8

What Makes a Good Test?

"There is nothing either good or bad,
but thinking makes it so."

—William Shakespeare, English Dramatist and Poet (1564–1616), *Hamlet,* Act 2, Scene 2.

What makes a good test? The question can be answered at a highly technical level, but that's not our goal here. In fact, it's helpful to step back from that debate and look at the question from a management point of view. How can you know if testing is being done well? How much credence can you put in test results?

You can never know for sure.

You may or may not agree with Hamlet that there is nothing either good or bad. For the sake of argument, though, let's suppose there is such a thing as a "good" test and ask the question, How can one know whether a particular test (or set of tests) is, indeed, good?

Let's start with something even better than "good" by looking at a definition of "perfect." A perfect set of tests would have the following characteristics:

a. It would detect every bug in a system.
b. It would never detect a non-bug as a bug.
c. It would give us complete confidence that it has done *a* and *b*.
d. It would accomplish *a*, *b*, and *c* quickly and cheaply enough for our needs.

Now consider a system under test. In the simplest case, if that system were perfectly bug free (a situation likely to exist only in our dreams), then any test that finds no bugs meets condition *a.* Some tests that we would consider really lousy could meet that condition as well, but when run against a bug-free, *perfect* system, they would look pretty good.

Or would they? We don't know in advance whether we're testing a bug-free system or a louse-y system. (If we did, why would we need to test?) So, imagine two sets of tests: perfect tests and lousy tests. When run against our bug-free, perfect system, both sets of tests reveal no bugs. So, on the basis of their bug-finding alone, we couldn't tell the difference between a perfect test and a lousy test.

In fact, what might be an adequate test for one implementation of a system might be a lousy test for another implementation of the same system. In other words, "goodness" cannot be a property of a test, but only a property of the relationship between a test and an implementation.

Going one step further, the same test of the same implementation might be adequate for one organization but lousy for another. For example, a test that is adequate for a system to be used internally by a single organization might be totally inadequate if the same implementation were sold as a software product to a thousand organizations. In other words, "goodness" cannot be a property of tests *and* implementations, but only a property of the *relationship* among tests, implementations, and situations.

So, you can never tell for sure, and you never can tell by looking at a test in isolation, whether a test is good—but you do have many ways to tell whether a test is likely to be bad. Meta-tests play an important role. Later, in Chapter 9, we examine some indicators of "bad" tests.

You can assess goodness only after the fact.

If you knew how many bugs were in a system, you could at least begin to assess the goodness, or not-badness, of a set of tests. For instance, after a system has been in use for a few years, a prudent manager will have statistics on how many bugs were shipped with the system. By keeping track of what bugs turn up in use, then

analyzing them to see what they're like, you will have at least some kinds of information, such as,

- *how good your testing was, and in what ways*
- *how testing might be improved in the future*
- *what kinds of bugs your testing characteristically missed*

Knowing such information allows you to make better estimates in the future, even if you don't improve your testing process. Such information may also be used to improve the development process—although in this regard, most likely I'm dreaming again.

Unfortunately, you never know for sure how many bugs were shipped because bugs can turn up in a product many, many years later, or never. Thirty years after I wrote *The Psychology of Computer Programming,* and after more than 200,000 copies of the published book had been sold, I received a letter documenting an "obvious" error that nobody else has ever pointed out to me. That was a useful lesson in author humility, but it also suggests that similar "obvious" errors can remain dormant in products—especially when the product is software—for a very long time. Software may even have bugs that were not bugs when the system shipped, such as functions that fail when an application is used with new hardware or a new operating system.

Moreover, if you wait five years to assess the goodness of a set of tests, what good is the information? Your testers, if they're still around, probably now use different tools and techniques. You may not have the original source code or bug reports, or you may have them and be unable to read or understand them. But if you wait only six months or a year, and keep good records, you may be able to assess the quality of a set of tests in time for the assessment to be useful in improving future tests.

Can you do better than simply waiting for experience? You can review your coverage and your oracles against your theories of failure. You can vary your tests randomly or arbitrarily and notice how problems emerge. You can compare different kinds of testing in parallel, such as beta testing compared to internal testing, or reviews compared to dynamic testing.

You may want to insert bugs intentionally.

Sometimes, you can gain quantitative estimates of how many problems might remain in software by seeding (or "bebugging," a term I believe I coined—and wrote about—in *The Psychology of Computer Programming).* Insert known bugs without telling the testers, then estimate the number of remaining unknown bugs by the percentage of known bugs they find.

For bebugging to work at all reasonably, the distribution of inserted bugs must closely match the (unknown) distribution of the unknown bugs. What I learned to do was leave (but document secretly) bugs that were made naturally but had been found by the developers, perhaps in code reviews, perhaps in unit testing. This gives the bugs a naturalness, but of course isn't necessarily a reliable sample of not-found bugs. Still, the practice provides some information if some known bugs aren't found. Be careful, though—it doesn't give a great deal of reliable information even if *all* the known bugs are found.

Estimates of goodness are always statistical.

In the end, though, an estimate of goodness is just that—an estimate. We can only estimate goodness statistically, because we can never know for sure how many bugs are, or were, in a given system. The fewer bugs there actually are in a system, the more the statistics are in our favor. For example, if there are ten bugs, a 50-percent error means we might be off by five bugs, but for 14,000 bugs, we might be off by 7,000.

To make your testers look good, you may want to come into test with as few bugs as they can reasonably manage. As an added benefit, testers are free to look for the difficult bugs if they aren't spending time finding the easy ones—noise that should have been removed during unit testing. Unfortunately, many managers judge testers by how many bugs they find, which means that poor-quality systems make testers look better. If they were testing a perfect system, they'd never find any bugs, and probably would be fired as incompetent. Under such a system of measurement, a lousy developer is a tester's best friend.

Why, then, is this flawed system so popular? One reason is the testers' lack of effort to explain how they look for bugs and why their strategy should inspire respect. Another reason is that when managers and developers *assume* that a product works, telling them it works seems to provide no information. That is, "no information" seems to equal "no value." It's the same reason why people don't replace the batteries in their smoke alarms—most of the time, a nonfunctioning smoke alarm is behaviorally indistinguishable from one that works. Sadly, the most common reminder to replace the batteries is a fire.

You can estimate not-badness.

At its deepest technical level, testing involves some rather esoteric mathematics and logic. To assess whether these esoteric activities have been done well, most managers must rely on second and third opinions from independent experts. There are, however, many assessments of not-badness most managers can make themselves by answering the following kinds of questions:

- *Does testing purport to give me the information I'm after?* If it doesn't, it's obviously not good.
- *Is it documented?* If not, have you personally observed the testing, or is it observed, reported, or performed by someone you trust?
- *Is it honest?* There are numerous ways test documentation can be fudged, intentionally or unintentionally.
- *Can I understand it?* If you can't, how can you possibly know whether it's good or bad?
- *Does it at least cover what matters most?* You can't generally test every path (remember the impossibility of exhaustive testing), but at the very least, a set of tests should visit each line of code once.
- *Is it actually finished?* Anyone can check a box on a test plan. Do you have ways of knowing what was actually done?
- *Can I tell the difference between a test and a demonstration?* Demonstrations are designed to make a system look

good. Tests should be designed to make it look the way it truly is.

- *Are trends and status reports overly simplistic and regular?* In real projects, tests and test activities come in many sizes and shapes. If test status reports show extremely predictable trends, testing may be shallow or the reports may be leaving out something important.
- *Are there inconsistencies between different kinds of test activities?* For instance, if beta testers find bugs that the internal test team doesn't find, or if performance testing finds functional bugs, that may be a sign that one or the other process is not working well.
- *Are managers visible?* They shouldn't hang over people's shoulders, but often testing can be improved simply by the presence of a curious manager who is known for paying attention.

Unfortunately, many fallacies and falsifications can distort tests, causing testing to turn sour. Managers need to understand and guard against them, possibly by applying methods detailed in the next several chapters.

Summary

You'll never know for sure whether your testing was done well, but there are many ways to know or estimate if it was done badly.

Common Mistakes

1. *Not thinking about what information you're after:* Testing is difficult enough when you *do* think about what you're after, and more or less impossible when you don't. You won't often know in advance what information you seek—in fact, in most instances, you'll have only an approximate idea. So, you need to think about your ultimate testing goals and about how you're going to *learn* what *other* information you're going to want.

2. *Measuring testers by how many bugs they find:* Testers will respond to this kind of measurement, but probably not the way you intend them to. The quantity of bugs found will increase, but the quality of information harvested will diminish.

3. *Believing you can know for sure how good a test is:* Be vigilant and skeptical when evaluating the accuracy and appropriateness of a test. If you aren't, you're going to get slapped down by the nature of the universe, which doesn't favor perfectionism.

4. *Failing to take context into account:* There are few, if any, tests that are equally significant in all circumstances. If you don't take potential usage patterns into account, your tests will be ineffectual.

5. *Testing without knowledge of the product's internal structure:* There are an infinite number of ways to replicate specific behavior on a finite series of tests. Knowing about the structure of the software you're testing can help you to identify special cases, subtle features, and important ranges to try—all of which help narrow the inference gap between what the software can do and what it will do during actual use. Charles Babbage, the maker of the very first computer, knew this almost 200 years ago, so there's no reason for you not to know it.

6. *Testing with too much knowledge of the product's internal structure:* It's too easy to make allowances for what you think you know is going on inside the black box. Typical users probably won't know enough to do that, so some of your tests had better simulate activities likely to be performed by naïve users.

7. *Giving statistical estimates of bugs as if the numbers were fixed, certain numbers:* Always give a range when stating the estimated number of bugs (for example, say, "There are somewhere in the range of thirty to forty bugs in this release."). Even better, give a statistical distribution, or a graph.

8. *Failing to apply measures of "badness" to your tests:* Use a checklist, asking questions such as the ones in this chapter.

9. *Not ensuring that development is done well:* Poorly developed code needs good testing but usually receives poor testing, thus compounding problems. What use are good tests of shoddy code?

10. *Not considering the loss of testing efficiency caused by numerous found bugs:* A "perfect" testing session is one entirely dedicated to test design and execution (this is the exception, not the rule). Set-up, bug investigation, and reporting take time away from test design and execution. Finding lots of bugs might make testers look good, but finding lots of bugs slows down testing, reduces coverage, or both.