An American National Standard

IEEE Standard for
Software Unit Testing

Sponsor

**Software Engineering Technical Committee
of the
IEEE Computer Society**

Approved December 11, 1986
Reaffirmed December 2, 1993

IEEE Standards Board

Approved July 28, 1986

American National Standards Institute

Foreword

Objectives

This standard's primary objective is to specify a standard approach to software unit testing that can be used as a basis for sound software engineering practice.

A second objective is to describe the software engineering concepts and testing assumptions on which this standard approach is based. This information is contained in Appendix B. Note that Appendix B is not a part of this standard.

A third objective is to provide guidance and resource information to assist with the implementation and usage of the standard unit testing approach. This information is contained in Appendixes A, C, and D. Note that these Appendixes are not a part of this standard.

Motivation

A consensus definition of sound unit testing provides a baseline for the evaluation of specific approaches. It also aids communication by providing a standard decomposition of the unit testing process.

Audience

The primary audience for this standard is unit testers and unit test supervisors. This standard was developed to assist those who provide input to, perform, supervise, monitor, and evaluate unit testing.

Relationship with Other Software Engineering Standards

ANSI/IEEE Std 829-1983, IEEE Standard for Software Test Documentation, describes the basic information needs and results of software testing. This unit testing standard requires the use of the test design specification and test summary report specified in ANSI/IEEE Std 829-1983.

This standard is one of a series aimed at establishing the norms of professional practice in software engineering. Any of the other software engineering standards in the series may be used in conjunction with it.

Terminology

Terminology in this standard is consistent with ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology. To avoid inconsistency when the glossary is revised, its definitions are not repeated in this standard.

The *test unit* referred to in this standard is a specific case of the test item referred to in ANSI/ IEEE 829-1983. The term *test unit* is used because of this standard's narrower scope.

The use of the term *specification*, *description*, or *document* refers to data recorded on either an electronic or paper medium.

The word *must* and imperative verb forms identify mandatory material within the standard. The words *should* and *may* identify optional material.

Overview

The unit testing process is composed of three *phases* that are partitioned into a total of eight basic activities as follows:
(1) *Perform the test planning*
      (a) Plan the general approach, resources, and schedule
      (b) Determine features to be tested
      (c) Refine the general plan
(2) *Acquire the test set*
      (a) Design the set of tests
      (b) Implement the refined plan and design

(3) *Measure the test unit*
        (a) Execute the test procedures
        (b) Check for termination
        (c) Evaluate the test effort and unit

The major dataflows into and out of the phases are shown in Fig A.
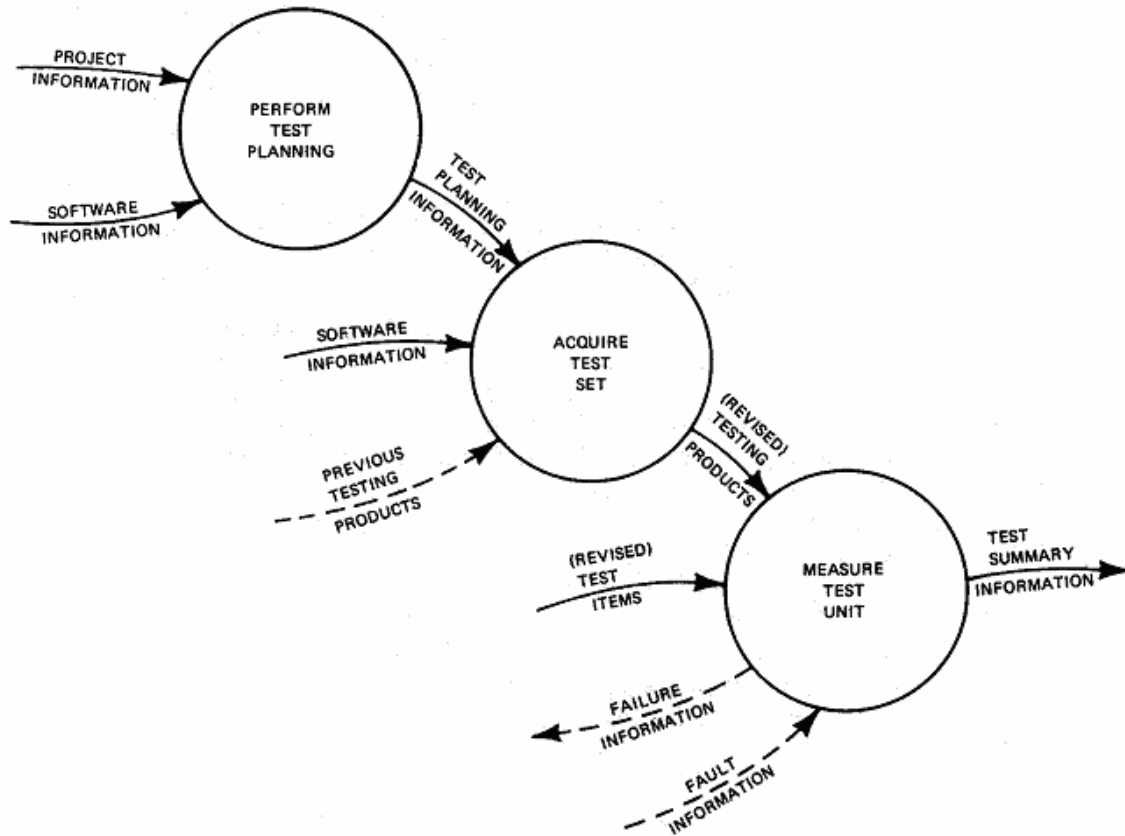
Fig A
Major Dataflows of the Software
Unit Testing Phases

    Within a phase, each basic activity is associated with its own set of inputs and outputs and is composed of a series of tasks. The inputs, tasks, and outputs for each activity are specified in the body of this standard.

    The set of outputs from all activities must contain sufficient information for the creation of at least two documents – a test design specification and a test summary report. Both documents must conform to the specifications in ANSI/IEEE Std 829-1983.

History

Work on this standard began in February 1983, following announcement of the formation of the task group in the technical and commercial press in late 1982. The project authorization request was approved by the IEEE Standards Board on June 23, 1983 following the second meeting. A total of seven meetings held throughout the United States at three month intervals produced the draft submitted for ballot in March 1985. A total of over 90 persons contributed to the initial development of this standard. Contributors are those individuals who either attended a working-group meeting, submitted written comments on a draft, or both.

This standard was developed by a working group with the following members:

David Gelperin, *Chairperson*   **Pat Wilburn**, *Cochairperson*

A. Frank Ackerman	Ken Foster	John Owens
Craig Adams	John Fox	William Perry
David Adams	Roger Fujii	Gerald Peterson
Jack Barnard	Ross Gagliano	Bob Poston
Wanda Beck	Mark Gerhard	Patricia Powell
Boris Beizer	Ed Gibson	Samuel T. Redwine, Jr
K. Mack Bishop	Therese Gilbertson	Sanford Rosen
Jill E. Boogaard	Gary Girard	Hans Schaefer
Milt Boyd	Keith Gordon	Eric Schnellman
Nathan B. Bradley	Paul Grizenko	Harvey Schock
Martha Branstad	Jeff Grove	Al Sema, Jr
Fletcher Buckley	Ismet Gungor	Harlan Seyfer
John W. Cain	Mark Heinrich	Victor Shtern
Christopher Cooke	Rudolph Hodges	Rick Simkin
L. L. Doc Craddock	R. A. Kessler	Wayne Smith
Palmer Craig	Tom Kurihara	Harry Sneed
Michael Cramer	Costas Labovites	Hugh B. Spillane
Dave Dahlinghaus	Frank LaMonica	Ben Sun
Noah Davids	F. C. Lim	Murray Tabachnick
Henry Davis	Philip C. Marriott	Barbara Taute
Bruce Dawson	Debra L. McCusker	Leonard Tripp
Claudia Dencker	Charlie McCutcheon	William S. Turner III
Michael Deutsch	Rudolf van Megen	John Vance
Judie Divita	Denis Meredith	Guy Vogt
Jim Dobbins	Edward Miller, Jr	Dolores Wallace
David C. Doty	William Milligan	John Walter
Bill Dupras	Marcus Mullins	John C. Wang
Jim Edwards	W. M. Murray	Cheryl Webb
Karen Fairchild	Bruce Nichols	William Wilson
Peter Farrell-Vinay	Dennis Nickle	Ed Yasi
Thom Foote-Lennox	Larry Nitzsche	Natalie C. Yopconka

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

Frank Ackerman	Russell Gustin	Bill Perry
Leo Beltracchi	Virl Haas	Poul Grav Peterson
Ron Berlack	William Hanrahan	Donald Pfeiffer
William Boll, Jr	Carolyn Harrison	Sharon Cobb-Pierson
F. Buckley	G.B. Hawthorne	Robert Poston
Douglas Burt	Clark Hay	Thomas Radi
Homer Carney	Leslie Heselton, III	Meir Razy
Skip Carpenter, Jr	Mark Holthouse	Larry Reed
Jung Chung	John Horch	R. Waldo Roth
Won Lyang Chung	Frank Jay	Raymond Sandborgh
Francois Coallier	Laurel Kaleda	Hans Schaefer
Richard Coates	Adi Kasad	Norm Schneidewind
A.J. Cote, Jr	Ron Kenett	Eric Schnellman
Patricia Daggett	Bob Kessler	Wolf Schnoege
Jim Darling	Gary Kroc	Robert Schueppert
N.S. Davids	Robert Lane	David Schultz
Henry Davis	Greg Larsen	Leonard Seagren
Peter Denny	F .C. Lim	Gary Shea
A. Dniestrowski	Bertil Lindberg	Craig Shermer
David Dobratz	Gary Lindsay	Robert Shillato
David Doty	Ben Livson	David Siefert
Walter DuBlanica	M. Malagarois	David Simkins
R.E. Dwyer	W .A. Mandeville	Shirley Gloss-Soler
Mary Eads	Stuart Marcotte	William Sutcliffe
John Earls	Philip C. Marriott	K.C. Tai
L.G. Egan	Roger Martin	Barbara Taute
John Fendrich	Paul Mauro	Paul Thompson
Glenn Fields	Belden Menkus	Terrence Tillmanns
Charles Finnell	Jerome Mersky	R.L. Van Tilburg
Jim Flournoy	Gene Morun	Dolores Wallace
Violet Foldes	Hironobu Nagano	M.L. Weisbein
Julian Forster	Geraldine Neidhart	Joseph Weiss
Rick Fredrick	G.A. Niblack	N.P. Wilburn
Lee Gagner	Dennis Nickle	Paul Wolfgang
M. Galinier	Larry Nitzsche	Charles Wortz
David Gelperin	Peter Olsen	Edward Yasi
L.M. Gunther	Wilma Osborne	Zhou Zhi Ying
David Gustafson		Natalie Yopconka

When the IEEE Standards Board approved this standard on December 11, 1986, it had the following membership:

John E. May, *Chairman* **Irving Kolodny**, *Vice Chairman*

Sava I. Sherr, *Secretary*

James H. Beall	Jack Kinn	Robert E. Rountree
Fletcher J. Buckley	Joseph L. Koepfinger[*]	Martha Sloan
Paul G. Cummings	Edward Lohse	Oley Wanaselja
Donald C. Fleckenstein	Lawrence V. McCall J.	Richard Weger
Jay Forster	Donald T. Michael[*]	William B. Wilkens
Daniel L. Goldberg	Marco W. Migliaro	Helen M. Wood
Kenneth D. Hendrix	StanleyOwens	Charles J. Wylie
lrvin N. Howell	John P. Riganati	Donald W. Zipse
	Frank L. Rose	

[*]Member emeritus

Contents

An American National Standard

IEEE Standard for
Software Unit Testing

1. Scope and References

1.1 Inside the Scope. Software unit testing is a process that includes the performance of test planning, the acquisition of a test set, and the measurement of a test unit against its requirements. Measuring entails the use of sample data to exercise the unit and the comparison of the unit's actual behavior with its required behavior as specified in the unit's requirements documentation.

This standard defines an integrated approach to systematic and documented unit testing. The approach uses unit design and unit implementation information, in addition to unit requirements, to determine the completeness of the testing.

This standard describes a testing process composed of a hierarchy of phases, activities, and tasks and defines a minimum set of tasks for each activity. Additional tasks may be added to any activity.

This standard requires the performance of each activity. For each task within an activity, this standard requires either that the task be performed, or that previous results be available and be reverified. This standard also requires the preparation of two documents specified in ANSI/IEEE Std 829-1983[2][1]. These documents are the Test Design Specification and the Test Summary Report.

General unit test planning should occur during overall test planning. This general unit test planning activity is covered by this standard, although the balance of the overall test planning process is outside the scope of this standard.

This standard may be applied to the unit testing of any digital computer software or firmware. However, this standard does *not* specify any class of software or firmware to which it must be

applied, nor does it specify any class of software or firmware that must be unit tested. This standard applies to the testing of newly developed and modified units.

This standard is applicable whether or not the unit tester is also the developer.

1.2 Outside the Scope. The results of some overall test planning tasks apply to all testing levels (for example, identify security and privacy constraints). Such tasks are not considered a part of the unit testing process, although they directly affect it.

While the standard identifies a need for failure analysis information and software fault correction, it does not specify a software debugging process.

This standard does not address other components of a comprehensive unit verification and validation process, such as reviews (for example, walkthroughs, inspections), static analysis (for example, consistency checks, data flow analysis), or formal analysis (for example, proof of correctness, symbolic execution).

This standard does not require the use of specific test facilities or tools. This standard does not imply any particular methodology for documentation control, configuration management, quality assurance, or management of the testing process.

1.3 References. This standard shall be used in conjunction with the following publications.
[1] ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology.[2]
[2] ANSI/IEEE Std 829-1983, IEEE Standard for Software Test Documentation.

---

[1] The numbers in brackets correspond to the references listed in 1.3 of this standard

[2] This publications are available from American National Standards Institute, Sales Department, 1430 Broadway, New York, NY 10018 and from IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854.

2. Definitions

This section defines key terms used in this standard but not included in ANSI/IEEE Std 729-1983 [1] or ANSI/IEEE Std 829-1983 [2].

characteristic. *See*: **data characteristic** or **software characteristic**.

data characteristic. An inherent, possibly accidental, trait, quality, or property of data (for example, arrival rates, formats, value ranges, or relationships between field values).

feature. *See*: **software feature**.

incident. *See*: **software test incident**.

nonprocedural programming language. A computer programming language used to express the parameters of a problem rather than the steps in a solution (for example, report writer or sort specification languages). Contrast with **procedural programming language**.

procedural programming language. A computer programming language used to express the sequence of operations to be performed by a computer (for example, COBOL). Contrast with **nonprocedural programming language**.

software characteristic. An inherent, possibly accidental, trait, quality, or property of software (for example, functionality, performance, attributes, design constraints, number of states, lines of branches).

software feature. A software characteristic specified or implied by requirements documentation (for example, functionality, performance, attributes, or design constraints).

software test incident. Any event occurring during the execution of a software test that requires investigation.

state data. Data that defines an internal state of the test unit and is used to establish that state or compare with existing states.

test objective. An identified set of software features to be measured under specified conditions by comparing actual behavior with the required behavior described in the software documentation.

test set architecture. The nested relationships between sets of test cases that directly reflect the hierarchic decomposition of the test objectives.

test unit.[3] A set of one or more computer program modules together with associated control data, (for example, tables), usage procedures, and operating procedures that satisfy the following conditions:

(1) All modules are from a single computer program

(2) At least one of the new or changed modules in the set has not completed the unit test[4]

(3) The set of modules together with its associated data and procedures are the sole object of a testing process

unit. *See*: **test unit**.

unit requirements documentation. Documentation that sets forth the functional, interface, performance, and design constraint requirements for the test unit.

3. Unit Testing Activities

This section specifies the activities involved in the unit testing process and describes the associated input, tasks, and output. The activities described are as follows:

(1) Perform test planning phase

(a) Plan the general approach, resources, and schedule

(b) Determine features to be tested

(c) Refine the general plan

(2) Acquire test set phase

(a) Design the set of tests

(b) Implement the refined plan and design

(3) Measure test unit phase

(a) Execute the test procedures

(b) Check for termination

(c) Evaluate the test effort and unit

When more than one unit is to be unit tested (for example, all those associated with a software project), the Plan activity should address the total set of test units and should not be repeated for each test unit. The other activities must be performed at least once for each unit.

Under normal conditions, these activities are sequentially initiated except for the Execute and

---

[3] A test unit may occur at any level of the design hierarchy from a single module to a complete program. Therefore, a test unit may be a module, a few modules, or a complete computer program along with associated data and procedures.

[4] A test unit may contain one or more modules that have already been unit tested.

Check cycle as illustrated in Fig 1. When performing any of the activities except Plan, improper performance of a preceding activity or external events (for example, schedule, requirements, or design changes) may result in the need to redo one or more of the preceding activities and then return to the one being performed.
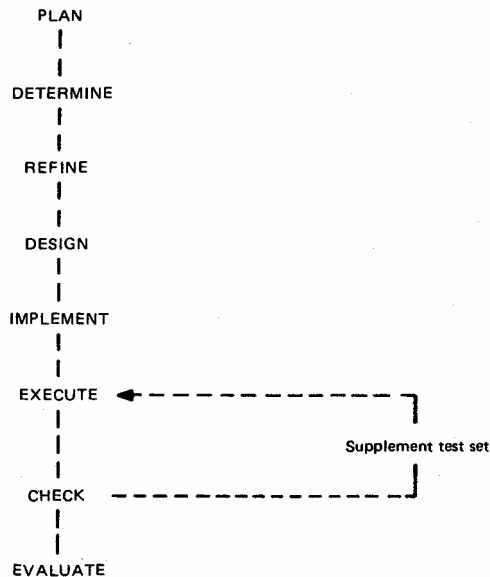

Fig 1
Unit Testing Activities

During the testing process, a test design specification and a test summary report must be developed. Other test documents may be developed. All test documents must conform to the ANSI/IEEE Std 829-1983 [2]. In addition, all test documents must have identified authors and be dated.

The test design specification will derive its information from the Determine, Refine, and Design activities. The test summary report will derive its information from all of the activities.

3.1 Plan the General Approach, Resources, and Schedule. General unit test planning should occur during overall test planning and be recorded in the corresponding planning document.

3.1.1 Plan Inputs

(1) Project plans

(2) Software requirements documentation

3.1.2 Plan Tasks

(1) *Specify a General Approach to Unit Testing*. Identify risk areas to be addressed by the testing. Specify constraints on characteristic determination (for example, features that must be tested), test design, or test implementation (for example, test sets that must be used).

Identify existing sources of input, output, and state data (for example, test files, production files, test data generators). Identify general techniques for data validation. Identify general techniques to be used for output recording, collection, reduction, and validation. Describe provisions for application software that directly interfaces with the units to be tested.

(2) *Specify Completeness Requirements*. Identify the areas (for example, features, procedures, states, functions, data characteristics, instructions) to be covered by the unit test set and the degree of coverage required for each area.

When testing a unit during software development, every software feature must be covered by a test case or an approved exception. The same should hold during software maintenance for any unit testing.

When testing a unit implemented with a procedural language (for example, COBOL) during software development, every instruction that can be reached and executed must be covered by a test case or an approved exception, except for instructions contained in modules that have been separately unit tested. The same should hold during software maintenance for the testing of a unit implemented with a procedural language.

(3) *Specify Termination Requirements*. Specify the requirements for normal termination of the unit testing process. Termination requirements must include satisfying the completeness requirements.

Identify any conditions that could cause abnormal termination of the unit testing process (for example, detecting a major design fault, reaching a schedule deadline) and any notification procedures that apply.

(4) *Determine Resource Requirements*. Estimate the resources required for test set acquisition, initial execution, and subsequent repetition of testing activities. Consider hardware, access time (for example, dedicated computer time), communications or system software, test tools, test files, and forms or other supplies. Also consider the need for unusually large volumes of forms and supplies.

Identify resources needing preparation and the parties responsible. Make arrangements for these resources, including requests for resources that require significant lead time (for example, customized test tools).

Identify the parties responsible for unit testing and unit debugging. Identify personnel requirements including skills, number, and duration.

(5) *Specify a General Schedule*. Specify a schedule constrained by resource and test unit availability for all unit testing activity.

3.1.3 Plan Outputs

(1) General unit test planning information (from 3.1.2(1) through (5) inclusive)

(2) Unit test general resource requests – if produced from 3.1.2(4)

3.2 Determine Features To Be Tested

3.2.1 Determine Inputs

(1) Unit requirements documentation

(2) Software architectural design documentation – if needed

3.2.2 Determine Tasks

(1) *Study the Functional Requirements*. Study each function described in the unit requirements documentation. Ensure that each function has a unique identifier. When necessary, request clarification of the requirements.

(2) *Identify Additional Requirements and Associated Procedures*. Identify requirements other than functions (for example, performance, attributes, or design constraints) associated with software characteristics that can be effectively tested at the unit level. Identify any usage or operating procedures associated only with the unit to be tested. Ensure that each additional requirement and procedure has a unique identifier. When necessary, request clarification of the requirements.

(3) *Identify States of the Unit*. If the unit requirements documentation specifies or implies multiple states (for example, inactive, ready to receive, processing) software, identify each state and each valid state transition. Ensure that each state and state transition has a unique identifier. When necessary, request clarification of the requirements.

(4) *Identify Input and Output Data Characteristics*. Identify the input and output data structures of the unit to be tested. For each structure, identify characteristics, such as arrival rates, formats, value ranges, and relationships between field values. For each characteristic, specify its valid ranges. Ensure that each characteristic has a unique identifier. When necessary, request clarification of the requirements.

(5) *Select Elements to be Included in the Testing*. Select the features to be tested. Select the associated procedures, associated states, associated state transitions, and associated data characteristics to be included in the testing.

Invalid and valid input data must be selected. When complete testing is impractical, information regarding the expected use of the unit should be used to determine the selections. Identify the risk associated with unselected elements.

Enter the selected features, procedures, states, state transitions, and data characteristics in the *Features to be Tested* section of the unit's Test Design Specification.

3.2.3 Determine Outputs

(1) List of elements to be included in the testing (from 3.2.2(5))

(2) Unit requirements clarification requests – if produced from 3.2.2(1) through (4) inclusive

3.3 Refine the General Plan

3.3.1 Refine Inputs

(1) List of elements to be included in the testing (from 3.2.2(5))

(2) General unit test planning information (from 3.1.2(1) through (5) inclusive)

3.3.2 Refine Tasks

(1) *Refine the Approach*. Identify existing test cases and test procedures to be considered for use. Identify any special techniques to be used for data validation. Identify any special techniques to be used for output recording, collection, reduction, and validation.

Record the refined approach in the *Approach Refinements* section of the unit's test design specification.

(2) *Specify Special Resource Requirements*. Identify any special resources needed to test the unit (for example, software that directly interfaces with the unit). Make preparations for the identified resources.

Record the special resource requirements in the *Approach Refinements* section of the unit's test design specification.

(3) *Specify a Detailed Schedule*. Specify a schedule for the unit testing based on support software, special resource, and unit availability and integration schedules. Record the schedule in the *Approach Refinements* section of the unit's test design specification.

3.3.3 Refine Outputs

(1) Specific unit test planning information (from 3.3.2(1) through (3) inclusive)

(2) Unit test special resource requests – if produced from 3.3.2(2).

3.4 Design the Set of Tests

3.4.1 Design Inputs

(1) Unit requirements documentation

(2) List of elements to be included in the testing (from 3.2.2(5))

(3) Unit test planning information (from 3.1.2(1) and (2) and 3.3.2(1))

(4) Unit design documentation

(5) Test specifications from previous testing – if available

3.4.2 Design Tasks

(1) *Design the Architecture of the Test Set*. Based on the features to be tested and the conditions specified or implied by the selected associated elements (for example, procedures, state transitions, data characteristics), design a hierarchically decomposed set of test objectives so that each lowest-level objective can be directly tested by a few test cases. Select appropriate existing test cases. Associate groups of test-case identifiers with the lowest-level objectives. Record the hierarchy of objectives and associated test case identifiers in the *Test Identification* section of the unit's test design specification.

(2) *Obtain Explicit Test Procedures as Required*. A combination of the unit requirements documentation, test planning information, and test-case specifications may implicitly specify the unit test procedures and therefore minimize the need for explicit specification. Select existing test procedures that can be modified or used without modification.

Specify any additional procedures needed either in a supplementary section in the unit's test design specification or in a separate procedure specification document. Either choice must be in accordance with the information required by ANSI/IEEE Std 829-1983 [2]. When the correlation between test cases and procedures is not readily apparent, develop a table relating them and include it in the unit's test design specification.

(3) *Obtain the Test Case Specifications*. Specify the new test cases. Existing specifications may be referenced.

Record the specifications directly or by reference in either a supplementary section of the unit's test design specification or in a separate document. Either choice must be in accordance with the information required by ANSI/ IEEE Std. 829-1983 [2].

(4) *Augment, as Required, the Set of Test-Case Specifications Based on Design Information*. Based on information about the unit's design,

update as required the test set architecture in accordance with 3.4.2(1). Consider the characteristics of selected algorithms and internal data structures.

Identify control flows and changes to internal data that must be recorded. Anticipate special recording difficulties that might arise, for example, from a need to trace control flow in complex algorithms or from a need to trace changes in internal data structures (for example, stacks or trees). When necessary, request enhancement of the unit design (for example, a formatted data structure dump capability) to increase the testability of the unit.

Based on information in the unit's design, specify any newly identified test cases and complete any partial test case specifications in accordance with 3.4.2(3).

(5) *Complete the Test Design Specification*. Complete the test design specification for the unit in accordance with ANSI/IEEE Std 829-1983 [2].

3.4.3 Design Outputs

(1) Unit test design specification (from 3.4.2(5))

(2) Separate test procedure specifications – if produced from 3.4.2(2)

(3) Separate test-case specifications – if produced from 3.4.2(3) or (4)

(4) Unit design enhancement requests – if produced from 3.4.2(4)

3.5 Implement the Refined Plan and Design

3.5.1 Implement Inputs

(1) Unit test planning information (from 3.1.2(1), (4), and (5) and 3.3.2(1) through (3) inclusive)

(2) Test-case specifications in the unit test design specification or separate documents (from 3.4.2(3) and (4))

(3) Software data structure descriptions

(4) Test support resources

(5) Test items

(6) Test data from previous testing activities – if available

(7) Test tools from previous testing activities – if available

3.5.2 Implement Tasks

(1) *Obtain and Verify Test Data*. Obtain a copy of existing test data to be modified or used without modification. Generate any new data required. Include additional data necessary to ensure data consistency and integrity. Verify all

data (including those to be used as is) against the software data structure specifications. When the correlation between test cases and data sets is not readily apparent, develop a table to record this correlation and include it in the unit's test design specification.

(2) *Obtain Special Resources*. Obtain the test support resources specified in 3.3.2(2).

(3) *Obtain Test Items*. Collect test items including available manuals, operating system procedures, control data (for example, tables), and computer programs. Obtain software identified during test planning that directly interfaces with the test unit.

When testing a unit implemented with a procedural language, ensure that execution trace information will be available to evaluate satisfaction of the code-based completeness requirements.

Record the identifier of each item in the *Summary* section of the unit's test summary report.

3.5.3 Implement Outputs

(1) Verified test data (from 3.5.2(1))

(2) Test support resources (from 3.5.2(2))

(3) Configuration of test items (from 3.5.2(3))

(4) Initial summary information (from 3.5.2 (3))

3.6 Execute the Test Procedures

3.6.1 Execute Inputs

(1) Verified test data (from 3.5.2(1))

(2) Test support resources (from 3.5.2(2))

(3) Configuration of test items (from 3.5.2(3))

(4) Test-case specifications   (from 3.4.2(3) and (4))

(5) Test procedure specifications (from 3.4.2 (2)) – if produced

(6) Failure analysis results (from debugging process) – if produced

3.6.2 Execute Tasks

(1) *Run Tests*. Set up the test environment. Run the test set. Record all test incidents in the *Summary of Results* section of the unit's test summary report.

(2) *Determine Results*. For each test case, determine if the unit passed or failed based on required result specifications in the case descriptions. Record pass or fail results in the *Summary of Results* section of the unit's test summary report. Record resource consumption data in the *Summary of Activities* section of the report. When testing a unit implemented with a procedural language, collect execution trace summary information and attach it to the report.

For each failure, have the failure analyzed and record the fault information in the *Summary of Results* section of the test summary report. Then select the applicable case and perform the associated actions.

Case 1:   *A Fault in a Test Specification or Test Data*. Correct the fault, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun the tests that failed.

Case 2:   *A Fault in Test Procedure Execution*. Rerun the incorrectly executed procedures.

Case 3:   *A Fault in the Test Environment (for example, system software)*. Either have the environment corrected, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun the tests that failed OR prepare for abnormal termination by documenting the reason for not correcting the environment in the *Summary of Activities* section of the test summary report and proceed to check for termination (that is, proceed to activity 3.7).

Case 4:   *A Fault in the Unit Implementation*. Either have the unit corrected, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun all tests OR prepare for abnormal termination by documenting the reason for not correcting the unit in the *Summary of Activities* section of the test summary report and proceed to check for termination (that is, proceed to activity 3.7).

Case 5:   *A Fault in the Unit Design*. Either have the design and unit corrected, modify the test specification and data as appropriate, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun all tests OR prepare for abnormal termination by documenting the reason for not correcting the design in the *Summary of Activities* section of the test summary report and proceed to check for termination (that is, proceed to activity 3.7).

NOTE: The cycle of Execute and Check Tasks must be repeated until a termination condition defined in 3.1.2(3) is satisfied (See Fig 3). Control flow within the Execute activity itself is pictured in Fig 2.
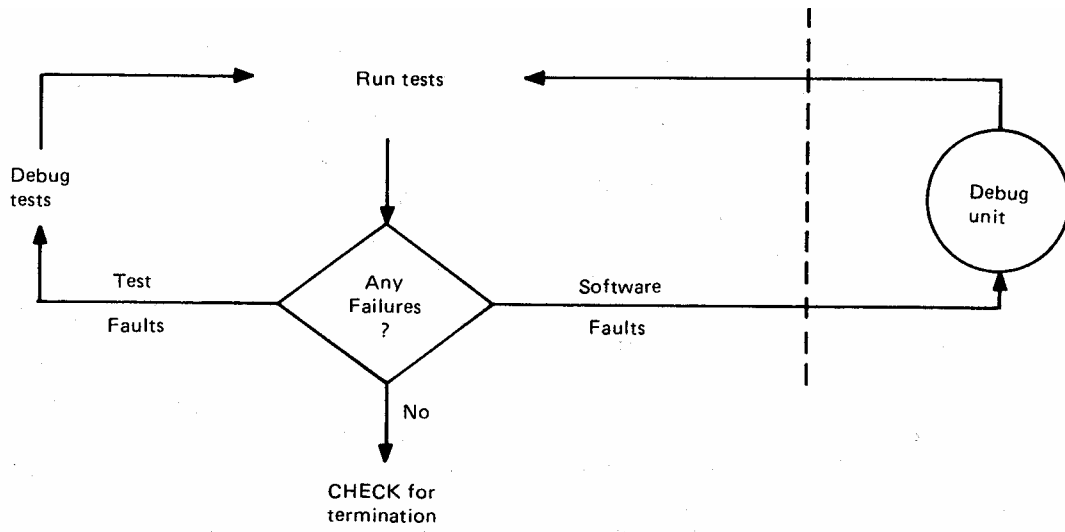
Fig 2
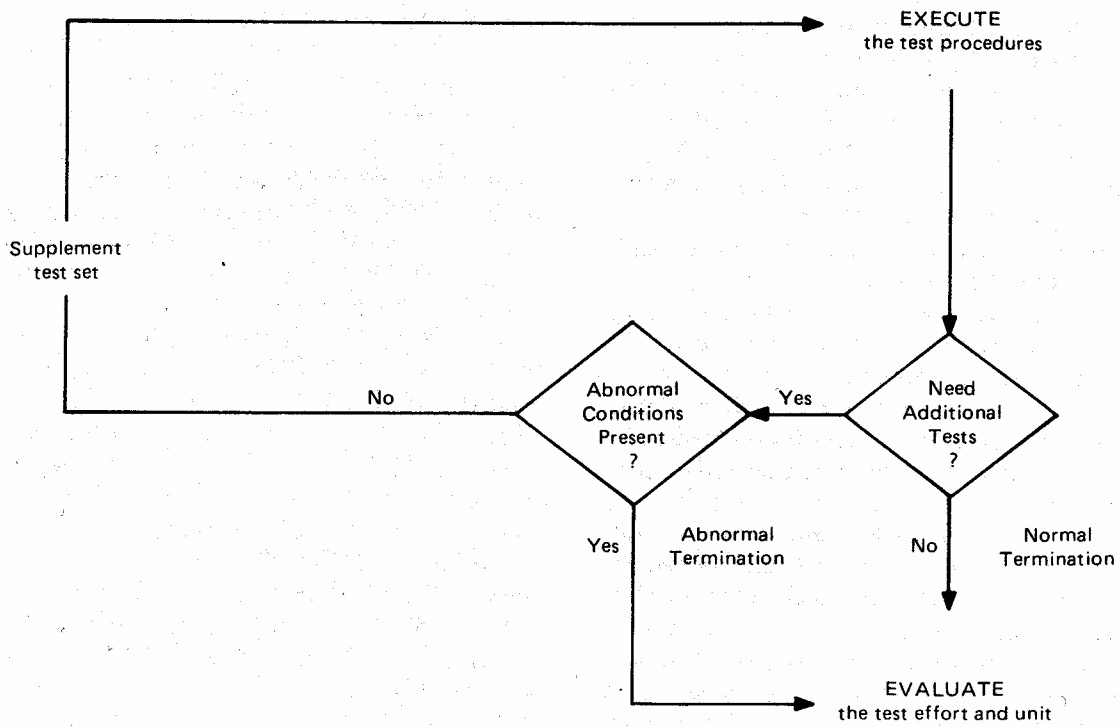**Control Flow Within the Execute Activity**

Fig 3
Control Flow Within the Check Activity

3.6.3 Execute Outputs

(1) Execution information logged in the test summary report including test outcomes, test incident descriptions, failure analysis results, fault correction activities, uncorrected fault reasons, resource consumption data and, for procedural language implementations, trace summary information (from 3.6.2(1) and (2))

(2) Revised test specifications – if produced from 3.6.2(2)

(3) Revised test data – if produced from 3.6.2(2)

3.7 Check for Termination

3.7.1 Check Inputs

(1) Completeness and termination requirements (from 3.1.2(2) and (3))

(2) Execution information (from 3.6.2(1) and (2))

(3) Test specifications (from 3.4.2(1) through (3) inclusive) – if required

(4) Software data structure descriptions – if required

3.7.2 Check Tasks

(1) *Check for Normal Termination of the Testing Process*. Determine the need for additional tests based on completeness requirements or concerns raised by the failure history. For procedural language implementations, analyze the execution trace summary information (for example, variable, flow).

If additional tests are not needed, then record normal termination in the *Summary of Activities* section of the test summary report and proceed to evaluate the test effort and unit (that is, proceed to activity 3.8).

(2) *Check for Abnormal Termination of the Testing Process*. If an abnormal termination condition is satisfied (for example, uncorrected major fault, out of time) then ensure that the specific situation causing termination is documented in the *Summary of Activities* section of the test summary report together with the unfinished testing and any uncorrected faults. Then proceed to evaluate the test effort and unit (that is, proceed to activity 3.8).

(3) *Supplement the Test Set*. When additional tests are needed and the abnormal termination conditions are not satisfied, supplement the test set by following steps (a) through (e).

(a) Update the test set architecture in accordance with 3.4.2(1) and obtain additional test-case specifications in accordance with 3.4.2(3).

(b) Modify the test procedure specifications in accordance with 3.4.2(2) as required.

(c) Obtain additional test data in accordance with 3.5.2(1).

(d) Record the addition in the *Summary of Activities* section of the test summary report.

(e) Execute the additional tests (that is, return to activity 3.6).

3.7.3 Check Outputs

(1) Check information logged in the test summary report including the termination conditions and any test case addition activities (from 3.7.2(1) through (3) inclusive)

(2) Additional or revised test specifications – if produced from 3.7.2(3)

(3) Additional test data – if produced from 3.7.2(3)

3.8 Evaluate the Test Effort and Unit

3.8.1 Evaluate Inputs

(1) Unit Test Design Specification (from 3.4.2(5)

(2) Execution information (from 3.6.2(1) and (2))

(3) Checking information (from 3.7.2(1) through (3) inclusive)

(4) Separate test-case specifications (from 3.4.2(3) and (4)) – if produced

3.8.2 Evaluate Tasks

(1) *Describe Testing Status*. Record variances from test plans and test specifications in the *Variances* section of the test summary report. Specify the reason for each variance.

For abnormal termination, identify areas insufficiently covered by the testing and record reasons in the *Comprehensiveness Assessment* section of the test summary report.

Identify unresolved test incidents and the reasons for a lack of resolution in the *Summary of Results* section of the test summary report.

(2) *Describe Unit's Status*. Record differences revealed by testing between the unit and its requirements documentation in the *Variances* section of the test summary report.

Evaluate the unit design and implementation against requirements based on test results and detected fault information. Record evaluation information in the *Evaluation* section of the test summary report.

(3) *Complete the Test Summary Report*. Complete the test summary report for the unit in accordance with ANSI/IEEE Std 829-1983 [2].

(4) *Ensure Preservation of Testing Products*. Ensure that the testing products are collected,

organized, and stored for reference and reuse. These products include the test design specification, separate test-case specifications, separate test procedure specifications, test data, test data generation procedures, test drivers and stubs, and the test summary report.

3.8.3 Evaluate Outputs

(1) Complete test summary report (from 3.8.2(3))

(2) Complete, stored collection of testing products (from 3.8.2(4))

Appendixes

(These Appendixes are not a part of ANSI /IEEE Std 1008-1987, IEEE Standard for Software Unit Testing, but are included for information only.)

Appendix A

Implementation and Usage Guidelines

This section contains information intended to be of benefit when the standard is being considered for use. It is therefore recommended that this section be read in its entirety before any extensive planning is done.

A1. Use of the Standard

The standard can be used
(1) As a basis for comparison to confirm current practices
(2) As a source of ideas to modify current practices
(3) As a replacement for current practices

A2. Additional Testing Requirements

Requirements such as the amount of additional test documentation (for example, test logs), the level of detail to be included, and the number and types of approvals and reviews must be specified for each project. Factors, such as unit criticality, auditing needs, or contract specifications will often dictate these requirements. The standard leaves it to the user to specify these requirements either by individual project or as organizational standards. If the requirements are project specific, they should appear in the project plan, quality assurance plan, verification and validation plan, or overall test plan.

A3. Additional Test Documentation

The information contained in the test design specification and the test summary report is considered an absolute minimum for process visibility. In addition, it is assumed that any test information need can be satisfied by the set of test documents specified in ANSI/IEEE Std 829 -1983 [2], either by requiring additional content in a required document or by requiring additional documents.

A4. Approvals and Reviews

If more control is desired, the following additional tasks should be considered:
(1) Approval of general approach at the end of Plan
(2) Approval of identified requirements at the end of Determine
(3) Approval of specific plans at the end of Refine
(4) Approval of test specifications at the end of Design
(5) Review of test readiness at the end of Implement
(6) Review of test summary report at the end of Evaluate

A5. Audit Trails

It is assumed that auditing needs are taken into account when specifying control requirements. Therefore, the set of test documents generated together with the reports from test reviews should be sufficient to supply all required audit information.

A6. Configuration Management

Configuration management should be the source of the software requirements, software architectural design, software data structure, and unit requirements documentation. These inputs must be managed to ensure confidence that we have current information and will be notified of any changes.

The final unit testing products should be provided to configuration management. These

outputs must be managed to permit thorough and economical regression testing. See ANSI/IEEE Std 828-1983, IEEE Standard for Software Configuration Management Plans, for details.

A7. Determination of Requirements-Based Characteristics

Psychological factors (for example, self-confidence, a detailed knowledge of the unit design) can make it very difficult for the unit developer to determine an effective set of requirements-based elements (for example, features, procedures, state transitions, data characteristics) to be included in the testing. Often, this determination should be made by someone else.

There are several ways to organize this separation.

(1) Developers determine these elements for each other.

(2) Developers fully test each other's code. This has the added advantage that at least two developers will have a detailed knowledge of every unit.

(3) A separate test group should be available. The size of the project or the criticality of the software may determine whether a separate group can be justified.

If developers determine requirements-based elements for their own software, they should perform this determination before software design begins.

A8. User Involvement

If the unit to be tested interacts with users (for example, menu displays), it can be very effective to involve those users in determining the requirements-based elements to be included in the testing. Asking users about their use of the software may bring to light valuable information to be considered during test planning. For example, questioning may identify the relative criticality of the unit's functions and thus determine the testing emphasis.

A9. Stronger Code-Based Coverage Requirements

Based on the criticality of the unit or a shortage of unit requirement and design information (for

example, during maintenance of older software), the code-based coverage requirement specified in 3.1.2(2) could be strengthened. One option is to strengthen the requirement from instruction coverage to branch coverage (that is, the execution of every branch in the unit).

A10. Code Coverage Tools

An automated means of recording the coverage of source code during unit test execution is highly recommended. Automation is usually necessary because manual coverage analysis is unreliable and uneconomical. One automated approach uses a code instrumentation and reporting tool. Such a tool places software probes in the source code and following execution of the test cases provides a report summarizing data and control-flow information. The report identifies unexecuted instructions. Some tools also identify unexecuted branches. This capability is a feature in some compilers.

A11. Process Improvement

To evaluate and improve the effectiveness of unit testing, it is recommended that failure data be gathered from those processes that follow unit testing, such as integration test, system test, and production use. This data should then be analyzed to determine the nature of those faults that should have been detected by unit testing but were not.

A12. Adopting the Standard

Implementing a new technical process is itself a process that requires planning, implementation, and evaluation effort. To successfully implement a testing process based on this standard, one must develop an implementation strategy and tailor the standard. Both activities must reflect the culture and current abilities of the organization. Long-term success will require management commitment, supporting policies, tools, training, and start-up consulting. Management can demonstrate commitment by incorporating the new process into project tracking systems and performance evaluation criteria.

A13. Practicality of the Standard

This standard represents consensus on the definition of good software engineering practice. Some organizations use practices similar to the process specified here while others organize this work quite differently. In any case, it will involve considerable change for many organizations that choose to adopt it. That change involves new policies, new standards and procedures, new tools, and new training programs. If the differences between the standard and current practice are too great, then the changes will need to be phased in. The answer to the question of practicality is basically one of desire. How badly does an organization want to gain control of its unit testing?

Appendix B

Concepts and Assumptions

B1. Software Engineering Concepts

The standard unit testing process specified in this standard is based on several fundamental software engineering concepts which are described in B1.1 through B1.8 inclusive.

B1.1 Relationship of Testing to Verification and Validation. Testing is just one of several complementary verification and validation activities. Other activities include technical reviews (for example, code inspections), static analysis, and proof of correctness. Specification of a comprehensive verification and validation process is outside the scope of this standard.

B1.2 Testing As Product Development. Testing includes a product development process. It results in a *test set* composed of data, test support software, and procedures for its use. This product is documented by test specifications and reports. As with any product development process, test set development requires planning, requirements (test objectives), design, implementation, and evaluation.

B1.3 Composition of Debugging. The debugging process is made up of two major activities. The objective of the first activity, *failure analysis*, is to locate and identify all faults responsible for a failure. The objective of the second, fault correction, is to remove all identified faults while avoiding the introduction of new ones.

Specification of the process of either failure analysis or fault correction is outside the scope of this standard.

B1.4 Relationship of Testing to Debugging. Testing entails attempts to cause failures in order to detect faults, while debugging entails both failure analysis to locate and identify the associated faults and subsequent fault correction. Testing may need the results of debugging's failure analysis to decide on a course of action. Those actions may include the termination of testing or a request for requirements changes or fault correction.

B1.5 Relationship Between Types of Units. A one-to-one relationship between design units, implementation units, and test units is not necessary. Several design units may make up an implementation unit (for example, a program) and several implementation units may make up a test unit.

B1.6 Need for Design and Implementation Information. Often, requirements information is not enough for effective testing, even though, fundamentally, testing measures actual behaviour against required behaviour. This is because it's usually not feasible to test all possible situations and requirements often do not provide sufficient guidance in identifying situations that have high failure potential. Design and implementation information often are needed, since some of these

high-potential situations result from the design and implementation choices that have been made.

B1.7 Incremental Specification of Elements To Be Considered in Testing. Progressively more detailed information about the nature of a test unit is found in the unit requirements documentation, the unit design documentation, and finally in the unit's implementation. As a result, the elements to be considered in testing may be built up incrementally during different periods of test activity.

For procedural language (for example, COBOL) implementations, element specification occurs in three increments. The first group is specified during the Determine activity and is based on the unit requirements documentation. The second group is specified during the Design activity and is based on the unit design (that is, algorithms and data structures) as stated in a software design description. The third group is specified during the Check activity and is based on the unit's code.

For nonprocedural language (for example, report writer or sort specification languages) implementations, specification occurs in two increments. The first is during the Determine activity and is based on requirements and the second is during Design and is based on the non-procedural specification.

An incremental approach permits unit testing to begin as soon as unit requirements are available and minimizes the bias introduced by detailed knowledge of the unit design and code.

B1.8 Incremental Creation of a Test Design Specification. Information recorded in the test design specification is generated during the Determine, Refine, and Design activities. As each of these test activities progress, information is recorded in appropriate sections of the specification. The whole document must be complete at the end of the final iteration of the Design activity.

B1.9 Incremental Creation of the Test Summary Report. Information recorded in the test summary report is generated during all unit testing activities expect Plan. The report is initiated during Implement, updated during Execute and Check, and completed during Evaluate.

B2. Testing Assumptions

The approach to unit testing specified in this standard is based on a variety of economic, psychological, and technical assumptions. The significant assumptions are given in B2.1 through B2.7 inclusive.

B2.1 The objective of unit testing is to attempt to determine the correctness and completeness of an implementation with respect to unit requirements and design documentation by attempting to uncover faults in:

(1) The unit's required features in combination with their associated states (for example, inactive, active awaiting a message, active processing a message)

(2) The unit's handling of invalid input

(3) Any usage or operating procedures associated only with the unit

(4) The unit's algorithms or internal data structures, or both

(5) The decision boundaries of the unit's control logic

B2.2 Testing entails the measurement of behavior against requirements. Although one speaks informally of *interface testing*, *state testing*, or even *requirement testing*, what is meant is measuring actual behavior associated with an interface, state, or requirement, against the corresponding required behavior. Any verifiable unit testing process must have documented requirements for the test unit. This standard assumes that the documentation of unit requirements exists before testing begins.

B2.3 Unit requirements documentation must be thoroughly reviewed for completeness, testability, and traceability. This standard assumes the requirements have been reviewed either as a normal part of the documentation review process or in a special unit requirements review.

B2.4 There are significant economic benefits in the early detection of faults. This implies that test set development should start as soon as practical following availability of the unit requirements documentation because of the resulting requirements verification and validation. It also implies that as much as practical should be tested at the unit level.

B2.5 The levels of project testing (for example, acceptance, system, integration, unit) are specified in project plans, verification and validation plans, or overall test plans. Also included is the unit test planning information that is applicable to all units being tested (for example, completeness requirements, termination requirements, general resource requirements). Subsequently, based on an analysis of the software design, the test units will be identified and an integration sequence will be selected.

B2.6 The availability of inputs and resources to do a task is the major constraint on the sequencing of activities and on the sequencing of tasks within an activity. If the necessary resources are available, some of the activities and some of the tasks within an activity may be performed concurrently.

B2.7 This standard assumes that it is usually most cost-effective to delay the design of test cases based on source-code characteristics until the set of test cases based on requirements and design characteristics has been executed. This approach minimizes the code-based design task. If code-based design is started before test execution data is available, it should not start until the test cases based on unit requirements and design characteristics have been specified.

Appendix C

Sources for Techniques and Tools

C1. General

Software tools are computer programs and software techniques are detailed methods that aid in the specification, construction, testing, analysis, management, documentation, and maintenance of other computer programs. Software techniques and tools can be used and reused in a variety of development environments. Their effective use increases engineering productivity and software quality.

The references given in C2 of this Appendix contain information on most of the testing techniques and tools in use today. The set of references is not exhaustive, but provides a comprehensive collection of source material. To keep up to date, the reader is encouraged to obtain information on recent IEEE tutorials and recent documents in the Special Publications series of the National Bureau of Standards.[5] Current information on test tools can be obtained from the Federal Software Testing Center[6] and software tool data bases are accessible through the Data & Analysis Center for Software.[7]

A set of general references on software testing is listed in Appendix D.

C2. References

BEIZER, BORIS. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1983. This book presents a collection of experience-based test techniques. It describes several test design techniques together with their mathematical foundations. The book describes various techniques (decision tables and formal grammars) that provide a precise specification of the input and software. It also discusses a data-base-driven testing technique. Many techniques are based on the author's first-hand experience as director of testing and quality assurance for a telecommunications software producer. The inclusion of experiences and anecdotes makes this book enjoyable and informative.

HOUGHTON, Jr, RAYMOND C. *Software*

---

[5] The NBS publications and software tools survey may be obtained from Superintendent of Documents, US Government Printing Office, Washington, DC 20402.

[6] Information regarding test tools may be obtained by contacting Federal Software Testing Center, Office of Software Development, General Services Administration, 5203 Leesburg Pike, Suite 1100, Falls Church, VA 22041.

[7] Information regarding the tools data base may be obtained from Data & Analysis Center for Software (DACS), RADC/ISISI, Griffiss AFB NY 13441.

Development Tools: A Profile. IEEE Computer vol. 16, no 5, May 1983.[8] The Institute of Computer Science and Technology of the National Bureau of Standards studied the software tools available in the early 1980's. This article reports the results of that study and analyzes the information obtained. Various categorizations of the tools are presented, with tools listed by their characteristics. The lists incorporate percentage summaries based on the total number of tools for which information was available.

OSD / DDT & E Software Test and Evaluation Project, Phases land II, Final Report, vol. 2, *Software Test and Evaluation: State-of-the-Art Overview*. School of Information and Computer Science, Georgia Institute of Technology, June 1983, 350 pp.[9] This report contains a concise overview of most current testing techniques and tools. A set of references is provided for each one. A set of test tool data sheets containing implementation details and information contacts is also provided.

POWELL, PATRICIA B. (ed). *Software Validation, Verification, and Testing Technique and Tool Reference Guide*. National Bureau of Standards Special Publication 500-93, 1982. Order from GPO SN-003-003-02422-8.[5] Thirty techniques and tools for validation, verification, and testing are described. Each description includes the basic features of the technique or tool, its input, its output, and an example. Each description also contains an assessment of effectiveness and usability, applicability, an estimate of the learning time and training, an estimate of needed resources, and associated references.

PRESSON, EDWARD. *Software Test Handbook: Software Test Guidebook*. Rome Air Development Center RADC-TR-84-53, vol. 2 (of two) March 1984. Order from NTIS A147-289. This guidebook contains guidelines and methodology for software testing including summary descriptions of testing techniques, typical paragraphs specifying testing techniques for a Statement of Work, a cross-reference to government and commercial catalogs listing

automated test tools, and an extensive bibliography.

REIFER, DONALD J. *Software Quality Assurance Tools and Techniques*. John D. Cooper and Matthew J. Fisher (eds). Software Quality Management, New York: Petrocelli Books, 1979, pp. 209-234. This paper explains how modern tools and techniques support an assurance technology for computer programs. The author first develops categories for quality assurance tools and techniques (aids) and discusses example aids. Material on toolsmithing is presented next. Finally, an assessment is made of the state of the technology and recommendations for improving current practice are offered.

SOFTFAIR 83. *A Conference on Software Development Tools, Techniques, and Alternatives*. IEEE Computer Society Press, 1983. This is the proceedings of the first of what is likely to be a series of conferences aimed at showing the most promising approaches within the field of software tools and environments. It is a collection of 42 papers covering abroad range of software engineering tools from research prototypes to commercial products.

Software Aids and Tools Survey. Federal Software Management Support Center, Office of Software Development, Report OIT /FSMC-86/ 002, 1985.[6] The purpose of this document is to support management in various government agencies in the identification and selection of software tools. The document identifies and categorizes tools available in the marketplace in mid 1985. Approximately 300 tools are presented with various data concerning each one's function, producer, source language, possible uses, cost, and product description. The survey is expected to be updated periodically.

---

[8] Information regarding IEEE Computer Society publications may be obtained from IEEE Computer Society Order Department, PO Box 80452, Worldway Postal Center, Los Angeles, CA 90080.

[9] The Georgia Technology report may be obtained from Documents Librarian, Software Test and Evaluation Project, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332.

Appendix D

General References

This section identifies a basic set of reference works on software testing. While the set is not exhaustive, it provides a comprehensive collection of source material. Additional references focusing specifically on testing techniques and tools are contained in Appendix C.

CHANDRASEKARAN, B. and RADICCHI, S., (ed) *Computer Program Testing*, North-Holland, 1981. The following description is from the editors Preface:

> "The articles in this volume, taken as a whole, provide a comprehensive, tutorial discussion of the current state of the art as well as research directions in the area of testing computer programs. They cover the spectrum from basic theoretical notions through practical issues in testing programs and large software systems to integrated environments and tools for performing a variety of tests. They are all written by active researchers and practitioners in the field."

DEUTSCH, MICHAEL S. *Software Verification and Validation*. ENGLEWOOD CLIFFS: Prentice-Hall, 1982. The following description is taken from the Preface.

> "The main thrust of this book is to describe verification and validation approaches that have been used successfully on contemporary large-scale software projects. Methodologies are explored that can be pragmatically applied to modern complex software developments and that take account of cost, schedule, and management realities in the actual production environment. This book is intended to be tutorial in nature with a 'This is how it's done in the real world' orientation. Contributing to this theme will be observations and recounts from actual software development project experiences in industry."

Guideline for Lifecycle Validation, Verification, and Testing of Computer Software. Federal Information Processing Standards (FIPS) Publication 101.[10] Order from NTIS FIPSPUB101 1983 (See Appendix C). This guideline presents an integrated approach to validation, verification, and testing that should be used throughout the software lifecycle. Also included is a glossary of technical terms and a list of supporting ICST publications. An Appendix provides an outline for formulating a VV & T plan.

HETZEL, WILLIAM, *The Complete Guide to Software Testing*. QED Information Sciences, 1984. This book covers many aspects of software verification and validation with a primary emphasis on testing. It contains an overview of test methods and tools including sample reports from several commercially available tools. The book is especially useful when used for viewing testing from a management perspective and discussing many of the associated management issues. An extensive bibliography is included.

McCABE, THOMAS J. (ed). *Structured Testing*. IEEE Computer Society Press, Cat no EHO 200-6, 1983.[8] This IEEE Tutorial is a collection of papers focusing on the relationship between testing and program complexity. The first two papers define cyclomatic complexity and describe an associated technique for developing program test cases. The third paper describes a systematic approach to the development of system test cases. The fourth paper provides general guidelines for program verification and testing. The balance of the papers deal with complexity and reliability.

MILLER, EDWARD and HOWDEN, WILLIAM E. (ed). *Tutorial: Software Testing & Validation Techniques* (2nd ed) IEEE Computer Society Press, Cat no EHO 180-0, 1981.[8] This IEEE Tutorial is a collection of some significant papers dealing with various aspects of software testing. These aspects include theoretical foundations, static analysis, dynamic analysis, effectiveness assessment, and software management. An extensive bibliography is included.

MYERS, GLENFORD J. *The Art of Software Testing*. New York: Wiley-Interscience, 1979. This book contains practical, *How To Do It* technical information on software testing. The main emphasis is on methodologies for the design of effective test cases. It also covers psychological and economic issues, managerial aspects of testing, test tools, debugging, and code inspections. Comprehensive examples and checklists support the presentation.

---

[10] The FIPS VV & T Guideline may be obtained from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

POWELL, PATRICIA B. (ed). *Plan for Software Validation, Verification, and Testing*. National Bureau of Standards Special Publication 500-98, 1982.[5] Order from GPO SN-003-003-02449-0 (See Appendix C). This document is for those who direct and those who implement computer projects. It explains the selection and use of validation, verification, and testing (VV & T) tools and techniques. It explains how to develop a plan to meet specific software VV & T goals.

Acknowledgment