CHAPTER 5

SOFTWARE TESTING

Antonia Bertolino
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Research Area of S. Cataldo
56100 PISA (Italy)
bertolino@iei.pi.cnr.it

Table of Contents

1 INTRODUCTION

Testing is an important, mandatory part of software development; it is a technique for evaluating product quality and also for indirectly improving it, by identifying defects and problems.

As more extensively discussed in the *Software Quality* chapter of the Guide to the SWEBOK, the right attitude towards quality is one of prevention: it is obviously much better to avoid problems, rather than repairing them. Testing must be seen as a means primarily for checking whether the prevention has been effective, but also for identifying anomalies in those cases in which, for some reason, it has been not. It is perhaps obvious, but worth recognizing, that even after successfully completing an extensive testing campaign, the software could still contain faults; nor is defect free code a synonymous for quality product. The remedy to system failures that are experienced after delivery is provided by (corrective) maintenance actions. Maintenance topics are covered into the *Software Maintenance* chapter of the Guide to the SWEBOK.

In the years, the view of Software Testing has evolved towards a more constructive attitude. Testing is no longer seen as an activity that starts only after the coding phase is complete, with the limited purpose of detecting failures. Software testing is nowadays seen as an activity that should encompass the whole development process, and is an important part itself of the actual product construction. Indeed, planning for testing should start since the early stages of requirement analysis, and test plans and procedures must be systematically and continuously refined as the development proceeds. These activities of planning and designing tests constitute themselves a useful input to designers for highlighting potential weaknesses (like, e.g., design oversights or contradictions, and omissions or ambiguities in the documentation).

In the already referred *Software Quality* (SQ) chapter of the Guide to the SWEBOK, activities and techniques for quality analysis are categorized into*: static techniques* (no code execution), and *dynamic techniques* (code execution). Both categories are useful. Although this chapter focuses on testing, that is dynamic (see Sect. 2), static techniques are as important for the purposes of evaluating product quality and finding defects. Static techniques are covered into the SQ Knowledge Area description.

2 DEFINITION OF THE SOFTWARE TESTING KNOWLEDGE AREA

Software testing consists of the **dynamic** verification of the behavior of a program on a **finite** set of test cases, suitably **selected** from the usually infinite executions domain, against the specified **expected** behavior.

In the above definition, and in the following as well, underlined words correspond to key issues in identifying the Knowledge Area of Software Testing. In particular:

- **dynamic:** this term means *testing always implies executing the program on (valued) inputs*. To be precise, the input value alone is not always sufficient to determine a test, as a complex, non deterministic system might react with different behaviors to a same input, depending on the system state. In the following, though, the term "input" will be maintained, with the implied convention that it also includes a specified input state, in those cases in which it is needed. Different from testing, and complementary with it, are static analysis techniques, such as peer review and inspection (that sometimes are improperly referred to as "static testing"); these are not considered as part of this Knowledge Area (nor is program execution on symbolic inputs, or symbolic evaluation);
- **finite:** for even simple programs, so many test cases are theoretically possible that exhaustive testing could

require even years to execute. This is why in practice the whole test set can generally be considered infinite. But, the number of executions which can realistically be observed in testing must obviously be finite. Clearly, "enough" testing should be performed to provide reasonable assurance. Indeed, testing always implies a *trade-off* between limited resources and schedules, and inherently unlimited test requirements: this conflict points to well known problems of testing, both technical in nature (criteria for deciding test adequacy) and managerial in nature (estimating the effort to put in testing);

- **selected:** the many proposed *test techniques* essentially differ in how they select the (finite) test set, and testers must be aware that different selection criteria may yield largely different effectiveness. How to identify the most suitable selection criterion under given conditions is a very complex problem; in practice risk analysis techniques and test engineering expertise are applied;
- **expected:** it must be possible (although not always easy) to decide whether the observed outcomes of program execution are acceptable or not, otherwise the testing effort would be useless. The observed behavior may be checked against user's expectations (commonly referred to as testing for validation) or against a specification (testing for verification). The test pass/fail decision is commonly referred in the testing literature to as the *oracle problem*, which can be addressed with different approaches, for instance by human inspection of results or by comparison with an existing reference system. In some situations, the expected behavior may only be partially specified, i.e., only some parts of the actual behavior need to be checked against some stated assertion.

2.1 Conceptual Structure of the Breakdown

Software testing is usually performed at **different levels** along the development process. That is to say, the **target of the test** can vary: a whole system, parts of it (related by purpose, use, behavior, or structure), a single module.

The testing is conducted in view of a specific purpose (**test objective**), which is stated more or less explicitly, and with varying degrees of precision. Stating the objective in precise, quantitative terms allows for establishing control over the test process.

One of testing aims is to expose failures (as many as possible), and many popular **test techniques** have been developed for this objective. These techniques variously attempt to "break" the program, by running one [or more] test[s] drawn from identified classes of (deemed equivalent) executions. The leading principle underlying such techniques is being as much systematic as possible in identifying a representative set of program behaviors (generally in the form of subclasses of the input domain). However, a comprehensive view of the Knowledge Area of Software Testing as a means for quality must include other as important objectives for testing, e.g., reliability measurement, usability evaluation, contractor's acceptance, for which different approaches would be taken. Note that the test objective varies with the test target, i.e., in general *different purposes are addressed at the different levels of testing*.

The test target and test objective together determine how the test set is identified; both with regard to its consistency -*how much testing is enough for achieving the stated objective?*- and its composition -*which test cases should be selected for achieving the stated objective?*- (although usually the "*for achieving the stated objective*" part is left implicit and only the first part of the two italicized questions above is posed). Criteria for addressing the first question are referred to as *test adequacy criteria*, while for the second as *test selection criteria*.

Sometimes, it can happen that confusion is made between test objectives and techniques. Test techniques are to be viewed as aids that help to ensure the achievement of test objectives. For instance, branch coverage is a popular test technique. Achieving a specified branch coverage measure should not be considered *per se* as the objective of testing: it is a means to improve the chances of finding failures (by systematically exercising every program branch out of a decision point). To avoid such misunderstandings, a clear distinction should be made between **test measures** which evaluate the thoroughness of the test set, like measures of coverage, and those which instead provide an evaluation of the program under test, based on the observed test outputs, like reliability.

Testing concepts, strategies, techniques and measures need to be integrated into a defined and controlled process, which is run by people. The **test process** supports testing activities and provide guidance to testing teams, from test planning to test outputs evaluation, in such a way as to provide justified assurance that the test objectives are met cost-effectively.

Software testing is a very expensive and labor-intensive part of development. For this reason, **tools** are instrumental for automated test execution, test results logging and evaluation, and in general to support test activities. Moreover, in order to enhance cost-effectiveness ratio, a key issue has always been pushing test automation as much as possible.

2.2 Overview

Following the above-presented conceptual scheme, the Software Testing Knowledge Area description is organized as follows.

Part A deals with *Testing Basic Concepts and Definitions*. It covers the basic definitions within the Software Testing field, as well as an introduction to the terminology. In the same part, the scope of the Knowledge Area is laid down, also in relation with other activities.

Part B deals with *Test Levels*. It consists of two (orthogonal) subsections: B.1 lists the levels in which the testing of large software systems is traditionally subdivided. In B.2 testing for specific conditions or properties is instead considered, and is referred to as "Objectives of testing". Clearly not all types of testing apply to every system, nor has every possible type been listed, but those most generally applied.

As said, several *Test Techniques* have been developed in the last two decades according to various criteria, and new ones are still proposed. "Generally accepted" techniques are covered in Part C.

Test-related Measures are dealt in Part D.

Finally, issues relative to *Managing the Test Process* are covered in Part E.

Existing tools and concepts related to supporting and automating the activities into the test process are not addressed here. They are covered within the Knowledge Area description of *Software Engineering Tools and Methods* in this Guide.

3 BREAKDOWN OF TOPICS FOR THE SOFTWARE TESTING KNOWLEDGE AREA

This section gives the list of topics identified for the Software Testing Knowledge Area, with succinct descriptions and references. Two levels of references are provided with topics: the recommended references within brackets, and additional references within parentheses. In particular, the recommended references for Software Testing have been identified into selected book chapters (for instance, Chapter 1 of reference Be is denoted as Be:c1), or, in some cases, sections (for instance, Section 1.4 of Chapter 1 of Be is denoted as Be:c1s1.4). The Further Readings list includes several refereed journal and conference papers and some relevant standards, for a deeper study of the pointed arguments.

A chart in Figure 1 gives a graphical presentation of the top-level decomposition of the breakdown for the Software Testing Knowledge Area. The finer decomposition of the five level 1 topics into the lowest level entries is then summarised by the following five tables (note that two alternative decompositions are proposed for the level 1 topic of Testing Techniques)

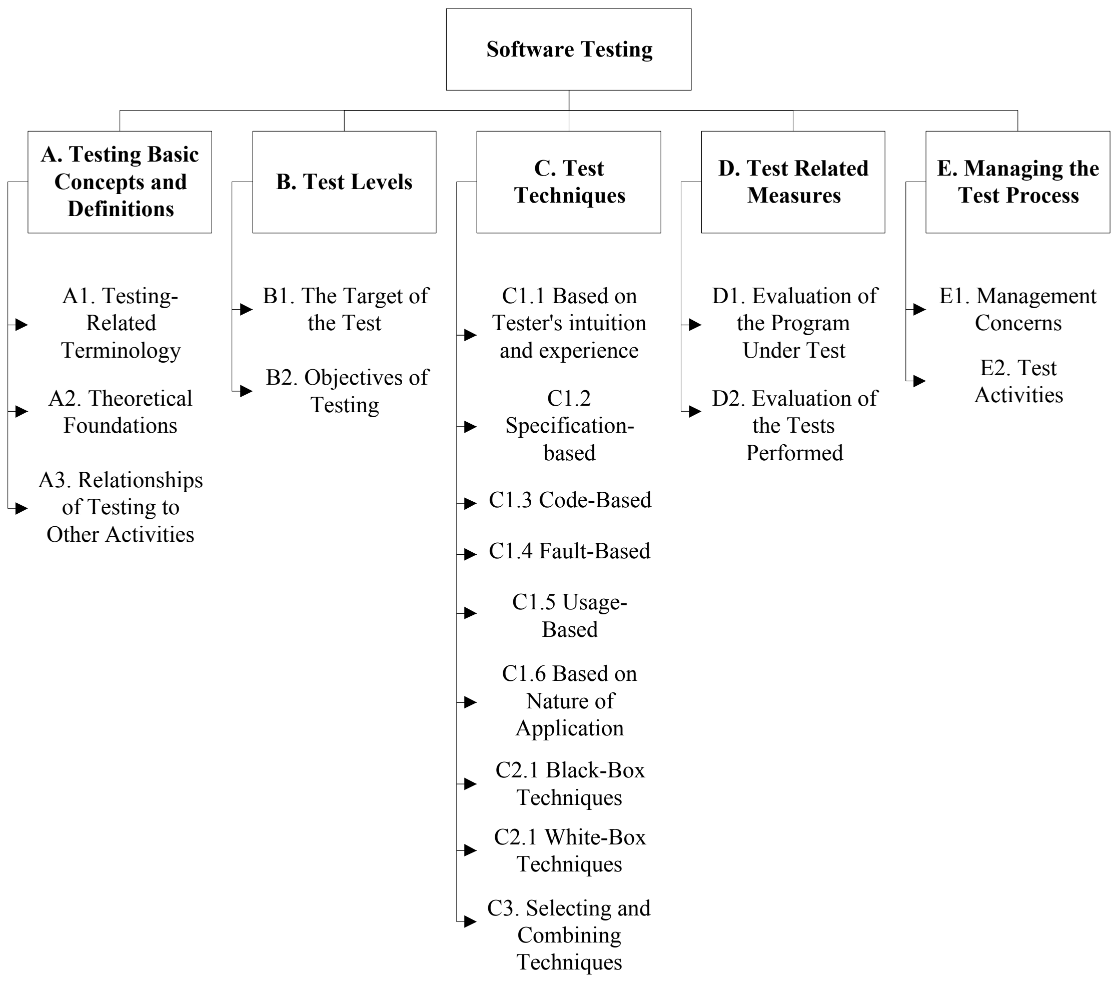
Software Testing
A. Testing Basic Concepts and Definitions
B. Test Levels
C. Test Techniques
D. Test Related Measures
E. Managing the Test Process
A1. Testing-Related Terminology
A2. Theoretical Foundations
A3. Relationships of Testing to Other Activities
B1. The Target of the Test
B2. Objectives of Testing
C1.1 Based on Tester's intuition and experience
C1.2 Specification-based
C1.3 Code-Based
C1.4 Fault-Based
C1.5 Usage-Based
C1.6 Based on Nature of Application
C2.1 Black-Box Techniques
C2.1 White-Box Techniques
C3. Selecting and Combining Techniques
D1. Evaluation of the Program Under Test
D2. Evaluation of the Tests Performed
E1. Management Concerns
E2. Test Activities

Table 1-A: Decomposition for Testing Basic Concepts and Definitions		
A. Testing Basic Concepts and Definitions	*A1. Testing-related terminology*	Definitions of testing and related terminology
		Faults vs. Failures
	A2. Theoretical foundations	Test selection criteria/Test adequacy criteria (or stopping rules)
		Testing effectiveness/Objectives for testing
		Testing for defect removal
		The oracle problem
		Theoretical and practical limitations of testing
		The problem of infeasible paths
		Testability
	A3. Relationships of testing to other activities	Testing vs. Static Analysis Techniques
		Testing vs. Correctness Proofs and Formal Verification
		Testing vs. Debugging
		Testing vs. Programming
		Testing within SQA
		Testing within Cleanroom
		Testing and Certification

Table 1-B: Decomposition for Test Levels		
B. Test Levels	*B1. The target of the test*	Unit testing
		Integration testing
		System testing
	B2. Objectives of testing	Acceptance/qualification testing
		Installation testing
		Alpha and Beta testing
		Conformance testing/ Functional testing/ Correctness testing
		Reliability achievement and evaluation by testing
		Regression testing
		Performance testing
		Stress testing
		Back-to-back testing
		Recovery testing
		Configuration testing
		Usability testing

Table 1-C: Decomposition for Test Techniques

C. Test Techniques	C1: (criterion "base on which tests are generated")	*C1.1 Based on tester's intuition and experience*	Ad hoc
		C1.2 Specification-based	Equivalence partitioning
			Boundary-value analysis
			Decision table
			Finite-state machine-based
			Testing from formal specifications
			Random testing
		C1.3 Code-based	Reference models for code-based testing (flow graph, call graph)
			Control flow-based criteria
			Data flow-based criteria
		C1.4 Fault-based	Error guessing
			Mutation testing
		C1.5 Usage-based	Operational profile
			SRET
		C1.6 Based on nature of application	Object-oriented testing
			Component-based testing
			Web-based testing
			GUI testing
			Testing of concurrent programs
			Protocol conformance testing
			Testing of distributed systems
			Testing of real-time systems
			Testing of scientific software
	C2: (criterion "ignorance or knowledge of implementation")	*C2.1 Black-box techniques*	Equivalence partitioning
			Boundary-value analysis
			Decision table
			Finite-state machine-based
			Testing from formal specifications
			Error guessing
			Random testing
			Operational profile
			SRET
		C2.2 White-box techniques	Reference models for code-based testing (flow graph, call graph)
			Control flow-based criteria
			Data flow-based criteria
			Mutation testing
	C3 Selecting and combining techniques		Functional and structural
			Coverage and operational/Saturation effect

Table 1-D: Decomposition for Test Related Measures

D. Test Related Measures	*D.1 Evaluation of the program under test*	Program measurements to aid in planning and designing testing
		Types, classification and statistics of faults
		Remaining number of defects/Fault density
		Life test, reliability evaluation
		Reliability growth models
	D.2 Evaluation of the tests performed	Coverage/thoroughness measures
		Fault seeding
		Mutation score
		Comparison and relative effectiveness of different techniques

Table 1-E: Decomposition for Managing the Test Process

E. Managing the Test Process	*E.1 Management concerns*	Attitudes/Egoless programming
		Test process
		Test documentation and workproducts
		Internal vs. independent test team
		Cost/effort estimation and other process measures
		Termination
		Test reuse and test patterns
	E.2 Test activities	Planning
		Test case generation
		Test environment development
		Execution
		Test results evaluation
		Problem reporting/Test log
		Defect tracking

A. Testing Basic Concepts and Definitions

A1. Testing-related terminology

- Definitions of testing and related terminology [Be:c1; Jo:c1,2,3,4; Ly:c2s2.2] (610)

A comprehensive introduction to the Knowledge Area of Software Testing is provided by the core references. Moreover, the IEEE Standard Glossary of Software Engineering Terminology (610) defines terms for the whole field of software engineering, including testing-related terms.

- Faults vs. Failures [Ly:c2s2.2; Jo:c1; Pe:c1; Pf:c7] (FH+; Mo; ZH+:s3.5; 610; 982.2:fig3.1.1-1; 982.2:fig6.1-1)

Many terms are used in the software literature to speak of malfunctioning, notably *fault*, *failure*, *error*, and several others. Often these terms are used interchangeably. However, in some cases they are given a more precise meaning (unfortunately, not in consistent ways between different sources), in order to identify the subsequent steps of the cause-effect chain that originates somewhere, e.g., in the head of a designer, and eventually leads to the system's user observing an undesired effect. This terminology is precisely defined in the IEEE Standard 610.12-1990, Standard Glossary of Software Engineering Terminology (610) and is also discussed in more depth in the Software Quality Knowledge Area (Chapter 11, Sect. 7). What is essential to discuss Software Testing, as a minimum, is to clearly distinguish between the *cause* for a malfunctioning, for which either of the terms *fault* or *defect* will be used here, and an undesired effect observed in the system delivered service, that will be called a *failure*. It is important to clarify that testing can reveal failures, but then it is the faults that can and must be removed.

However, it should also be recognized that not always the cause of a failure can be unequivocally identified, i.e., no theoretical criteria exists to uniquely say what the fault was that caused a failure. One may choose to say the fault was what had to be modified to remove the problem, but other modifications could have worked just as well. To avoid ambiguities, some authors instead of faults prefer to speak in terms of *failure-causing inputs* (FH+), i.e., those sets of inputs that when executed cause a failure.

A2. Theoretical foundations

- Test selection criteria/Test adequacy criteria (or stopping rules) [Pf:c7s7.3; ZH+:s1.1] (We-b; WW+; ZH+)

A test criterion is a means of deciding which a suitable set of test cases should be. A criterion can be used for selecting the test cases, or for checking if a selected test suite is adequate, i.e., to decide if the testing can be stopped. In mathematical terminology it would be a decision predicate defined on triples (P, S, T), where P is a program, S is the specification (intended here to mean in general sense any relevant source of information for testing) and T is a test set. Some generally used criteria are mentioned in Part C.

- Testing effectiveness/Objectives for testing [Be:c1s1.4; Pe:c21] (FH+)

Testing amounts at observing a sample of program executions. The selection of the sample can be guided by different objectives: it is only in light of the objective pursued that the effectiveness of the test set can be evaluated. This important issue is discussed at some length in the references provided.

- Testing for defect identification [Be:c1; KF+:c1]

In testing for defect identification a successful test is one that causes the system to fail. This is quite different from testing to demonstrate that the software meets its specification, or other desired properties, whereby testing is successful if no (important) failures are observed.

- The oracle problem [Be:c1] (We-a; BS)

An oracle is any (human or mechanical) agent that decides whether a program behaved correctly on a given test, and produces accordingly a verdict of “pass” or “fail”. There exist many different kinds of oracles; oracle automation can be very difficult and expensive.

- Theoretical and practical limitations of testing [KF+:c2] (Ho)

Testing theory warns against putting a not justified level of confidence on series of passed tests. Unfortunately, most established results of testing theory are negative ones, i.e., they state what testing can never achieve (as opposed to what it actually achieved). The most famous quotation in this regard is Dijkstra aphorism that “program testing can be used to show the *presence* of bugs, but never to show their absence”. The obvious reason is that complete testing is not feasible in real systems. Because of this, testing must be driven based on risk, i.e., testing can also be seen as a risk management strategy.

- The problem of infeasible paths [Be:c3]

Infeasible paths, i.e., control flow paths which cannot be exercised by any input data, are a significant problem in path-oriented testing, and particularly in the automated derivation of test inputs for code-based testing techniques.

- Testability [Be:c3,c13] (BM; BS; VM)

The term of software testability has been recently introduced in the literature with two related, but different meanings: on the one hand as the degree to which it is easy for a system to fulfill a given test coverage criterion, as in (BM); on the other hand, as the likelihood (possibly measured statistically) that the system exposes a failure under testing, *if* it is faulty, as in (VM, BS). Both meanings are important.

A3. Relationships of testing to other activities

Here the relation between the Software Testing and other related activities of software engineering is considered. Software Testing is related to, but different from, static analysis techniques, proofs of correctness, debugging and programming. On the other side, it is informative to consider testing from the point of view of software quality analysts, users of CMM and Cleanroom processes, and of certifiers.

- Testing vs. Static Analysis Techniques [Be:c1; Pe:c17p359-360] (1008:p19)
- Testing vs. Correctness Proofs and Formal Verification [Be:c1s5; Pf:c7]
- Testing vs. Debugging [Be:c1s2.1] (1008:p19)
- Testing vs. Programming [Be:c1s2.3]
- Testing within SQA (see the SQ Chapter in this Guide)
- Testing within CMM (Po:p117-123)
- Testing within Cleanroom [Pf:c8s8.9]
- Testing and Certification (WK+)

B. Test Levels

B1. The target of the test

Testing of large software systems usually involves more steps [Be:c1; Jo:c12; Pf:c7].

Three big test stages can be conceptually distinguished, namely Unit, Integration and System. No process model is implied in this Guide, nor any of those three stages is assumed to have a higher importance than the other two. Depending on the development model followed, these three stages will be adopted and combined in different paradigms, and quite often more than one iteration between them is necessary.

- Unit testing [Be:c1; Pe:c17; Pf:c7s7.3] (1008)

Unit testing verifies the functioning in isolation of software pieces that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. A test unit is defined more precisely in the IEEE Standard for Software Unit Testing [1008], that also describes an integrated approach to systematic and documented unit testing. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools, and might involve the same programmers. Clearly, unit testing starts after coding is quite mature, for instance after a clean compile.

- Integration testing [Jo:c12,13; Pf:c7s7.4]

Integration testing is the process of verifying the interaction between system components (possibly, and hopefully, already tested in isolation). Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured systems. Modern systematic integration strategies are rather architecture driven, which implies integrating the software components or subsystems based on identified functional threads: integration testing is a continuous activity, at each stage of which testers must abstract away lower level perspectives and concentrate on the perspectives of the level they are integrating. Except for small, simple systems, systematic, incremental integration testing strategies are to be preferred to putting all components together at once, that is pictorially said "big-bang" testing.

- System testing [Jo:c14; Pf:c8]

System testing is concerned with the behavior of a whole system. The majority of functional failures should have been already identified during unit and integration testing. System testing should compare the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

B2. Objectives of Testing [Pe:c8; Pf:c8s8.3]

Testing of a software system (or subsystem) can be aimed at verifying different properties. Test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as conformance testing, "correctness" testing, functional testing. However several other non-functional properties need to be tested as well, including conformance, reliability and usability among many others.

References cited above give essentially a collection of the potential different purposes. The topics separately listed below (with the same or additional references) are those most often cited in the literature. Note that some kinds of testing are more appropriate for custom made packages (e.g., installation testing), while others for generic products (e.g., beta testing).

- Acceptance/qualification testing [Pe:c10; Pf:c8s8.5] (12207:s5.3.9)

Acceptance testing checks the system behavior against the customer's requirements (the "contract"); the customers undertake (or specify) typical tasks to check their requirements. This testing activity may or may not involve the developers of the system.

- Installation testing [Pe:c9; Pf:c8s8.6]

After completion of system and acceptance testing, the system is verified upon installation in the target environment, i.e., system testing is conducted according to the hardware configuration requirements. Installation procedures are also verified.

- Alpha and Beta testing [KF+:c13]

Before releasing the system, sometimes it is given in use to a small representative set of potential users, in-house (alpha testing) or external (beta testing), who report potential experienced problems with use of the product. Alpha and beta use is often uncontrolled, i.e., the testing does not refer to a test plan.

- Conformance testing/Functional testing/Correctness testing [KF+:c7; Pe:c8] (WK+)

Conformance testing is aimed at verifying whether the observed behavior of the tested system conforms to its specification.

- Reliability achievement and evaluation by testing [Pf:c8s.8.4; Ly:c7] (Ha; Musa and Ackermann in Po:p146-154)

By testing failures can be detected, and afterwards, if the faults that are the cause of the identified failures are efficaciously removed, the software will be more reliable. In this sense, testing is a means to improve reliability. On the other hand, by randomly generating test cases accordingly to the operational profile, statistical measures of reliability can be derived. Using reliability growth models, both objectives can be pursued together (see also part D.1).

- Regression testing [KF+:c7; Pe:c11,c12; Pf:c8s8.1] (RH)

According to (610), regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects [...]". In practice, the idea is to show that previously passed tests, still do. [Be] defines it as any repetition of tests intended to show that the software's behavior is unchanged except insofar as required. Obviously a tradeoff must be found between the assurance given by regression testing every time a change is made and the resources required to do that.

Regression testing can be conducted at each of the test levels in B.1, and may apply to functional and non-functional testing.

- Performance testing [Pe:c17; Pf:c8s8.3] (WK+)

This is specifically aimed at verifying that the system meets the specified performance requirements, e.g., capacity and response time. A specific kind of performance testing is volume testing (Pe:p185, p487; Pf:p349), in which internal program or system limitations are tried.

- Stress testing [Pe:c17; Pf:c8s8.3]

Stress testing exercises a system at the maximum design load as well as beyond it.

- Back-to-back testing

A same test set is presented to two implemented versions of a system, and the results are compared with each other.

- Recovery testing [Pe:c17; Pf:c8s8.3]

It is aimed at verifying system restart capabilities after a "disaster".

- Configuration testing [KF+:c8; Pf:c8s8.3]

In those cases in which a system is built to serve different users, configuration testing analyzes the system under the various specified configurations.

- Usability testing [Pe:c8; Pf:c8s8.3]

It evaluates the ease of using and learning the system (and system user documentation) by the end users, as well as the effectiveness of system functioning in supporting user tasks, and finally the ability of recovering from user's errors.

C. Test Techniques

In this section, two alternative classifications of test techniques are proposed. It is arduous to find a homogeneous criterion for classifying all techniques, as there exist many and very disparate.

The first classification, from C1.1 to C1.6, is based on how tests are generated, i.e., respectively from: tester's intuition and expertise, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage or finally the nature of application, which in some case can require knowledge of specific test problems and of specific test techniques.

The second classification is the classical distinction of test techniques between *black-box* and *white-box* (pictorial terms derived from the world of integrated circuit testing). Test techniques are here classified according to whether the tests rely on information about how the software has been designed and coded (white-box, somewhere also said glass-box), or instead only rely on the input/output behavior, without no assumption about what happens in between the "pins" (precisely, the entry/exit points) of the system (black box). Clearly this second classification is more coarse than the first one, and it does not allow us to categorize the techniques specialized on the nature of application (section C1.6) nor ad hoc approaches, because these can be either black-box or white-box. Also note that as new technologies such as Object Oriented or Component-based become more and more widespread, this split becomes more of a theoretical than a practical scope, as information about code and design is hidden or simply not available.

A final section, *C3,* deals with combined use of more techniques.

C1: CLASSIFICATION "based on how tests are generated"

C1.1 Based on tester's intuition and experience [KF+:c1]

Perhaps the most widely practiced technique remains *ad hoc testing*: tests are derived relying on the tester skill and intuition ("exploratory" testing), and on his/her experience with similar programs. While a more systematic approach is advised, ad hoc testing might be useful (but only if the tester is really expert!) to identify special tests, not easily "captured" by formalized techniques. Moreover it must be reminded that this technique may yield largely varying degrees of effectiveness.

C1.2 Specification-based

- Equivalence partitioning [Jo:c6; KF+:c7]

The input domain is subdivided into a collection of subsets, or "equivalent classes", which are deemed equivalent according to a specified relation, and a representative set of tests (sometimes even one) is taken from within each class.

- Boundary-value analysis [Jo:c5; KF+:c7]

Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many defects tend to concentrate near the extreme values of inputs. A simple, and often worth, extension of this technique is *Robustness Testing*, whereby test cases are also chosen outside the domain, in fact to test program robustness to unexpected, erroneous inputs.

- Decision table [Be:c10s3] (Jo:c7)

Decision tables represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Test cases are systematically derived by considering every possible combination of conditions and actions. A related techniques is *Cause-effect graphing* [Pf:c8].

- Finite-state machine-based [Be:c11; Jo:c4s4.3.2]

By modeling a program as a finite state machine, tests can be selected in order to cover states and transitions on it, applying different techniques. This technique is suitable for transaction-processing, reactive, embedded and real-time systems.

- Testing from formal specifications [ZH+:s2.2] (BG+; DF; HP)

Giving the specifications in a formal language (i.e., one with precisely defined syntax and semantics) allows for automatic derivation of functional test cases from the specifications, and at the same time provides a reference output, an oracle, for checking test results. Methods for deriving test cases from model-based (DF, HP) or algebraic specifications (BG+) are distinguished.

- Random testing [Be:c13; KF+:c7]

Tests are generated purely random (not to be confused with statistical testing from the operational profile, where the random generation is biased towards reproducing field usage, see C1.5). Actually, therefore, it is difficult to categorize this technique under the scheme of "base on which tests are generated". It is put under the Specification-based entry, as at least the domain must be known, to be able to pick random points within it.

C1.3 Code-based

- Reference models for code-based testing (flowgraph, call graph) [Be:c3; Jo:c4].

In code-based testing techniques, the control structure of a program is graphically represented using a flowgraph, i.e., a directed graph whose nodes and arcs correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs the transfer of control between nodes.

- Control flow-based criteria [Be:c3; Jo:c9] (ZH+:s2.1.1)

Control flow-based coverage criteria aim at covering all the statements or the blocks in a program, or specified combinations of them. Several coverage criteria have been proposed (like Decision/Condition Coverage), in the attempt to get good approximations for the exhaustive coverage of all control flow paths, that is unfeasible for all but trivial programs.

- Data flow-based criteria [Be:c5] (Jo:c10; ZH+:s2.1.2)

In data flow-based testing, the control flowgraph is annotated with information about how the program variables are defined and used. Different criteria exercise with varying degrees of precision how a value assigned to a variable is used along different control flow paths. A reference notion is a definition-use pair, which is a triple (d,u,V) such that: V is a variable, d is a node in which V is defined, and u is a node in which V is used; and such that there exists a path between d and u in which the definition of V in d is used in u.

C1.4 Fault-based (Mo)

With different degrees of formalization, fault based testing techniques devise test cases specifically aimed at revealing categories of likely or pre-defined faults.

- Error guessing [KF+:c7]

In error guessing, test cases are specifically designed by testers trying to figure out those, which could be the most plausible faults in the given program. A good source of information is the history of faults discovered in earlier projects, as well as tester's expertise.

- Mutation testing [Pe:c17; ZH+:s3.2-s3.3]

A mutant is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all generated mutants: If a test case is successful in identifying the difference between the program and a mutant, the latter is said to be killed. Originally conceived as a technique to evaluate a test set (see D.2.2), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants are killed or tests are specifically designed to kill (survived) mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, also more complex, (i.e., real) faults will be found. For the technique to be effective, a high number of mutants must be automatically derived in systematic way.

C1.5 Usage-based

- Operational profile [Jo:c14s14.7.2; Ly:c5; Pf:c8]

In testing for reliability evaluation, the test environment must reproduce as closely as possible the product use in operation. In fact, from the observed test results one wants to infer the future reliability in operation. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation.

- (Musa's) SRET [Ly:c6]

Software Reliability Engineered Testing (SRET) is a testing methodology encompassing the whole development process, whereby testing is "designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field".

C1.6 Based on nature of application

The above techniques apply to all types of software, and their classification is based on how test cases are derived. However, for some kinds of applications some additional know-how is required for test derivation. Here below a list of few "specialized" testing fields is provided, based on the nature of the application under test.

- Object-oriented testing [Jo:c15; Pf:c7s7.5] (Bi)
- Component-based testing
- Web-based testing
- GUI testing (OA+)
- Testing of concurrent programs (CT)
- Protocol conformance testing (Sidhu and Leung in Po:p102-115; BP)
- Testing of distributed systems
- Testing of real-time systems (Sc)
- Testing of scientific software

C2: CLASSIFICATION "ignorance or knowledge of implementation"

As explained at the beginning of Section C, here below an alternative classification of the same test techniques cited so far is proposed (just the headings are mentioned), based on whether knowledge of implementation is exploited to derive the test cases (white-box), or not (black-box).

C2.1 Black-box techniques

- Equivalence partitioning [Jo:c6; KF+:c7]
- Boundary-value analysis [Jo:c5; KF+:c7]
- Decision table [Be:c10s3] (Jo:c7)
- Finite-state machine-based [Be:c11; Jo:c4s4.3.2]
- Testing from formal specifications [ZH+:s2.2] (BG+; DF; HP)

- Error guessing [KF+:c7]
- Random testing [Be:c13; KF+:c7]
- Operational profile [Jo:c14s14.7.2; Ly:c5; Pf:c8]
- (Musa's) SRET [Ly:c6]

C2.2 White-box techniques

- Reference models for code-based testing (flowgraph, call graph) [Be:c3; Jo:c4].
- Control flow-based criteria [Be:c3; Jo:c9] (ZH+:s2.1.1)
- Data flow-based criteria [Be:c5] (Jo:c10; ZH+:s2.1.2)
- Mutation testing [Pe:c17; ZH+:s3.2-s3.3]

C3 Selecting and combining techniques

- Functional and structural [Be:c1s.2.2; Jo:c1, c11s11.3; Pe:c17] (Po:p3-4; Po:Appendix 2)

Specification-based and code-based test techniques are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternative, but rather as complementary: in fact, they use different sources of information, and have proved to highlight different kinds of problems. They should be used in combination, compatibly with budget availability.

- Coverage and operational/Saturation effect (Ha; Ly:p541-547; Ze)

Test cases can be selected in deterministic way, according to one of the various listed techniques, or randomly drawn from some distribution of inputs, such as it is usually done in reliability testing. There are interesting considerations one should be aware of, about the different implications of each approach.

D. Test related measures

Measurement is instrumental to quality analysis. Indeed, product evaluation is effective only when based on quantitative measures. Measurement is instrumental also to the optimal planning and execution of tests, and several process measures can be used by the test manager to monitor progress. Measures relative to the test process for management purposes are considered in part E.

A wider coverage of the topic of quality measurement, including fundamentals, measures and techniques for measurement, is provided in the Software Quality chapter of the Guide to the SWEBOK. A comprehensive reference is provided by the IEEE Standard. 982.2 "Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software", which was originally conceived as a guide to using the companion standard 982.1, that is the Dictionary. However, the guide is also a valid and very useful reference by itself, for selection and application of measures in a project.

Test related measures can be divided into two classes: those relative to evaluating the program under test, and those relative to evaluating the test set. The first class, for instance, includes measures that count and predict either faults (e.g., fault density) or failures (e.g., reliability). The second class instead evaluates the test suites against selected test criteria; notably, this is what is usually done by measuring the code coverage achieved by the executed tests.

D1. Evaluation of the program under test (982.2)

- Program measurements to aid in planning and designing testing. [Be:c7s4.2; Jo:c9] (982.2:sA16, BMa)

Measures based on program size (e.g., Source Lines of Code, function points) or on program structure (e.g., complexity) is useful information to guide the testing. Structural measures can also include measurements among program modules, in terms of the frequency with which modules call each other.

- Types, classification and statistics of faults [Be:c2; Jo:c1; Pf:c7] (1044, 1044.1; Be:Appendix; Ly:c9; KF+:c4, Appendix A)

The testing literature is rich of classifications and taxonomies of faults. Testing allows for discovering defects. To make testing more effective it is important to know which types of faults could be found in the application under test, and the relative frequency with which these faults have occurred in the past. This information can be very useful to make quality predictions as well as for process improvement. The topic "Defect Characterization" is also covered more deeply in the SQA Knowledge Area. An IEEE standard on how to classify software "anomalies" (1044) exists, with a relative guide (1044.1) to implement it. An important property for fault classification is orthogonality, i.e., ensuring that each fault can be unequivocally identified as belonging to one class.

- Fault density [Pe:c20] (982.2:sA1; Ly:c9)

In common industrial practice a product under test is assessed by counting and classifying the discovered faults by their types (see also A1). For each fault class, fault density is measured by the ratio between the number of faults found and the size of the program.

- Life test, reliability evaluation [Pf:c8] (Musa and Ackermann in Po:p146-154)

A statistical estimate of software reliability, that can be obtained by operational testing (see in B.2), can be used to evaluate a product and decide if testing can be stopped.

- Reliability growth models [Ly:c7; Pf:c8] (Ly:c3, c4)

Reliability growth models provide a prediction of reliability based on the failures observed under operational testing. They assume in general that the faults that caused the observed failures are fixed (although some models also accept imperfect fixes) and thus, on average, the product reliability exhibits an increasing trend. There exist now tens of published models, laid down on some common assumptions as well as on differing ones. Notably, the models are divided into *failures-count* and *time-between-failures* models.

D2. Evaluation of the tests performed

- Coverage/thoroughness measures [Jo:c9; Pf:c7] (982.2:sA5-sA6)

Several test adequacy criteria require the test cases to systematically exercise a set of elements identified in the program or in the specification (see Part C). To evaluate the thoroughness of the executed tests, testers can monitor the elements covered, so that they can dynamically measure the ratio (often expressed as a fraction of 100%) between covered elements and the total number. For example, one can measure the percentage of covered branches in the program flowgraph, or of exercised functional requirements among those listed in the specification document. Code-based adequacy criteria require appropriate instrumentation of the program under test.

- Fault seeding [Pf:c7] (ZH+:s3.1)

Some faults are artificially introduced into the program before test. When the tests are executed, part of these seeded faults will be revealed, as well as possibly genuine faults. Depending on which and how many of the artificial faults are hit, testing effectiveness can be evaluated; also, one could estimate how many of the genuine faults should remain.

- Mutation score [ZH+:s3.2-s3.3]

Mutation testing has been described before (within C1.4). The proportion between killed mutants and the total number of generated mutants can be a measure of the effectiveness of the executed test set.

- Comparison and relative effectiveness of different techniques [Jo:c8,c11; Pe:c17; ZH+:s5] (FW; Weyuker in Po p64-72; FH+)

Several studies have been recently conducted to compare the relative effectiveness of different test techniques. It is important to be precise relative to the property against which the techniques are being assessed, i.e., what "effectiveness" is exactly meant for. Possible interpretations are how many tests are needed to find the first failure, or the ratio of the number of faults found by the testing to all the faults found during and after the testing, or of how much reliability is improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the above specified notions of "effectiveness".

E. Managing the Test Process

E1. Management concerns

- Attitudes/Egoless programming [Be:c13s3.2; Pf:c7]

A very important component of successful testing is a positive and collaborative attitude towards testing activities. Managers should revert a negative vision of testers as the destroyers of developers' work and as heavy budget consumers. On the contrary, they should foster a common culture towards software quality, by which early failure discover is an objective for all involved people, and not only of testers.

- Test process [Be:c13; Pe:c1,c2,c3,c4; Pf:c8] (Po:p10-11; Po:Appendix 1; 12207:s5.3.9;s5.4.2;s6.4;s6.5)

A process is defined as "a set of interrelated activities, which transform inputs into outputs"[12207]. Test activities conducted at different levels (see B.1) must be organized, together with people, tools, policies, measurements, into a well defined process, which is integral part to the life cycle. This test process needs control and continuous improvement. In the IEEE/EIA Standard 12207.0 testing is not described as a stand alone process, but principles for testing activities are included along with the five primary life cycle processes, as well as along with the supporting process.

- Test documentation and workproducts [Be:c13s5; KF+:c12; Pe:c19; Pf:c8s8.8] (829)

Documentation is an integral part of the formalization of the test process. The IEEE standard for Software Test Documentation [829] provides a good description of test documents and of their relationship with one another and with the testing process. Test documents includes, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log and Test Incident or Problem Report. The program under test, with specified version and identified hw/sw requirements before testing can begin, is documented as the Test Item. Test documentation should be produced and continually updated, at the same standards as other types of documentation in development.

- Internal vs. independent test team [Be:c13s2.2-2.3; KF+:c15; Pe:c4; Pf:c8]

Formalization of the test process requires formalizing the test team organization as well. The test team can be composed of members internal to the project team (but not directly involved in code development), or of external members, in the latter case bringing in an unbiased, independent perspective, or finally of both internal and external members. The decision will be determined by considerations of costs, schedule, maturity levels of the involved organizations, and criticality of the application.

- Cost/effort estimation and other process measures [Pe:c4, c21] (Pe: Appendix B; Po:p139-145; 982.2:sA8-sA9)

In addition to those discussed in Part D, several measures relative to the resources spent on testing, as well as to the relative effectiveness in fault finding of the different test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as: number of test cases specified, number of test cases executed, number of test cases passed, number of test cases failed, and similar.

Evaluation of test phase reports is often combined with root cause analysis to evaluate test process effectiveness in finding faults as early as possible. Moreover, the resources that are worth spending in testing should be commensurate to the use/criticality of the application: the techniques listed in part C have different costs, and yield different levels of confidence in product reliability.

- Termination [Be:c2s2.4; Pe:c2]

A critical task of the test manager is to decide how much testing is enough and when a test stage can be terminated. Thoroughness measures such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability, provide useful support, but are not sufficient by themselves. The decision involves also considerations about the costs and risks incurred by potentially remaining failures, as opposed to the costs implied by further continuing to test.

- Test reuse and test patterns [Be:c13s5]

To carry out testing or maintenance in an organized and cost/effective way, the means used to test each part of the system should be reused systematically. At all levels of testing, test scripts, test cases, and expected results should be carefully defined and documented so that they may be reused. This repository of test materials must be configuration controlled, so that changes to system requirements or design can be reflected in changes to the scope of the tests conducted.

The test solutions adopted for testing some application type under certain circumstances, with the motivations behind the decisions taken, form a test pattern, that can itself be documented for later reuse in similar projects.

E2. Test Activities

Here below a brief overview of test activities is given; as often implied by the following description, successful management of test activities strongly depends from the Software Configuration Management process (see Chapter 7 in this Guide).

- Planning [KF+:c12; Pe:c19; Pf:c7s7.6] (829:s4; 1008:s1, s2, s3)

Like any other part of project management, testing activities must be planned. Key aspects of test planning include co-ordination of personnel needed, management of available test facilities and equipment (which may include magnetic media, test plans and procedures), and planning for possible undesirable outcomes. If more than one baseline of the system is being maintained, then a major planning consideration is the time and effort needed to ensure the test environment is set to the proper configuration.

- Test case generation [KF+:c7] (Po:c2; 1008:s4, s5)

Generation of test cases is based on the level of testing to be performed, and the particular testing techniques. Test cases should be configuration controlled and include the expected results for each test.

- Test environment development [KF+:c11]

The environment used for testing should be compatible with the software development environment. It should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials.

- Execution [Be:c13; KF+:c11] (1008:s6, s7;)

Execution of tests is generally performed by testing engineers with oversight by quality assurance personnel and, in some cases, customer representatives. Execution of tests should embody the basic principles of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the same results. Hence testing should be performed in accordance with documented procedures using a clearly defined version of the system under test.

- Test results evaluation [Pe:c20,c21] (Po:p18-20; Po:p131-138)

The results of testing must be evaluated to determine if the test was successful, and to derive specific test measures. In most cases, ‘successful’ means that the system performed as expected, and did not have any major unexpected outcomes. On the other side, not all unexpected outcomes are necessarily faults, but could be judged as just noise. Before a failure can be removed, analysis and debugging effort is needed to isolate, identify and describe it. When test results are particularly important, a formal review board may be convened to evaluate test results.

- Problem reporting/Test log [KF+:c5; Pe:c20] (829:s9-s10)

All testing activities should be entered into a test log to identify when a test was conducted, who performed the test, what system configuration was the basis for testing, and other relevant identification information. Unexpected or incorrect test results should be recorded in a problem reporting system. The problem reporting system’s data forms the basis for later debugging and fixing the problems which were observed as failures during testing. Also anomalies not classified as faults could be documented, in case they later turn out to be more serious than judged. Test Reports are also an input to the Change Management system (which is a part of the Configuration Management system).

- Defect tracking [KF+:c6]

Failures observed during testing are often due to faults or defects in the system. Such defects should be analyzed to determine when they were introduced into the system, what kind of error caused them to be created (e.g. poorly defined requirements, incorrect variable declaration, memory leak, programming syntax error, etc.), and when they could have been first observed in the system. Defect tracking information is used to determine what aspects of system development need improvement and how effective have been previous analyses and testing.

4 Breakdown Rationale

The conceptual scheme followed in decomposing the Software Testing Knowledge Area is described in Section 2.1. Level 1 topics include five entries, labeled from A to E, that correspond to the fundamental and complementary concerns forming the Software Testing knowledge: Basic Concepts and Definitions, Levels, Techniques, Measures, and Process. There is not a standard way to decompose the Software Testing Knowledge Area, each book on Software Testing would structure its table of contents in different ways. However any thorough book on Software Testing would cover these five topics. A sixth level 1 topic would be Test Tools. These are not covered here, but in the *Software Engineering Tools and Methods* chapter of the Guide to the SWEBOK.

The breakdown is three levels deep. The second level is for making the decomposition more understandable. The selection of level 3 topics, that are the subjects of study, has been quite difficult. Finding a breakdown of topics that is "generally accepted" by all different communities of potential users of the Guide to the SWEBOK is challenging for Software Testing, because there still exists a wide gap between the literature on Software Testing and current industrial test practice. There are topics that have been taking a relevant position in the academic literature for many years now, but are not generally used in industry, for example data-flow based or mutation testing. The position taken in writing this document has been to include any relevant topics in the literature, even those that are likely not considered so relevant by practitioners at the current time. The proposed breakdown of topics for Software Testing is thus considered as an inclusive list, from which each stakeholder can pick according to his/her needs.

However, under the precise definition for "generally accepted" adopted in the Guide to the SWEBOK (i.e., *knowledge to be included in the study material of a software engineering with four years of work experience*), some of the included topics (like the examples above) would be only lightly (if at all) covered in a curriculum of a software engineer with four years of experience. The recommended references have been therefore selected accordingly, i.e., they provide reading material according to this meaning of "generally accepted", while the more advanced topics are covered in the Further Reading list.

5 Matrix of Topics vs. Reference Material

A. Testing Basic Concepts and Definitions	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**	**[ZH+]**
Definitions of testing and related terminology	C1	C1,2,3,4	C2S2.2				
Faults vs. Failures		C1	C2S2.2		C1	C7	
Test selection criteria/Test adequacy criteria (or stopping rules)						C7S7.3	S1.1
Testing effectiveness/Objectives for testing	C1S1.4				C21		
Testing for defect identification	C1			C1			
The oracle problem	C1						
Theoretical and practical limitations of testing				C2			
The problem of infeasible paths	C3						
Testability	C3,13						
Testing vs. Static Analysis Techniques	C1				C17		
Testing vs. Correctness Proofs and Formal Verification	C1S5					C7	
Testing vs. Debugging	C1S2.1						
Testing vs. Programming	C1S2.3						
Testing within SQA							
Testing within CMM							
Testing within Cleanroom						C8S8.9	
Testing and Certification							

B. Test Levels	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**
Unit testing	C1				C17	C7S7.3
Integration testing		C12,13				C7S7.4
System testing		C14				C8
Acceptance/qualification testing					C10	C8S8.5
Installation testing					C9	C8S8.6
Alpha and Beta testing				C13		
Conformance testing/ Functional testing/ Correctness testing				C7	C8	
Reliability achievement and evaluation by testing			C7			C8S8.4
Regression testing				C7	C11,12	C8S8.1
Performance testing					C17	C8S8.3
Stress testing					C17	C8S8.3
Back-to-back testing						
Recovery testing					C17	C8S8.3
Configuration testing				C8		C8S8.3
Usability testing					C8	C8S8.3

C. Test Techniques	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**	**[ZH+]**
Ad hoc				C1			
Equivalence partitioning		C6		C7			
Boundary-value analysis		C5		C7			
Decision table	C10S3						
Finite-state machine-based	C11	C4S4.3.2					
Testing from formal specifications							S2.2
Random testing	C13			C7			
Reference models for code-based testing (flow graph, call graph)	C3	C4					
Control flow-based criteria	C3	C9				C7	
Data flow-based criteria	C5						
Error guessing						C7	
Mutation testing					C17		S3.2, 3.3
Operational profile		C14S14.7.2	C5			C8	
SRET			C6				
Object-oriented testing		C15				C7S7.5	
Component-based testing							
Web-based testing							
GUI testing							
Testing of concurrent programs							
Protocol conformance testing							
Testing of distributed systems							
Testing of real-time systems							
Testing of scientific software							
Functional and structural	C1S2.2	C1,11S11.3			C17		
Coverage and operational/Saturation effect							

D. Test Related Measures	[Be]	[Jo]	[Ly]	[KF+]	[Pe]	[Pf]	[ZH+]
Program measurements to aid in planning and designing testing.	C7S4.2	C9					
Types, classification and statistics of faults	C2	C1				C7	
Remaining number of defects/Fault density					C20		
Life test, reliability evaluation						C8	
Reliability growth models			C7			C8	
Coverage/thoroughness measures		C9				C7	
Fault seeding						C7	
Mutation score							S3.2, 3.3
Comparison and relative effectiveness of different techniques		C8,11			C17		S5

E. Managing the Test Process	[Be]	[Jo]	[Ly]	[KF+]	[Pe]	[Pf]
Attitudes/Egoless programming	C13S3.2					C7
Test process	C13				C1,2,3,4	C8
Test documentation and workproducts	C13S5			C12	C19	C8S8.8
Internal vs. independent test team	C13S2.2,2.3			C15	C4	C8
Cost/effort estimation and other process measures					C4,21	
Termination	C2S2.4				C2	
Test reuse and test patterns	C13					
Planning				C12	C19	C7S7.6
Test case generation				C7		
Test environment development				C11		
Execution	C13			C11		
Test results evaluation					C20,21	
Problem reporting/Test log				C5	C20	
Defect tracking				C6		

6 Recommended References for Software Testing

Be Beizer, B. *Software Testing Techniques* 2nd Edition. Van Nostrand Reinhold, 1990. [Chapters 1, 2, 3, 5, 7s4, 10s3, 11, 13]

Jo Jorgensen, P.C., *Software Testing A Craftsman's Approach*, CRC Press, 1995. [Chapters 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15]

KF+ Kaner, C., Falk, J., and Nguyen, H. Q., *Testing Computer Software*, 2nd Edition, Wiley, 1999. [Chapters 1, 2, 5, 6, 7, 8, 11, 12, 13, 15]

Ly Lyu, M.R. (Ed.), *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996. [Chapters 2s2.2, 5, 6, 7]

Pe Perry, W. *Effective Methods for Software Testing*, Wiley, 1995. [Chapters 1, 2, 3, 4, 9, 10, 11, 12, 17, 19, 20, 21]

Pf Pfleeger, S.L. *Software Engineering Theory and Practice*, Prentice Hall, 1998. [Chapters 7, 8]

ZH+ Zhu, H., Hall, P.A.V., and May, J.H.R. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29, 4 (Dec. 1997) 366-427. [Sections 1, 2.2, 3.2, 3.3,

APPENDIX A – LIST OF FURTHER READINGS

Books

Be Beizer, B. *Software Testing Techniques* 2nd Edition. Van Nostrand Reinhold, 1990.

Bi Binder, R. V., *Testing Object-Oriented Systems Models, Patterns, and Tools*, Addison-Wesley, 2000.

Jo Jorgensen, P.C., *Software Testing A Craftsman's Approach*, CRC Press, 1995.

KF+ Kaner, C., Falk, J., and Nguyen, H. Q., *Testing Computer Software*, 2nd Edition, Wiley, 1999.

Ly Lyu, M.R. (Ed.), *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996.

Pe Perry, W. *Effective Methods for Software Testing*, Wiley, 1995.

Po Poston, R.M. *Automating Specification-based Software Testing*, IEEE, 1996.

Survey Papers

ZH+ Zhu, H., Hall, P.A.V., and May, J.H.R. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29, 4 (Dec. 1997) 366-427.

Specific Papers

BG+ Bernot, G., Gaudel, M.C., and Marre, B. Software Testing Based On Formal Specifications: a Theory and a Tool. *Software Engineering Journal* (Nov. 1991) 387-405.

BM Bache, R., and Müllerburg, M. Measures of Testability as a Basis for Quality Assurance. *Software Engineering Journal,* 5 (March 1990) 86-92.

BMa Bertolino, A., Marrè, M. "How many paths are needed for branch testing?", *The Journal of Systems and Software*, Vol. 35, No. 2, 1996, pp.95-106.

BP Bochmann, G.V., and Petrenko, A. Protocol Testing: Review of Methods and Relevance for Software Testing. *ACM Proc. Int. Symposium on Sw Testing and Analysis (ISSTA' 94),* (Seattle, Washington, USA, August 1994) 109-124.

BS Bertolino, A., and Strigini, L. On the Use of Testability Measures for Dependability Assessment. *IEEE Transactions on Software Engineering,* 22, 2 (Feb. 1996) 97-108.

CT Carver, R.H., and Tai, K.C., Replay and testing for concurrent programs. *IEEE Software* (March 1991) 66-74

DF Dick, J., and Faivre, A. Automating The Generation and Sequencing of Test Cases From Model-Based Specifications. *FME'93: Industrial-Strenght Formal Method*, LNCS 670, Springer Verlag, 1993, 268-284.

FH+ Frankl, P., Hamlet, D., Littlewood B., and Strigini, L. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering,* 24, 8, (August 1998), 586-601.

FW Frankl, P., and Weyuker, E. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering,* 19, 3, (March 1993), 202-

Ha Hamlet, D. Are we testing for true reliability? *IEEE Software* (July 1992) 21-27.

Ho Howden, W.E., Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering,* 2, 3, (Sept. 1976) 208-215

HP Horcher, H., and Peleska, J. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4 (1995) 309-327.

Mo Morell, L.J. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering* 16, 8 (August 1990), 844-857.

MZ Mitchell, B., and Zeil, S.J. A Reliability Model Combining Representative and Directed Testing. *ACM/IEEE Proc. Int. Conf. Sw Engineering ICSE 18* (Berlin, Germany, March 1996) 506-514.

OA+ Ostrand, T., Anodide, A., Foster, H., and Goradia, T. A Visual Test Development Environment for GUI Systems. *ACM Proc. Int. Symposium on Sw Testing and Analysis (ISSTA' 98),* (Clearwater Beach, Florida, USA, March 1998) 82-92.

OB Ostrand, T.J., and Balcer, M. J. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of ACM*, 31, 3 (June 1988), 676-686.

RH Rothermel, G., and Harrold, M.J., Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22, 8 (Aug. 1996) 529-

Sc Schütz, W. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems Journal*. 7, 2, (Sept. 1994) 129-157.

VM Voas, J.M., and Miller, K.W. Software Testability: The New Verification. *IEEE Software*, (May 1995) 17-28.

We-a Weyuker, E.J. On Testing Non-testable Programs. *The Computer Journal*, 25, 4, (1982) 465-470

We-b Weyuker, E.J. Assessing Test Data Adequacy through Program Inference. *ACM Trans. on Programming Languages and Systems*, 5, 4, (October 1983) 641-655

WK+ Wakid, S.A., Kuhn D.R., and Wallace, D.R. Toward Credible IT Testing and Certification, *IEEE Software*, (August 1999) 39-47.

WW+ Weyuker, E.J., Weiss, S.N, and Hamlet, D. Comparison of Program Test Strategies in *Proc. Symposium on Testing, Analysis and Verification TAV 4* (Victoria, British Columbia, October 1991), ACM Press, 1-10.

Standards

610 IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology.

829 IEEE Std 829-1998, Standard for Software Test Documentation.

982.2 IEEE Std 982.2-1998, Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

1008 IEEE Std 1008-1987 (R 1993), Standard for Software Unit Testing.

1044 IEEE Std 1044-1993, Standard Classification for Software Anomalies.

1044.1 IEEE Std 1044.1-1995, Guide to Classification for Software Anomalies.

12207 IEEE/EIA 12207.0-1996, Industry Implementation of Int. Std. ISO/IEC 12207:1995, Standard for Information Technology-Software Life cycle processes.