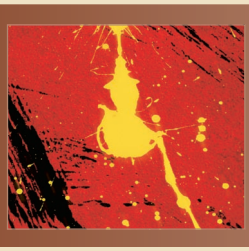
Seven Principles of Software Testing

Bertrand Meyer, ETH Zürich and Eiffel Software



Testing is about producing failures.

While everyone knows the theoretical limitations of software testing, in practice we devote considerable effort to this task and would consider it foolish or downright dangerous to skip it. Other verification techniques such as static analysis, model checking, and proofs have great potential, but none is ripe for overtaking tests as the dominant verification technique. This makes it imperative to understand the scope and limitations of testing and perform it right.

The principles that follow emerged from experience studying software testing and developing automated tools such as AutoTest (http://se.inf.ethz.ch/research/autotest).

DEFINING TESTING

As a verification method, testing is a paradox. Testing a program to assess its quality is, in theory, akin to sticking pins into a doll—very small pins, very large doll. The way out of the paradox is to set realistic expectations.

Too often the software engineering literature claims an overblown role for testing, echoed in the Wikipedia definition (http://en.wikipedia.org/wiki/Software_testing): "Software testing is the process used to assess the quality of computer software. Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate." In truth, testing a program tells us little about its quality, since 10 or even 10 million test runs are a drop in the ocean of possible cases.

There are connections between tests and quality, but they are tenuous: A successful test is only relevant to quality assessment if it previously failed; then it shows the removal of a failure and usually of a fault. (I follow the IEEE standard terminology: An unsatisfactory program execution is a "failure," pointing to a "fault" in the program, itself the result of a "mistake" in the programmer's thinking. The informal term "bug" can refer to any of these phenomena.)

If a systematic process tracks failures and faults, the record might give clues about how many remain. If the last three weekly test runs have evidenced 550, 540, and 530 faults, the trend is encouraging, but the next run is unlikely to find no faults, or 100. (Mathematical reliability models allow more precise estimates, credible in the presence of a sound long-term data collection process.)

The only incontrovertible connection is negative, a falsification in the Popperian sense: A failed test gives us evidence of nonquality. In addition, if the test previously passed, it indicates regression and points to possible quality problems in the program and the development process. The most famous quote about testing expressed this memorably: "Program testing," wrote Edsger Dijkstra, "can be used to show the presence of bugs, but never to show their absence!"

Less widely understood (and probably not intended by Dijkstra) is what this means for testers: the best possible self-advertisement. Surely, any technique that uncovers faults holds great interest for all "stakeholders," from managers to developers and customers.

Rather than an indictment, we should understand this maxim as a definition of testing. While less ambitious than providing "information about quality," it is more realistic, and directly useful.

> **Principle 1: Definition**
> *To test a program is to try to make it fail.*

This keeps the testing process focused: Its single goal is to uncover faults by triggering failures. Any inference about quality is the responsibility of quality assurance but beyond the scope of testing. The definition also reminds us that testing, unlike debugging, does not deal with correcting faults, only finding them.

TESTS AND SPECIFICATIONS

Test-driven development, given prominence by agile methods, has brought tests to the center stage, but

sometimes with the seeming implication that tests can be a substitute for specifications. They cannot. Tests, even a million of them, are instances; they miss the abstraction that only a specification can provide.

> **Principle 2: Tests versus specs**
> *Tests are no substitute for specifications.*

The danger of believing that a test suite can serve as specification is evidenced by several software disasters that happened because no one had thought of some extreme case. Although specifications can miss cases too, at least they imply an effort at generalization. In particular, specifications can serve to generate tests, even automatically (as in model-driven testing); the reverse is not possible without human intervention.

REGRESSION TESTING

A characteristic of testing as practiced in software is the deplorable propensity of previously corrected faults to resuscitate. The hydra's old heads, thought to have been long cut off, pop back up. This phenomenon is known as regression and leads to regression testing: Checking that what has been corrected still works. A consequence is that once you have uncovered a fault it must remain part of your life forever.

> **Principle 3: Regression testing**
> *Any failed execution must yield a test case, to remain a permanent part of the project's test suite.*

This principle covers all failures occurring during development and testing. It suggests tools for turning a failed execution into a reproducible test case, as have recently emerged: Contract-Driven Development (CDD), ReCrash, JCrasher.

ORACLES

A test run is only useful if you can unambiguously determine whether it passed. The criterion is called a *test oracle*. If you have a few dozen or perhaps a few hundred tests, you might afford to examine the results individually, but this does not scale up. The task cries for automation.

> **Principle 4: Applying oracles**
> *Determining success or failure of tests must be an automatic process.*

This statement of the principle leaves open the *form* of oracles. Often, oracles are specified separately. In research such as ours, they are built in, as the target software already includes contracts that the tests use as oracles.

Random testing often outperforms supposedly smart ideas.

> **Principle 4 (variant): Contracts as oracles**
> *Oracles should be part of the program text, as contracts. Determining test success or failure should be an automatic process consisting of monitoring contract satisfaction during execution.*

This principle subsumes the previous one but is presented as a variant so that people who do not use contracts can retain the weaker form.

MANUAL AND AUTOMATIC TEST CASES

Many test cases are *manual*: Testers think up interesting execution scenarios and devise tests accordingly. To this category we may add cases derived—according to principle 3—from the failure of an execution not initially intended as a test run. It is becoming increasingly realistic to complement these two categories by *automatic* test cases, derived from the specification through an automatic test generator. A process restricted to manual tests underutilizes the power of modern computers.

The approaches are complementary.

> **Principle 5: Manual and automatic test cases**
> *An effective testing process must include both manually and automatically produced test cases.*

Manual tests are good at depth: They reflect developers' understanding of the problem domain and data structure. Automatic tests are good at breadth: They try many values, including extremes that humans might miss.

TESTING STRATEGIES

We now move from testing practice to research investigating new techniques. Testing research is vulnerable to a risky thought process: You hit upon an idea that seemingly promises improvements and follow your intuition. Testing is tricky; not all clever ideas prove helpful when submitted to objective evaluation.

A typical example is random testing. Intuition suggests that any strategy using knowledge about the program must beat random input. Yet objective measures, such as the number of faults found, show that random testing often outperforms supposedly smart ideas. Richard Hamlet's review of random testing (*Encyclopedia of Software Engineering*, J.J. Marciniak, ed., Wiley, 1994, pp. 970-978) provides a fascinating confrontation of folk knowledge and scientific analysis.

There is no substitute for empirical assessment.

> ***Principle 6: Empirical assessment of testing strategies***
> *Evaluate any testing strategy, however attractive in principle, through objective assessment using explicit criteria in a reproducible testing process.*

I was impressed as a child by reading in *The Life of the Bee* (Fasquelle, 1901) by Maurice Maeterlinck (famous as the librettist of Debussy's

Pelléas et Mélisande) what happens when you put a few bees and a few flies in a bottle and turn the bottom toward the light source. As Figure 1 shows, bees, attracted by the light, get stuck and die of hunger or exhaustion; flies don't have a clue and try all directions—getting out within a couple of minutes.

Maeterlinck was a poet, not a professional biologist, and I don't know if the experiment holds up. But it is a good metaphor for cases of apparent stupidity outsmarting apparent cleverness, as happens in testing.

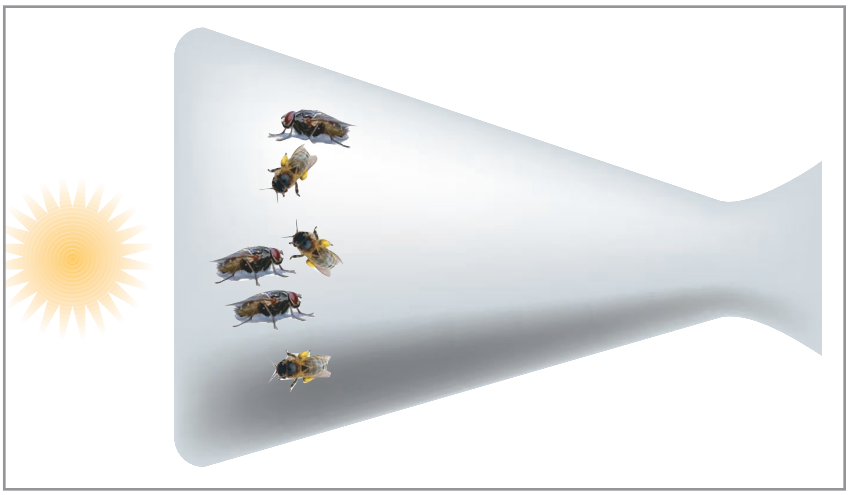


Figure 1. Smarter is not always better. Maeterlinck observed that if you put bees and flies into a bottle and turn the bottom toward the light source, the supposedly clever bees, attracted by the light, get stuck and die, while apparently stupid flies get out within a couple of minutes. Is this a metaphor for testing strategies?

ASSESSMENT CRITERIA

In applying the last principle, the issue remains of which criteria to use. The testing literature includes measures such as "number of tests to first failure." For the practitioner this is not the most useful: We want to find all faults, not just one. Granted, the idea is that the first fault will be corrected and the criterion applied again. But successive faults might be of a different nature; an automated process must trigger as many failures as possible, not stop at the first.

The *number* of tests is not that useful to managers, who need help deciding when to stop testing and ship, or to customers, who need an estimate of fault densities. More relevant is the *testing time* needed to uncover the faults. Otherwise we risk favoring strategies that uncover a failure quickly but only after a lengthy process of devising the test; what counts is total time. This is why, just as flies get out faster than bees, a seemingly dumb strategy such as random testing might be better overall.

Other measures commonly used include test coverage of various kinds (such as instruction, branch, or path coverage). Intuitively they seem to be useful, but there is little actual evidence that higher coverage has any bearing on quality. In fact, several recent studies suggest a negative correlation; if a module has higher test coverage, this is usually because the team knew it was problematic, and indeed it will often have more faults.

More than any of these metrics what matters is how fast a strategy can produce failures revealing faults.

> **Principle 7: Assessment criteria**
> *A testing strategy's most important property is the number of faults it uncovers as a function of time.*

The relevant function is fault count against time, $fc\ (t)$, useful in two ways: Researchers using a software base with known faults can assess a strategy by seeing how many of them it finds in a given time; project managers can feed $fc\ (t)$ into a reliability model to estimate how many faults remain, addressing the age-old question "when do I stop testing?"

We never strayed far from where we started. The first principle told us that testing is about producing failures; the last one is a quantitative restatement of that general observation, which also underlies all the others.

Bertrand Meyer *is professor of Software Engineering at ETH Zürich and chief architect at Eiffel Software in Santa Barbara, Calif. Contact him at bertrand.meyer@inf.ethz.ch.*

Editor: Mike Hinchey, Lero—The Irish Software Engineering Research Centre; mike.hinchey@lero.ie