SPECIAL ARTICLE

THE GROWTH OF SOFTWARE TESTING

We can trace the evolution of software test engineering by examining changes in the testing process model and the level of professionalism over the years. The current definition of a good software testing practice involves some preventive methodology.

DAVID GELPERIN and BILL HETZEL

Over the past four decades, as the use of digital computers diversified and increased with a corresponding increase in the cost of software failure, the importance of testing grew. During this period, our understanding of how to make testing a cost-effective process grew as well. This growth in understanding was marked by a progression of testing process models. As growth in understanding is never uniform over a large population, due to different needs and differential rates of communication and assimilation, examples of all models in the progression can be found in practice today.

With simultaneous multiple models has come confusion. Since testing is as old as coding, most people involved with software have a mental model of testing. However, there are significant differences in the definition of testing's scope and objectives that have resulted in different definitions of testing success. Unrecognized differences in mental models have resulted in confused communication between and among customers, managers, analysts, programmers and testers.

This article characterizes the major process models and describes some of the changes associated with testing's growth. A recent model based on professional standards is compared with the best industry practice to highlight differences.

OUTLINE OF THE MAJOR MODELS

We distinguish the four major testing models listed in Figure 1. These models are primarily differentiated by scope of activity and primary goal.

Scope of Activity

Both phase models focus on execution as the primary testing activity because execution achieves their primary objectives. While both models acknowledge test planning and data selection as prerequisites to effective execution, they strongly associate the word "testing" with "test execution" to the point where the terms are virtually synonymous. This strong association has resulted in the view that testing is a life cycle phase. This view of testing expressed in the phrase "code and test" took root early and still remains prevalent.

The life cycle models move the focus of testing from the end of a project to its beginning. This shift results in the view that testing is not a sequential phase affecting just the software implementation, but is on a parallel track affecting software requirements and designs as well. Testing is seen as strongly interacting with development activities from the very beginning of a project.

We distinguish testing (planning, design, implementation, execution) from other evaluation activities such as software analysis (e.g., timing analysis, data flow analysis, proof of correctness) and software reviews (including walkthroughs and inspections). The two life cycle models differ in the way they relate these other activities to testing. The evaluation model either includes all analysis and review activities as part of a larger process called life cycle testing [28] or excludes but combines them with a phase testing model into a process that has many names: verification, validation, and test [17]; verification and validation [13, 33]; life cycle validation [18]; life cycle verification; test and evaluation; product evaluation.

MODEL	PRIMARY GOAL
Phase Models	
Demonstration	To make sure that the software satisfies its specification
Destruction	To detect implementation faults
Life Cycle Models	
Evaluation	To detect requirements, design, and implementation faults
Prevention	To prevent requirements, design, and implementation faults

FIGURE 1. Major Testing Models

The prevention model includes certain review and analysis activities in testing and excludes others. The include/exclude decision is based on the purpose of the activity. Only review or analysis activities done to directly support test planning or test design or to evaluate a test product are included. These include: reviewing software requirements to determine if satisfactory test criteria can be defined, analyzing the software design to identify test conditions that will challenge the design, and reviewing products such as test plans and test data to determine their adequacy. A review of software requirements to support software design would not be considered a part of testing. The scope of prevention testing includes the use (review and analysis) and improvement of the software specifications (requirements and design) during test analysis and design as well as the use and improvement of code during test execution.

Primary Goals

One model says we test to demonstrate that some (possibly future) version of the software satisfies its specification, two models say we test to detect faults, and the fourth says we test to prevent faults. These three goals need not conflict and, in fact, are all present in the prevention model.

A conflict may arise between the goals of demonstration and fault detection if the strategy for achieving the demonstration is not carefully selected. Historically, with demonstration as the primary goal, testers used a strategy analogous to a constructive mathematical proof. They understood their job as that of building a test set T such that when the software passes all tests in T it could be claimed that the software satisfies its specification. Since only the set of all possible tests can prove that software satisfies its specification and the goal of demonstration gives little guidance in selecting a smaller set, the danger in this constructive approach is that the criteria for test selection will be arbitrarily chosen. Sometimes, the chosen criteria seemed to be that the test set should contain only tests that the software could pass. This clearly conflicts with the goal of detecting faults.

Using a demonstration strategy analogous to proof by contradiction significantly reduces the danger of conflict. Testers negate the hypothesis that the software satisfies its specifications and assume the software contains at least one fault. They must then build a test set T such that if the software contains a fault, then at least one test in T will fail. This strategy defines the ability to detect faults as a *test set adequacy criterion*. The satisfaction of this criterion can be evaluated before the software is used. The degree of satisfaction can be estimated by:

1. determining if the test set satisfies necessary conditions (e.g., testing every requirement at least once) associated with particular fault types;
2. determining if the test set satisfies sufficient conditions associated with particular fault types [21, 36]; and
3. inserting different types of faults in the software and determining whether the test set can detect them [10].

Even using a proof by contradiction demonstration strategy, a residual danger of conflict lies in the definition of stopping criteria. A fault detection emphasis could press for more tests to cover more fault classes while a demonstration emphasis could be content with managing the higher impact and higher probability faults. The danger is easily managed by clearly defining testing's contribution to product success. Testing, like everything else, can be either underdone or overdone.

Testing, like everything else, can be either underdone or overdone.

The prevention model is distinguished from the evaluation model more by mechanism than by goal. Both models focus on software requirements and design in order to avoid implementation problems, but only the prevention model sees test planning, test analysis, and test design activities playing a *major* role in this strategy. The evaluation model primarily relies on software analysis and review techniques which are seen as separate from testing.

EVOLUTION OF THE MODELS

We divide the past four decades into periods (as shown in Figure 2) based on the most influential testing model. We start each period, except the first, based on an associated publishing event, however each model existed and gathered influence before its milestone publication.

The publication milestone was not chosen to be the first appearance of the model, but rather a publication about the model that had the largest audience of practitioners. In each period, all previous models retain adherents.

The Debugging-Oriented Period

In the earliest days of digital computers, testing focused on hardware. Software problems, although undoubtedly present, were submerged in the concern for hardware reliability. The earliest articles on digital computer testing discuss hardware components.

The early view of programming was that you "wrote" a program and then you "checked it out." For many, the concepts of program checkout, debugging, and testing were not clearly differentiated. For some, "debugging" meant any activity involved with getting the bugs out and testing was viewed as one of these activities. For others, "debugging" and "testing" were used interchangeably. For still others, the two terms referenced distinct activities but the distinction was considered difficult to describe.

-1956	The Debugging-Oriented Period
1957–1978	The Demonstration-Oriented Period
1979–1982	The Destruction-Oriented Period
1983–1987	The Evaluation-Oriented Period
1988-	The Prevention-Oriented Period

FIGURE 2. Testing's Stages of Growth

The earliest article on program checkout was written by Alan Turing in 1949 [34]. The article discusses the use of assertions for what we would today call "proof of correctness." Another article by Turing appearing in 1950 [35] could be considered the first on program testing. It addressed the question "How would we know that a program exhibits intelligence?" If the requirement is to build such a program, then this question is a special case of "How would we know that a program satisfies its requirements?" Turing defined an operational test for intelligent behavior by a computer program. Turing's test required the behavior of the program and a reference system (a human) to be indistinguishable to an interrogator (tester).

The Demonstration-Oriented Period

In 1957, Charles Baker distinguished debugging from testing in a review [5] of Dan McCracken's book *Digital Computer Programming.* Program checkout was seen to involve two goals: "Make sure the program runs" and "Make sure the program solves the problem." He viewed the former as the focus of debugging and the latter as the focus of testing. The optimism expressed in the phrase "make sure" was often translated into the testing goal of showing that the software satisfies its requirements.

As computer applications increased in number, cost, and complexity during this period, testing came to assume more significance because of greater economic risk and greater development and maintenance experience. It was evident that computer systems contained many deficiencies, and the cost of recovering and fixing these problems was substantial. Users and managers began placing greater emphasis on "better testing" and defining "better" as "more effective at detecting problems before product delivery."

As computer applications increased in number, cost, and complexity, testing came to assume more significance because of greater economic risk.

During the demonstration-oriented period, the meaning of both "debugging" and "testing" included efforts to detect, locate, identify, and correct faults (see Figure 3). The distinction between the two activities rested on the definition of success (i.e., running or solving the problem). The emerging destruction model regrouped these tasks so that all fault detection is included in "testing" and all fault location, identification, and correction is included in "debugging." Although "debug" and "test" realigned their meanings, there was still a need to "make sure the program runs." Today, in many organizations the process that used to be called "debugging" is called "sanity testing."

The Destruction-Oriented Period

In 1979, [25] described the destruction testing model. Myers defined testing as "the process of executing a program with the intent of finding errors." This definition made implementation fault detection the primary goal. Myers's concern was that in pursuing the demonstration goal of showing that a program has no faults, one might subconsciously select test data that has a low probability of causing program failures. If the goal is to

Destruction model	Demonstration model		
	Debug	Test	
Test	Detect	Detect	Detect faults
Debug	Locate Identify Correct	Locate Identify Correct	Fix faults
	Make sure it runs	Make sure it solves the problem	

FIGURE 3. Different Definitions of "Test" and "Debug"

demonstrate that a program has faults, our test data will have a higher probability of detecting them and we become more successful in testing.

The shift in emphasis from demonstration to detection naturally led to the association of testing with other fault detection activities. Myers's book and other publications from the period [13, 27] discussed software analysis and review techniques along with testing. Articles by Deutsch [12] and Howden [22] contain early descriptions of combined fault detection approaches.

The Evaluation-Oriented Period

The Institute for Computer Sciences and Technology of the National Bureau of Standards published [17] in 1983. This guideline, specifically targeted at federal information processing systems (FIPS), describes a methodology that integrates analysis, review, and test activities to provide product evaluation during the software life-cycle. Each life-cycle phase has an associated set of activities and products. The guideline recommends three sets of evaluation techniques; the basic, the comprehensive and the critical. The sets are cumulative in that each is contained in its successor.

The collective philosophy of the FIPS methodology can be found in the following quote from the Overview section:

> "No single VV&T technique can guarantee correct, error-free software. However, a carefully chosen set of techniques for a specific project can help to ensure the development and maintenance of quality software for that project."

Additional guidance on the planning of integrated evaluation can be found in [4].

GROWTH OF PROFESSIONALISM

Beginning in the early 1970s, the level of professionalism associated with software testing increased significantly. Establishment of test positions, inauguration of national meetings, increase in test publications, and development of national standards are all indicators of this growing professionalism.

Software Test Engineering

Recognition of software test engineering as a specialty area has increased during the last 20 years. In many organizations, the scope and objectives of a variety of test positions have been defined. Titles such as "test manager," "lead tester," "test analyst," and "test technician" have become common. In other organizations, people with various quality assurance titles do significant amounts of testing.

Job listings explicitly requesting test skills are appearing with increasing frequency. Testing is even a career path in a few companies. In most, however, testers are still struggling to reach parity with developers. We note that the position of tester has not yet appeared on any major salary survey.

National Meetings

The first formal conference on software testing was organized by Bill Hetzel and held at the University of North Carolina in June 1972 [20]. Following this meeting, there was a series of academic testing workshops. The first was held in Fort Lauderdale, Fla. in 1978 and chaired by Ed Miller. The second was held in Catalina, Calif. in 1982 and was chaired by Bill Howden and Susan Gerhart. The third was held in Banff, Canada in 1986 and was chaired by Lori Clarke [32]. The next meeting will also be held in Banff this summer and will be chaired by Lee White and Leon Osterweil.

In addition to these workshops, other testing meetings [9, 31] were held. An annual industrial testing conference was inaugurated in 1984 by the Data Processing Managers Association. This continuing support for testing workshops and conferences indicates a recognition of testing's importance and the resulting need to share research results and industrial experiences.

Test Publications

The first software testing book [20] was published as an outgrowth of the 1972 conference. Starting with the publication of Myers's book [25] in 1979, testing books [6, 7, 9, 11, 13, 14, 15, 18, 19, 23, 27, 28, 29, 33] appeared at an average rate of almost two a year. Steady growth in the inventory of good books is another indicator of the importance placed on testing.

Since Turing's articles in 1950 [34, 35], hundreds of testing articles and reports have appeared. Extensive bibliographies of the testing literature from the demonstration-oriented period to the mid-1980s can be found in [7, 11, 19, 27]; a bibliography of literature from the debugging-oriented period can be found in [20]. However, there are still no professional journals devoted to software testing or the broader topic of software evaluation.

National Standards

Another sign of professional maturity was the development of two consensus standards on good test engineering practice. The first dealt with documentation and the second with unit testing. A task group of the IEEE Technical Committee on Software Engineering began work in 1979 on a standard for software test documentation. The resulting ANSI/IEEE standard [2] was published in 1983 and specifies the content and format of eight documents.

The task group did not try to standardize the best current practices, but tried instead to reach consensus on how test documentation ought to be done. Documentation was seen as a system of data structures that should satisfy the information and access requirements of its users. During the design of the test documentation system, issues of modularity, cohesion, coupling, reviewability and usability were carefully considered.

The major differences between the test documents defined in the ANSI/IEEE standard and typical current practice lie in the definition of the test plan and the test design specification. A test plan that conforms to the standard limits its scope to issues such as risk identification, overall strategy, task structure, resources, responsibilities, schedule, and contingencies. The identification and description of specific test cases and procedures is allocated to separate test specifications. Many current test plans include both planning and design issues. This coupling tends to delay consideration of the planning issues thereby limiting strategy choices.

In addition to separate case and procedure specifications, the standard identifies a test design specification. This document is the testing analog of a software architecture specification. Its purpose is to focus attention on the overall organization of the test set and the linkage of test set elements to software requirements and design components. It is meant to be the major window onto the vital issue of test set adequacy. In typical test documentation, this type of information is usually missing.

Following the documentation standard, a second task group began to develop a standard on software unit testing in 1984. The resulting ANSI/IEEE standard [3] was published in 1987. It specifies the phases, activities, tasks, and documents that comprise a comprehensive unit testing effort. Continuing a major theme of the documentation work and recognizing a strong parallel between test development and software development, the unit testing standard emphasizes the need to *design* the test set.

The unit testing standard requires two documents: a test design specification and a test summary report. In addition to emphasizing test design, this visibility requirement represents the biggest difference between the standard and most current practice. Typical unit testing is invisible and therefore undermanaged.

A PREVENTION TESTING METHODOLOGY

Starting in 1985, the authors generalized the IEEE unit testing process [3] to all levels of testing and integrated the results into a comprehensive methodology called the "Systematic Test and Evaluation Process" or STEP [19, chapter 3]. This methodology defines a system of testing tasks, products, and roles for the consistent and cost-effective achievement of test objectives.

STEP is based on a life cycle prevention model that sees testing parallel to development with an activity sequence containing planning, analysis (setting test requirements or objectives), design (specifying an architecture for the set of tests and details of individual cases and procedures), implementation (acquiring or developing test data, procedures, and test support software), execution (running and rerunning tests and determining the results), and maintenance (saving and updating the tests as the software changes).

Foundations

During the course of using STEP, it was observed that timely test planning, test analysis and test design have powerful side effects. They significantly improve the quality of software specifications in two ways. First, attempts to use the software requirement and design specifications during planning, analysis, and design invariably generate questions that reveal incompleteness, ambiguity, inconsistency, and incorrectness at a time when these problems are relatively cheap to correct. Thus software specifications, as well as code, are improved by their use in testing. This effect was noted over 10 years ago and described as follows:

"The test planning must begin early in the development cycle . . . Testers look at the specifications from an entirely different point of view than do the developers. As such, they can frequently spot flaws in the logic of the design, or the incompleteness of the specification. In doing this, they help the developers to produce a superior product, and accelerate the actual program development." [1, page 6]

The second way that timely testing improves software specifications is by building models (i.e., comprehensive sets of behavior samples) that show the consequences of the software specifications. Test case descriptions, with their definitions of input situations and required responses, serve as what-you-see-is-what-you-will-get examples of software behavior. The collection of behavior samples contained in a test specification clearly reveal to customers, analysts, designers and programmers the consequences of software requirements and design choices. The timely development of these models often leads to early recognition of the need for requirements and design changes. The use of a test case reference set to provide a working definition of software functionality accessible to both users and developers is proposed in [30].

Programmers want to do a good job. They understand that their goal is to produce effective software, not software that merely passes tests.

The assumption that test cases are known to programmers should be noted here. Some people express concern that if programmers know the tests, then they will develop incorrect code that passes. It is argued that programmers will do a better job when unbiased by test information. An analogy is drawn to students studying for a test. Since the test can not cover everything, revealing the questions could focus studying and those areas not covered might be ignored.

When students ignore areas, it results from confusing the goal of passing tests with the goal of learning the subject. We find no evidence of such confusion on the part of software professionals. Programmers want to do a good job. They understand that their goal is to produce effective software, not software that merely passes tests.

If a tester identifies an input situation that is not considered by a programmer, then a "secretive" testing strategy may reveal the problem, but only at execution time. This will significantly increase the cost of correction and will certainly create an "I gotcha" relationship between testers and programmers to the detriment of both. Conversely, a test-sharing strategy creates healthy relationships and challenges testers to develop a comprehensive test set early and to help programmers do a good job from the start.

This strategy also implies that testers should share responsibility for costly execution time failures. For those test groups that use this strategy and have the authority to delay product release, the exercise of that authority would be an admission of the failure of testing to prevent major problems.

The side effects of timely test planning and design (i.e., improved software specifications) are now understood to be a major contribution of good testing. Asking test-related questions and building test-related models early is often more important to software quality and cost-effective development than actually executing the tests. The prevention model uses these two effects as the primary focus of the testing process. From a prevention perspective, detection of a costly software requirements or design defect during test execution implies not only the failure of a test, but the failure of the testing process.

Timely test design also helps manage developer bias when developers design the tests. Since detailed knowledge of the software design and implementation can make it difficult to imagine some situations that need testing, one strategy puts test set design before software design. The strategy is composed of the following four elements:

1. Provide developers with clear definitions, guidelines, measures and examples of quality test plans and test designs;
2. Require test design based on requirements information before software design and test design based on software design before software implementation;
3. Require good documentation of test planning, design, and execution;
4. Have a second party approve the results.

Without the visibility provided by good documentation and objective measures of adequacy, second-party approval is not feasible. This strategy provides a cost-effective alternative to older strategies that recommend authors not test their own products.

The prevention model views testing as a form of risk management. It not only reduces project risk as outlined earlier, but must work to minimize the expected cost of software failure. Therefore, assessment of failure-impact risk (i.e., how much damage a software failure can do and the likelihood of the failure) is a prerequisite to test strategy development and test set design. Evaluation of the appropriateness of a testing strategy or test design requires consensus on the risk issues and their priorities.

Test planning should be done in two phases; master planning and detailed planning. A Master Test Plan should be developed at the start of a project to record

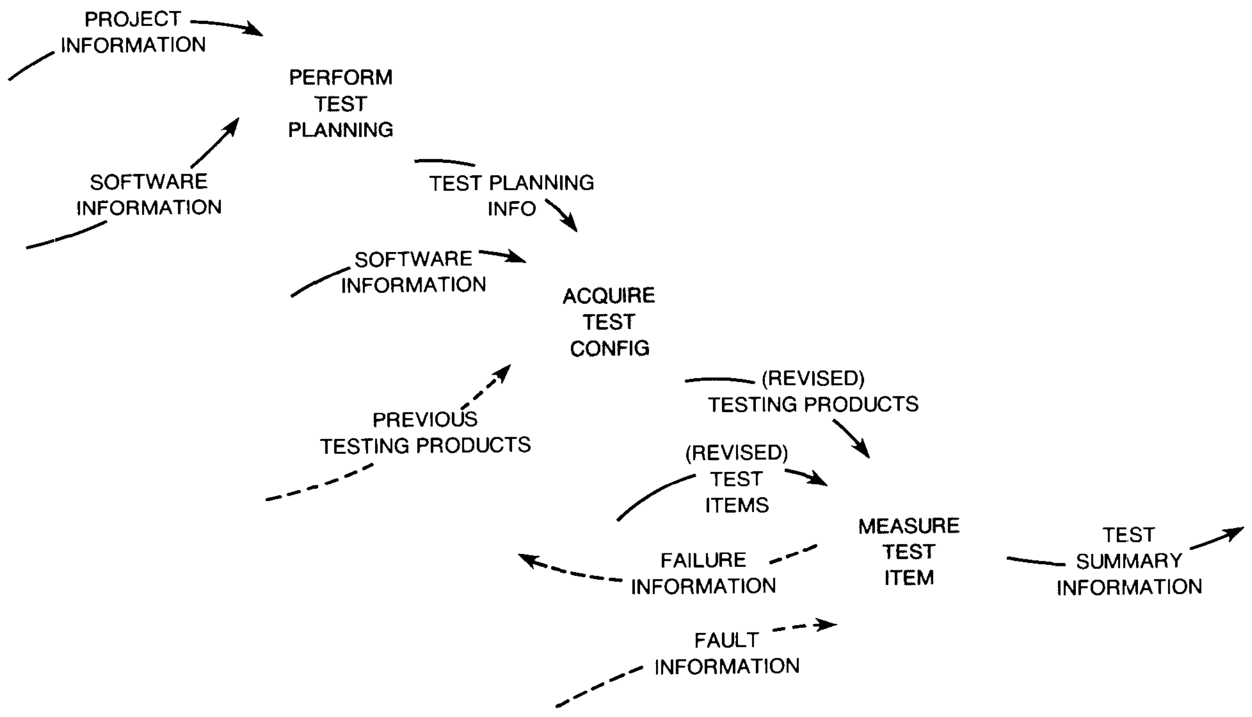

FIGURE 4. Basic Test Process Data Flow

the risk assessment and outline the testing strategy. It must include a description of the scope and objectives for each level of project testing (e.g., program, build, system, integration, acceptance). Detailed planning is a separate activity required for each level and triggered by the availability of approved requirements.

Outline Of A Methodology

We outline the structure of STEP in order to discuss similarities and differences with existing test methodologies [8, 17, 24, 26]. Figure 4 shows the data flow at a test level (system or program). Testing is decomposed into three phases—Planning, Acquisition, and Measurement.

During the Planning phase, information about the software to be tested and the project are used to develop test objectives and an overall testing approach. One output of this phase is a Master Test Plan that serves to guide the remainder of the testing activity and coordinate the test levels.

PLANNING
1. DEVELOP a master test plan
2. DETERMINE features to be tested
3. DEVELOP detailed test plans

ACQUISITION
4. DESIGN the tests
5. IMPLEMENT the tests

MEASUREMENT
6. EXECUTE the tests
7. CHECK termination
8. EVALUATE results

FIGURE 5. Testing Activities in STEP

During the Acquisition phase, more information about the software (requirements and design) along with any previous testing documentation and data are used to specify and implement a test configuration. The output of this phase is the test set and its documentation.

The input to the Measurement phase is the software to be tested and the test configuration. The output consists of test reports documenting the execution and evaluation activities along with records of any failures or other incidents observed.

The testing phases are composed of activities as shown in Figure 5.

Each of the eight activities is further decomposed into tasks. Figure 6 shows the tasks for developing a Master Test Plan.

In addition to a task architecture, STEP specifies the test products produced by its activities. There are two types: documentation products and implementation products (see Figure 7). STEP provides outlines and templates for the test documents. The outlines for the test plans, specifications, and reports are consistent with those in the ANSI/IEEE Test Documentation Standard [2].

At every level, the methodology calls for designing

P.1 DEVELOP a Master Test Plan
- P.1.1 Identify and prioritize software risk issues
- P.1.2 Identify organizational training needs
- P.1.3 Determine available resources
- P.1.4 Develop a comprehensive testing strategy
- P.1.5 Determine resource and staffing requirements
- P.1.6 Identify responsibilities
- P.1.7 Specify an overall schedule
- P.1.8 Review and revise the draft master plan

FIGURE 6. Tasks in STEP Activity 1

tests in three increments. The core increment is developed based on software requirements information, the second increment is based on software design information, and the final increment is based on software implementation information.

Comparison With Existing Methodologies

Based on scope, there are two kinds of published test methodologies. One kind only applies to program testing [8, 24], while the other encompasses all testing activities [17, 26]. Each of the life-cycle methodologies defines an activity structure and provides product descriptions. One [17] provides for incremental test design, while another [26] discusses risk assessment but not as a prerequisite for test planning and design. Only STEP addresses the use of test specifications as a requirements model.

PRESENT PRACTICE—MODEL VS. REALITY

Systematic, prevention test methodologies such as STEP are in use and proving cost-effective today. However, test methodology usage in most of industry is limited. Figure 8 identifies major differences between preventive testing and industry practice based on the knowledge and experience of the authors. Another source of information on industry practices was the results of a testing practices and trends survey completed at the 4th International Conference on Software Testing held in Washington, DC on June 14–16, 1987. Some results of the survey are shown in Figure 9.

Conference attendees were given a list of 20 test practices and asked to indicate whether each was common practice in their organizations (a *yes* response), a practice found on some projects or in some organi-

Documentation Products
1. Test Plans
2. Support Software Requirements
3. Coverage Inventories
4. Test Design Specifications
5. Test Reports

Implementation Products
6. Test Data Sets
7. Automated Test Procedures
8. Support Software & Documentation

FIGURE 7. Major Test Products in STEP

	Focus	Timing: Planning	Timing: Design	Coverage	Visibility
Prevention Testing	Prevention	Before & after software requirements	After software requirements	Largely known	Publicly-documented and reviewed
Industry Practice	Detection & Demonstration	After software design	After software design	Largely unknown	Publicly- or privately-documented or undocumented with little or no review

FIGURE 8. Major Differences between Prevention Testing and Industry Practice

zational units but not others (a *sometimes* response), or not normally used (a *no* response). A response of *don't know* was also available.

Figure 9 indicates that the most commonly reported practice was recording of defects found during testing (73 percent of the respondents considered this a normal practice in their organization). The least common practice was measuring code coverage (e.g., number of executable source lines not executed) achieved during testing (only five percent viewed this as normal practice).

Two additional observations seem especially important. First, only eight percent of the respondents regularly develop tests before coding. The concept of test design before code is a key element in cost-effective testing but it is clear that industry has a long way to go before this practice becomes the norm. Second, we note an inconsistency. A high percentage of the respondents felt that the testing in their organization was a systematic and organized activity (91 percent answered either *yes* or *sometimes* to this practice). However, only half regularly document their test designs, only half regularly save their tests for reuse after software changes, and an extremely small five percent provide regular measurements of code coverage. Such practices are more indicative of ad hoc approaches. Clearly, "systematic and organized" is in the eye of the beholder.

It should be understood that the results of this survey are strongly biased. People who attend technical conferences represent the leading edge of test awareness and sophistication. Therefore, the survey does not measure general industry practice but the practices of that small part of the software development community that is aware of the issues and wants to do something.

Some observers believe that general industry practice is much worse than the survey profile. Support for this belief can be found by comparing the corresponding practices in Figures 9 and 10. The findings in Figure 10 were extracted from a 1983 Government Accounting

	Test Practice	% Yes	% Sometimes
1	*Record* of *defects* found during testing is *maintained*	73	16
2	*Designated person is responsible for* the *test process*	65	13
3	*Test plan* describing objectives/ approach is *required*	61	29
4	Testing is a *systematic* and *organized* activity	61	30
5	*Full-time testers* perform system testing	62	19
6	Testing is *separated from development*	60	20
7	Tests are *required* to be *rerun* when software changes	51	35
8	*Tests* are *saved* and maintained for future use	51	28
9	*Test specifications* and designs are *documented*	48	36
10	Test procedure is *documented* in the *standards* manual	45	15
11	A *log of tests run* is maintained	42	35
12	A *record of the time spent* on testing is maintained	40	30
13	*Test documents* are formally peer-*reviewed*	31	29
14	*Full-time testers* perform integration testing	24	24
15	The *cost of testing* is measured and *tracked*	24	19
16	*Test training* is *provided* periodically	22	26
17	*Test results* are formally peer *reviewed*	20	31
18	*Users* are *heavily involved* in test activities	8	39
19	*Tests* are developed *before coding*	8	29
20	A *measurement of code coverage* achieved is *required*	5	16

FIGURE 9. Analysis of Industry Test Practice Usage

		% Yes	% No
1	Record of defects found during testing is maintained	30	70
3	Test plan describing objectives/approach is required	10	90
10	Test procedure is documented in the standards manual	44	56
20	A measurement of code coverage achieved is required	13	87

FIGURE 10. GAO 1983 Test Study Results

Office report on software testing practices in federal agencies [16].

Our sample of the best industry practices showed that even the best organizations have significant opportunities to improve their testing practices. It also showed that many are acting to improve and that there have been significant gains in putting effective test methods in place over the past several years. It further indicated that a period of fairly intense and continued assimilation and improvement lies ahead. Those organizations not actively improving their current testing practices may find they are actually losing ground due to a steadily moving industry standard.

Acknowledgments. We thank our friends and colleagues Boris Beizer, Fletcher Buckley, Richard Hamlet, Bill Howden, Tom Ostrand, and Elaine Weyuker for their help in reviewing this paper. They tried to make it better. If you like it, thank them; if you don't, blame us.

REFERENCES

1. *A Guide to Testing in a Complex System Environment.* Report GH20-1628. IBM, 1974.
2. *ANSI/IEEE STD 829—1983 Standard for Software Test Documentation.* Institute of Electrical and Electronics Engineers, New York, 1983.
3. *ANSI/IEEE STD 1008—1987 Standard for Software Unit Testing.* Institute of Electrical and Electronics Engineers, New York, 1986.
4. *ANSI/IEEE STD 1012—1986 Standard for Software Verification and Validation Plans.* Institute of Electrical and Electronics Engineers, New York, 1986.
5. Baker, C. Review of D.D. McCracken's "Digital Computer Programming". *Mathematical Tables and Other Aids to Computation 11,* 60 (Oct. 1957), 298–305.
6. Beizer, B. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold, New York, 1984.
7. Beizer, B. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1983.
8. Branstad, M.A. Cherniavsky, J.C., and Adrion, W.R. Validation, Verification, and Testing for the Individual Programmer. National Bureau of Standards Report NBS 500-56. Washington D.C., 1980.
9. Chandrasekaran, B., and Radicchi, S. Eds. Computer Program Testing, North-Holland Publishing Co., New York, 1981.
10. DeMillo, R.A., Lipton, R.J., and Sayward, F.G. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer 11,* 4 (Apr. 1978), 34–41.
11. DeMillo, R.A., McCracken, W.M., Martin, R.J., and Passafiume, J.F. *Software Testing and Evaluation.* Benjamin/Cummings Publishing, Menlo Park, Calif., 1987.
12. Deutsch, M.S. Software Project Verification and Validation. *Computer 14,* 4 (Apr. 1981), 54–70.
13. Deutsch, M.S. *Software Verification and Validation: Realistic Project Approaches.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
14. Dunn, R.H. *Software Defect Removal.* McGraw-Hill, New York, 1984.
15. Evans, M.W. Productive Software Test Management. John Wiley & Sons, New York, New York, 1984.
16. Greater Emphasis On Testing Needed To Make Computer Software More Reliable And Less Costly. GAO/IMTEC-84-2. Government Accounting Office, Washington, D.C., 1983.
17. Guideline for Lifecycle Validation, Verification, and Testing of Computer Software. National Bureau of Standards Report NBS FIPS 101. Washington, D.C., 1983.
18. Hausen, H.L. Ed. *Software Validation: Inspection–Testing–Verification–Alternatives.* North-Holland, Amsterdam, 1984.
19. Hetzel, W. *The Complete Guide to Software Testing, Second Edition.* QED Information Sciences, Wellesley, Mass., 1988.
20. Hetzel, W.C. Ed. *Program Test Methods.* Prentice-Hall, Englewood Cliffs, N.J., 1973.
21. Howden, W.E. *Algebraic Program Testing. Acta Informatica 10,* 1 (1978).
22. Howden, W.E. *Life-Cycle Software Validation. Computer 15,* 2 (Feb. 1982), 71–78.
23. Howden, W.E. *Functional Program Testing And Analysis.* McGraw-Hill, New York, 1987.
24. McCabe, T.J. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric.* National Bureau of Standards Report NBS 500-99. Washington, D.C., 1982.
25. Meyers, G.J. *The Art of Software Testing.* John Wiley & Sons, New York, 1979.
26. Military Standard for Defense System Software Development. DOD-STD-2167A. Department of Defense. Washington, D.C., 1987.
27. Miller, E., and Howden, W.E. Eds. *Tutorial: Software Testing and Validation Techniques.* IEEE Computer Society Press, New York, 1981.
28. Ould, M.A., and Unwin, C. Eds. *Testing in Software Development.* British Computer Society Monographs in Informatics, Cambridge University Press, Cambridge, England, 1986.
29. Perry, W.E. *A Structured Approach To Systems Testing.* QED Information Sciences, Wellesley, Mass., 1983.
30. Probert, R.L., and Ural, H. *High-Level Testing and Example-Directed Development of Software Specifications. J. Systems and Software 4,* 4 (Nov. 1984), 317–325.
31. *Proceedings of the National Conference on Software Test and Evaluation.* National Security Industrial Association, Washington, D.C., 1983.
32. *Proceedings of the Workshop on Software Testing.* July 15–17, 1986, Banff, Alberta, Canada, Sponsors ACM/SIGSOFT and IEEE/CS Software Engineering Technical Committee. IEEE Computer Society Press, New York, 1986.
33. Quirk, W.J. Ed. *Verification and Validation of Real-Time Software.* Springer-Verlag, Berlin, 1985.
34. Turing, A. *Checking a Large Routine.* Report of a Conference on High Speed Automatic Calculating-Machines (Jan. 1950), 67–69.
35. Turing, A. *Computing Machinery and Intelligence. Mind 59,* (Oct. 1950), 433–460.
36. White, L.J., and Cohen, E.I. *A Domain Strategy for Computer Program Testing.* IEEE Trans. Softw. Eng. *SE-6,* 3 (May 1980), 247–257.

Authors' Present Addresses: David Gelperin, Bill Hetzel, Software Quality Engineering, 3015 Hartley Road, Suite 16, Jacksonville, FL 32217.