The Problem with Testing

Norman Hines
JE Sverdrup Naval Systems Group

Testing is inefficient for the detection and removal of requirements and design defects. As a result, lessons learned in testing can only help prevent defects in the development of subsequent software and subsequent process improvement. Instead of testing out defects to achieve quality measures, quality should be designed into software. Thus test development should parallel the development of the software it tests.

Unlike other engineering disciplines, software development produces products of undetermined quality. Testing is then used to find defects to be corrected. Instead of testing to produce a quality product, software engineers should design in quality [1]. The purpose of testing should not be to identify defects inserted in earlier phases, but to demonstrate, validate, and certify the absence of defects.

Beginning with the Industrial Revolution, many technical fields evolved into engineering fields, but sometimes not until after considerable damage and loss of life. In each case, the less scientific, less systematic, and less mathematically rigorous approaches resulted in designs of inefficient safety, reliability, efficiency, or cost. Furthermore, while other engineering practices characteristically attempt to consciously prevent mistakes, software engineering seems only to correct defects after testing has uncovered them [2].

Many software professionals have espoused the opinion that there are "always defects in software [3]." Yet in the context of electrical, mechanical, or civil engineering the world has come to expect defect-free circuit boards, appliances, vehicles, machines, buildings, bridges, etc.

Follow the Basics

All models of the software development life cycle center upon four phases: requirements analysis, design, implementation, and testing. The waterfall model requires each phase to act on the entire project. Other models use the same phases, but for intermediate releases or individual software components.

Software components should not be designed until their requirements have been identified and analyzed. Software components should not be implemented until they have been designed. Even if a software component contains experimental features for a prototype, or contains only some of the final system's features as an increment, that prototype or incremental software component should be designed before it is implemented.

Software components cannot be tested until after they have been implemented. Defects in software cannot be removed until they have been identified. Defects are often injected during requirements analysis or design, but testing cannot detect them until after implementation. Testing is therefore inefficient for the detection of requirements and design defects, and thus inefficient for their removal.

Testing in the Life Cycle

Burnstein, et al. have developed a Testing Maturity Model (TMM) [4] similar to the Capability Maturity Model® [5]. The TMM states that to view testing as the fourth phase of software development is at best Level 2. However, it is physically impossible to test a software component until it has been implemented.

The solution to this difference of viewpoint can be found in TMM Level 3, which states that one should analyze test requirements at the same time as analyzing software requirements, design tests at the same time as designing software, and write tests at the same time as implementing code. Thus, test development parallels software development. Nevertheless, the tests themselves can only identify defects after the fact.

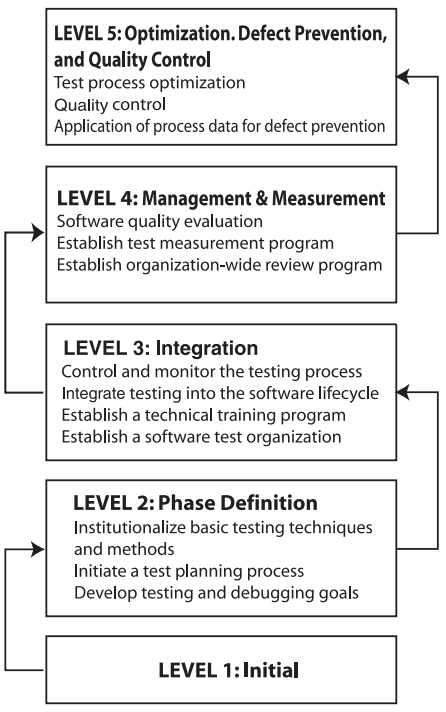
Furthermore, testing can only prove the existence of defects, not their absence. If testing finds few or no defects, it is either because there are no defects, or because the testing is not adequate. If testing finds too many defects, it may be the product's fault, or the testing procedures themselves.

Branch coverage testing cannot exercise all paths under all states with all possible data. Regression testing can only exercise portions of the software, essentially sampling usage in the search for defects.

The clean-room methodology uses statistical quality certification and testing based on anticipated usage. Usage probability distributions are developed to determine the systems most likely used most often [6]. However, clean-room testing is predicated upon mathematical proof of each software product; testing is supposed to confirm quality, not locate defects. This scenario-based method of simulation and statistically driven testing has been reported as 30 times better than classical coverage testing [7].

Page-Jones dismisses mathematical proofs of correctness because they must be based on assumptions [8], yet both testing and correctness verification are done against the software's requirements. Both are therefore based on the same assumptions; incorrect assumptions result in incorrect conclusions. This indictment of proofs of correctness must also condemn testing for the same reason.

Table 1: *TMM Levels*



The clean-room methodology's rigorous correctness verification approaches zero defects prior to any execution [9], and therefore prior to any testing. Correctness verification by mathematical proof seems better than testing to answer the question, "Does the software product meet requirements?"

Properly done test requirements analysis, design, and implementation that parallels the same phases of software development may help in early defect detection. However, done improperly (as when developers test their own software), this practice may result in tests that only test the parts that work, and in software that passes its tests but nevertheless contains defects. Increasingly frustrated users insist that there are serious defects in the software, while increasingly adversarial developers insist that the tests reveal no defects.

Test requirements analysis done separately from software requirements analysis can make successful testing impossible. A multi-million dollar project was only given high-level requirements, from which the software developers derived their own set of (often-undocumented) lower-level requirements, to which they designed and implemented the software. After the software had been implemented, a test manager derived his own set of lower-level requirements, one of which had not even been considered by the developers. The design and test requirements were mutually exclusive in this area, so it was impossible for the software to pass testing. This failure scrapped the entire project and destroyed several careers [10].

Defect Removal and Prevention

Test-result-driven defect removal is detective work; the maintenance programmer must identify and track down the cause within the software. Defect removal is also similar to archeology, since all previous versions of the software, and documentation of all previous phases of the development may have to be researched, if available. Using testing to validate that software is not defective [9], rather than to identify and remove defects, moves their removal from detection to comparative analysis [7].

TMM Level 3 integrates testing into the software lifecycle. This includes testing each procedure or module as soon as possible after each is written. Integration testing is also done as soon as possible after the components are integrated. Nevertheless, the concept of defect prevention is not addressed until TMM Level 4, and then only as a philosophy for the entire testing process.

Testing cannot prevent the occurrence of defects; it can only aid in the prevention of their recurrence in future components. This is why neither CMM nor TMM discusses actual defect prevention, or more accurately, *subsequent* defect prevention until Level 5. Waiting until one has reached Level 5 before trying to prevent defects can be very costly, both in terms of correcting defects not prevented and in lost business and goodwill from providing defective software.

Waiting until implementation to test a component for defects caused in much earlier phases seems too much of a delay; yet, an emphasis in testing for defect prevention is exactly that. An ounce of prevention may be worth a pound of cure, but one cannot use a cure as if it were a preventative.

There are several methods currently available to accomplish defect prevention at earlier levels of maturity such as Cleanroom Software Engineering [9], Zero Defect Software [11], and other provably correct software methods [2, 12].

Software Quality and Process Improvement

Gene Krinz, Mission Operations' director for the NASA space shuttle, is quoted as saying about the quality of the flight software, "You can't test quality into the software [11]." Clean-room methods teach that one can neither *test in* nor *debug in* quality [9].

If quality was not present in the requirements analysis, design, or implementation, testing cannot put it there. One of TMM's Level 3 maturity goals is software quality evaluation. While many quality attributes may be measured by testing, and many quality goals may be linked to testing's ultimate objectives, most aspects of software quality come from the quality of its design.

Procedure coupling and cohesion [13], measures of object-oriented design quality such as encapsulation, connascence, encumbrance, class cohesion, type conformance, closed behavior [8], and other quantitative measures of software quality, are established in the design phase. They should be measured soon after each component is designed; do not wait until after implementation to measure them with testing.

Some authors have suggested that analyzing, designing, and implementing tests in parallel with the products to be tested will somehow improve the processes used to develop those products. [3] However, since the software product testers should be different from those who developed it, there needs to be some way for the testers to communicate their process improvement lessons learned to the developers. Testers and developers should communicate effectively; every developer should also act as a tester (but only for components developed by others).

Designing in Quality

One of the maturity subgoals of subsequent defect prevention is establishing a causal analysis mechanism to identify the root causes of defects. Already there is evidence that most defects are created in the requirements analysis and design phases [11]. Some have put the percentage of defects caused in these two phases at 70 percent [3].

Clear communication and careful documentation are required to prevent injecting defects during the requirements analysis phase. Requirements are characteristically inconsistent, incomplete, ambiguous, nonspecific, duplicate, and inconstant. Interface descriptions, prototypes, use cases, models, contracts, pre- and post-conditions, etc. are all useful tools.

To prevent injecting defects during the design phase, software components must never be designed until a large part of their requirements have been identified and analyzed. The design should be thorough, using such things as entities and relationships, data and control flow, state transitions, algorithms, etc. Peer reviews, correctness proofs and verifications, etc. are good ways to demonstrate that a design satisfies its requirements.

Preventing the injection of defects during the implementation phase requires that software components never be implemented until they have been designed. It is far too easy to implement a software component while the design is still evolving, sometimes just in the developer's mind. Poor documentation and a lack of structure in the code usually accompany an increased number of defects per 100 lines of code [3]. As I mentioned earlier, this applies even to prototype and incremental software components; those experimental or partial features should be designed before implementation.

The clean-room method has an excellent track record of near defect-free software development, as documented by the Software Technology Support Center, Hill AFB, Utah, regardless of Daich's statements to the contrary [3]. Clean-room is compatible with CMM Levels 2 through 5 [9], and can be implemented in phases at all these levels [6].

Conclusion

It is my dream that software engineering will become as much of an engineering discipline as the others; users will have just as much confidence that their software is as defect free as their cars, highway bridges, and aircraft.

Testing should be used to demonstrate the absence of defects, not to identify defects inserted in earlier phases. It should be used to certify that the software components implement their designs, and that these designs satisfy their requirements.

Analyzing testing requirements should be done in parallel with analyzing the software components' requirements. Tests should be designed in parallel with designing the components. Test implementation should occur in parallel with implementing the components, and developing integration tests should be done in parallel with integration.

The source of software defects is a lack of discipline in proper requirements analysis, design, and implementation processes. Testing must physically occur after implementation, so reliance on it to detect defects delays their correction. Until software defects are attacked at their source, software will continue to be developed as if it were an art form rather than a craft, engineering discipline, or a science.◆

References

1. Humphrey, W. S., Making Software Manageable, *CROSSTALK,* December 1996, pp. 3-6.
2. Baber, R. L., *The Spine of Software: Designing Provably Correct Software: Theory and Practice,* John Wiley & Sons Ltd., Chichester, United Kingdom, 1987.
3. Daich, G. T., Emphasizing Software Test Process Improvement, *CROSSTALK,* June 1996, pp. 20-26, and Daich, Gregory T., Letters to the Editor, *CROSSTALK,* September 1996, pp. 2-3, 30.
4. Burnstein, I.; Suwannasart, T.; and Carlson, C.R., Developing a Testing Maturity Model: Part I, *CROSSTALK,* August 1996, pp. 21-24; Part II, *CROSSTALK*, September 1996, pp. 19-26.
5. Paulk, M. C.; Curtis, B.; Chrissis, M. B.; and Weber, C. V., *Capability Maturity Model℠ for Software, Version 1.1,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1993.
6. Hausler, P. A.; Linger, R. C.; and Trammel, Adopting Cleanroom Software Engineering with a Phased Approach, *IBM Systems Journal,* volume 33, number 1, 1994, p. 95.
7. Bernstein, L.; Burke Jr., E. H.; and Bauer, W. F., Simulation- and Modeling-Driven Software Development, *CROSSTALK,* July 1996, pp. 25-27.
8. Page-Jones, M., *What Every Programmer Should Know About Object-Oriented Design,* Dorset House Publishing, New York, New York, 1995.
9. Linger, R.C., Cleanroom Software Engineering: Management Overview, *Cleanroom Pamphlet,* Software Technology Support Center, Hill Air Force Base, Utah, April 1995.
10. Unpublished CMM Tutorial, information withheld to protect the people involved.
11. Schulmeyer, G. G., *Zero Defect Software,* McGraw-Hill, Inc., New York, New York, 1990.
12. Martin, J., *System Design from Provably Correct Constructs,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.
13. Page-Jones, M., *The Practical Guide to Structured Systems Design,* second edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

About the Author

Norman Hines is a software developer for JE Sverdrup, working at the Naval Air Warfare Center, China Lake, Calif. He is currently working on projects that integrate weapon simulation systems with actual range data in real time and post mission. He has more than 20 years experience in software development, as well as a bachelor's degree in mathematics and business administration from University of Wisconsin–Platteville, a master's degree in business administration from University of Michigan, and a master's degree in computer science from California State University, Chico.

JE Sverdrup Naval Systems Group
900 N. Heritage Dr.
Ridgecrest, CA 93555
Phone: (760) 939-9460
E-mail: nwsverdrup@navair.navy.mil

"Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

— Donald Knuth