User Acceptance Testing – A Context-Driven Perspective

Michael Bolton, DevelopSense
michael@developsense.com
https://www.developsense.com
January 7, 2022

A version of this paper was originally published at the PNSQC Conference in 2007. I have made some minor edits to it.

Biography

Michael Bolton is a consulting software tester and testing teacher who helps people to solve testing problems that they didn't realize they could solve. In 2006, he became co-author (with James Bach) of Rapid Software Testing (RST), a methodology and mindset for testing software expertly and credibly in uncertain conditions and under extreme time pressure. Since then, he has flown over a million miles to teach RST in 35 countries on six continents.

Michael has over 30 years of experience testing, developing, managing, and writing about software. For over 20 years, he has led DevelopSense, a Toronto-based testing and development consultancy. Prior to that, he was with Quarterdeck Corporation for eight years, during which he managed the company's flagship products and directed project and testing teams both in-house and around the world.

Contact Michael at michael@developsense.com, on Twitter @michaelbolton, or through his Web site, http://www.developsense.com.

Abstract: Hang around a software development project for long enough and you'll probably hear things that seem in conflict: “We need to keep the customers satisfied”; “the customer is always right”; and “the customers don't know what they want.”

When it comes to testing, it might be a good idea 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 obvious to everyone what “user acceptance testing” means? Is because there is no effective difference between user acceptance testing and other testing activities?

In my view, there are so many possible interpretations of what might constitute “user acceptance testing” that the term isn’t very helpful until we’ve sorted these things out. We’re on better ground when we establish a contextual framework and understand what people mean by “user”, by “acceptance”, and by “testing”. Here, I’ll discuss the challenges of user acceptance testing, and propose some remedies that testers and teams 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 development work was deemed to be "done"[1].

When all of the user stories associated with the current project were finished development and testing, the project changed focus and directed its attention to… "user acceptance testing". This involved a month or so of work in which developers worked primarily in support of the testing group, rather than the other way around. The testing group performed harsh, complex, aggressive tests, while the developers contributed new test code and fixed 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 or user acceptance tests. 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 assess compliance with standards or requirements, based on specific examples
- a set of tests that are performed for a customer to demonstrate functionality
- a set of tests that are performed *by* a customer to demonstrate functionality
- not tests at all, but a slam-dunk demo
- outside beta testing
- prescribed test activities that absolutely must be performed, problem-free, before the user will take the product happily
- prescribed comparison of the product to do demonstrate consistency with explicit requirements that must be performed as a stipulation in a contract
- any testing that is not done by a developer
- experiential tests that are performed by real users

[1] Why the quotes? See https://www.developsense.com/blog/2010/09/done-the-relative-rule-and-the-unsettling-rule/

- tests that are performed by stand-ins or surrogates for real users
- in Agile projects, prescribed automated checks that mark a code-complete milestone
- in Agile projects, automated checks 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, contexts, and experiences.

If we are to do any kind of testing well, it is vital to avoid being fooled by *shallow agreement*—believing that we understand each other when we really do not. Avoid shallow agreement begins by gaining understanding of the ways in which other people might be saying and thinking profoundly different things, even though they sound alike.

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[2]. 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
- 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[3]
- Business analysts
- Architects
- Content providers
- Technical Support
- Sales people
- Sales support
- Marketing people
- The shareholders of the company
- The CEO
- 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 some notion of "UAT". Moreover, some of these roles have different (and sometimes competing) motivations. 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.

[2] Here's a useful little trick for identifying internal roles: in your head, walk through your company's buildings and offices, and think of everyone who works in each one of those rooms.

[3] 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.

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:

> Evaluating a product by learning about it through experiencing, exploring, and experimenting, which includes questioning, studying, modeling, observation, inference, etc.[4]

Testing also entails, at least, critical evaluation and risk analysis.

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.

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.[5]

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

---

[4] In the original version of this paper, we said "questioning a product in order to evaluate it". That's still true, but we've extended our definition since then. See http://www.satisfice.com/blog/archives/856.

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

- 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.

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[6].

In *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." (my question mark)

[6] This was true at the time of writing in the original edition of this paper.

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 of accepting another party's work.*

It's whatever the tester and the acceptor agree upon; whatever the key is to open the acceptor’s 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 demonstrates 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 herself 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. I am saying 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, typically in the form of automated checks. This kind of testing is by its nature entirely confirmatory unless and until a check suggests some kind of failure and human tester gets involved again to investigate.

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 common assumptions of this variety of UAT is 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 putting the product in front of someone who is unlikely to have testing skills, and unlikely to find bugs—and at the very time when the development team is least inclined 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 be even worse, though, to discover 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 strategies for preventing 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 bearpit with the developers, as eXtreme Programming advocates recommend, 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 testing generally. User stories are helpful, but they form a pretty weak basis for testing.

That's because user stories, in my experience, tend to describe simple, atomic tasks. User stories 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.

To address this, mandate testers to focus on *problems*. Direct testers to apply critical thinking and systems thinking, using science and the scientific method. Tell stories about bugs, how those bugs were found, and the techniques that helped to find them. At the same time, recognize the kinds of bugs that those techniques couldn't have found; and identify techniques that wouldn't find *those* bugs but that would find *other* bugs. Encourage testers to consider those "-ilities" beyond capability.

User Acceptance Testing as Usability Testing

User acceptance testing might be testing focused on usability. In this, there is an important distinction to be made between ease of *learning* and ease of *use*.

Here's a trivial example: compared to a program that has only a command-line interface, 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 experienced 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.[7]

User Acceptance Testing as Validation

In general, with confirmatory automated checks, a single bit of specified information (yes/no, pass/fail, true/false), is required for a test to "pass"; in testing, we consider many more bits of information, and many of those bits are unspecified in advance of the test.

- "When a developer says 'it works', he really means 'it appears to fulfill some requirement to some degree.'"
  - 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.'"
  - Jerry Weinberg

Conformance to documented requirements is at often at issue in a contractual, time-and-materials development model, where the product must pass a "user acceptance test" as a condition of sale. When such projects are in their later stages, they're are often behind schedule; people are tired and grumpy; lots of bugs have been found and fixed. There's lots of pressure to end the project, and there's a corresponding disincentive to find problems.

At this point, the skillful tester faces a dilemma: should she look actively for problems (thereby annoying both the client and his own organization should she find one), or should she be a "team player"?

My final take about this 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, consistent, up to date? There's a vast difference between requirements and requirements *documents*.

Second, shouldn't 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.

---

[7] 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 sometimes circumstances in which relations between the development organization and the customer are so fraught that the customer actively wants to reject the software. There may be all kinds of reasons for this. In the face of a problematic projects, customers might be trying to find someone to blame; they may want to show the vendor's malfeasance or incompetence to protect themselves from their own games of schedule chicken; they may want to avoid paying the bill.

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. In the 1990s, I I was a program manager for Quarterdeck, the company that produced DESQview and other mass-market products such as QEMM-386 and CleanSweep[8]. 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.

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.

Many companies regularly release beta versions of their software[9]. 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.

In those days, I was younger, and inexperienced at recognizing process traps. I read books that told me how important "user acceptance testing was" and agonized and tried for ages to figure

[8] Those of you with large amounts of gray hair may remember these products.
[9] Yes, I know: "…and call them 'releases'.")

out how to implement it in the context of mass-market commercial software. It was a long time before I realized that we didn't need to do what people were talking about in the books, in context-free ways. Those things didn't fit our context.

This was a big lesson: don't listen to any statement or proclamation—especially about process stuff—from someone that doesn't establish the context in which their advice might be expected to succeed or fail. Without a healthy dose of consideration for 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 frame requirements in terms of Behaviour Driven Development, or Acceptance Test Drive Development, or contract tests.

The claim is that these approaches can help add to a common understanding between developers and the business people. These approaches may have value in that they provide *checkable examples* of expected behaviour, but are design activity far more than they are testing activities. As such, they shouldn't be confused with testing the software, and examples shouldn't be misrepresented as tests.

User Acceptance Tests as Milestones

Checked examples 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 acceptance tests run green." Ready to go—but *where*?

Something is "done" or "complete" relative to a particular perspective[10], and it's important to be clear on what that perspective is, what the milestone represents. I recall J.B. Rainsberger at one point, in a mailing list, saying something like "a green bar doesn't tell you you're done; it tells you that you're ready for a real tester to kick the snot out of it."

Automated acceptance checks may be very useful tools for detecting particular problems, by encoding specific test ideas, procedures, and data, and re-running them frequently; "change detectors", as Cem Kaner has called them. This can be an exceedingly powerful means for recognizing easy bugs and coding errors.

That said, computers are exceedingly reliable, but the software running on them may not be doing what we believe it's doing, and automated checks are software. Whatever else we might believe, software is not *doing the testing*.

Computers and software don't have the capacity to recognize problems as problems; they must very explicitly programmed to alert us to specific problems in very specific ways. They are not social agents. They certainly don't have the imagination or cognitive skills to say, "What if...?" or "That's funny…" as human testers do.

[10] https://www.developsense.com/blog/2010/09/done-the-relative-rule-and-the-unsettling-rule/

Developers and people fascinated by programming tend to like automated checking. There are good reasons for that. A few quick and simple checks can confirm that the product meets the developer's and the team's intentions to some degree, without obvious problems. Getting to that point represents a good milestone, but it's important to keep clear on the difference between a milestone and a finish line.

User Acceptance Testing as Experiential Testing

In the last couple of years, James Bach and I have developed our notion of *experiential testing*.

A software product not simply the code. Software contains code, of course, but the product is really the experience that we provide for people as they try to get work done, accomplish goals, make money, have fun, and so forth. Testing that doesn't address the user's experience runs the risk of missing problems that matter.

Experiential testing is testing in which the tester's encounter with the product, and the actions that the tester performs, are practically indistinguishable from those of the contemplated user.

The tester, in this case, might be a genuine end user with *contributory expertise*[11] in the domain in which the product is set. Real end users lend credibility to experiential testing because they have domain expertise necessary to evaluate the state of the product in terms of the tasks that they intend to accomplish.

The tester might otherwise be someone with *interactional expertise*[12] sufficient to evaluate the product and identify problems that would represent threats to value for actual end users.

The object of the exercise here is to put the product in front of people and to show how it might behave in terms of real-world use.

User Acceptance Testing as Experiential AND Investigative Testing

Earlier in this paper, I identified modes of testing that might be ceremonial or demonstrative. Activity called testing can be experiential without being exploratory or experimental, as when a tester follows a set of scripted procedures, and doesn't deviate from them. That's fine, when the object of the game is to avoid or suppress the discovery of new problems.

On the other hand, the goal of experiential user acceptance testing might be to find problems that threaten value to people who matter. In this case, the tester must have a degree of freedom to explore the product, and freedom to design and perform experiments that challenge the software and people's assumptions about its goodness.

In this form of user acceptance testing, the responsible tester must have more than expertise in the product domain. The tester must also have contributory expertise with respect to testing

[11] People who can accomplish the work and/or advance the state of the art in a field have *contributory expertise*. See Collins and Evans, *Rethinking Expertise*.

[12] People who have expertise in a particular field, such that they're *not* capable of doing the work, but *are* able to understand and speak the practice language of the field pretty much as well as the contributory experts do. Collins and Evans call this *interactional expertise*.

itself. That is, the tester must have testing skills relevant to the task at hand. Such skills include the ability to

- learn and model the product rapidly
- model quality criteria for diverse stakeholders
- perform risk analysis and identify threats to value
- model coverage
- identify, develop, apply oracles (means of recognizing problems)
- obtain or develop tools
- design and perform experiments
- investigate bugs
- report on the testing work[13]

It's unusual for end users to have these skills without some specialized training in testing[14]. One good workaround is to pair skilled *responsible* testers[15] with expert users taking the role of *supporting* testers.

When to Do Acceptance Testing

It should be clear by now that there are many notions of acceptance testing, many people who might perform it, and many contexts in which it might happen. Given that, when should user acceptance testing be done? Here's a heuristic:

As soon as there's something for a user to observe and evaluate, such that a user is in a position to provide useful feedback to people on the development team, give the user opportunity to provide that feedback.

Throughout the project, advocate for opportunities to engage with users[16]. While the product is being designed, if there's a chance to engage users in review, offer that opportunity to users.

Show them designs, plans, prototypes, mockups, and solicit comments. Systematically question assumptions about the team's knowledge of the product domain and the problems to be solved. Consider engage users in helping to answer four questions: What are we building? Who are we building it for? What could go wrong? How would we know?[17].

Whether you've got a partially- or fully-built product, engage with users and show them the product whenever they're willing to look at it.

---

[13] This list is just a starter set. For a more comprehensive listing, see https://www.satisfice.com/download/elements-of-excellent-testing

[14] Alas, too often, testers may not have sufficient training or skill in these things either.

[15] See https://www.satisfice.com/blog/archives/1364.

[16] The idea here is to increase value-related and project-related testability; see https://www.satisfice.com/download/heuristics-of-software-testability

[17] https://www.developsense.com/blog/2018/03/four-and-more-questions/

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, on behalf of our clients, 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, too, about the distinction between a *decision rule* and a *test*. A decision rule produces a yes or no result that *prompts* a particular action; a test is information gathering that *informs* action.

Many people who want UAT are seeking decision rules and ceremony. That may be good enough in certain contexts. If it turns out that the purpose of your activity is ceremonial, it doesn't matter how badly you're testing. In fact, if you want confirmation or ceremony, 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.

If your goal is to find problems that matter, one key is to focus on finding problems, rather than on demonstration and confirmation. Another key is to diversify testing approaches to include automated checks, critical investigation, and experiential testing. Diversity of points of view and diversity of approaches are valuable in testing. Different minds will spot different patterns, and that’s all to the good.