Testing: Principles and Practice

STEPHEN R. SCHACH

Department of Computer Science, Vanderbilt University, Nashville, TN ⟨srs@vuse.vanderbilt.edu⟩

Testing is the process of determining whether a task has been correctly carried out. Testing should be performed throughout the software life cycle; it should not be restricted to verification (testing whether a phase has been carried out correctly) and validation (testing whether the completed product satisfies its specifications). The correction of a fault exposed by testing is termed *debugging*.

The goal of testing is to reveal faults. There are two types of testing: execution-based testing and nonexecution-based testing. It is impossible to execute a specification document; instead, it has to be reviewed carefully. Once executable code has been written it is possible to run test cases, that is, perform execution-based testing. Nevertheless, the existence of code does not preclude nonexecution-based testing because carefully reviewing code will uncover at least as many faults as running test cases. In this survey, the principles and practice of both execution-based and nonexecution-based testing are described.

NONEXECUTION-BASED TESTING

The principle underlying nonexecution-based testing techniques such as walkthroughs and inspections is that a review by a team of experts with a broad range of expertise increases the chance of finding a fault.

Nonexecution-based testing is remarkably effective. At the Jet Propulsion Laboratory (JPL), on average each two-hour inspection exposed four major and fourteen minor faults [Bush 1990], resulting in a saving of approximately $25,000 per inspection. Another JPL study [Kelly et al. 1992] shows that the number of faults detected decreases exponentially by phase. In other words, inspections result in faults being detected early in the software process, thereby saving both time and money.

The Cleanroom software development approach incorporates a number of different techniques, including an incremental life-cycle model, formal techniques for specification and design, and nonexecution-based module-testing techniques such as code reading and code inspections [Dyer 1992]. A critical aspect of the technique is that a module is not compiled until it has passed an inspection. As reported in Linger [1994], seventeen products totaling nearly one million lines of code were developed using Cleanroom. These included the 350,000-line Ericsson Telecom OS32 operating system. Overall, the weighted average testing fault rate was only 2.3 faults per KLOC.

An alternative nonexecution-based technique is *correctness proving*. This consists of using a mathematical proof to show that a product is correct, that is, satisfies its specifications. Dijkstra [1972] has stated that "the only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness." However, even if a product is proved correct, it must nevertheless be subjected to thorough execution-based testing [Schach 1996, Section 5.5.2].

Despite some problems with correctness proving, proofs are appropriate where indicated by risk or cost-benefit

analysis, and when human lives are at stake. Even when a full formal proof is not justified, the quality of software can be markedly improved through the use of informal proofs and by inserting assertions into the code.

EXECUTION-BASED TESTING

We need to test a number of different aspects of a software product, including its utility, reliability, robustness, performance, and correctness. There are two basic ways of systematically constructing test data to test the correctness of a module. The first is *testing to specifications* (or *black-box* testing). In this approach, the code is ignored; the only information used in drawing up test cases is the specification document. The other extreme is *testing to code* (or glass-box testing) where the test cases are based solely on the code. Neither approach is feasible because of the combinatorial explosion; there are simply too many test cases to consider.

The goal of execution-based testing is therefore to highlight as many faults as possible while accepting that there is no way to guarantee that all faults have been detected [Myers 1979; Beizer 1990]. A reasonable way to proceed is first to use black-box test cases (testing to specifications) and then to develop additional test cases using glass-box techniques (testing to code).

The art of black-box testing is to use the specifications to devise a small, manageable set of test cases to maximize the chances of detecting a previously undetected fault while minimizing the chances of wasting a test case by having the same fault detected by more than one test case. The favored technique for achieving this is equivalence testing combined with boundary-value analysis [Schach 1996, Section 12.15.1].

There are a number of different forms of glass-box testing, including statement, branch, and path coverage. The most powerful form of structural testing is *path coverage*, that is, testing all possible paths. However, in a product with loops the number of paths can be huge. In practice, therefore, techniques are used that reduce the number of paths to be examined while still being able to uncover more faults than would be possible using less comprehensive structural testing methods. One example is *all-definition-use-path coverage*. In this technique, each occurrence of a variable in the source code is labeled either as a *definition* of the variable or as a *use* of the variable. Next, all paths between the definition of a variable and the use of that definition are identified, nowadays by means of a CASE tool. Finally, a test case is set up for each such path. All-definition-use-path coverage is favored because large numbers of faults can frequently be detected using relatively few test cases [Schach 1996, Section 12.16.1].

FUTURE PROSPECTS

Software is tested in order to detect faults. However, instead of using better techniques for detecting faults, it is more effective to employ software-development approaches (such as Cleanroom) that reduce the number of faults in the software. Thus, the future role of testing will be to prevent faults rather than to detect them.

ACKNOWLEDGMENT

This survey was based on material taken from Stephen R. Schach, *Classical and Object-Oriented Software Engineering*, Third Edition, Richard D. Irwin Inc., 1996, pages 109–133 and 405–420.

REFERENCES

BEIZER, B. 1990. *Software Testing Techniques,* 2nd ed. Van Nostrand Reinhold, New York.

BUSH, M. 1990. Improving software quality: The use of formal inspections at the Jet Propulsion Laboratory. In *Proceedings of the 12th International Conference on Software Engineering.* (Nice, France), 196–199.

DIJKSTRA, E. W. 1972. The humble programmer. *Commun. ACM 15,* 10 (Oct.), 859–866.

DYER, M. 1992. *The Cleanroom Approach to Quality Software Development.* Wiley, New York.

KELLY, J. C., SHERIF, J. S., AND HOPS, J. 1992. An analysis of defect densities found during software inspections. *J. Syst. Softw. 17,* 1 (Jan.), 111–117.

LINGER, R. C. 1994. Cleanroom process model. *IEEE Software 11,* 3 (Mar.), 50–58.

MYERS, G. 1979. *The Art of Software Testing.* Wiley, New York.

SCHACH, S. R. 1996. *Classical and Object-Oriented Software Engineering,* 3rd ed. Richard D. Irwin, Inc., Chicago, IL.