***An American National Standard***

**IEEE Standard for Software Unit**

**Testing**

Conteúdo

[1. Scope and References 4](#_Toc530401293)

[1.1. Inside the Scope 4](#_Toc530401294)

[1.2. Outside the Scope 4](#_Toc530401295)

[1.3. References 5](#_Toc530401296)

[2. Definitions 5](#_Toc530401297)

[3. Unit Testing Activities 6](#_Toc530401298)

[3.1. Plan the General Approach, Resources, and Schedule. 7](#_Toc530401299)

[3.1.1. Plan Inputs 7](#_Toc530401300)

[3.1.2. Plan Tasks 7](#_Toc530401301)

[3.1.3. Plan Outputs 8](#_Toc530401302)

[3.2. Determine Features To Be Tested 8](#_Toc530401303)

[3.2.1. Determine Inputs 8](#_Toc530401304)

[3.2.3. Determine Tasks 8](#_Toc530401305)

[3.2.3. Determine Outputs 9](#_Toc530401306)

[3.3. Refine the General Plan 9](#_Toc530401307)

[3.3.1. Refine Inputs 9](#_Toc530401308)

[3.3.2. Refine Tasks 9](#_Toc530401309)

[3.3.3. Refine Outputs 10](#_Toc530401310)

[3.4. Design the Set of Tests 10](#_Toc530401311)

[3.4.1. Design Inputs 10](#_Toc530401312)

[3.4.2. Design Tasks 10](#_Toc530401313)

[3.4.3. Design Outputs 11](#_Toc530401314)

[3.5. Implement the Refined Plan and Design 11](#_Toc530401315)

[3.5.1. Implement Inputs 11](#_Toc530401316)

[3.5.2. Implement Tasks 11](#_Toc530401317)

[3.5.3. Implement Outputs 12](#_Toc530401318)

[3.6. Execute the Test Procedures 12](#_Toc530401319)

[3.6.1. Execute Inputs 12](#_Toc530401320)

[3.6.2. Execute Tasks 12](#_Toc530401321)

[3.6.3. Execute Outputs 13](#_Toc530401322)

[3.7. Check for Termination 13](#_Toc530401323)

[3.7.1. Check Inputs 13](#_Toc530401324)

[3.7.2. Check Tasks 13](#_Toc530401325)

[3.7.3. Check Outputs 14](#_Toc530401326)

[3.8. Evaluate the Test Effort and Unit 14](#_Toc530401327)

[3.8.1. Evaluate Inputs 14](#_Toc530401328)

[3.8.2. Evaluate Tasks 14](#_Toc530401329)

[3.8.3. Evaluate Outputs 15](#_Toc530401330)

[Appendix A 15](#_Toc530401331)

[Appendix B 18](#_Toc530401332)

[Appendix C 20](#_Toc530401333)

[Appendix D 23](#_Toc530401334)

[Acknowledgment 24](#_Toc530401335)

# Scope and References

## 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]. 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.

## 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.

## 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] ANSI/IEEE Std 829-1983, IEEE Standard for Software Test Documentation.

# 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, arrivalrates, 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 parametersof 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 ofoperations 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 withexisting states.

**test objective:** An identified set of software features to be measured under specified conditions by comparingactual 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 hierarchicdecomposition of the test objectives.

**test unit:** 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

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.

# 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 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.

## Plan the General Approach, Resources, and Schedule.

**Recursos Utilizados:** IntelliJ, Google Sheets, Word;

**Calendário de Atividades:**

**12/11/2018** – Início do planeamento de atividades, avaliação dos recursos necessários, criação do repositório no GitHub e configuração do Gradle;

**14/11/2018** – Distribuição de tarefas;

**16/11/2018** – Elaboração de tabelas ECP;

**17/11/2018** – Finalização de tabelas ECP, elaboração de tabelas BVA e Test Cases;

**18/11/2018** – Finalização de tabelas BVA e Test Cases, início dos testes de caixa preta e início da elaboração do relatório;

**19/11/2018** – Finalização dos testes de caixa preta e finalização do relatório (#1 Milestone);

### 3.1.1. Plan Inputs

1) Project plans

2) Software requirements documentation (Documentos de Requisitos)

## 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 Þles, 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.

1. *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 na 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.

1. *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 notiÞcation procedures that apply.

1. *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 Þles, 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 signiÞcant 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.

1. *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 requestsÑif produced from 3.1.2 (4)

* 1. Determine Features To Be Tested

3.2.1. Determine Inputs

**SRS? Nada a declarar**

(1) Unit requirements documentation

(2) Software architectural design documentation if needed

3.2.3. Determine Tasks

1. *Study the Functional Requirements.*

**getBicycle()** – Método que permite alugar uma bicicleta a um utilizador;

**returnBikycle()** – Método que permite devolver uma bicicleta alugada por um utilizador;

**bicycleRentalFee()** – Método que calcula a quantia a pagar pelo aluguer de uma bicicleta;

**addCredit()** – Método que permite adicionar fundos à carteira de um utilizador;

**registerUser()** – Método que permite registar um utilizador;

**verifycredit()** – Método que verifica se um utilizador tem os fundos mínimos suficientes para alugar uma bicicleta;

1. *Identify Additional Requirements and Associated Procedures. (Nada a declarar)*

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.

1. *Identify States of the Unit. (Diagrama de estados?)*

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.

1. *Identify Input and Output Data Characteristics and Select Elements to be Included in the Testing*

|  |  |  |
| --- | --- | --- |
| **EQUIVALENCE CLASS PARTITIONING getBicycle()** | | |
| **REQUIREMENTS** | **VALID CLASS** | **INVALID CLASS** |
| **Nº INPUTS** | 3 | !=3 |
| **INPUT TYPES** | IDDeposit: Inteiro | IDDeposit: != Inteiro |
| IDUser: Inteiro | IDUser: != Inteiro |
| starttime: Inteiro | starttime: != Inteiro |
| **SPECIFIC X VALUE** | IDDeposit: >=0 | IDDeposit: <0 |
| IDUser: >0 | IDUser: <=0 |
| starttime: >=0 | starttime: <0 |

|  |  |  |
| --- | --- | --- |
| **EQUIVALENCE CLASS PARTITIONING returnBikycle()** | | |
| **REQUIREMENTS** | **VALID CLASS** | **INVALID CLASS** |
| **Nº INPUTS** | 3 | !=3 |
| **INPUT TYPES** | IDDeposit: Inteiro | IDDeposit: != Inteiro |
| IDUser: Inteiro | IDUser: != Inteiro |
| endtime: Inteiro | endtime: != Inteiro |
| **SPECIFIC X VALUE** | IDDeposit: >=0 | IDDeposit: <0 |
| IDUser: >0 | IDUser: <=0 |
| endtime: >=0 | endtime: <0 |

|  |  |  |
| --- | --- | --- |
| **EQUIVALENCE CLASS PARTITIONING bicycleRentalFee()** | | |
| **REQUIREMENTS** | **VALID CLASS** | **INVALID CLASS** |
| **Nº INPUTS** | 4 | !=4 |
| **INPUT TYPES** | rentalProgram: Inteiro | rentalProgram: !=Inteiro |
| initTime: Inteiro | initTime: !=Inteiro |
| endTime: Inteiro | endTime: !=Inteiro |
| nRentals: Inteiro | nRentals: !=Inteiro |
| **SPECIFIC X VALUE** | rentalProgram: (1 ou 2) | rentalProgram: != (1 ou 2) |
| initTime: >=0 | initTime: <0 |
| endTime: >=0 | endTime: <0 |
| nRentals: >=0 | nRentals: <0 |

|  |  |  |
| --- | --- | --- |
| **EQUIVALENCE CLASS PARTITIONING addCredit()** | | |
| **REQUIREMENTS** | **VALID CLASS** | **INVALID CLASS** |
| **Nº INPUTS** | 2 | !=2 |
| **INPUT TYPES** | IDUser: Inteiro | IDUser: != Inteiro |
| amount: Inteiro | amount: != Inteiro |
| **SPECIFIC X VALUE** | IDUser: >=0 | IDUser: <0 |
| amount: >0 | amount: <1 |

|  |  |  |
| --- | --- | --- |
| **EQUIVALENCE CLASS PARTITIONING registerUser()** | | |
| **REQUIREMENTS** | **VALID CLASS** | **INVALID CLASS** |
| **Nº INPUTS** | 3 | !=3 |
| **INPUT TYPES** | IDUser: Inteiro | IDUser: != Inteiro |
| name: String | name: != String |
| rentalProgram: Inteiro | rentalProgram: != Inteiro |
| **SPECIFIC X VALUE** | IDUser: >=0 | IDUser: <0 |
| name: "teste" | name: null |
| rentalProgram: 1 ou 2 | 4 |

|  |  |  |
| --- | --- | --- |
| **EQUIVALENCE CLASS PARTITIONING verifyCredit()** | | |
| **REQUIREMENTS** | **VALID CLASS** | **INVALID CLASS** |
| **Nº INPUTS** | 1 | !=1 |
| **INPUT TYPES** | IDUser: Inteiro | IDUser: != Inteiro |
| **SPECIFIC X VALUE** | IDUser: >=0 | IDUser: <0 |

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

## 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.

1. *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 testdesign specification.

1. *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 requestsÑif produced from 3.3.2 (2).

## 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 identiÞers in the *Test IdentiÞcation* 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 modiÞcation. 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 speciÞcation.

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

Record the speciÞcations directly or by reference in either a supplementary section of the unit’s test

design speciÞcation 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 SpeciÞcations 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

difÞculties that might arise, for example, from a need to trace control ßow 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 test-ability of the unit.

Based on information in the unit’s design, specify any newly identiÞed test cases and complete any

partial test case speciÞcations in accordance with 3.4.2 (3).

(5) *Complete the Test Design SpeciÞcation.* Complete the test design speciÞcation for the unit in accordance

with ANSI/IEEE Std 829-1983 [2].

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

8

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

### 3.4.3. Design Outputs

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

(2) Separate test procedure specificationsÑif produced from 3.4.2 (2)

(3) Separate test-case specicationsÑif produced from 3.4.2 (3) or (4)

(4) Unit design enhancement requestsÑif produced from 3.4.2 (4)

## Implement the Refined Plan and Design

### 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 speciÞcations in the unit test design speciÞcation 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 activitiesÑif available

(7) Test tools from previous testing activitiesÑif available

* + 1. Implement Tasks

(1) *Obtain and Verify Test Data.* Obtain a copy of existing test data to be modiÞed or used without modi

Þcation. 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 speciÞcations. 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 unitÕs test design speciÞcation.

(2) *Obtain Special Resources.* Obtain the test support resources speciÞed 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 identiÞed 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 identiÞer of each item in the *Summary* section of the unitÕs test summary report.

* + 1. Implement Outputs

(1) VeriÞed test data (from 3.5.2 (1))

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

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

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

* 1. Execute the Test Procedures
     1. Execute Inputs

(1) VeriÞed test data (from 3.5.2 (1))

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

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

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

(5) Test procedure speciÞcations (from 3.4.2 (2))Ñif produced

(6) Failure analysis results (from debugging process)Ñif produced

* + 1. 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 unitÕs test summary report.

(2) *Determine Results.* For each test case, determine if the unit passed or failed based on required result

speciÞcations in the case descriptions. Record pass or fail results in the *Summary of Results* section

of the unitÕs 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.

NOTEÑThe 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.

*Case 1: A Fault in a Test SpeciÞcation 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

speciÞcation 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).

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

10

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

**Fig 2 Ñ Control Flow Within the Execute Activity**

**Fig 3 Ñ Control Flow Within the Check Activity**

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

11

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

* + 1. 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 speciÞcationsÑif produced from 3.6.2 (2)

(3) Revised test dataÑif produced from 3.6.2 (2)

* 1. Check for Termination
     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 speciÞcations (from 3.4.2 (1) through (3) inclusive)Ñif required

(4) Software data structure descriptionsÑif required

### 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, ßow).

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 satis

Þed (for example, uncorrected major fault, out of time) then ensure that the speciÞc situation causing

termination is documented in the *Summary of Activities* section of the test summary report

together with the unÞnished 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 satisÞed, 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 speci

Þcations in accordance with 3.4.2 (3).

(b) Modify the test procedure speciÞcations 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).

* + 1. 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 speciÞcationsÑif produced from 3.7.2 (3)

(3) Additional test dataÑif produced from 3.7.2 (3)

* 1. Evaluate the Test Effort and Unit
     1. Evaluate Inputs

(1) Unit Test Design SpeciÞcation (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 speciÞcations (from 3.4.2 (3) and (4))Ñif produced

### Evaluate Tasks

(1) *Describe Testing Status.* Record variances from test plans and test speciÞcations in the *Variances*

section of the test summary report. Specify the reason for each variance.

For abnormal termination, identify areas insufÞciently 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 speciÞcation, separate

test-case speciÞcations, separate test procedure speciÞcations, test data, test data generation procedures,

test drivers and stubs, and the test summary report.

* + 1. Evaluate Outputs

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

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

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

13

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

# Appendix A

**(informative)**

**Implementation and Usage Guidelines**

This section contains information intended to be of beneÞt 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 conÞrm 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 speciÞed for each project.

Factors, such as unit criticality, auditing needs, or contract speciÞcations 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 speciÞc, they should appear in the project plan,

quality assurance plan, veriÞcation and validation plan, or overall test plan.

**A3.** Additional Test Documentation

The information contained in the test design speciÞcation 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 satis

Þed by the set of test documents speciÞed 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 identiÞed requirements at the end of Determine

3) Approval of speciÞc plans at the end of ReÞne

4) Approval of test speciÞcations 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

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

14

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

**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 sufÞcient to supply

all required audit information.

**A6.** Configuration Management

ConÞguration management should be the source of the software requirements, software architectual design,

software data structure, and unit requirements documentation. These inputs must be managed to ensure con-

Þdence that we have current information and will be notiÞed of any changes.

The Þnal unit testing products should be provided to conÞguration management. These outputs must be

managed to permit thorough and economical regression testing. See ANSI/IEEE Std 828-1983, IEEE Standard

for Software ConÞguration Management Plans, for details.

**A7.** Determination of Requirements-Based Characteristics

Psychological factors (for example, self-conÞdence, a detailed knowledge of the unit design) can make it

very difÞcult 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 otherÕs 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 justiÞed.

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 unitÕs 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 speciÞed 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).

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

15

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

**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-ßow information. The report identiÞes 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 reßect 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 deÞnition of good software engineering practice. Some organizations

use practices similar to the process speciÞed 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?

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

16

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

# Appendix B

(informative)

**Concepts and Assumptions**

**B1.** Software Engineering Concepts

The standard unit testing process speciÞed 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 veriÞcation and validation activities. Other activities include

technical reviews (for example, code inspections), static analysis, and proof of correctness. SpeciÞcation of

a comprehensive veriÞcation 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 speciÞcations 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 Þrst 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 identiÞed faults while avoiding the introduction of new ones. SpeciÞcation 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.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

17

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

**B1.6** Need for Design and Implementation Information

Often, requirements information is not enough for effective testing, even though, fundamentally, testing

measures actual behavior against required behavior. This is because its usually not feasible to test all possible

situations and requirements often do not provide sufÞcient guidance in identifying situations that have

high failure potential. Design and implementation information often are needed, since some of these highpotential

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

**B1.7** Incremental SpeciÞcation 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 Þnally 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 speciÞcation occurs in three

increments. The Þrst group is speciÞed during the Determine activity and is based on the unit requirements

documentation. The second group is speciÞed 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 speci-

Þed during the Check activity and is based on the unit’s code.

For nonprocedural language (for example, report writer or sort speciÞcation languages) implementations,

speciÞcation occurs in two increments. The Þrst is during the Determine activity and is based on requirements

and the second is during Design and is based on the nonprocedural speciÞcation.

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 speciÞcation is generated during the Determine, ReÞne, and Design

activities. As each of these test activities progress, information is recorded in appropriate sections of the

speciÞcation. The whole document must be complete at the end of the Þnal 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 speciÞed in this standard is based on a variety of economic, psychological, and

technical assumptions. The signiÞcant 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:

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

18

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

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 veriÞable

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 signiÞcant economic beneÞts 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 veriÞcation 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 speciÞed in project

plans, veriÞcation 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 identiÞed 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 sourcecode

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 speciÞed.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

19

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

# Appendix C

(informative)

**Sources for Techniques and Tools**

**C1.** General

Software tools are computer programs and software techniques are detailed methods that aid in the speciÞ-

cation, 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 Center6 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 speciÞcation of the input and software. It also discusses a data-base-driven

testing technique. Many techniques are based on the authorÕs Þrst-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 Development Tools: A ProÞle. *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 1980Õs. 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.

5The NBS publications and software tools survey may be obtained from Superintendent of Documents, US Government Printing

OfÞce, Washington, DC 20402.

6Information regarding test tools may be obtained by contacting Federal Software Testing Center, OfÞce of Software Development,

General Services Administration, 5203 Leesburg Pike, Suite 1100, Falls Church, VA 22041.

7Information regarding the tools data base may be obtained from Data & Analysis Center for Software (DACS), RADC/ISISI, GrifÞss

AFB NY 13441.

8Information regarding IEEE Computer Society publications may be obtained from IEEE Computer Society Order Department, PO

Box 80452, Worldway Postal Center, Los Angeles, CA 90080.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

20

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

OSD/DDT & E Software Test and Evaluation Project, Phases I and 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, VeriÞcation, and Testing Technique and Tool Reference*

*Guide.* National Bureau of Standards Special Publication 500-93, 1982. Order from GPO SN-003-003-

02422-8. Thirty techniques and tools for validation, veriÞcation, 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 DevelopmentCenter

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. 209Ð234. This paper

explains how modern tools and techniques support an assurance technology for computer programs. The

author Þrst 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 Þrst of what is likely to be a series of conferences

aimed at showing the most promising approaches within the Þeld of software tools and environments. It is a

collection of 42 papers covering a broad range of software engineering tools from research prototypes to

commercial products.

*Software Aids and Tools Survey.* Federal Software Management Support Center, OfÞce of Software Development,

Report OIT/FSMC-86/002, 1985. The purpose of this document is to support management in various

government agencies in the identiÞcation and selection of software tools. The document identiÞes 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.

9The 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.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

21

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

# Appendix D

**(informative)**

**General References**

This section identiÞes 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 speciÞcally 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:

ÒThe 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 Þeld.Ó

DEUTSCH, MICHAEL S. *Software VeriÞcation and Validation.* ENGLEWOOD CLIFFS: Prentice-Hall,

1982. The following description is taken from the Preface.

ÒThe main thrust of this book is to describe veriÞcation 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 ÒThis is how itÕs done in the real worldÓ orientation. Contributing to this theme will be observations

and recounts from actual software development project experiences in industry.Ó

*Guideline for Lifecycle Validation, VeriÞcation, and Testing of Computer Software.* Federal Information

Processing Standards TIPS) Publication 101.10 Order from NTIS FIPSPUB101 1983 (See Appendix C).

This guideline presents an integrated approach to validation, veriÞcation, 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 veriÞcation 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 Þrst two papers deÞne 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 veriÞcation and testing. The balance of the

papers deal with complexity and reliability.

10The FIPS VV & T Guideline may be obtained from National Technical Information Service, 5285 Port Royal Road, SpringÞeld, VA

22161.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

22

ANSI/IEEE

Std 1008-1987 IEEE STANDARD FOR

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 signiÞcant 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.

POWELL, PATRICIA B. (ed). *Plan for Software Validation, VeriÞcation, 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, veriÞcation, and testing (VV & T) tools and techniques. It explains how to develop a

plan to meet speciÞc software VV & T goals.

# Acknowledgment

Appreciation is expressed to the following companies and organizations for contributing the time of their

employees to make possible the development of this text:

Algoma Steel

Applied Information Development

AT & T Bell Labs

AT & T Information Systems

Automated Language Processing Systems

Bank of America

Bechtel Power

Bell Canada

Boeing Computer Services

Boston University

Burroughs, Scotland

CAP GEMINI DASD

Central Institute for Industrial Research, Norway

Communications Sciences

Conoco

Digital Equipment Corp

US Department of the Interior

US Department of Transportation

Data Systems Analysts

E-Systems

K.A. Foster, Inc

General Dynamics

Georgia Tech

General Services Administration

Honeywell

Hughes Aircraft

IBM

IBM Federal Systems Division

International Bureau of Software Test

Johns Hopkins University Applied Physics Laboratory

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on October 17,2012 at 09:50:33 UTC from IEEE Xplore. Restrictions apply.

23

ANSI/IEEE

SOFTWARE UNIT TESTING Std 1008-1987

Lear Siegler

Logicon

Management and Computer Services

Martin Marietta Aerospace

McDonald-Douglas

Medtronic

MichaelSoft Inc.

Micom

Mitre

M. T. ScientiÞc Consulting

NASA

National Bureau of Standards

NCR

Product Assurances Consulting

Professional Systems & Technology

Programming Environments

Quality Assurance Institute

RCA

Reynolds & Reynolds

Rolm Telecommunications

Rome Air Development Center

Sallie Mae

Seattle’s First National Bank

SHAPE, BELGIUM

Software Engineering Service, Germany

Software Quality Engineering

Software Research Associates

Solo Systems

Sperry

SQS GmbH, Germany

Tandem Computers

Tektronix

Televideo

Tenn Valley Authority

Texas Instruments

Time

University of DC

University of Texas, Arlington

US Army Computer Systems Command

Warner Robins ALC

Westinghouse Hanford