How To Talk About Software Testing

By James Bach, Satisfice, Inc.

Managers, developers, and even testers often have questions about testing that need answers:

- "Why didn't we find that bug before we released?"
- "Why don't we do prevention instead of testing?"
- "Testing would be better if we used {X} practice, just like {successful company Y}!"
- "Why can't we automate all the testing?"
- "Why does testing take so long? How much testing is enough?"
- "Why do we need dedicated testers? Why don't we just get user feedback instead of testing? Why can't developers do the testing? Why don't we just get everyone to test?"

Questions like these often arise when there is trouble in the organization that seems to come from testing or to surround the testing process. But to deal with these questions you first must understand what testing is and what it isn't. Then you must understand the parts of testing, how they relate together, and what words describe them. Only then can you productively talk about testing without being crippled by unhelpful concepts that would otherwise be embedded in your speech.

This document has two sections:

- Testing Basics (stuff you need to know about testing to talk about it coherently)
- Testing Conversation Patterns (kinds of conversations you might have about testing)

Testing Basics

A powerful definition of testing is *the process of evaluating a product by learning about it through exploration and experiment.* That means it's not the same as review, inspection, or "quality assurance" although it plays a role in those things.

By "evaluate" I mean primarily identifying anything about the product that is potentially troublesome. Another word for potential trouble is risk. Testing is therefore a process of analyzing business risk associated with the product (from now on let's just call that "product risk").

1. The essence of testing is to believe in the existence of trouble and to eagerly seek it.

Testing is an inherently skeptical process. The essence of testing largely lies in how you think about what you see. A tester should be negatively biased (because it is usually better to think you see a problem and be wrong, than to think there is no problem and be wrong). When a competent tester sees a product that appears at first glance to work, his reaction is not "it works!" but rather that "it's possible that it's good enough to release, but it's also possible that there are serious problems that haven't yet been revealed." Only when sufficient testing has been done (meaning sufficient coverage with sufficiently sensitive oracles relative to business risk) can a tester offer a well-founded opinion of the status of the product.

Testing never "proves" that the product fulfills its requirements. This is part of a profound truth about science in general: you can disprove a theory about the state of the natural world based on one single fact, but you can never prove that the theory is true, because that would require collecting every possible fact. Consider a barrel of apples. You can pick one apple and see that it is rotten, and from that you could conclude that the barrel fails to fulfill the requirements of containing only fresh apples. But if that one apple is fresh, you could not conclude that all the other ones are also fresh.

The testing process does not determine that the product is "done" or ready to be released. In other words, there is no way to establish unambiguous, algorithmic, or mathematical criteria that can dictate a good and responsible business decision about releasing software. Instead testing uncovers the data needed for management (meaning whoever has responsibility to make the release decision) to make a sufficiently informed decision to release. Deploying software is always a complex business decision rather than a simple technical one.

The testing process is all about learning. Anyone who is not learning while testing is also not testing. We seek to learn the status of the product, of course, but we also seek to learn how we might be fooled into overlooking important problems. Good testing requires continual refreshment of our means of detecting trouble, based in part on studying the bugs that were found only after the product was released.

The mission of testing is to inform stakeholders. Specifically, testing exists to provide stakeholders with the right information to make sufficiently well-informed decisions about the product. The testing process itself does not and cannot determine if the product is "good enough to release," since that is strictly a stakeholder decision.

Why this matters: *When we fail to understand and respect the essence of testing, management and other outsiders to the process will have inflated expectations about what testing can provide and may use testing as a scapegoat for problems that originate elsewhere.*

2. A test is an experiment performed by a person on a product.

Consider a "play" in the theatre. A script for a play is often called a play, but the real play is the performance itself, with the actors on the stage. People may even speak of the "play" as a set of performances (e.g. "the play ran for four weeks on Broadway").

Think of a test in the same way. We can informally speak of a test specification as a "test," but try to remember at all times that the test specification never fully specifies the actual test that is performed, because that involves human attention and judgment. We can also speak of the "same test" over time even though the test is evolving and perhaps never performed exactly the same way twice.

Just be aware that, in truth, a “test” isn’t really a test except in the moment it is being performed. That’s when it becomes real, for better (when a skilled and motivated tester is performing it) or for worse (when an incompetent or inattentive tester is at the wheel). Many tests begin as open questions and sketches and become more defined and systematic with time. Testing is inherently exploratory, just like the development process for the product itself. This is even true for tests that employ automated elements, since that automation must be prototyped and debugged and maintained.

Why this matters: *This definition frames testing as an inherently human process that can be aided by tools but not fully automated. Defining it that way helps defend the testing process against attempts to oversimplify and dismiss it.*

3. The motivation for testing is risk.

If there is no product risk, then there is no need to test. Testing begins with a sort of faith that product risk exists; usually stemming from various risk indicators, such as the existence of complexity and the potential for harm if the code behaves badly. As testing proceeds, and finds problems or fails to find them, faith about risk becomes replaced with fact-based reasoning about risk, until that very process of reasoning leads testers to the conclusion that enough is known to allow management to make reasonably informed decisions about the product.

The ability to think systematically about risk is therefore a key to professional testing.

Why this matters: *Although risk is fundamental to testing, few people are trained to think about risk. This leads to haphazard test strategy and wasted effort. Meanwhile, any time you design a test you must be able to answer this question: why does this test need to exist? If your answer is “because the product might not work” that’s not good enough. What specifically won’t work? In what way might it not work? Is that sort of failure likely? Is this the only test that covers the product? What specific value does this specific test bring?*

4. Anyone who does testing is, at that moment, a tester.

Doing testing well requires certain intellectual, social, and technical skills, but-- just as with cooking--literally anyone can do testing to some degree, and literally anyone can contribute to an excellent testing process.

Some people specialize in testing and are full-time testers. But for the purposes of this document, the word “tester” applies to anyone-- developer, manager, etc.-- who is currently engaged in an attempt to test a product.

It’s worth distinguishing between two kinds of testers: responsible and supporting. A responsible tester is any tester who is accountable for the quality of his own work. In other words, responsible testers normally work without supervision. Responsible testers also decide on their own tactics and techniques, and make their own decisions about when to stop testing. Supporting testers are not expected to control their own test strategy. They include anyone who drops in to help the testing process or any other tester who works only under the direct supervision of a test lead. Often the task of writing formal bug reports is reserved for responsible testers.

Why this matters: *For many years, testing was handled mostly by specialists, who had the time and focus to learn deep truths about testing. But with the advent of “Agile,” many people now get involved with testing without making it their specialty and without having the time to devote to planning or self-improvement as a tester. That means it is especially important to discuss and review the essentials of testing explicitly, as an organization, rather than assuming that testing will automatically be performed to a professional standard by people who are steeped in their trade.*

5. Testing is not verification; testing includes verification.

You can only verify something that has a definite truth value: true or false. But the quality of a product does not have a definite truth value, partly because “quality” is a multidimensional concept that involves subjective tradeoffs. You can verify that a movie has received 46.7 on the “tomatometer,” but you cannot verify that the movie is actually worse than one that received 53.2 from the same website. Just as people can reasonably disagree about the quality of movies, people disagree about the quality of software. In that sense you can assess quality-- you can come to a fact-based judgment of it-- but not verify it. Quality cannot meaningfully be reduced to a single dimension. For instance, I can verify that, given “2+2” as input, just after startup, a particular calculator at a particular time displayed a “4” on its screen. But this is not the same as verifying that “addition works.” Another way of saying this is that verification establishes facts, but no set of set of facts with finite coverage is logically equivalent to a sweeping generalization.

In Rapid Software Testing methodology, we call verification “checking,” which is short for “fact checking.” But testing is bigger than checking. Testing does make use of verification as a tactic, but testing also involves critical thinking, tacit knowledge, and social competence. Testing involves hunches. Testing is a process of sensemaking and theory-building based on the facts that are observed. This is far more than simple verification.

Unlike verification, testing does attempt to make reasonable generalizations— leaps of judgment— grounded in facts, that management can use as one basis for decisions. Testing results in an assessment of the quality of a product.

Why this matters: *Verification is often easy. Testing is hard. It can be seductive to perform fact checks instead of tests because checks involve no judgment and therefore no possibility of anyone disputing your judgment. They are safe and objective. But that very quality also makes them systematically shallow and blinkered. A strategy focused solely on verification would fail to notice big product risks that any reasonable human tester should detect and investigate.*

6. Performing tests is just one part of testing.

Any time you are experimenting and exploring a product for the purpose of discovering bugs (which includes using automation to help you do that) you are performing tests.

But here are some things that are also testing: conferring with the team, designing tests, developing data sets, developing tools, using tools to create and perform checks, studying specifications, writing bug reports, tracking bugs, studying the technology used to create the product, learning about users, planning, etc. Any of these activities are testing when performed for the purposes of fulfilling the mission of testing.

In other words, testing is a bigger process than merely performing tests, and you can't just point to a set of test cases and honestly say "that's the testing, right there." It isn't. Testing is the entirety of your process that delivers the value of the testing mission.

Why this matters: *Good testing requires that testers have the time and resources to learn, model, design, record, collect, discuss, etc. The assumption that the process consists simply of writing test cases and then running them is not just wrong but terribly damaging. It forces testers to do rushed, bad work that results only in shallow testing that will miss important bugs.*

7. A responsible tester must think differently than a developer.

A developer must think of one good way to make things work; a tester strives to imagine 999 ways it could fail. This should not be characterized as "constructive vs. destructive" since the tester is in no way destroying anything (except maybe our illusions of confidence). The distinction is more "optimistic vs. pessimistic" or "imperative (do this!) vs. hypothetical (what if?)"

So, testers live in a world of many hypothetical failures, almost all of which don't happen, but many of which we are obliged to investigate-- because some happen. To a developer's eye, testing can look like an endless, fruitless loop. Where does it ever end? In fact, a major aspect of skilled testing is knowing how to make a case that it is time to stop testing; this is rarely a simple determination.

One way to put it is that development feels like a convergent task, moving toward completion, whereas testing seems to push in the opposite direction: opening up new possibilities and looking into each one, which leads to even more possibilities, and so on. For the same person to think like a developer and like a tester at the same time is quite difficult, even painful, requiring extraordinary discipline. This is not to say that developers should not include unit level checks in their code. Those are important. But unit "tests" are not really tests, they are low-level fact checks, and bear little resemblance to the skeptical and creative process of testing.

Testers and developers necessarily work by different incentives: the thrill of building something obviously good vs. the thrill of discovering hidden flaws. When they work together, they can build something that truly is good and doesn't just seem that way at first blush.

Coding is a useful skill for testers, but not all testers should be coders. Testers who are coders often focus on writing tools to help testing instead of directly and interactively testing the product. Testing benefit from diversity, including all forms of diversity, but especially cognitive diversity represented by some people who think more like developers and understand technology more deeply, as well as some people who focus on the behavior and look of the product and understand the users more deeply. It's good to have some people who want to "get things done" (even if the work is not the best it could be) and others who want to "do the job right" (even if it takes longer to get things done).

Why this matters: *The reason people specialize as testers is to allow them to focus on problem discovery without being hindered by biases that come from hoping that the product will work, or believing that it cannot fail, or being too tired from making it work to put the required energy in to detecting failure. For any non-specialist who occasionally does testing, these are dangerous biases that must be managed.*

8. The five parts of a test are: tester, procedure, coverage, oracles, and the motivating risk.

- **Tester.** There cannot be a test without a tester. The tester is the human being who takes responsibility for the development, operation, and maintenance of the test. The tester is not just a steward for the test, the tester is part of the test. The tester's judgment and attentiveness are a vital part of any good test. A corollary to this is that when a tester gives a test to another tester, that changes the test. Two competent testers can follow the same formal test procedure, but they will not be performing exactly the same test, because each test depends on a variety of human factors to make it work. Another corollary to this principle is that giving a good test procedure to a low-skilled tester will diminish or even destroy the value of that test; whereas giving a poor test procedure to an excellent tester may result is good testing getting done, because that skilled tester will automatically correct the design or compensate in some other way.

- **Procedure.** A procedure is the way that a test is performed. The procedure may or may not be written down, but it must exist, or there won't be a test. Even if a procedure is written down, the parts performed by a tester (as opposed to that done by tools) will always involve unwritten elements supplied automatically by the know-how of the tester. If tools are used to perform a test, then the tools are part of the test procedure.

- **Coverage.** *Coverage is what was tested.* There are many kinds of coverage, but among them are these broad categories: structure, function, data, interfaces, platform, operations, and timing. You need to know, at least in broad terms, what your tests cover and what they don't cover. Code coverage is one kind of structural coverage, and there are tools which can measure that. Data coverage, by contrast, is not so easily measured, since there are so many kinds of data and they can be combined in so many ways. Measuring that kind of coverage would require keeping track of all the data you test with and comparing that with all the data you could have used.

  All coverage is based on a model, by which I mean some particular way of describing, depicting, or conceiving of the product; a model is a representation of the thing it models. For instance, code coverage is based on a model of code, which is usually the human readable source code itself. The usual form that models take when discussing test coverage is a list or an outline. A model of browsers for the purposes of browser coverage would be a list of browsers and a list of relevant features of browsers.

  You could say that a model provides a perspective on the product, and to find every important bug we need to test from different perspectives.

  A test condition is defined as something about the product that could be examined during a test, or that could influence the outcome of a test. Basically, a test condition is something that could be tested. If you model the product in some way, such as listing all its error messages, then each error message is a test condition. If you have a list of buttons that can be pushed, then each button is a test condition. Every feature of the product is a test condition, and so is every line of code. We cannot test all possible conditions, but we do need to know how to identify them and talk about them.

- **Oracles.** Oracles are how problems are detected. No matter what your coverage is, you can't find a bug without some oracle. An oracle is defined as a means to detect a bug once it has been encountered. There are many kinds of oracles, each of which are sensitive to different kinds of bugs, but every oracle ultimately comes down to some concept of consistency. We can detect a bug when the product behaves in a manner that is not consistent with something important, such as a specification, or user expectation, or some state of the world that it's supposed to represent, etc. Testing can be characterized as the search for important inconsistencies between the state of the product as it is and as it ought to be.

- **Motivating Risk.** The motivating risk for a test is the probability and the impact of a problem with the product that justifies designing and performing that specific test. All testing is both motivated by our beliefs about risk and all good testing leads to a better understanding of the actual risk posed by the product. Ultimately, the purpose of testing is to establish and maintain a good understanding of risk so that everyone on the team, including developers and management, can make the right decisions at the right time to develop the product.

***Why this matters:** Discussing, evaluating, and defending your tests begins with understanding these five elements that every test must have. When you explain your tests and show how they fit in the overall test strategy, you will have to speak to each of these essential elements.*

Conversation Patterns for Testing

"Why didn't we find that bug before we released?"

This may be interpreted as:

1. That is the sort of interesting bug we would have wanted to find before release.
2. We tried to find every interesting bug before release.
3. Yet, we did not find that bug, which is disappointing.
4. Something about our process must have been wrong, which led to that disappointment.
5. What went wrong?

But premise #4 is incorrect. It is not necessarily true that something must go wrong for a bug to remain undetected. That's because testing is necessarily a sampling process. We cannot test everything, and we don't even try to do so. We make educated guesses about risk, and while we can improve our education and make better guesses, we can't remove the element of guessing entirely. In other words, even the best test process that it is possible to perform may lead to some disappointment. Testing just isn't a sure thing.

However, it may very well be true that something did go wrong with our process, especially if the bug that escaped is a particularly bad one. Treat every escaped bug as an opportunity to ask healthy and constructive questions about what happened and why.

Suggested response: "Let's look into it and find out."

No need to feel defensive about this. No one can rationally expect perfection, but our clients can expect testers to learn from experience. But beware of reducing the situation to a simple one-dimensional explanation. When investigating and discussing the matter more deeply, keep these issues in mind:

1. **Testability.** Was there anything about the design the of the product, or organization of the project, which allowed this bug to hide from us. Is there some identifiable improvement in those things that would make it harder for future bugs to hide?
2. **Opportunity cost.** If you had done what was necessary to find that escaped bug, would other bugs have escaped instead?
3. **Hindsight bias.** When analyzing how to avoid future bugs, it is tempting to focus on the exact circumstances of the bug that escaped, because you know that bug actually occurred. But that is too narrow. The next bug to escape won't be exactly like that one, but may be similar, so think about the set of all bugs that are similar but not the same, and what you could do to find any future bug belonging to that set.
4. **Automation.** Is there an automatic way to catch bugs like that? Are such bugs important and prevalent enough to justify that automation?
5. **Tester Agency.** Does the reason the bug escaped imply anything about the focus, commitment, or skills of the tester? Perhaps the test strategy and tool set are fine, and the tester just had a bad day. Or perhaps no one has taken enough ownership of the testing process.

"Why don't we do prevention instead of testing?"

On its face, this question asserts a false choice. But the questioner probably meant to say "Perhaps it would be better to put more of our effort into prevention and less into testing."

Suggested response: Affirm the value of both activities, but turn the conversation to the central issue, which is risk: "We must do both. For instance, we do lots of things to prevent our building from burning down. We have circuit breakers to prevent electrical fires. But we also have smoke alarms and fire stations, just in case our prevention measures fail. The depth of our testing should relate to the potential risk we perceive, and as that risk falls, so might our investment in testing. Of course, the ultimate prevention is not to develop a product in the first place. Assuming we want to create something new in the marketplace, however, a certain amount of risk is inevitable."

"Testing would be better if we used {X} practice, just like {successful company Y}!"

No matter what field of craftsmanship you look at, there are a few things that you will always see. One of them is practitioners who are aware of a variety of practices (i.e. methods, tools, and any other heuristics that are available for use) and make choices about which ones to use in which situations. Methodology should be context-driven, not driven by blind pursuit of whatever is popular at the moment.

When someone suggests that there is a successful company (Facebook and Google are often cited) that gets its success through not doing certain things or always doing other things, this is probably not coming from a consideration of problems and how to solve them. It's based on incomplete rumors. We don't know what Facebook and Google actually do; nor why they chose to do it that way. And we aren't in a position to evaluate that. We may be hearing a self-congratulatory myth.

Furthermore, what makes companies successful is primarily their business models and intellectual properties, not their engineering excellence. Famous companies can generally afford to be shockingly wasteful and still make a profit. And anyone who has worked in a famously successful company can tell you it's a constant challenge to get people to believe in the possibility of failure. Success creates a haze of complacency which is toxic to process improvement.

Suggested Response: Affirm that good ideas can come from anywhere. Remind that responsible technical organizations probably do not thoughtlessly follow trends for the sake of trends. Then offer to seriously consider the practice. If a team wants to try it out, encourage them to make an experiement. But keep certain caveats in mind and be prepared to discuss them:

1. **Skills.** Does the new practice require special skills to use correctly? Is any outside training required?
2. **Opportunity cost.** If you deploy this practice what will be pushed aside? What will you not have time to do?
3. **Evaluation.** How will you know if it is going well? How will you detect new problems that are created?
4. **Progress horizon.** How much time do you need to give the experiment in order to declare it a success or failure? When is it reasonable to expect results?
5. **Required support.** How much cooperation and what infrastructure is needed to try it out? Can it be done on the cheap?

"Why can't we automate all the testing?"

This is a reasonable question if you believe that testing is the same as fact checking. Fact checking *can* be automated, although it might be expensive and slow to create and maintain it. Nevertheless it is possible.

But testing is much more than fact checking. Testing encompasses the entire process of building mental models of the product and of risk that are required in order to make a compelling, credible report about the readiness of that product for release. Remember, you are not verifying facts when you test, you are making an assessment.

The short reason we can't automate "all the testing" is that to make an assessment of a product is a non-algorithmic, exploratory, socially situated learning process. But here is a slightly longer version of that: Some bugs are easy to anticipate and easily triggered, and easy to spot when they manifest themselves (call those bugs "shallow"), but many bugs are not easy to anticipate, or not easily triggered, or not easy to detect unless you know just what to look for (call those bugs "deep"). Testers are not just looking for shallow bugs, but for all *important* bugs. To find deep bugs requires insight of a kind that often comes only after playing with the product for an extended period of time. It may require noticing something strange and following-up on it. It may require doing complex operations that are easy for a human to do, yet expensive to automate. No good tester has all the good testing ideas right at the beginning of a project. Meanwhile, to automate something requires that you have a specific formal procedure that will spot every kind of important bug. There is no such thing. Big bugs don't follow any pre-ordained set of rules, and without rules we don't know what code to write to find them.

One more reason we can't just automate all the testing is that testing requires social competence. A test must make judgments about what kind of data matters, what kinds of problems matter, which testing is more important or less important, etc. These decisions are based on an understanding of the shifting and evolving values of the stakeholders of the product. These decisions must be defensible. All that requires social competence.

Automated checking only succeeds as a substitute for testing where there are no deep bugs that matter. Wherever you find that, it’s usually in a very stable codebase or in a product that has customers who are tolerant about failure.

Suggested response: “Look. We can’t automate parenting, management, getting to know people, falling in love, grief counseling, medical care, government, and no one is wondering when we can fire all the programmers and replace them with automated programmers. So, why do you think a skilled activity like hunting for a massive and completely unexpected bug is an exception to that? We can automate lots of interesting things that are part of testing, however. Wherever we can do that in a cost-effective way, we should.”

“Why does testing take so long? How much testing is enough?”

These questions can only be answered in context. Some testing takes longer, and some testing can be done quite quickly. It depends on many factors, all of which can be called aspects of testability. See the *Heuristics of Testability* document for details.

Suggested response: “Which specific testing are you talking about? If we are talking about something specific, then we can reach a specific answer. Otherwise here is the general answer: testing takes however long it takes for the tester to build a compelling enough case that the product is tested well enough so that the stakeholders are able to make decisions based on a good enough knowledge of the risks associated with the product.”

“Why do we need dedicated testers? Why don’t we just get user feedback instead of testing? Why can’t developers do the testing? Why don’t we just get everyone to test?”

Often the foundational issue that gives rise to this question is a lack of understanding about what skilled testing is and what skilled testers do. The questioner may consider testing skills to be ubiquitous. The questioner may really be saying “since anyone can test as easily as anyone else, why bother with full-time testers?”

However, it may be that the lack of understanding is about roles: maybe the questioner is questioning the very idea of having people specialize or focus on any one activity, rather than fluidly moving from one activity to another over time. See the *How to Think About Roles and Actors* document for a list of dynamics that affect roles and the people who play them. For instance, one benefit of a having a person specialize in a role is *readiness*. If you only test once-in-a-while, you are probably not looking ahead and planning and preparing to perform testing. While a dedicated tester would be preparing, a casual tester if doing some entirely different job.

Suggested responses: “Dedicating people to a complex task improves their efficiency. This involves the improvement of competence, commitment, readiness, and coordination. Of course, in a low risk situation, we might not need such efficiency and effectiveness. Yes, we can get feedback from users, too, and we should, but you still need people who can process that information. Users are terrible bug reporters, and developers don’t have the time and usually not the patience to follow-up on all those half-baked reports. Certainly, everyone on the team can help testing. But someone needs to be responsible for it, or it will turn into chaos.”