steps at a trade show. Naturally, this list is not exhaustive; there are plenty of other potential motivations for testing

---

[4] Kaner, Cem, *The Ongoing Revolution in Software Testing*. http://www.kaner.com/pdfs/TheOngoingRevolution.pdf, PNSQC, 2004

What is Acceptance?

Now that we've looked at testing, let's look at the notion of *acceptance*.

In *Testing Computer Software*, Cem Kaner, Hung Nguyen, and Jack Falk talk about acceptance testing as something that the test team does as it accepts a build from the developers. The point of this kind of testing is to make sure that the product is acceptable *to the testing team*, with the goal of making sure that the product is stable enough to be tested. It's a short test of mainstream functions with mainstream data. Note that the expression *user* acceptance testing doesn't appear in TCS, which is the best-selling book on software testing in history.

Lessons Learned in Software Testing, on which Kaner was the senior author with James Bach and Brett Pettichord, neither the term "acceptance test" nor "user acceptance test" appears at all. Neither term seems to appear in Black Box Software Testing, by Boris Beizer. Beizer uses "acceptance test" several times in Software Testing Techniques, but doesn't mention what he means by it.

Perry and Rice, in their book *Surviving the Top Ten Challenge of Software Testing*, say that "Users should be most concerned with validating that the system will support the needs of the organization. The question to be answered by user acceptance testing is 'will the system meet the business or operational needs in the real world?'". But what kind of testing *isn't* fundamentally about that? Thus, in what way is there anything special about user acceptance testing? Perry and Rice add that user acceptance testing includes "Identifying all the business processes to be tested; decomposing these processes to the lowest level of complexity, and testing real-life test cases (people or things *(?)*) through those processes."

Finally, they beg the question by saying, "the nuts and bolts of user acceptance test is (sic) beyond the scope of this book."

Without a prevailing definition in the literature, I offer this definition: *Acceptance testing is any testing done by one party for the purpose accepting another party's work*. It's whatever the tester and the acceptor agree upon; whatever the key is to open the gate for acceptance—however secure or ramshackle the lock.

In this light, user acceptance testing could appear at any point on a continuum, with probing, investigative tests at one end, and softball, confirmatory tests at the other.

User Acceptance Testing as Ceremony

In some cases, UAT is not testing at all, but a ceremony. In front of a customer, someone operates the software, without investigation, sometimes even without confirmation. Probing tests have been run before; this thing called a user acceptance test is a feel-good exercise. No one is obliged to be critical in such a circumstance; in fact, they're required to take the opposite position, lest they be tarred with the brush of *not being a team player*. This brings us to an observation about expertise that might be surprising: for this kind of dog and pony show, the expert tester shows his expertise by *never finding a bug*.

For example, when the Queen inspects the troops, does anyone expect her to perform an actual inspection? Does she behave like a drill sergeant, checking for errant facial hairs? Does she ask a soldier to disassemble his gun so that she can look down the barrel of it? In this circumstance, the inspection is ceremonial. It's not a fact-finding mission; it's a stroll. We might call that kind of inspection a formality, or pro forma, or ceremonial, or perfunctory, or ritual; the point is that it's not an investigation at all.

User Acceptance Testing as Demonstration

Consider the case of a test drive for a new car. Often the customer has made up his mind to purchase the car, and the object is to familiarize himself with the vehicle and to confirm the wisdom of his choice. Neither the salesman nor the customer wants problems to be found; that would be a disaster. In the case, the "test" is a mostly ceremonial part of an otherwise arduous process, and everyone actively uninterested in finding problems and just wants to be happy. It's a feel-good occasion.

This again emphasizes the idea of a user acceptance test as a formality, a ceremony or demonstration, performed after all of the regular testing has been done. I'm not saying that this is a bad thing, by the way—just that if there's any disconnect between expectations and execution, there will be trouble—especially if the tester, by some catastrophe, actually does some investigative testing and finds a bug.

User Acceptance Testing as Smoke Test

As noted above, Kaner, Falk, and Nguyen refer to acceptance testing as a checkpoint such that the testers accept or reject a build from the developers. Whether performed by automation or by a human tester, this form of testing is relatively quick and light, with the intention of determining whether the build is complete and robust enough for further testing.

On an agile project, the typical scenario for this kind of testing is to have "user acceptance tests" run continuously or at any time, often via automated scripts. This kind of testing is by its nature entirely confirmatory unless and until a human tester gets involved again.

User Acceptance Testing as Mild Exercise

Another kind of user acceptance testing is more than a ceremony, and more than just a build verification script. Instead, it's a hoop through which the product must jump in order to pass, typically performed at a very late stage in the process, and usually involving some kind of demonstration of basic functionality that an actual user might perform. Sometimes a real user runs the program; more often it's a representative of a real user from the purchasing organization. In other cases, the seller's people—a salesperson, a product manager, a development manager, or even a tester—walk through some user stories with the buyer watching. This kind of testing is essentially confirmatory in nature; it's more than a demo, but less than a really thorough look at the product.

The object of this game is still that Party B is supposed to accept that which is being offered by Party A. In this kind of user acceptance testing, there may be an opportunity for B to raise concerns or to object in some other way. One of the assumptions of this variety seems to be that the users are seeing the application for the first time, or perhaps for the first time since they saw the prototype. At this stage, we're asking someone who is unlikely to have testing skills to find bugs hat they're unlikely to find, at the very time when we're least likely to fix them. A fundamental restructuring of the GUI or the back-end logic is out of the question, no matter how clunky it may be, so long as it barely fits the user's requirements. If the problem is one that requires no thinking, no serious development work, and no real testing effort to fix, it *might* get fixed. That's because every change is a risk; when we change the software late in the game, we risk throwing away a lot that we know about the product's quality. Easy changes, typos and such, are potentially palatable. The only other kind of problem that will be addressed at this stage is the opposite extreme—the one that's so overwhelmingly bad that the product couldn't possibly ship. Needless to say, this is a bad time to find this kind of problem.

It's almost worse, though, to find the middle ground bugs—the mundane, workaday kinds of problems that one would hope to be found earlier, that will irritate customers and that really do need to be fixed. These problems will tend to cause contention and agonized debate of a kind that neither of the other two extremes would cause, and that costs time.

There are a couple of preventative strategies for this catastrophe. One is to involve the user continuously in the development effort and the project community, as the promoters of the Agile movement suggest. Agilists haven't solved the problem completely, but they have been taking some steps in some good directions, and involving the user closely is a noble goal. In our shop, although our business analyst not sitting in the Development bearpit, as eXtreme Programming recommends, she's close at hand, on the same floor. And we try to make sure that she's at the daily standup meetings. The bridging of understanding and the mutual adjustment of expectations between the developers and the business is much easier, and can happen much earlier in this way of working, and that's good.

Another antidote to the problem of finding bad bugs too late in the game—although rather more difficult to pull off successfully or quickly—is to improve your testing generally. User stories are nice, but they form a pretty weak basis for testing. That's because, in my experience, they tend to be simple, atomic tasks; they tend to exercise happy workflows and downplay error conditions and exception handling; they tend to pay a lot of attention to capability, and not to the other quality criteria—reliability, usability, scalability, performance, installability, compatibility, supportability, testability, maintainability, portability, and localizability. Teach testers more about critical thinking and about systems thinking, about science and the scientific method. Show them bugs, talk about how those bugs were found, and the techniques that found them. Emphasize the critical thinking part: recognize the kinds of bugs that those techniques couldn't have found; and recognize the techniques that wouldn't find those bugs but that would find other bugs. Encourage them to consider those other "-ilities" beyond capability.

User Acceptance Testing as Usability Testing

User acceptance testing might be focused on usability. There is an important distinction to be made between ease of *learning* and ease of *use*. An application with a graphical user interface may provide excellent affordance—that is, it may expose its capabilities clearly to the user—but that affordance may require a compromise with efficiency, or constrain the options available to the user. Some programs are very solicitous and hold the user's hand, but like an obsessive parent, that can slow down and annoy the mature user. So: if your model for usability testing involves a short test cycle, consider that you're seeing the program for much less time than you (or the customers of your testing) will be using it. You won't necessarily have time to develop expertise with the program if it's a challenge to learn but easy to use, nor will you always be able to tell if the program is both hard to learn *and* hard to use. In addition, consider a wide variety of user models in a variety of roles—from trainees to experts to managers. Consider using personas, a technique for creating elaborate and motivating stories about users.[5]

User Acceptance Testing as Validation

In general, with confirmation, one bit of information is required to pass the test; in investigation, many bits of information are considered.

- "When a developer says 'it works', he really means 'it appears to fulfill some requirement to some degree.'" (One or more successes)
  - James Bach
- "When you hear someone say, 'It works,' immediately translate that into, 'We haven't tried very hard to make it fail, and we haven't been running it very long or under very diverse conditions, but so far we haven't seen any failures, though we haven't been looking too closely, either.' (Zero or more successes)
  - Jerry Weinberg

Validation seems to be used much more often when there is some kind of contractual model, where the product must pass a user acceptance test as a condition of sale. At the later stages, projects are often behind schedule, people are tired and grumpy, lots of bugs have been found and fixed, and there's lots of pressure to end the project, and a corresponding disincentive to find problems. At this point, the skilful tester faces a dilemma: should he look actively for problems (thereby annoying the client and his own organization should he find one), or should he be a team player?

My final take about the validation sense of UAT: when people describe it, they tend to talk about validating the requirements. There are two issues here. First, can you describe all of the requirements for your product? Can you? Once you've done that, can you test for them? Are the requirements all clear, complete, up to date? The context-driven school loves talking about requirements, and in particular, pointing out that there's a vast difference between requirements and requirements *documents*.

Second, shouldn't the requirements be validated as the software is being built? Any software development project that hasn't attempted to validate requirements up until a test cycle, late in the game, called "user acceptance testing" is likely to be in serious trouble, so I can't imagine that's what they mean. Here I agree with the Agilistas again—that it's helpful to validate requirements continuously throughout the project, and to adapt them when new information comes in and the context changes. Skilled testers can be a boon to the project when they supply new, useful information.

---

[5] Cooper, Alan, *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity*. Pearson Education, 2004.

User Acceptance Testing as Assigning Blame

There are some circumstances in which relations between the development organization and the customer are such that the customer actively wants to reject the software. There are all kinds of reasons for this; the customer might be trying to find someone to blame, and they want to show the vendor's malfeasance or incompetence to protect themselves from their own games of schedule chicken. This is testing as scapegoating; rather than a User Acceptance Test, it's more of a User Rejection Test. In this case, as in the last one, the tester is actively trying to find problems, so she'll challenge the software harshly to try to make it fail. This isn't a terribly healthy emotional environment, but context-driven thinking demands that we consider it.

User Acceptance Testing When The User is Other Software

There is yet another sense of the idea of UAT: that the most direct and frequent user of a piece of code is not a person, but other software. In *How to Break Software,* James Whittaker talks about a four-part user model, in which humans are only one part. The operating system, the file system, and application programming interfaces, or APIs, are potential users of the software too. Does your model of "the user" include that notion? It could be very important; humans can tolerate a lot of imprecision and ambiguity that software doesn't handle well.

User Acceptance Testing As Beta Testing

There's another model, not a contract-driven model, in which UAT is important. I mentioned DESQview earlier; I worked for Quarterdeck, the company that produced DESQview and other mass-market products such as QEMM and CleanSweep. Those of you with large amounts of gray in the beard will remember it. We didn't talk about user acceptance testing very much in the world of mass-market commercial software. Our issue was that there was no single user, so user acceptance testing wasn't our thing. Requirements traceability matrices don't come up there either. We *did* talk about beta testing, and we did some of that—or rather we got our users to do it. It took us a little while to recognize that we weren't getting a lot of return on our investment in time and effort. Users, in our experience, didn't have the skills or the motivation to test our product. They weren't getting paid to do it, their jobs didn't depend on it, they didn't have the focus, and they didn't have the time. Organizing them was a hassle, and we didn't get much worthwhile feedback, though we got some.

Microsoft regularly releases beta versions of its software (yes, I know, "and calls them releases"). Seriously, this form of user acceptance testing has yet another motivation: it's at least in part a marketing tool. It's at least in part designed to get customers interested in the software; to treat certain customers as an elite; to encourage early adopters. It doesn't do much for the testing of the product, but it's a sound marketing strategy.

One of the first papers that I wrote after leaving Quarterdeck addresses beta testing issues; you can find it on my Web site.

In those days, I was younger, and inexperienced at recognizing process traps. I read books and agonized and tried for ages to figure out how to implement UAT at Quarterdeck. It was a long time before I realized that we didn't need to do it; it didn't fit. This was a big lesson, and I hope I can impart it to some people: don't listen to any statement or proclamation—especially about process stuff, in my opinion—from someone that doesn't establish the context in which their advice might be expected to succeed or fail. Without a healthy dose of context, there's a risk of pouring effort or resources into things that don't matter, and ignoring things that do matter.

User Acceptance Tests as Examples

In the Agile world, it's becoming increasing popular to create requirements documents using a free tool called Fit (which works with Excel spreadsheets and Word documents) or Fitnesse (which works on a Wiki Web page). The basic idea is to describe requirements in natural language, and to supplement the descriptions with tables of data. These tables contain input data and references to specific functions in the program. Developers write code (called "fixtures") to link the function to the framework, which then feeds the tabular data to the application. The framework returns the results of the function and adds colour to the cells in the table—green for successes, and red for failures.

A number of Agilists call these User Acceptance Tests. I much prefer to take Brian Marick's perspective: the tables provide *examples* of expected behaviour much more than they test the software. This attempt to create a set of tools (tools that are free, by the way) that help add to a common understanding between developers and the business people is noble—but that's a design activity far more than a testing activity. That's fine when Fit or Fitnesse tests are examples, but sometimes they are misrepresented as tests. This leads to a more dangerous view…

User Acceptance Tests as Milestones

Fitnesse tests are sometime used as milestones for the completion of a body of work, to the extent that the development group can say "The code is ready to go when all of the Fitnesse tests run green." Ready to go—but *where*? At the point the Fitnesse stories are complete, the code is ready for some serious testing. I'd consider it a mistake to say that the code was ready for production. It's good to have a ruler, but it's important to note that rulers can be of differing lengths and differing precision. In my opinion, we need much more attention from human eyes on the monitor and human hands on the keyboard. Computers are exceedingly reliable, but the programs running on them may not be, and test automation is software. Moreover, computers don't have the capacity to recognize problems; they have to be very explicitly trained to look for them in very specific ways. They certainly don't have the imagination or cognitive skills to say, "What if?"

My personal jury is still out on Fitnesse. It's obviously a useful tool for recording test ideas and specific test data, and for rerunning them frequently. I often wonder how many of the repeated tests will find or prevent a bug. I think that when combined with an exploratory strategy, Fitnesse has some descriptive power, and provides some useful insurance, or "change detectors", as Cem calls them.

I've certainly had to spend a lot of time in the care and feeding of the tool, time which might have been better spent testing. There are certain aspects of Fitnesse that are downright clunky—the editing control in the Wiki is abysmal. I'm not convinced that all problems lend themselves to tables, and I'm not sure that all people think that way. Diversity of points of view is a valuable asset for a test effort, if your purpose is to find problems. Different minds will spot different patterns, and that's all to the good.

I frequently hear people—developers, mostly, saying things like, "I don't know much about testing, and that's why I like using this tool" without considering all of the risks inherent in that statement. I think the Agile community has some more thinking to do about testing. Many of the leading voices in the Agile community advocate automated acceptance tests as a hallmark of Agilism. I think automated acceptance tests are nifty in principle—but in practice, what's in them?

"When all the acceptance tests pass for a given user story, that story is considered complete." What might this miss? User stories can easily be atomic, not elaborate, not end-to-end, not thorough, not risk-oriented, not challenging. All forms of specification are to some degree incomplete; or unreadable; or both

User Acceptance Testing as Probing Testing by Users or Testers

A long time ago, I learned a couple of important lessons from two books. In *The Art of Software Testing,* Glenford Myers suggests that testing, real testing where we're trying to find problems, depends upon us actively searching for failures. We have to find ways to break the software. All good books on software testing, in my estimation, repeat this principle at some point. Testers can't prove the conjecture that the software works, but at least they can disprove the conjecture that it will fail based on some test. Tests that are designed to pass are relatively weak, and when those tests pass, we don't learn much. In contrast, tests designed to expose failures are powerful, and when those tests themselves fail to find a problem, we gain confidence. In his book The Craft of Software Testing, Brian Marick gave an example in which one powerful, hostile programmer test can provide seven different points of confirmation, but when we're probing the software, we advocate exploratory testing in additional to scripted testing. Exploratory testing is simultaneous design, execution, and learning. There's nothing particularly new about it—long before computers, expert testers have used the result of their last test to inform their next one. The way we like to think about it and to teach it, exploratory testing encourages the tester to turn her brain on and follow heuristics that lead towards finding bugs.

When performed by expert testers, this line of investigation is oriented towards finding out new information about the software's behaviour. This exposing the system to weird input, resource starvation,

Much of the testing is focus on traceability, repeatability, decidability, and accountability. In some contexts, that could be a lot of busywork—it would be inappropriate, I would argue, to apply these approaches to testing video games. But I still contend that a testing a product oriented towards a technical or organizational problem requires investigative behaviour.

By the way, when I said earlier that the majority of user acceptance tests were confirmatory, I don't have any figures on this breakdown; I can't tell you what percentage of organizations take a confirmatory view of UAT, and which ones actually do some investigation. I have only anecdotes, and I have rumours of practice. However, to the context-driven tester, such figures wouldn't matter much. The only interpretation that matters in the moment is the one taken by the prevailing culture, where you're at. I used to believe it was important for the software industry to come to agreements on certain terms and practices. That desire is hamstrung by the fact that we would have a hard time coming to agreement on what we meant by "the software industry”, when we consider all of the different contexts in which software is developed. On a similar thread, we’d have a hard time agreeing on who should set and hold the definitions for those terms. This is a very strong motivation for learning and practicing context-driven thinking.

Conclusion

Context-driven thinking is all about appropriate behaviour, solving a problem that actually exists, rather than one that happens in some theoretical framework. It asks of everything you touch, “Do you really understand this thing, or do you understand it only within the parameters of your context? Are we folklore followers, or are we investigators?” Context-driven thinkers try to look carefully at what people say, and how different cultures perform their practices. We're trying to make better decisions for ourselves, based on the circumstances in which we're working. This means that context-driven testers shouldn’t panic and attempt to weasel out of the service role: “That’s not user acceptance testing, so since our definition doesn’t agree with ours, we’ll simply not do it.” We don’t feel that that’s competent and responsible behaviour.

So I’ll repeat the definition. Acceptance testing is any testing done by one party for the purpose of accepting another party's work. It's whatever the acceptor says it is; whatever the key is to open the gate—however secure or ramshackle the lock. The key to understanding acceptance testing is to understand the dimensions of the context.

Think about the distinctions between ceremony, demonstration, self-defense, scapegoating, and real testing. Think about the distinction between a decision rule and a test. A decision rule says yes or no; a test is information gathering. Many people who want UAT are seeking decision rules. That may be good enough. If it turns out that the purpose of your activity is ceremonial, it doesn't matter how badly you're testing. In fact, the less investigation you’re doing, the better—or as someone once said, if something isn’t worth doing, it’s certainly not worth doing well.