# User Acceptance Testing

Michael Bolton
DevelopSense
Toronto, Ontario, Canada
mb@developsense.com
http://www.developsense.com
+1 (416) 656-5160

# Acknowledgements


- For this talk in particular, I'm deeply indebted to James Bach
- For all the talks that I do, I'm indebted to
  - James Bach
  - Cem Kaner
  - Jerry Weinberg
  - George Carlin
  - Richard Feynman

# What Might Be Wrong Here?

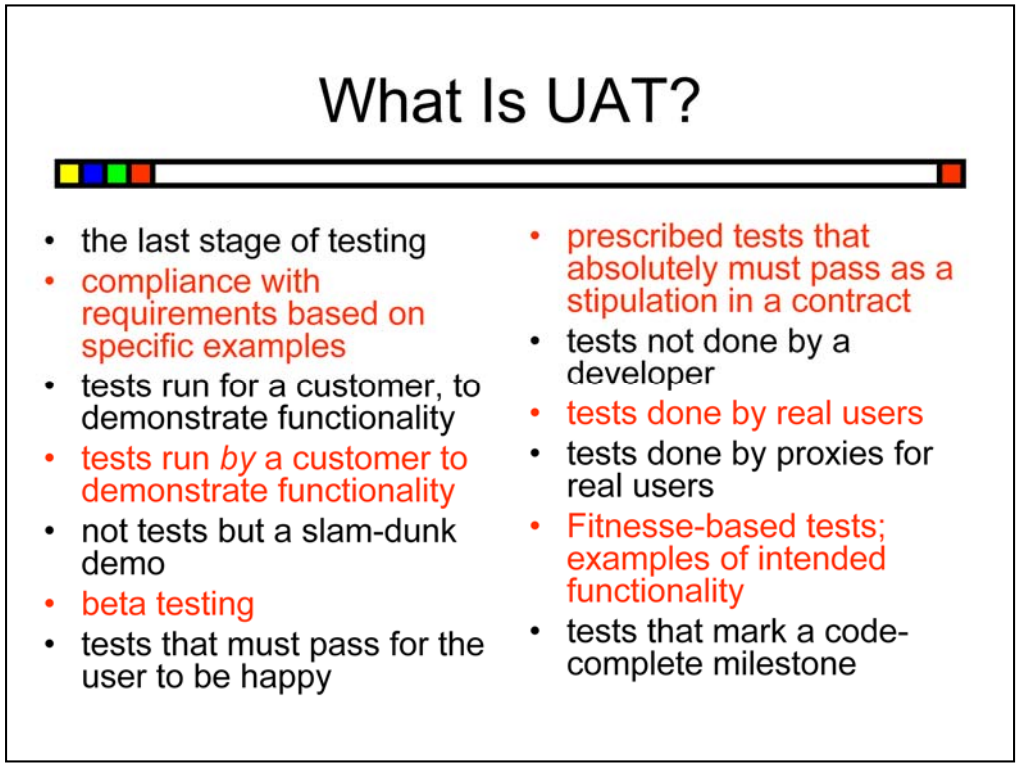
From the Fitnesse mailing list:

>>…because we decide the acceptance criteria up-front, we know, objectively, when we are finished. Being able to code to a test eliminates speculative coding and decreases bug rate as well. It may be a small amount of more work up front, but it decreases time from development to production, so the entire cycle is reduced.

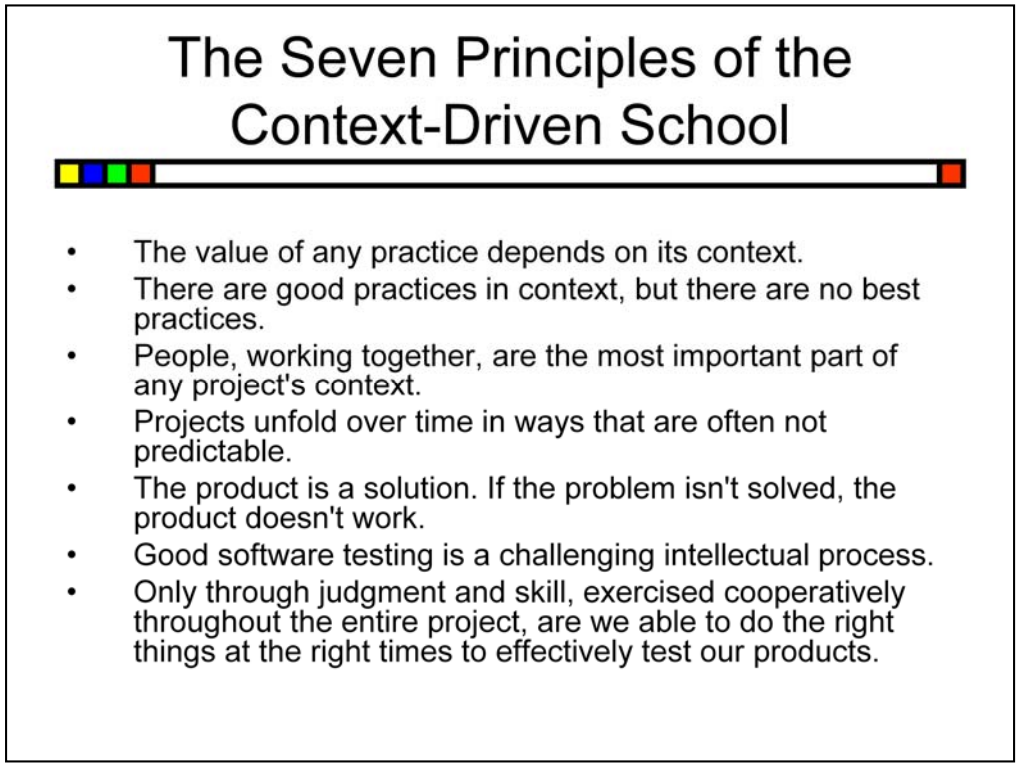
>The above is golden and I am sure that I will paraphrase and reuse it many times in the coming weeks and months!

# Question

What *is*
User Acceptance Testing
to *you*?



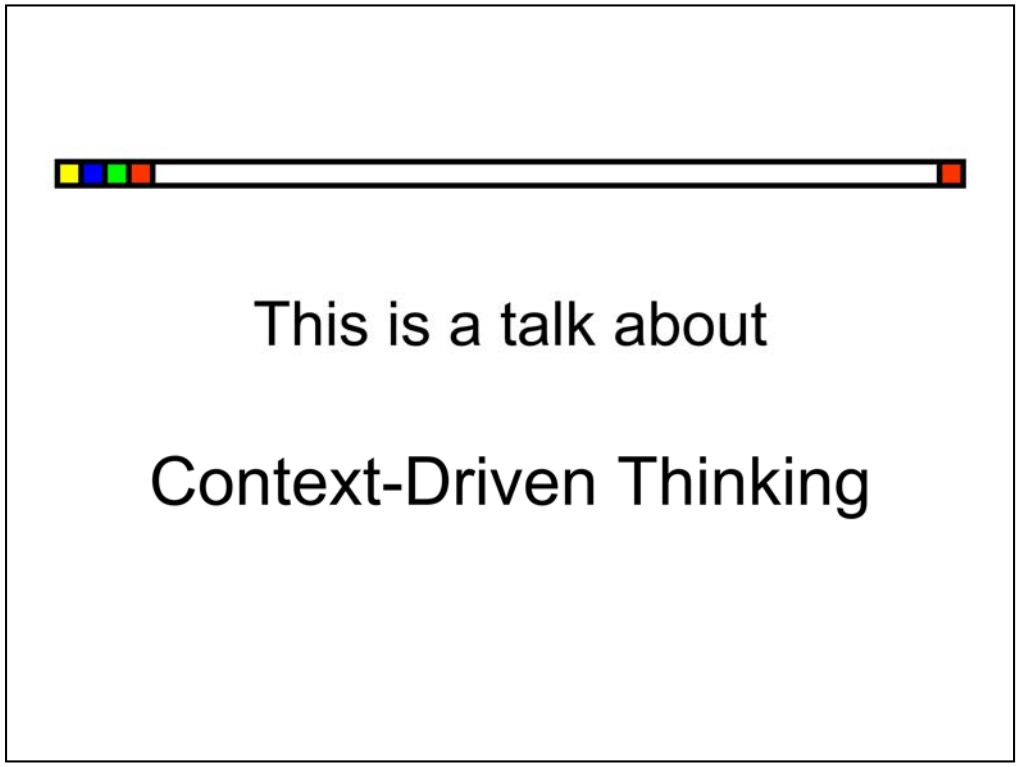

What is User Acceptance Testing? With all of these possible interpretations, can it really be different from "testing"? To paraphrase Gertrude Stein, is there any *there* there?



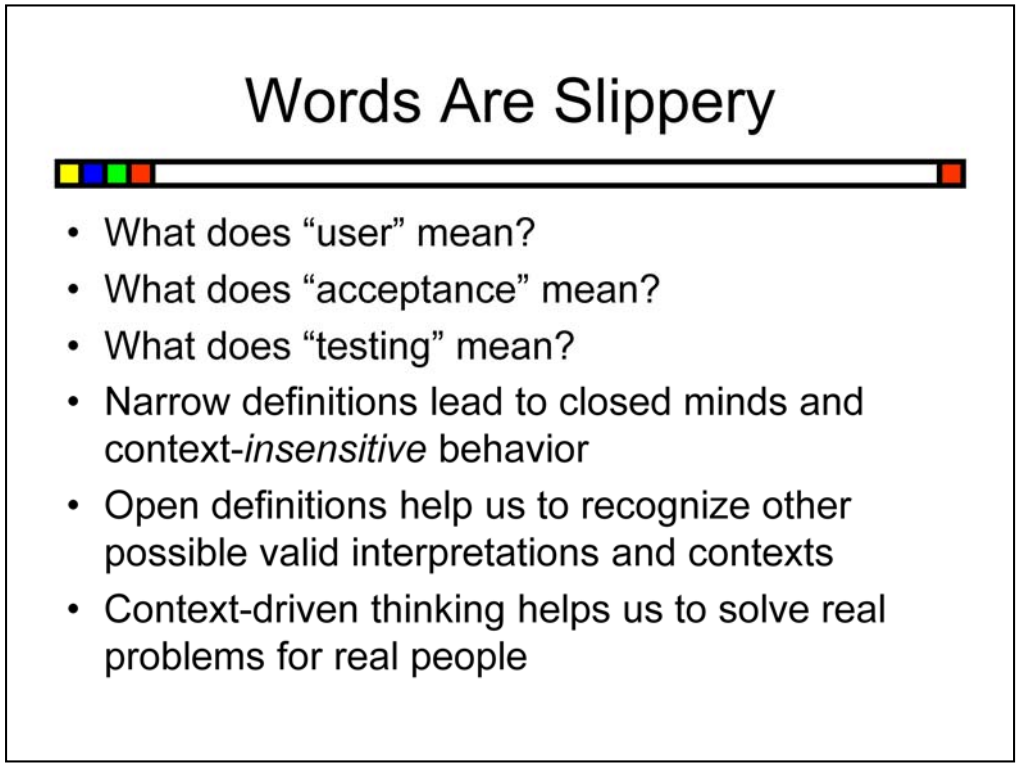

This last point is the most important subject that I'd like to address today. Far more than that, I think it's the most important subject for our profession as technical people in general, and as testers in particular. I believe that we must, as competent and responsible technical workers, as *knowledge* workers, continuously question all of our processes in a critical way. It's only by doing that, I argue, that we can be more sure that we are performing valuable and honourable services for our customers. It's the difference between working to a purpose and going through the motions.

This is a talk about

User Acceptance Testing




Understanding user acceptance testing is really an exercise in context-driven thinking. The definition of User Acceptance Testing can't be applied in a meaningful way without reference to a specific context.

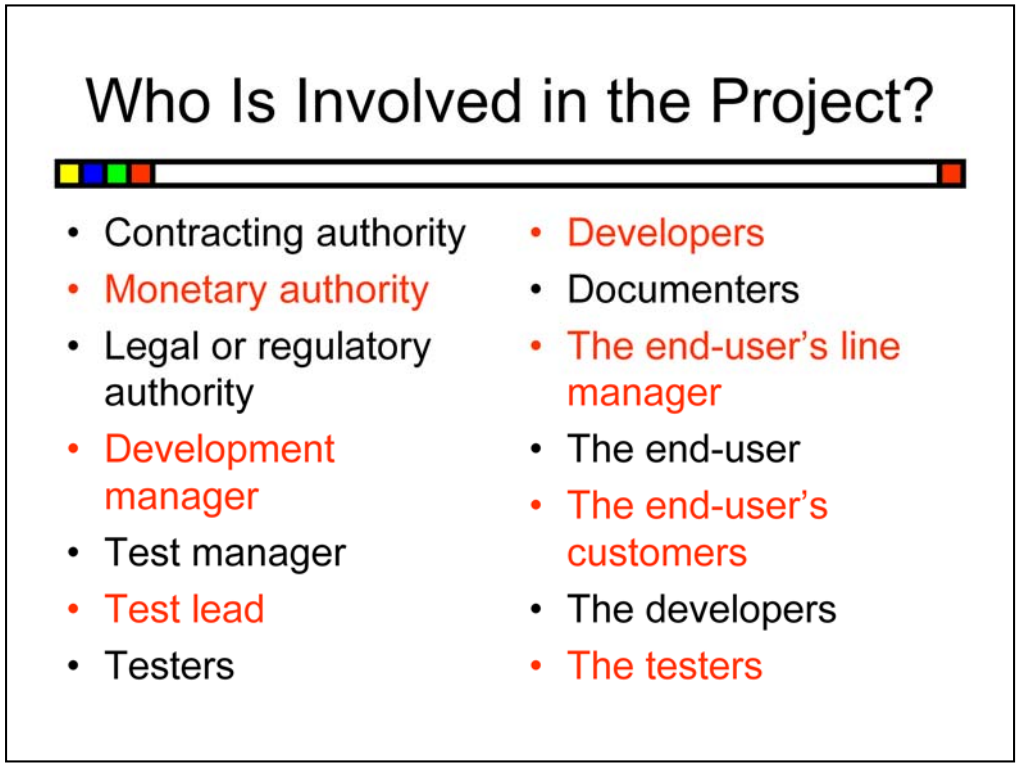



I’d like to highlight the notions that words (like user acceptance testing) are fundamentally ambiguous; that we have different points of view that are rooted in our own cultures and circumstances and experiences; that it is vital to try to gain understanding of the ways in which other people, even though they sound alike, might be saying and thinking profoundly different things. Rather than saying “that’s not User Acceptance Testing”, I’d like to encourage you to say, “that’s not User Acceptance Testing *the way we do it*— but how might it be rational, appropriate, or even crucial, to do it some other way in different circumstances?” 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.

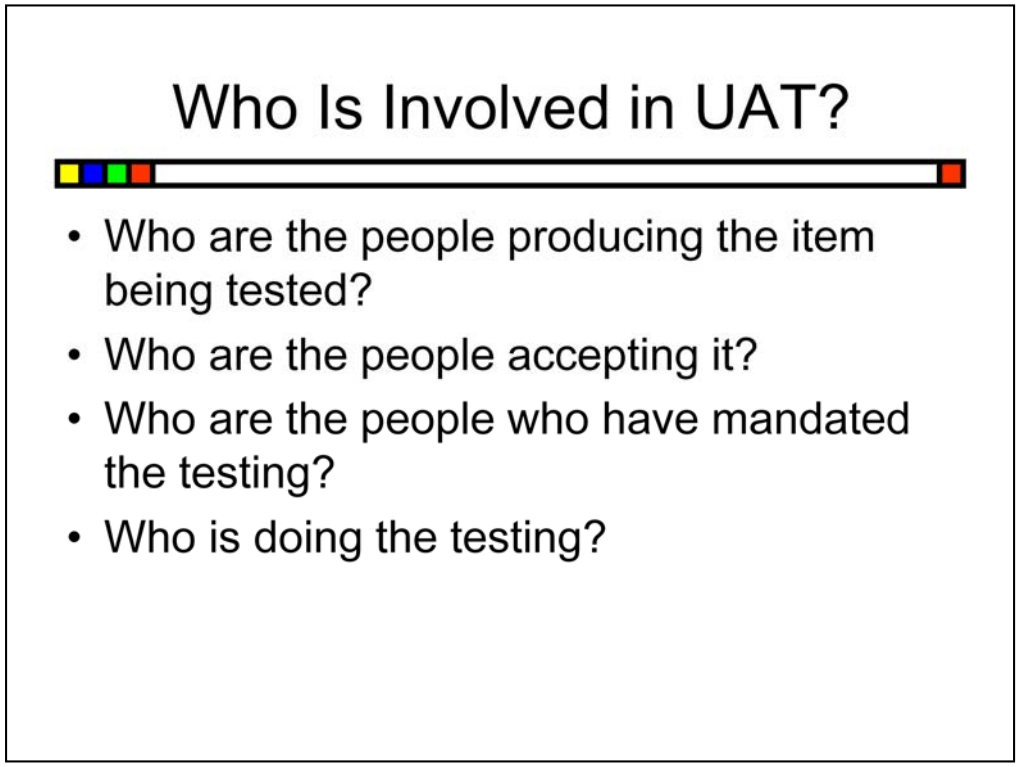
In order to make sense of our context, it’s important early on to visit item 3 on the Context-Driven Testing list: people, working together, are the most important part of any project’s context. From Josh Kerievsky, I learned the term “project community”—although I believe it was Alastair Cockburn who coined it. In each project that I’ve worked on, I’ve found it very useful to build a list of parties who might have an interest in the testing of a product, or who might be affected by it. I try to do this early in the project, and to revisit it frequently.

# Question

Who might be interested in User Acceptance Test results?




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?



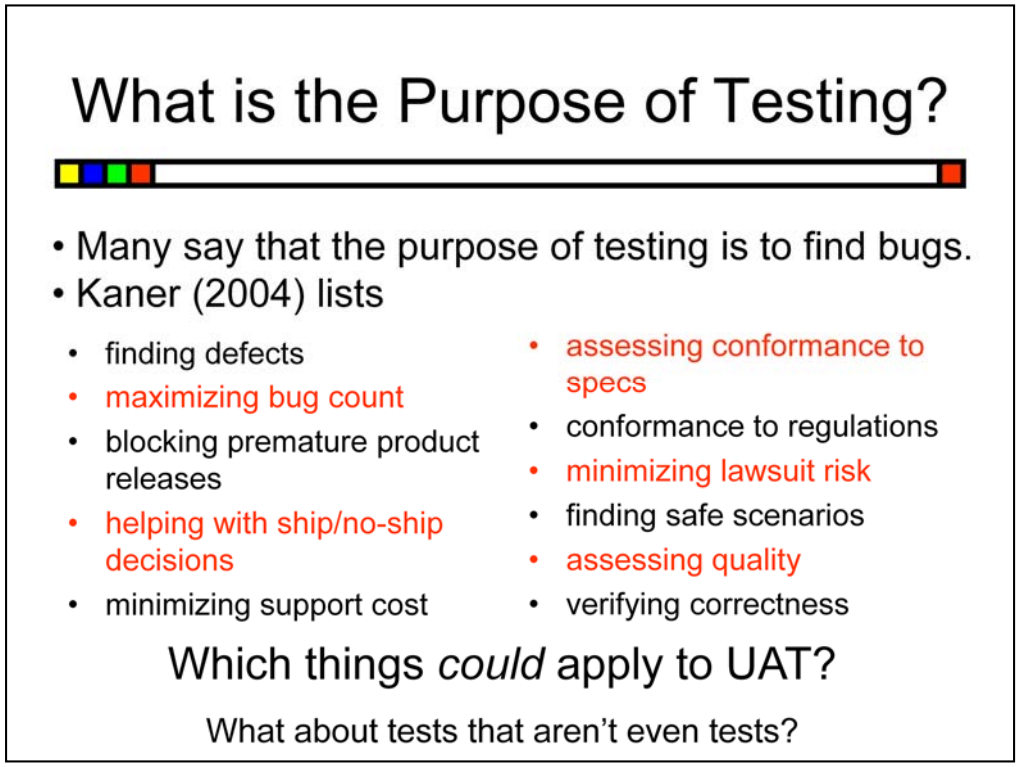

I've listed fourteen project roles on a previous page. When we list user roles in my seminars and courses, we typically get to thirty with no effort at all. Thirty roles in the project community times four roles within the acceptance test gives us a huge number of potential interaction models for a UAT project. There may be lots of project roles that I've left out, but the mere act of thinking about this should remind us that there are plenty of roles in any project community, and these roles have different (and sometimes antagonistic) motivations. My statistics is a little rusty, but for just the roles I've listed, I believe the general formula would be 14 times 13 times 12 times 11, which is 24,000 some-odd combinations, if we believed that every combination were valid. They won't be of course, but would "over 50" suffice to make the point? Which is this: 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?

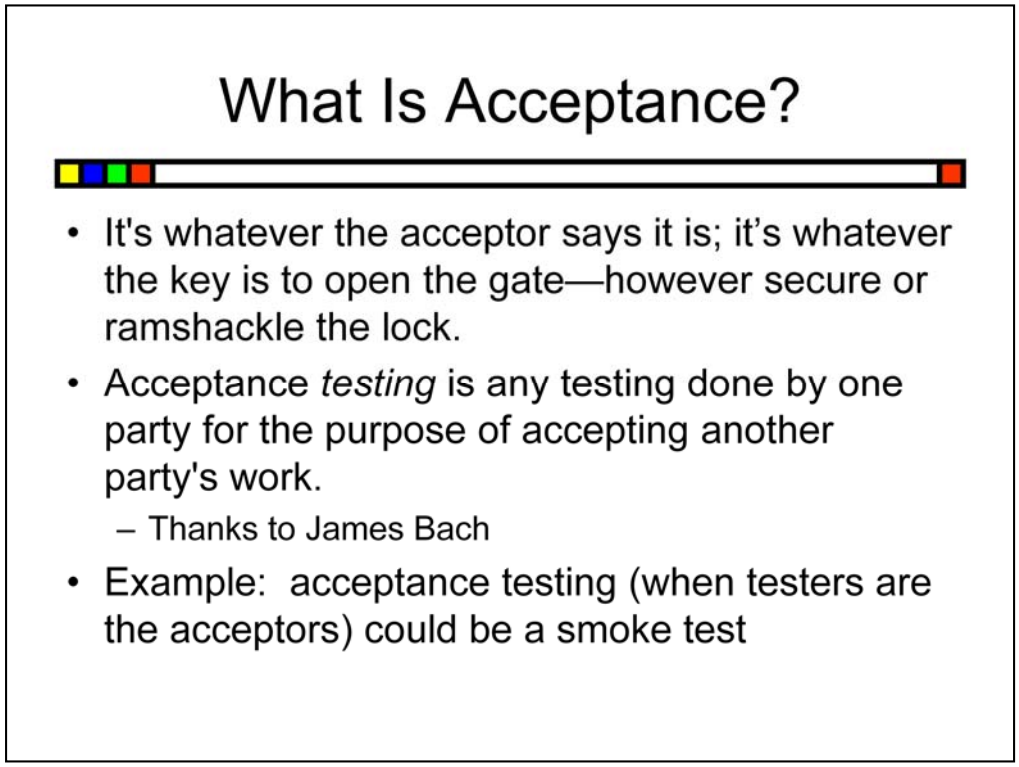
- Testing is a technical investigation of a product, done to expose quality-related information.
  - Cem Kaner, “The Ongoing Revolution in Software Testing”, 2005
- Questioning the product in order to evaluate it.
  - James Bach, “Rapid Software Testing”, 2005

# Question

What is the purpose of testing?




I would add assessing of compatibility, assessing readiness for deployment, regression testing, or ensuring that that which used to work still works—we can all add to the list, and that's great.




A context-driven approach encourages us to ask questions about what people mean when they say things—so what could people mean by "acceptability" or "acceptance"? Just as with "testing" itself, they could mean a bunch of things.

In TCS, 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 Cem was the senior author with James Bach and Brett Pettichord, neither the term acceptance test nor "user acceptance test" appears at all. In Black Box Software Testing, by Boris Beizer, neither the term acceptance test nor "user acceptance test" appears at all.

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? They 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 wimp out and say, "the nuts and bolts of user acceptance test is (sic) beyond the scope of this book."

# What is Testing?

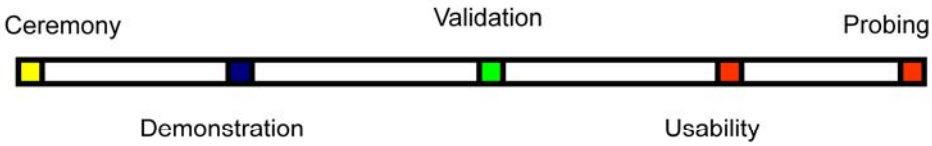
# Confirmation vs. Investigation

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.

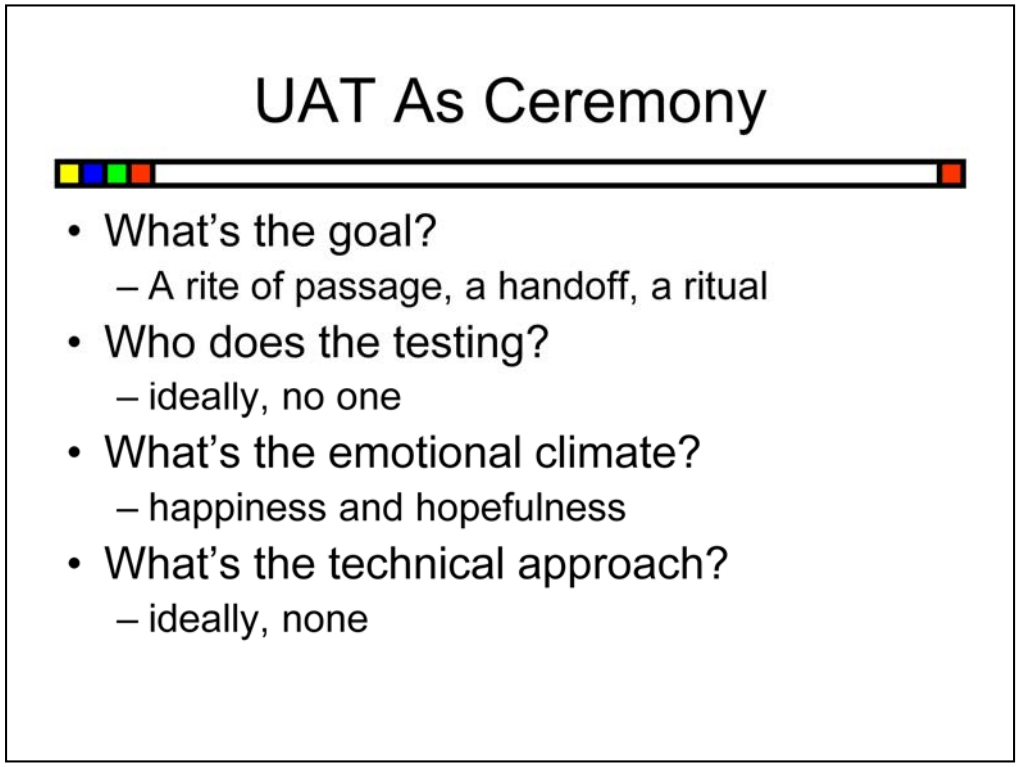
*Cem Kaner*

*The Ongoing Revolution in Software Testing, 2004*

# Some Motivations for User Acceptance Testing

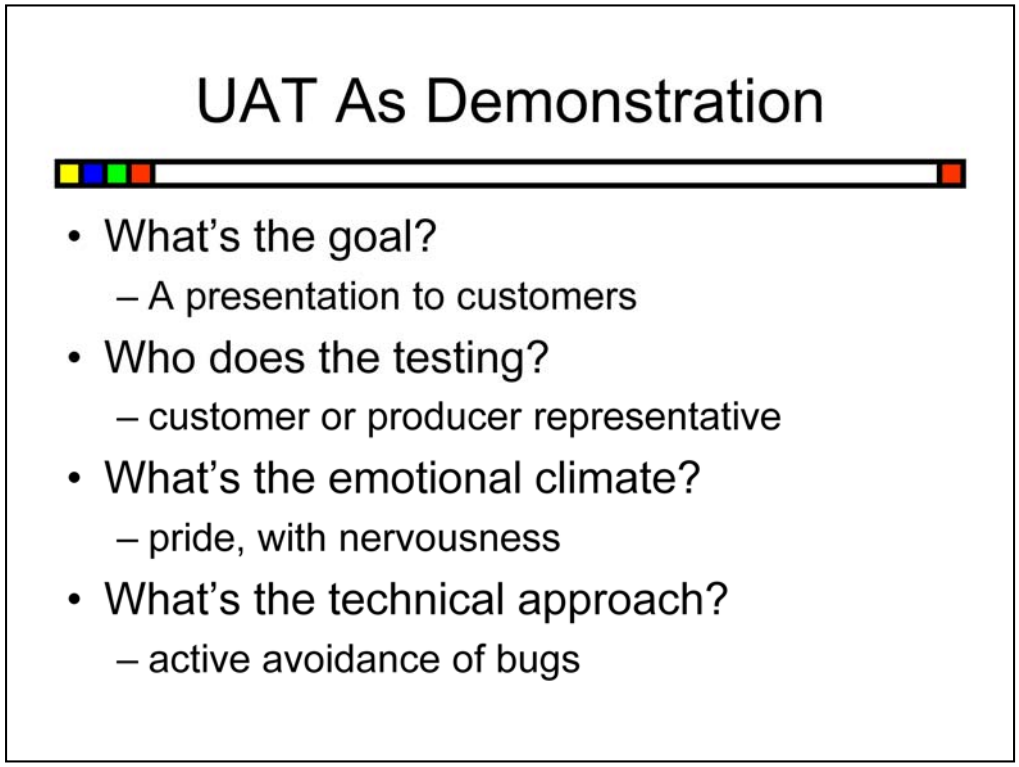



- The following slides provide some paradigmatic examples
- They're intended to be controversial to some
- If you accept the example, can you imagine others?
- If you reject the example, is it because that's not your context?

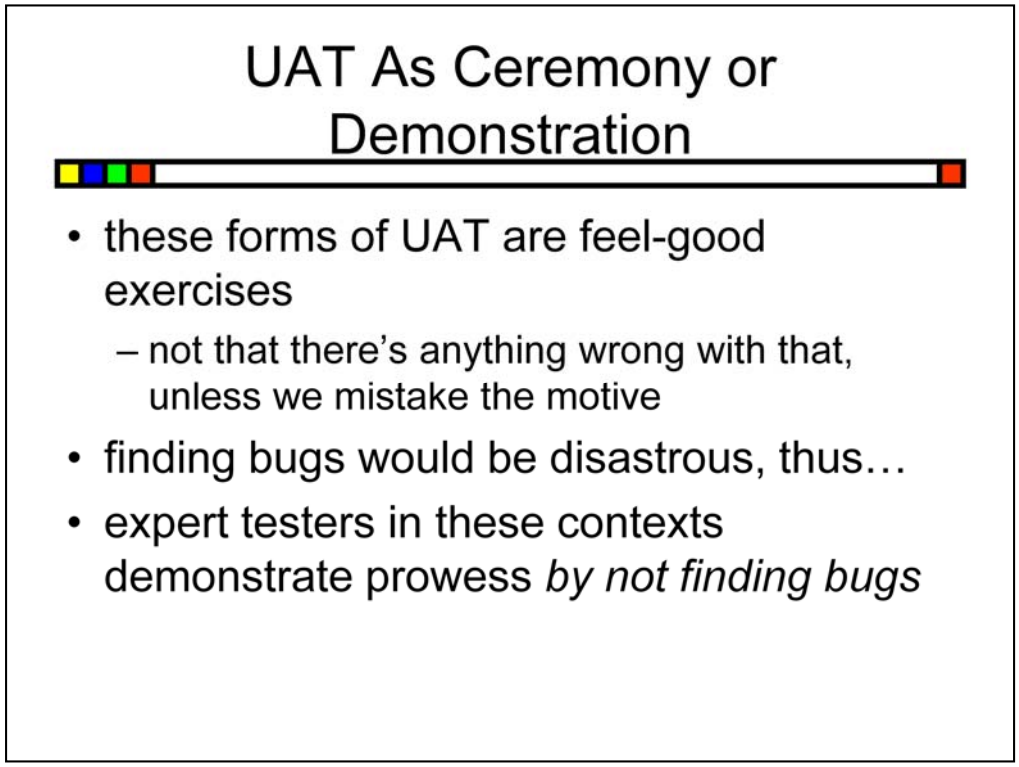



In some cases, UAT is not testing at all, but a ceremory. In front of a customer (who may or may not be a bigwig), someone operates the software, without investigation, even without confirmation. Tests have been run before; this thing called a user acceptance test is a feel-good exercise. No one is obliged 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*.

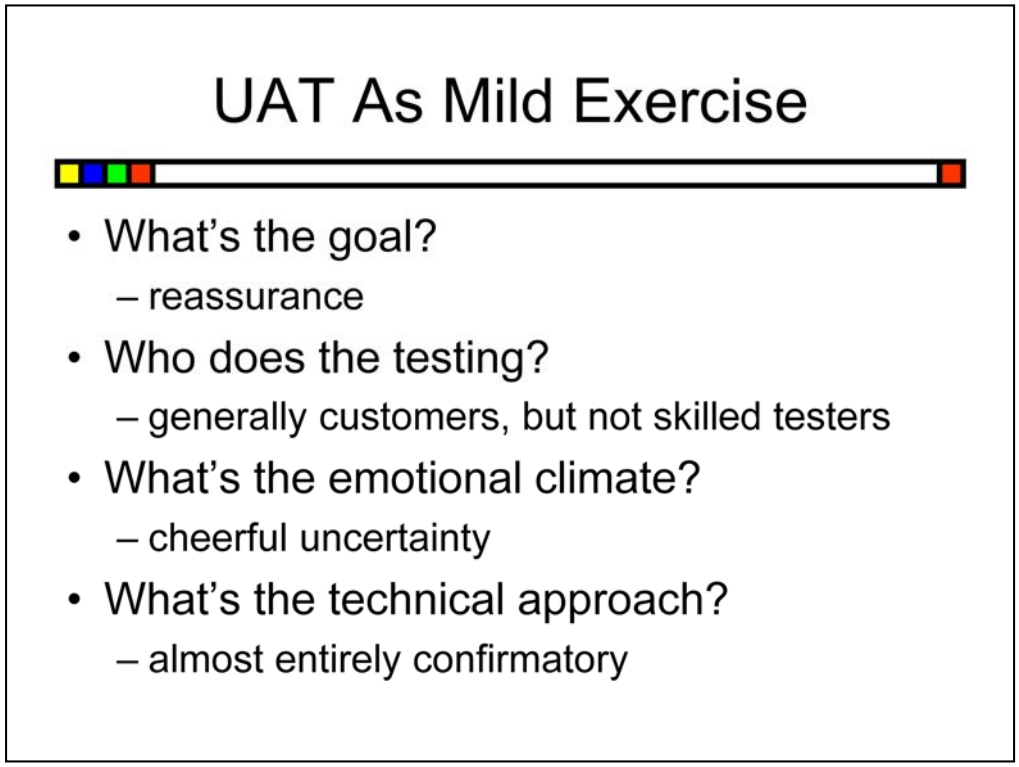
For example, when the Queen inspects the troops, does anyone expect her to perform an actual inspection? Does she behave like a drill sergeant, check for errant facial hairs? Would we not consider it strange if she asked a soldier to disassemble his gun so that she could look down the barrel of it? Of course we would—and that's because of the context of the situation. 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 inspection at all.




Another great example, which James Bach suggested to me—this actually happened to me. I bought a car a few years ago—it's a 20-year-old BMW. On that test drive for a new car, the salesman doesn't want problems to be found; that would be a disaster. Not even the *customer* wants to find a problem at that stage. It was a ceremonial part of an otherwise arduous process, and everyone just wants to be happy. In this case, both the customer and the salesperson are actively uninterested in finding problems; it's a feel-good occasion.

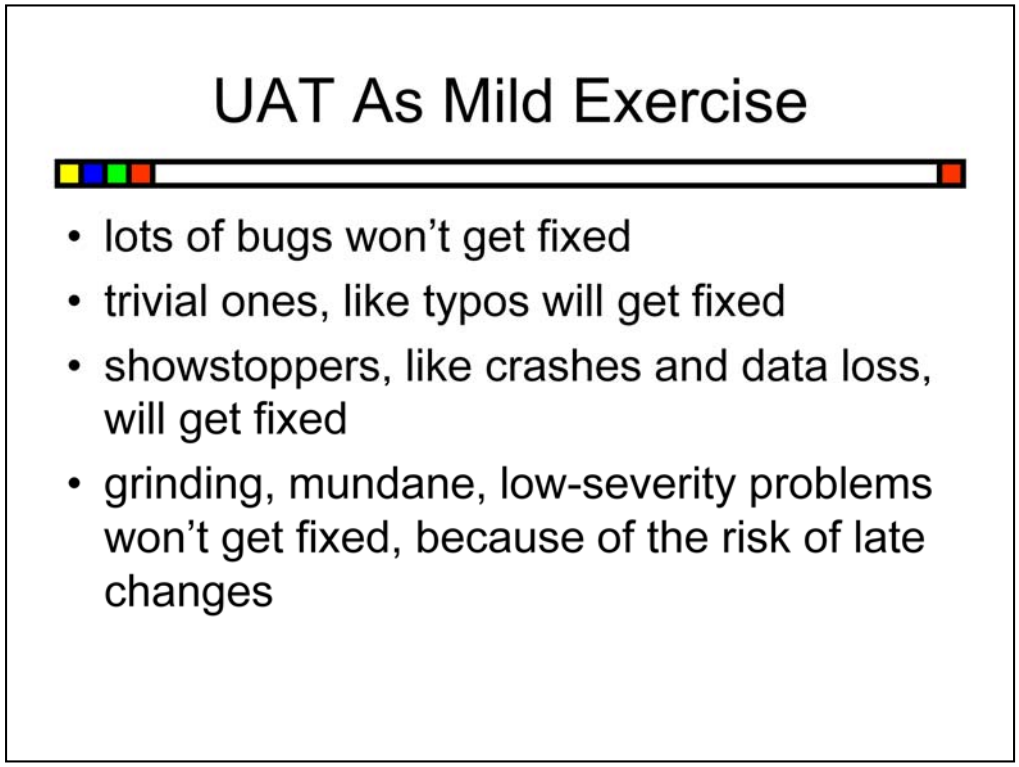



We often see in testing literature the idea of a user acceptance test as a formality, a ceremony or demonstration, performed after all of the regular testing has been done. I'm not saying that this is a bad thing, by the way—just that if there's any disconnect between expectations and execution, there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




Some versions of user acceptance testing are a little less perfunctory and pro forma. There's a form of user acceptance testing is a very late stage in the process, a hoop through which the product must jump in order to pass. Typically it involves 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 a tester—walk through some user stories. 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.

This form of user acceptance testing confuses me, quite frankly, especially as it gets closer to the confirmatory end of the scale. I hear about it a lot; it's one of the few kinds of user acceptance testing that actually shows up in the literature. Whenever it does, I observe mixed messages about it. One of the assumptions of this variety seems to be that the users are seeing the application for the first time, or 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 that they're unlikely to find, at the very time when we're least likely to fix them.

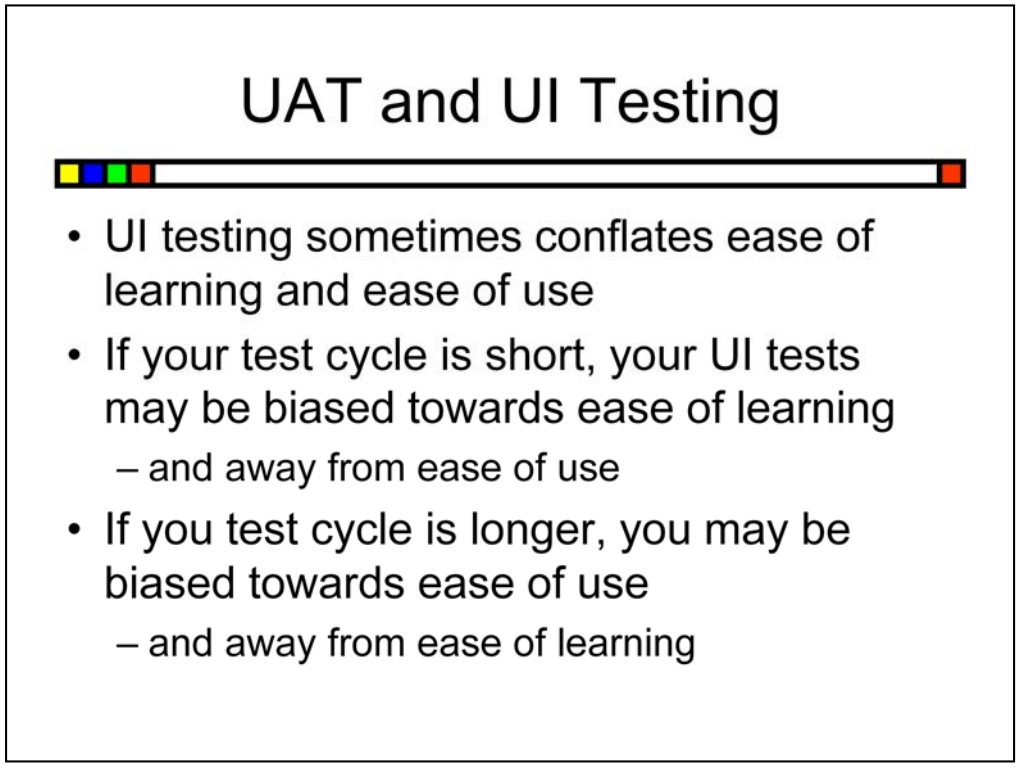



The object of this game is still that Party B is supposed to accept that which is being offered by Party A. In this kind of user acceptance testing, there may be an opportunity for B to raise concerns or to object in some other way. If the problem is one that requires no thinking, no serious development 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. 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. 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.

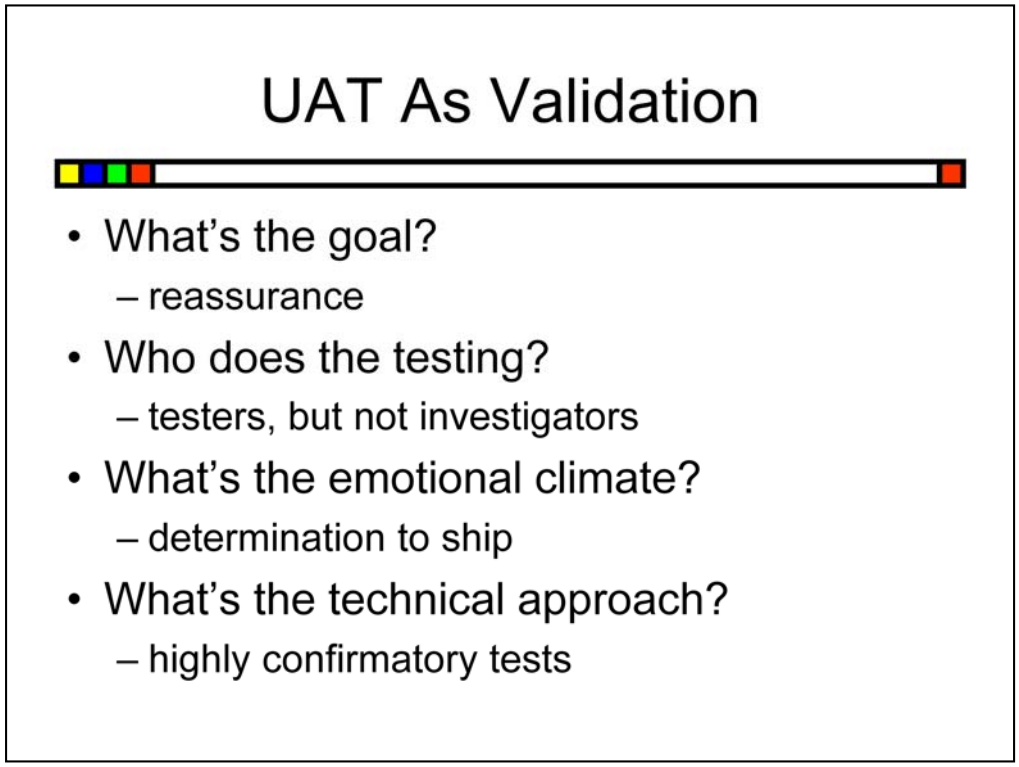
There are a couple of preventative strategies for this catastrophe. One is to involve the user continuously in the development effort and the project community. This is something that promoters of the Agile movement suggest. I think this is a really good thing. I've been doing some consulting lately in a shop that calls itself "agile". I don't think the Agilistas have 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 finding bad bugs 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.



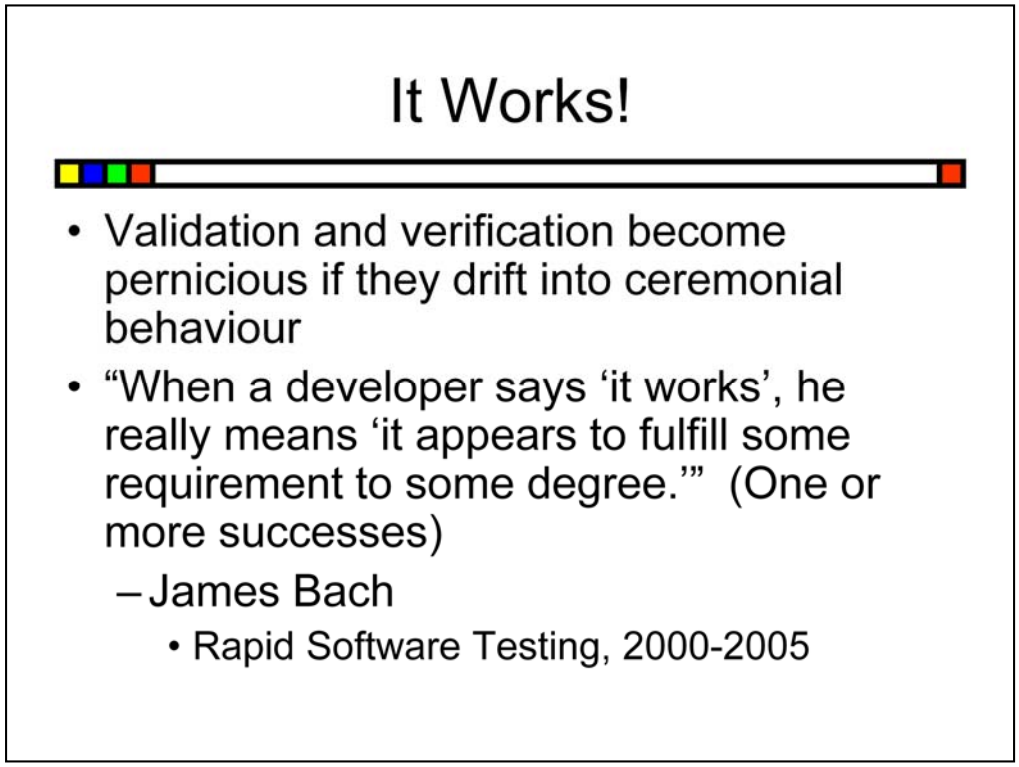

User acceptance testing at this stage might pay *some* attention to usability; here's something I'd like to say about that. If you're in the position where you're accepting software, I think it's important to watch out here for the distinction between ease of *learning* and ease of *use*. In my own past, I look to the difference between DOS-with-DESQview and Windows. DESQview was a multitasking environment for regular DOS programs. Its user interface was keyboard-oriented and very terse, or subtle, if you like, but it was fast and clean and simple, and it stayed out of the way. Windows gave an impression of user-friendliness that was hard to resist for a lot of people; it was prettier, mouse-centric, obvious in certain ways. I found then—as I still do now—that there are lots of tasks that still take longer in Windows. Windows has also become more and more cluttered as it has taken on more and more features and functions. There's no denying its success, but compared to DESQview, I would say that Windows is easy to learn, but ultimately harder to use. 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 to 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.

Usability testing late in the development cycle is usually a mug's game. Big changes won't happen; usability concerns might get addressed in a later version of the product. If you're going to do UI testing, do it early.

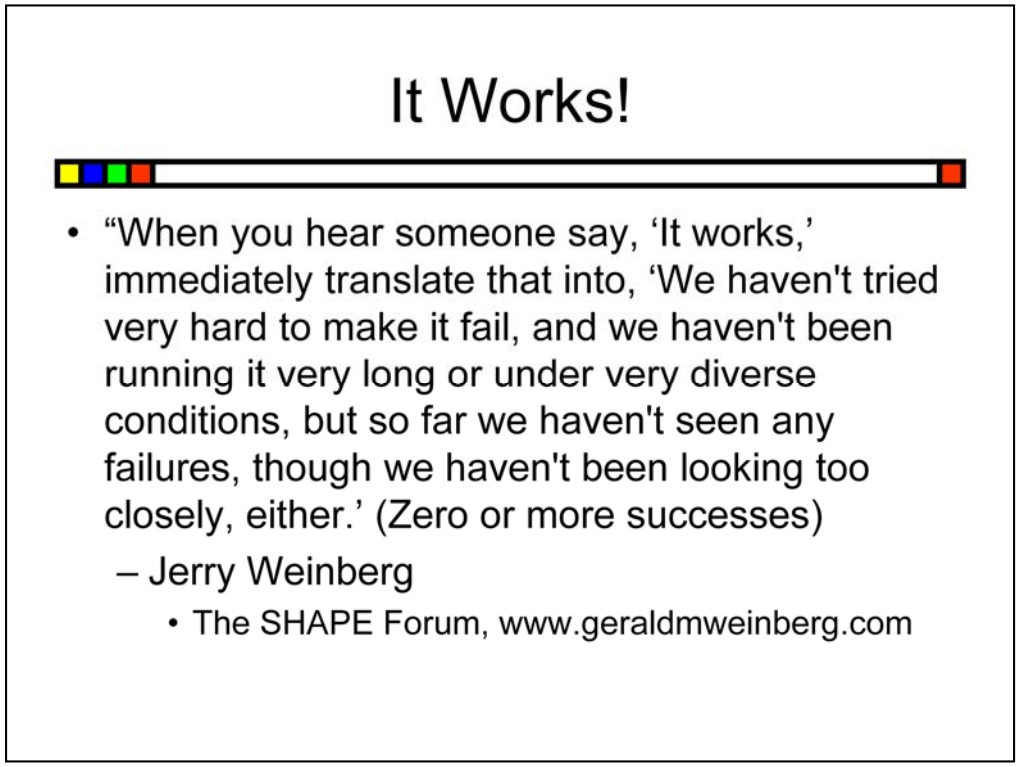



Confirmation should be investigative; pernicious if it slips into ceremonial behaviour.

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

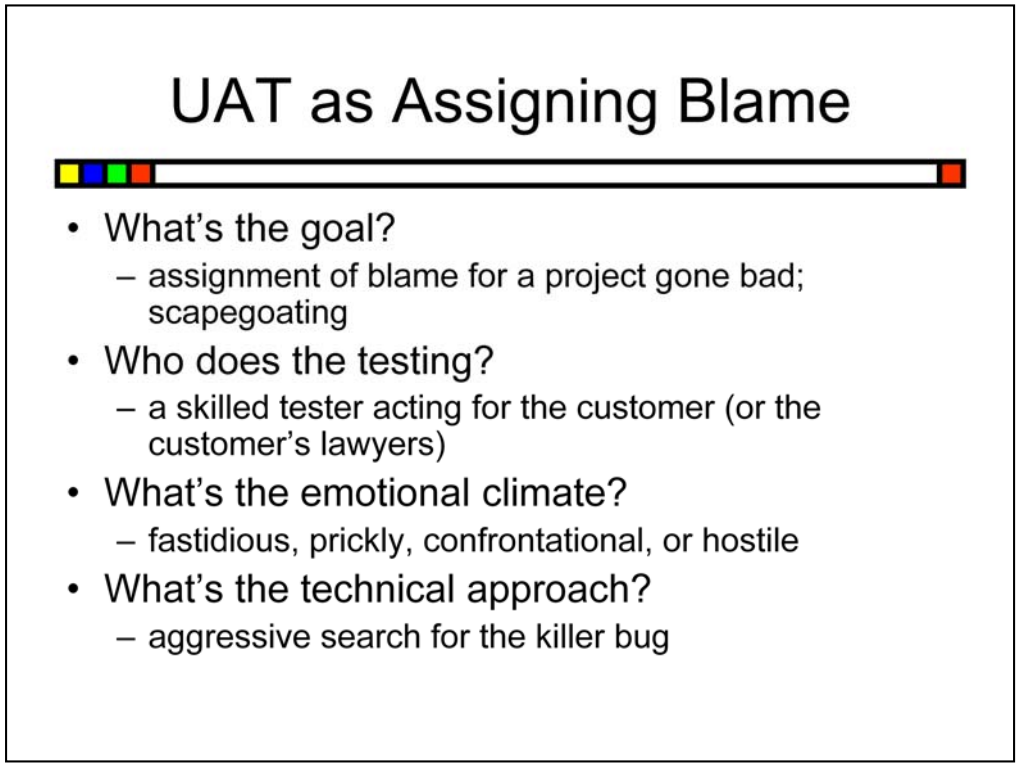



Validation seems to be used much more often when there is some kind of contractual model, where the product must pass a user acceptance test as a condition of sale. At this point, assuming a typical product, it's late, people are tired, lots of bugs have been found and fixed, and we just want to get it over with. There's lots of pressure to get it done, and there's lots of 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?

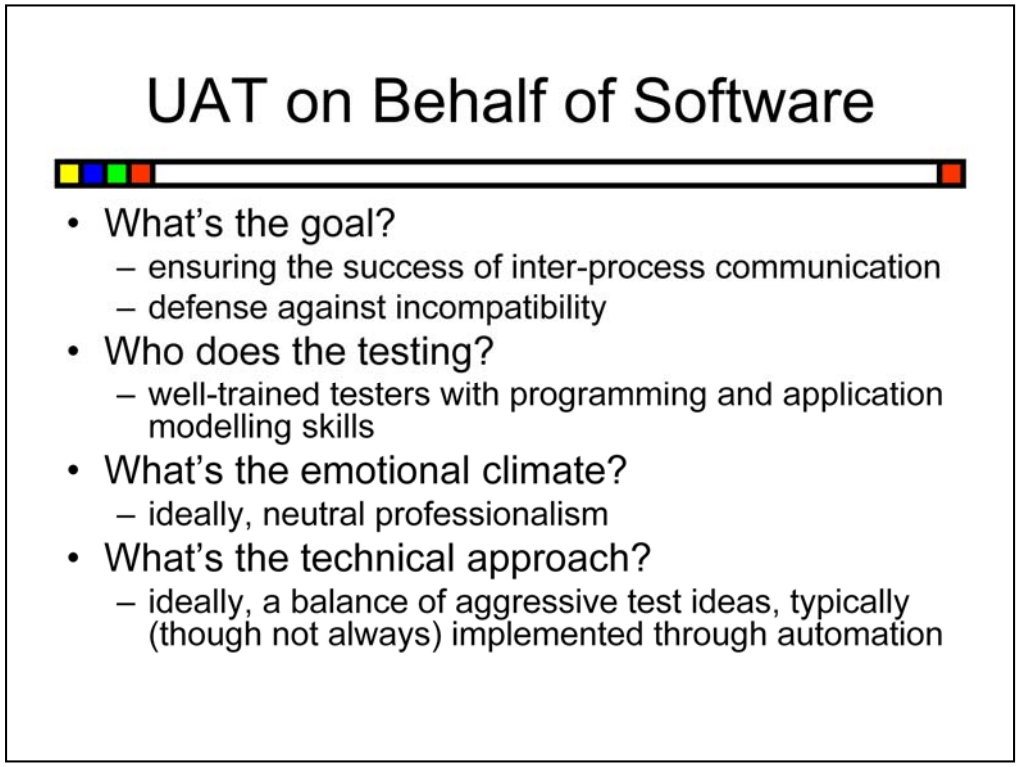



My final take about the validation sense of UAT: when people describe it, they tend to talk about validating the requirements. There are two issues here. First, can you describe all of the requirements for your product? Can you? Once you've done that, can you test for them? Are the requirements all clear, 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*.

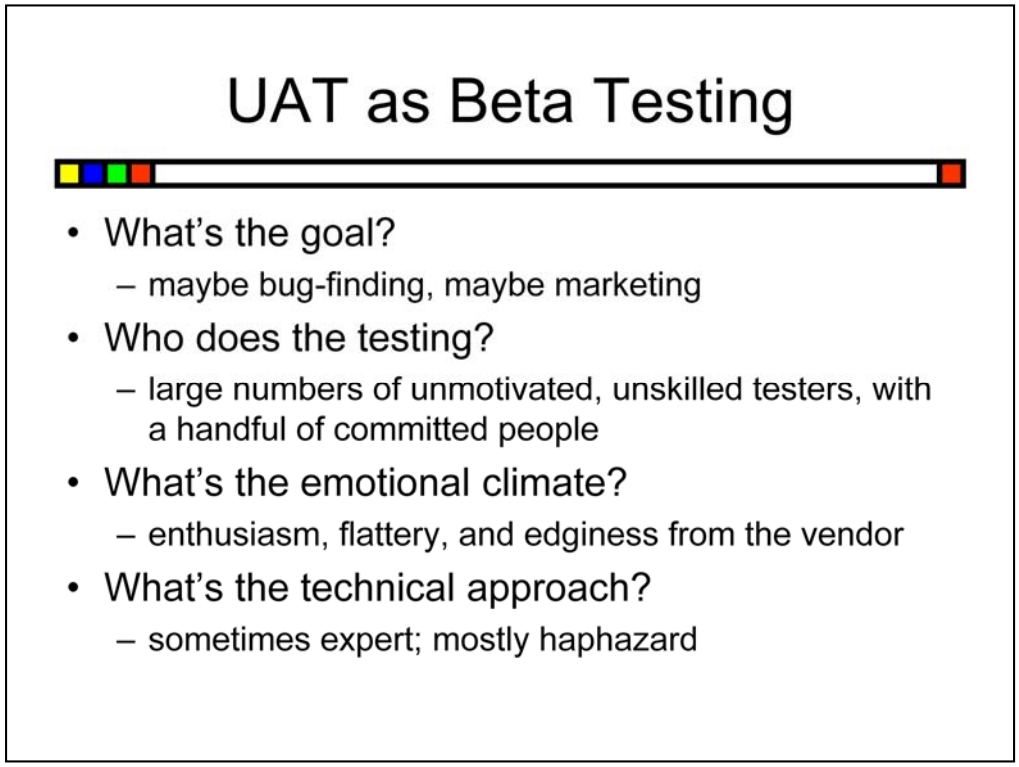
Second, shouldn't the requirements be validated as the software is being built? Any software development project that hasn't attempted to validate requirements up until a test cycle, late in the game, called "user acceptance testing" is likely to be in serious trouble, so I can't imagine that's what 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.




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.



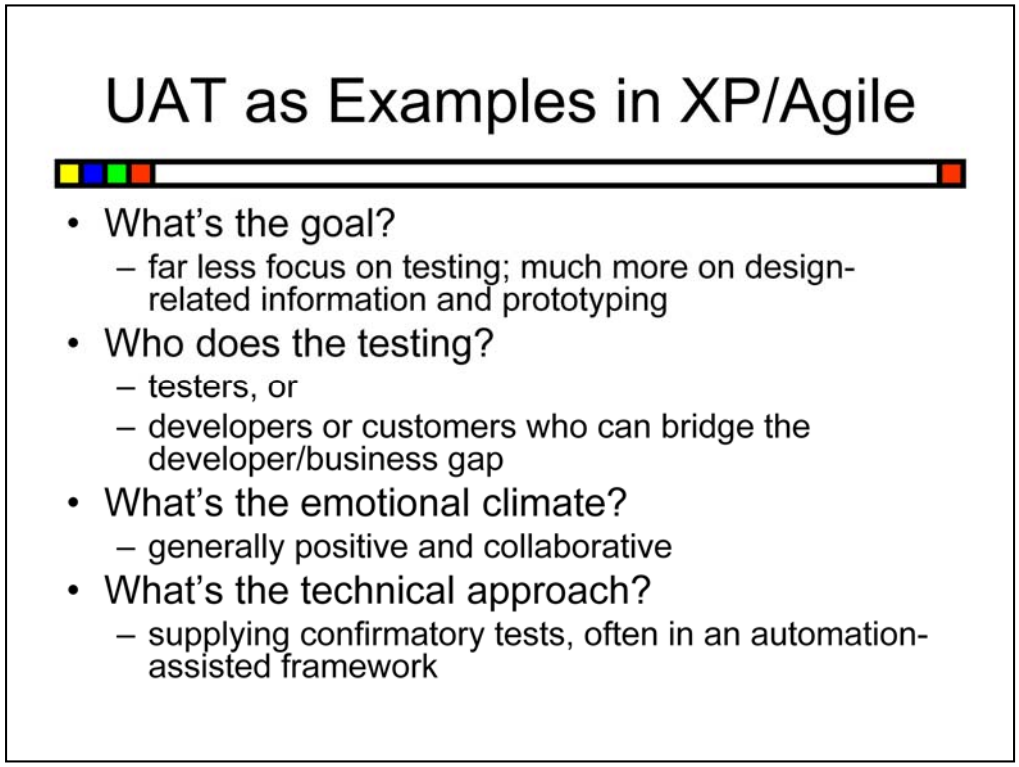

There is yet another sense of the idea of UAT: that the most direct and frequent user of a piece of code is not a person, but other software. In his book "How to Break Software"—one of my favourites—James Whittaker talks about a four-part user model, in which humans are only one part. The operating system, 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.




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.

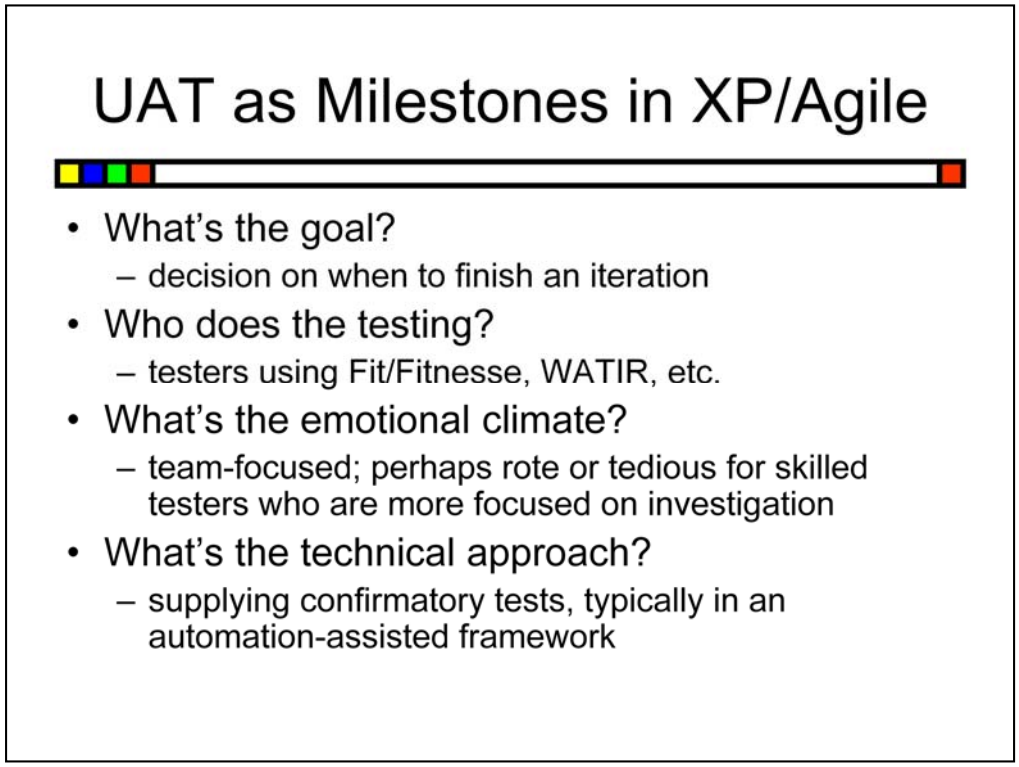
Microsoft regularly releases beta versions of its software (yes, I know, "and calls them releases"). Seriously, this form of user acceptance testing has yet another motivation: it's at least in part a marketing tool. It's at least in part designed to get customers interested in the software; to treat 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.



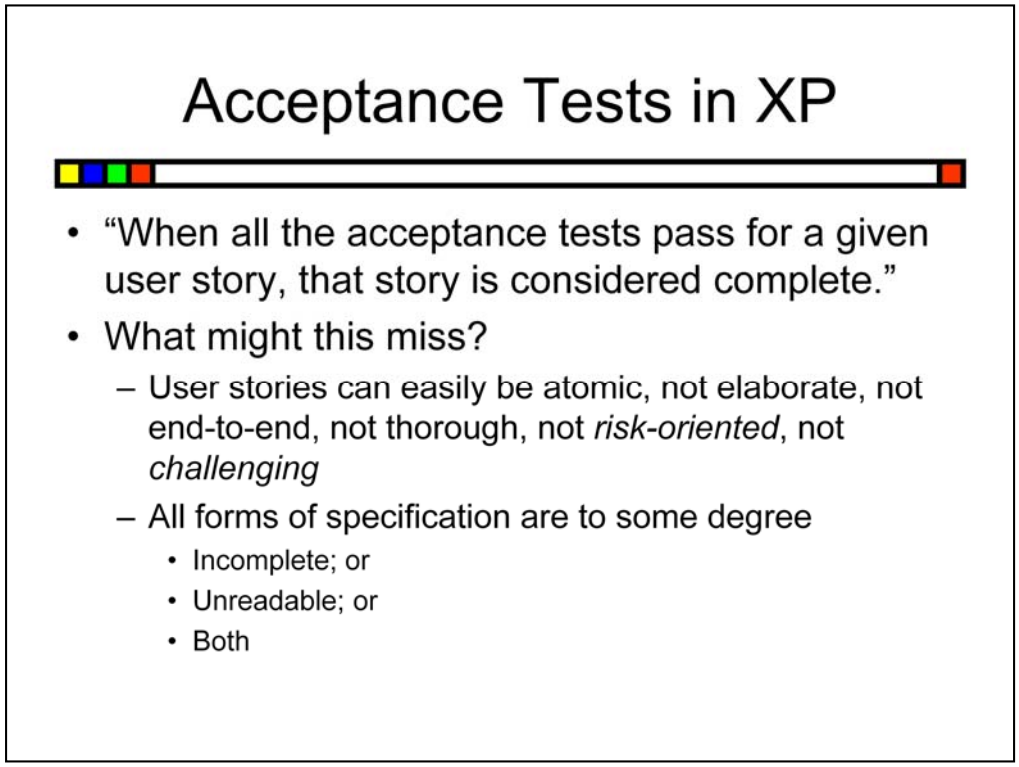

In the Agile world, it's becoming increasing popular to create requirements documents using a tool called Fit (which works with Excel spreadsheets and Word documents) or Fitnesse, which works on a wiki Web page. The basic idea is to create code and a series of tables of data that exercise the product and ask it questions about its state. Developers write code (called "fixtures") that feed the tabular data to the application. The tables are stored in Word or Excel or in a simple markup file within a Wiki page, and they're intended to be run frequently. Upon being run, the tool adds colour to the cells in the table—green for successes, and red for failures.

A number of the Agilistas 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…




Fitnesse tests are sometime used as milestones for the completion of a period 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." I agree that the code is ready to go, but where? I contend that 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?"

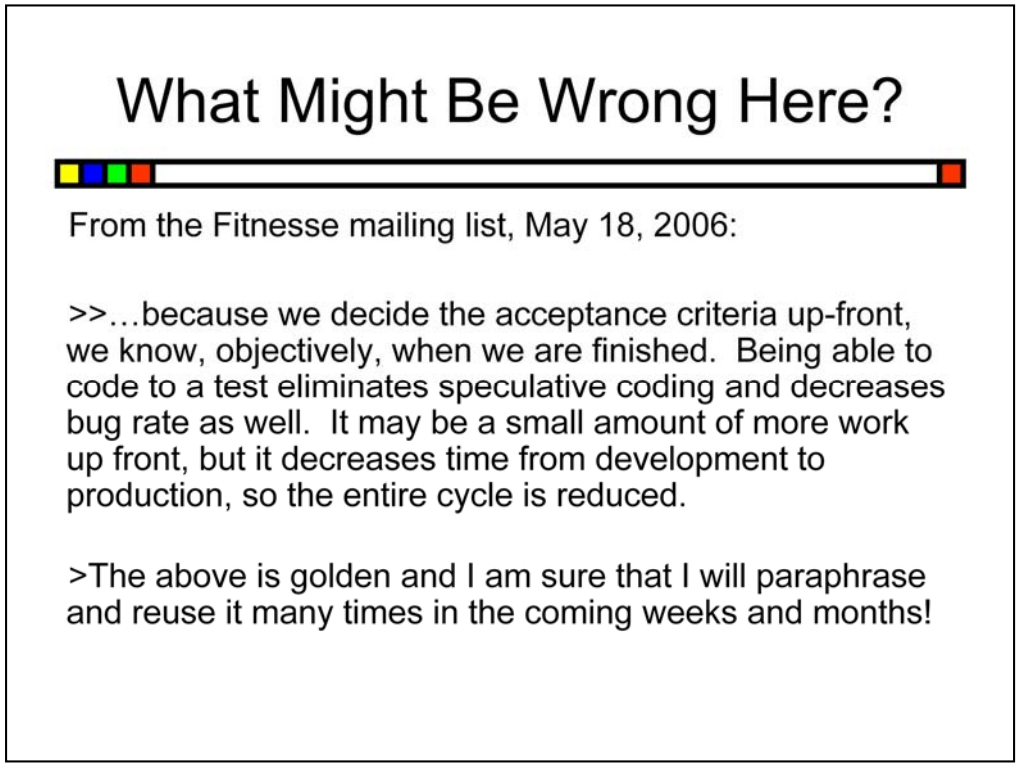
My personal jury is still out on Fitnesse. It's obviously a useful tool for recording test ideas and specific test data, and for rerunning them frequently. I often wonder how many of the repeated tests will find or prevent a bug. I think that when combined with an exploratory strategy, Fitnesse has some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



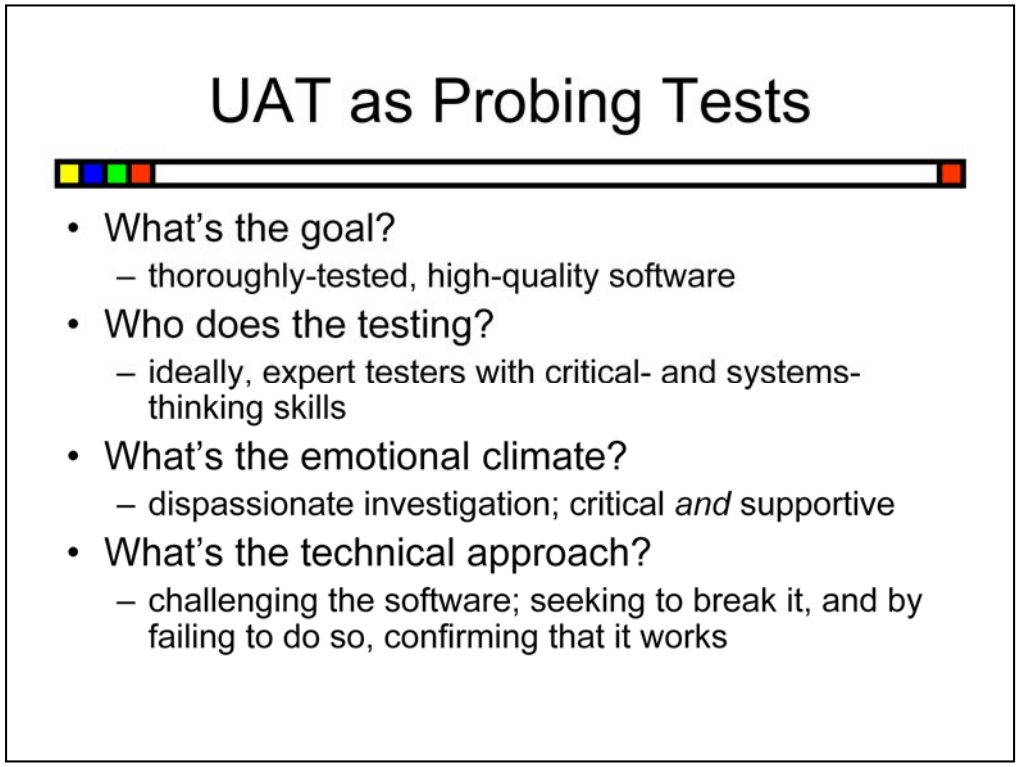

I've certainly had to spend a lot of time in the care and feeding of the tool, time which might have been better spent testing. There are certain aspects of Fitnesse that are downright clunky—the editing control in the Wiki is abysmal. I'm not convinced that all problems lend themselves to tables, and 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?

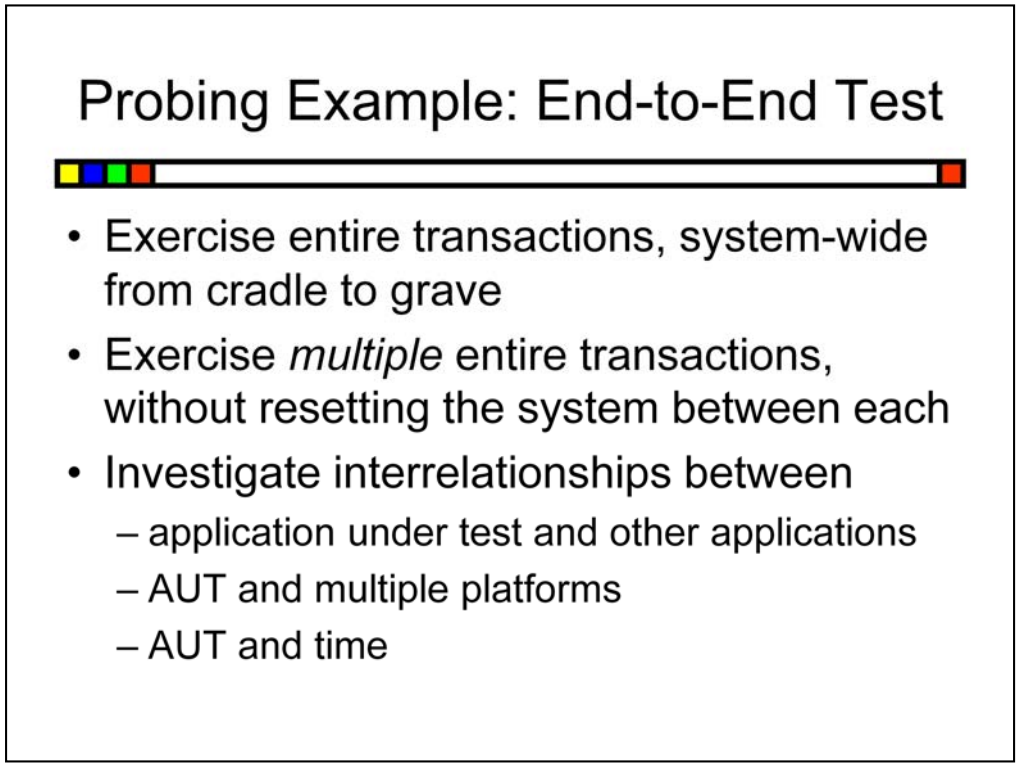
So: my point of view is that Fitnesse is a terrific thing, and can help testing enormously. It's just not the *only* thing.




A long time ago, I learned a couple of important lessons from two books: The Art of Software Testing, by Glenford Myers. The first is that testing, real testing where we're trying to find problems, depends upon us searching for failures. We have to find ways to break the software. All good books on software testing, in my estimation, repeat this principle. We can't prove the conjecture that the software works, but at least we can disprove the conjecture that it will fail. Tests designed to expose failures are powerful, and when those tests themselves fail, we gain confidence. Tests that are designed to pass are relatively weak, and when those tests pass, we don't gain much confidence. In his book The Craft of Software Testing, Brian Marick gave an excellent example in which one powerful hostile test can provide seven different points of confirmation.

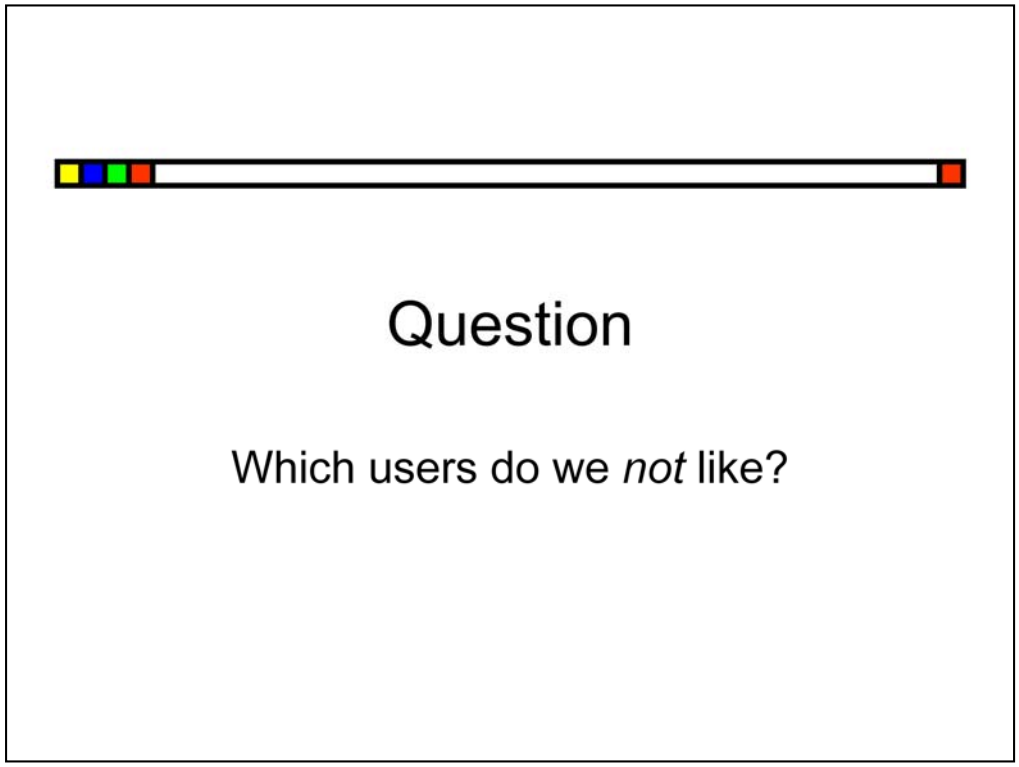



Recently, I've been involved with an organization that produces software that is used by a passel of big banks. User acceptance testing here takes a couple of months. There's a month in-house, when the tables turn and the developers are primarily in the business of supporting, rather than the other way around. That's called User Acceptance Testing. Then there's a month of testing at the big banks that use the software. It's called User Acceptance Testing. So it's not short, and it's not ceremonial. At the core, it's not a terribly complicated application, but since it involves thousands of transactions of some quantity of money each, since there are fraud and privacy issues on the table, there is real risk. After we've spent a month testing aggressively, the banks typically take several weeks to run their own tests. In this case—if you're a tester working for the banks, you are actively trying to find problems during the user acceptance test. Here you really a defender of the gates; heaven forfend, but if we've screwed up somehow, our software could sink your software.

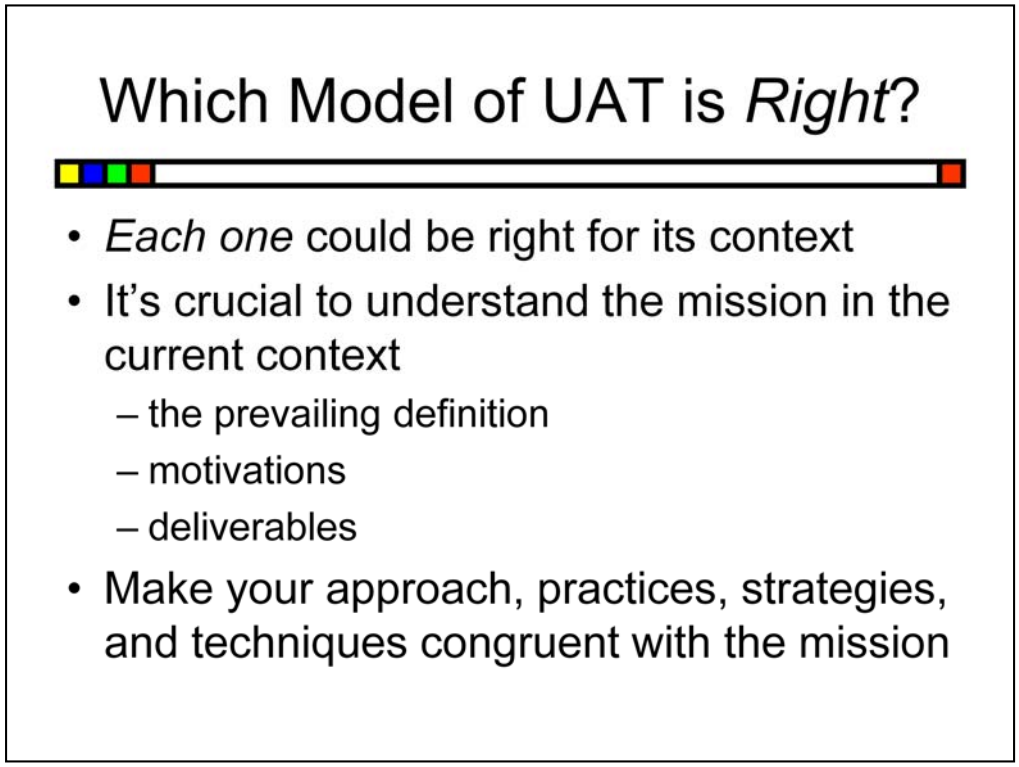



Still, there's a risk that the model of testing in this context, to my mind, can be heavily biased towards confirmation—making sure that stuff works—rather than investigation—finding out new information about the software's behaviour, exposing it to weird input, resource starvation, and heavy loads; in practice making sure that stuff doesn't fail. 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.

In these contexts, 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, great 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 and hunches that lead towards finding bugs.




Make a catalog of people that we don't like. We don't like novice users, and we don't like experienced users either. Be as extensive and/or as funny as possible. Then consider seriously: *how are we failing to meet the needs of these people when we test?*




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. Context-driven thinkers would support descriptive dictionaries, rather than prescriptive ones.

Instead, context-driven testers should use some heuristics to help sort out the nature of the task at hand. 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?" We try to look carefully at what people say, and how different cultures practice this. 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.

# References

- Kaner's papers:  www.kaner.com
- Bach's papers: www.satisfice.com
- The Shape Forum: www.geraldmweinberg.com/shape.html
- Lessons Learned in Software Testing (Kaner, Bach, and Pettichord)
- Black Box Software Testing and Software Testing Techniques (Beizer)
- Top Ten Challenges of Software Testing (Perry and Rice)
- *Any* and *all* books by Jerry Weinberg
- My own Web site at www.developsense.com

# Questions and Comments?

- Here and now
- Later, here at the conference
- By email: mb@developsense.com
- By phone: +1 (416) 656-5160
- The Software-Testing mailing list
  - register at groups.yahoo.com
- The SHAPE Forum