CHAPTER 5

SOFTWARE TESTING

ACRONYMS

TDD Test-Driven Development

XP Extreme Programming

INTRODUCTION

Testing is performed to evaluate and improve product quality by identifying defects and problems.

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

In the above definition, italicized words correspond to key issues in identifying the Knowledge Area of Software Testing. In particular:

- *Dynamic:* This term means that testing always implies executing the program on (valued) inputs. To be precise, the input value alone is not always sufficient to determine a test, since a complex, nondeterministic system might react to the same input with different behaviors, depending on the system state. In this KA, though, the term "input" will be maintained, with the implied convention that its meaning also includes a specified input state in those cases in which it is needed. Different from (dynamic) testing and complementary to it are static techniques, as described in the Software Quality KA.
- *Finite:* Even in simple programs, so many test cases are theoretically possible that exhaustive testing could require months or years to execute. This is why, in practice, the whole test set can generally be considered infinite. Testing always implies a tradeoff between limited resources and schedules on the one hand and inherently unlimited test requirements on the other.
- *Selected:* The many proposed test techniques differ essentially in how they select the test set, and software engineers must be aware that different selection criteria may yield vastly different degrees of effectiveness. How to identify the most suitable selection criterion under given conditions is a 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 expectations (commonly referred to as testing for validation), against a specification (testing for verification), or, finally, against the anticipated behavior from implicit requirements or reasonable expectations. (See "Acceptance Tests" in the Software Requirements KA).

In recent years, the view of software testing has matured into a constructive one. 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 now seen as an activity that should encompass the whole development and maintenance process and is itself an important part of the actual product construction. Indeed, planning for testing should start with the early stages of the requirement process, and test plans and procedures must be systematically and continuously developed—and possibly refined—as development proceeds. These test planning and designing activities provide useful input for designers in highlighting potential weaknesses (like design oversights or contradictions and omissions or ambiguities in the documentation).

Currently, the right attitude towards quality is considered one of prevention: it is obviously much better to avoid problems than to correct them. Testing must be seen, then, primarily as a means not only for checking whether the prevention has been effective, but also for identifying faults in those cases where, for some reason, it has not been effective. It is perhaps obvious but worth recognizing that, even after successful completion of an extensive testing campaign, the software could still contain faults. The remedy for software failures experienced after delivery is provided by corrective maintenance actions. Software maintenance topics are covered in the Software Maintenance KA.

In the Software Quality KA (see "Software Quality Management Techniques"), software quality management techniques are notably categorized into *static* techniques (no code execution) and *dynamic* techniques (code execution). Both categories are useful. This KA focuses on dynamic techniques.

Software testing is also related to software construction (see "Construction Testing" in that KA). In particular, unit and integration testing are intimately related to software construction, if not part of it.

BREAKDOWN OF TOPICS FOR SOFTWARE TESTING

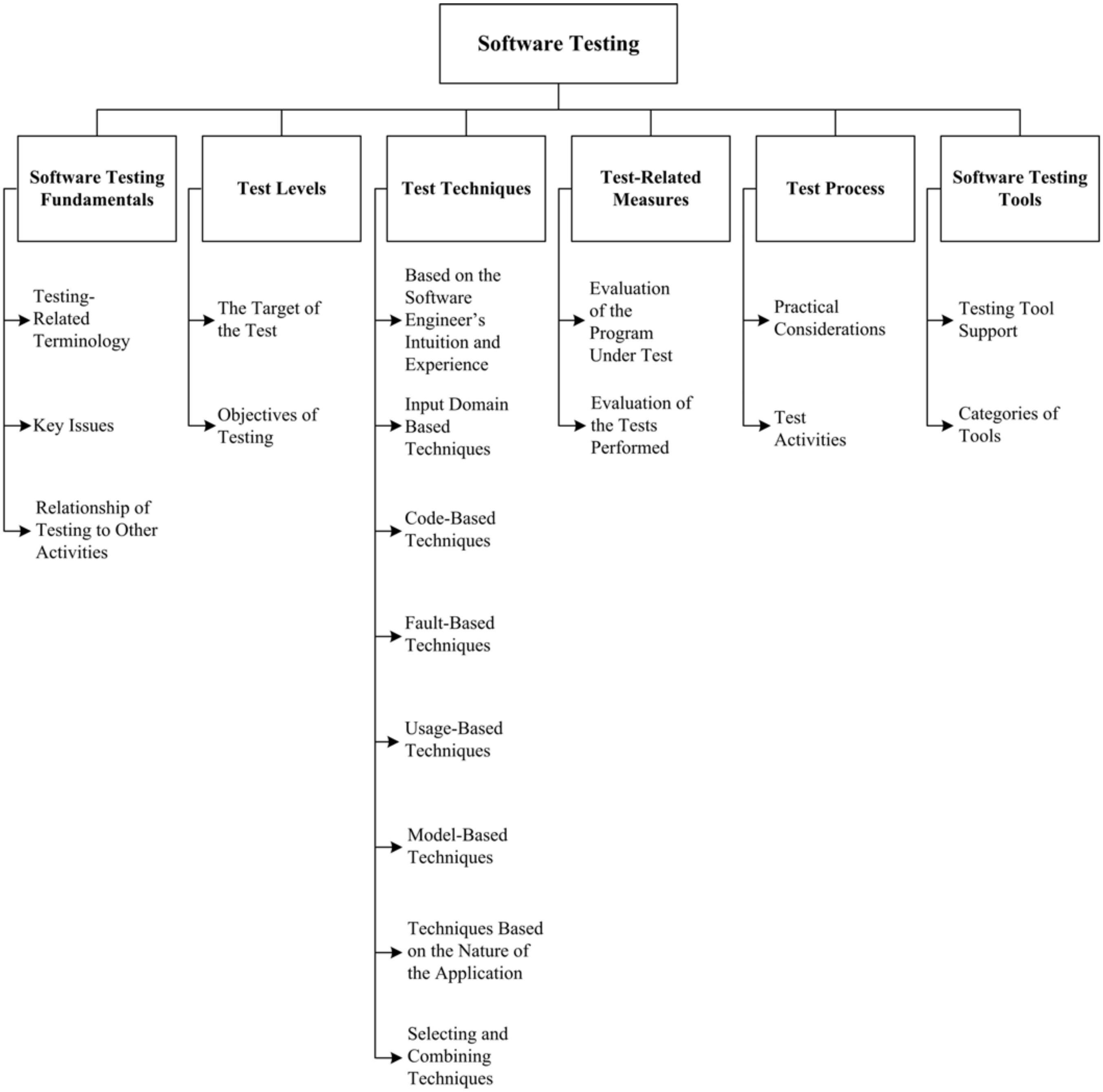

Figure 1: Breakdown of Topics for the Software Testing KA

The breakdown of topics for the Software Testing KA is shown in Figure 1. A more detailed breakdown is provided in Tables 1-A to 1-F.

The first subarea describes *Software Testing Fundamentals*. It covers the basic definitions in the field of software testing, the basic terminology and key issues, and software testing's relationship with other activities.

The second subarea, *Test Levels*, consists of two (orthogonal) topics: 2.1 lists the levels in which the testing of large software is traditionally subdivided, and 2.2 considers testing for specific conditions or properties and is referred to as *objectives of testing*. Not all types of testing apply to every software product, nor has every possible type been listed.

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 those addressing the second question are the test *selection criteria*.

Several *Test Techniques* have been developed in the past few decades, and new ones are still being proposed. Generally accepted techniques are covered in subarea 3.

Test-Related Measures are dealt with in subarea 4, while the issues relative to *Test Process* are covered in subarea 5. Finally *Software Testing Tools* are presented in subarea 6.

Table 1-A: Breakdown for Software Testing Fundamentals

1. Software Testing Fundamentals	1.1Testing-related terminology	Definitions of testing and related terminology
		Faults vs. Failures
	1.2 Key Issue	Test selection criteria/Test adequacy criteria (or stopping rules)
		Testing effectiveness/Objectives for testing
		Testing for defect identification
		The oracle problem
		Theoretical and practical limitations of testing
		The problem of infeasible paths
		Testability
	1.3 Relationship of testing to other activities	Testing vs. Static Software Quality Management Techniques
		Testing vs. Correctness Proofs and Formal Verification
		Testing vs. Debugging
		Testing vs. Programming

Table 1-B: Breakdown for Test Levels

2. Test Levels	2.1 The target of the test	Unit testing
		Integration testing
		System testing
	2.2 Objectives of testing	Acceptance/qualification testing
		Installation testing
		Alpha and Beta testing
		Reliability and evaluation achievement
		Regression testing
		Performance testing
		Security testing
		Stress testing
		Back-to-back testing
		Recovery testing
		Configuration testing
		Usability and human computer interaction testing

		Test-driven development

Table 1-C: Breakdown for Test Techniques		
3. Test Techniques	3.1 Based on the software engineer's intuition and experience	Ad hoc
		Exploratory testing
	3.2 Input domain-based techniques	Equivalence partitioning
		Pairwise testing
		Boundary-value analysis
		Random testing
	3.3 Code-based techniques	Control-flow-based criteria
		Data flow-based criteria
		Reference models for code-based testing (flowgraph, call graph)
	3.4 Fault-based techniques	Error guessing
		Mutation testing
	3.5 Usage-based techniques	Operational profile
		User observation heuristics
	3.6 Model-based testing techniques	Decision table
		Finite-state machine-based
		Testing from formal specifications
	3.7 Techniques based on the nature of the application	
	3.8 Selecting and combining techniques	Functional and structural
		Deterministic vs. random

Table 1-D: Breakdown for Test-Related Measures		
4. Test-Related Measures	4.1 Evaluation of the program under test	Program measurements to aid in planning and designing testing
		Fault types, classification, and statistics
		Fault density
		Life test, reliability evaluation
		Reliability growth models
	4.2 Evaluation of the tests performed	Coverage/thoroughness measures
		Fault seeding
		Mutation score
		Comparison and relative effectiveness of different techniques

Table 1-E: Breakdown for Test Process		
5. Test Process	5.1 Practical considerations	Attitudes/Egoless programming
		Test guides
		Test process management
		Test documentation and work products
		Internal vs. independent test team
		Cost/effort estimation and other process measures
		Termination
		Test reuse and patterns
	5.2 Test activities	Planning
		Test-case generation
		Test environment development
		Execution
		Test results evaluation
		Problem reporting/Test log
		Defect tracking

Table 1-F: Breakdown for Software Testing Tools		
6 Software Testing Tools	6.1 Testing tool support	Selecting tools
	6.2 Categories of tools	Test harness
		Test generators
		Capture/Replay tools
		Oracle/file comparators/assertion checking
		Coverage analyzer/Instrumenter
		Tracers
		Regression testing tools
		Reliability evaluation tools

1. Software Testing Fundamentals

1.1. Testing-related terminology

- Definitions of testing and related terminology [1*, c1, c2, 2*, c8].

A comprehensive introduction to the Software Testing KA is provided in the recommended references.

- Faults vs. Failures [1*, c1s5, 2*, c11].

Many terms are used in the software engineering literature to describe a malfunction: notably *fault*, *failure*, and *error,* among others. This terminology is precisely defined in [3] and [4]. It is essential to clearly distinguish between the *cause* of a malfunction (for which the term *fault* or *defect* will be used here) and an undesired effect observed in the system's delivered service (which will be called a *failure)*. Testing can reveal failures, but it is the faults that can and must be removed [5].

However, it should be recognized that the cause of a failure cannot always be unequivocally identified. No theoretical criteria exist to definitively determine what fault caused the observed failure. It might be said that it was the fault that had to be modified to remove the problem, but other modifications could have worked just as well. To avoid ambiguity, one could refer to *failure-causing inputs* instead of faults—that is, those sets of inputs that cause a failure to appear.

1.2. Key issues

- Test selection criteria/Test adequacy criteria (or stopping rules) [1*c1s14, c6s6, c12s7]

A test selection criterion is a means of deciding what a suitable set of test cases should be. A selection criterion can be used for selecting the test cases or for checking whether a selected test suite is adequate—that is, to decide whether the testing can be stopped [6]. See also the sub-topic *Termination*, under topic 5.1 *Practical considerations*.

- Testing effectiveness/Objectives for testing [1*,c13s11, c11s4].

Testing is the observation of a sample of program executions. Sample selection 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.

- Testing for defect identification [1*, c1s14].

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 specifications or other desired properties, in which case testing is successful if no (significant) failures are observed.

- The oracle problem [1*, c1s9, c9s7]

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

- Theoretical and practical limitations of testing [1*, c2s7]

Testing theory warns against ascribing an unjustified level of confidence to a series of passed tests. Unfortunately, most established results of testing theory are negative ones, in that they state what testing can never achieve as opposed to what it actually achieved. The most famous quotation in this regard is the Dijkstra aphorism that "program testing can be used to show the presence of bugs, but never to show their absence" [7]. The obvious reason for this is that complete testing is not feasible in real software. Because of this, testing must be driven based on risk and could be seen as a risk management strategy.

- The problem of infeasible paths [1*, c4s7]

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

- Testability [1*, c17s2]

The term "software testability" has two related but different meanings: on the one hand, it refers to the degree to which it is easy for software to fulfill a given test coverage criterion; on the other hand, it is defined as the likelihood, possibly measured statistically, that the software will expose a failure under testing *if* it is faulty. Both meanings are important.

1.3. Relationship of testing to other activities

Software testing is related to, but different from, static software quality management techniques, proofs of correctness, debugging, and programming. However, it is informative to consider testing from the point of view of software quality analysts and of certifiers.

- Testing vs. Static Software Quality Management Techniques. See also *Software Quality Management Processes* in the Software Quality KA. [1*, c12].
- Testing vs. Correctness Proofs and Formal Verification. See also the Software Engineering Models and Methods KA [1*, c17s2].
- Testing vs. Debugging. See also *Construction Testing* in the Software Construction KA and *Debugging Tools and Techniques* in the Computing Foundations KA [1*, c3s6].
- Testing vs. Programming. See also *Construction Testing* in the Software Construction KA [1*, c3s2].

2. Test Levels

2.1. *The target of the test [1*, c1s13, 2*, c8s1].*

Software testing is usually performed at different *levels* along the development and maintenance processes. That is to say, the target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or a whole system. Three test stages can be conceptually distinguished—namely, Unit, Integration, and System. No process model is implied, nor is any of those three stages assumed to have greater importance than the other two.

- Unit testing [1*, c3, 2*, c8]

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. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools; it might involve the programmers who wrote the code.

- Integration testing [1*, c7, 2*, c8]

Integration testing is the process of verifying the interaction between software components. Classical integration-testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.

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 software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once—which is pictorially called "big bang" testing.

- System testing [1*, c8, 2*, c8]

System testing is concerned with the behavior of a whole system. The majority of functional failures should already have been identified during unit and integration testing. System testing is usually considered appropriate for comparing the system to the nonfunctional system requirements—such as security, speed, accuracy, and reliability (see *Functional and NonFunctional Requirements* in the Software Requirements KA). External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

2.2. *Objectives of testing [1*, c1s7]*

Testing is conducted in view of a specific objective, which is stated more or less explicitly, and with varying degrees of precision. Stating the objective in precise, quantitative terms allows control to be established over the test process.

Testing 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, or *functional* testing. However, several other nonfunctional properties may be tested as well—including performance, reliability, and usability, among many others.

Other important objectives for testing include (but are not limited to) reliability measurement, usability evaluation, and acceptance, for which different approaches would be taken. Note that, in general, the test objective varies with the test target; different purposes being addressed at a different level of testing.

The sub-topics listed below are those most often cited in the literature. Note that some kinds of testing are more appropriate for custom-made software packages—*installation* testing, for example—and others for generic products, like *beta* testing.

- Acceptance/qualification testing [1*, c1s7, 2*, c8s4].

Acceptance testing checks the system behavior against the customer's requirements, however these may have been expressed; the customers undertake, or specify, typical tasks to check that their requirements have been met or that the organization has identified these for the software's target market. This testing activity may or may not involve the system's developers.

- Installation testing [1*, c12s2]

Usually after completion of system and acceptance testing, the software can be verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified.

- Alpha and beta testing [1*, c13s7, c16s6, 2*, c8s4]

Before the software is released, it is sometimes given to a small, representative set of potential users for trial use, either in-house (*alpha* testing) or external (*beta* testing). These users report problems with the product. Alpha and beta use is often uncontrolled and is not always referred to in a test plan.

- Reliability and evaluation achievement [1*, c15, 2*, c15s2]

In helping to identify faults, testing is a means to improve reliability. By contrast, by randomly generating test cases according to the operational profile, statistical measures of reliability can be derived. Using reliability growth models, both objectives can be pursued together [5] (see also sub-topic *Life test, reliability evaluation* under 4.1 *Evaluation of the program under test*).

- Regression testing [1*, c8s11, c13s3]

According to IEEE/ISO/IEC 24765:2009 Systems and Software Engineering Vocabulary [3], regression testing is the “selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.” In practice, the idea is to show that software that previously passed the tests still does (in fact, it is also referred to as non-regression testing). Specifically for incremental development, the purpose is to show that the software’s behavior is unchanged, except insofar as required. Obviously, a tradeoff must be made between the assurance given by regression testing every time a change is made and the resources required to do that. Regression testing refers to techniques for selecting, minimizing, and/or prioritizing a subset of the test cases in an existing test suite [8]. Regression testing can be conducted at each of the test levels described in topic 2.1 *The target of the test* and may apply to functional and nonfunctional testing.

- Performance testing [1*, c8s6]

This is specifically aimed at verifying that the software meets the specified performance requirements—for instance, capacity and response time.

- Security testing [1*, c8s3, 2*, c11s4]

This is focused on the verification that the software is protected from external attacks. In particular, security testing verifies the confidentiality, integrity, and availability of the systems and its data. Usually, security testing includes verification of misuse and abuse of the software or system (negative testing).

- Stress testing [1*, c8s8]

Stress testing exercises software at the maximum design load, as well as beyond it.

- Back-to-back testing [3]

IEEE/ISO/IEC Standard 24765 defines back-to-back testing as “testing in which two or more variants of a program are executed with the same inputs, the outputs are compared, and errors are analyzed in case of discrepancies.”

- Recovery testing [1*, c14s2]

Recovery testing is aimed at verifying software restart capabilities after a “disaster.”

- Configuration testing [1*, c8s5]

In cases where software is built to serve different users, configuration testing analyzes the software under various specified configurations.

- Usability and human computer interaction testing [9*, c6]

The main task of usability testing is to evaluate how easy it is for end users to use and learn the software. In general, it may involve the user documentation, the software functions in supporting user tasks, and the ability to recover from user errors. Specific attention is devoted to validating the software interface (human-computer interaction testing) (see *User Interface Design* in the Software Design KA).

- Test-driven development [1*, c1s16]

Test-driven development (TDD) originated as one of the core XP (extreme programming) practices and essentially consists of writing automated unit tests prior to the code under test (see also *Agile Methods* in the Software Engineering Models and Method KA). In this way, TDD promotes the use of tests as a surrogate for a requirements specification document rather than as an independent check that the software has correctly implemented the requirements. TDD is more a specification and programming practice than a testing strategy.

3. Test Techniques

One of the aims of testing is to reveal as much potential for failure as possible, and many techniques have been developed to do this. These techniques attempt to “break” the program by running one or more tests drawn from identified classes of executions deemed equivalent. The leading principle underlying such techniques is to be as systematic as possible in identifying a representative set of program behaviors; for instance, considering subclasses of the input domain, scenarios, states, and dataflow.

It is difficult to find a homogeneous basis for classifying all techniques, and the one used here must be seen as a compromise. The classification is based on how tests are generated: from the software engineer’s intuition and experience, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage, or, finally, the nature of the application. Sometimes these techniques are classified as *white-box* (also called *glass-box)* if the tests rely on information about how the software has been designed or coded, or as *black-box* if the test cases rely only on the input/output behavior. One last category deals with the combined use of two or more techniques. Obviously, these techniques are not used equally often by all practitioners. Included in the list are those that a software engineer should know.

3.1. Based on the software engineer’s intuition and experience

- Ad hoc

Perhaps the most widely practiced technique remains ad hoc testing: tests are derived relying on the software engineer’s skill, intuition, and experience with similar programs. Ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques.

- Exploratory testing

Exploratory testing is defined as simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible faults and failures, the risk associated with a particular product, and so on.

3.2. Input domain-based techniques

- Equivalence partitioning [1*, c9s4]

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

- Pairwise testing [1*, c9s3]

Test cases are derived by combining interesting values for every pair of a set of input variables instead of considering all possible combinations. Pairwise testing belongs to *combinatorial testing,* which in general also includes higher-level combinations than pairs: these techniques are referred to as *t-wise*, whereby every possible combination of *t* input variables is considered.

- Boundary-value analysis [1*, c9s5]

Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs. An extension of this technique is *robustness testing*, wherein test cases are also chosen outside the input domain of variables to test program robustness to unexpected or erroneous inputs.

- Random testing [1*, c9s7]

Tests are generated purely at random (not to be confused with statistical testing from the operational profile, as described in sub-topic 3.5 *Operational profile*). This form of testing falls under the heading of the input domain entry since the input domain (at least) must be known in order to be able to pick random points within it. Random testing provides a relatively simple approach to test automation; recently, enhanced forms have been proposed in which the random test sampling is directed by other input selection criteria [10].

3.3. Code-based techniques

- Control-flow-based criteria [1*, c4]

Control-flow-based coverage criteria are aimed at covering all the statements, blocks of statements, or specified combinations of statements in a program. Several coverage criteria have been proposed, like condition/decision coverage and modified condition/decision coverage. The strongest of the control-flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in the flowgraph. Since path testing is generally not feasible because of loops, other less stringent criteria tend to be used in practice—such as statement, branch, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage is said to have been achieved.

- Data-flow-based criteria [1*, c5]

In data-flow-based testing, the control flowgraph is annotated with information about how the program variables are defined, used, and killed (undefined). The strongest criterion, all definition-use paths, requires that, for each variable, every control-flow path segment from a definition of that variable to a use of that definition is executed. In order to reduce the number of paths required, weaker strategies such as all-definitions and all-uses are employed.

- Reference models for code-based testing (flowgraph, call graph) [1*, c4]

Although not a technique in itself, the control structure of a program is graphically represented using a flowgraph in code-based testing techniques. A flowgraph is a directed graph the nodes and arcs of which correspond to program elements (see *Graphs and Trees* in the Mathematical Foundations KA). For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs may represent the transfer of control between nodes.

3.4. Fault-based techniques [1, c1s14]*

With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or predefined faults. To better focus the test case generation or selection, a *fault model* could be introduced that classifies the different types of faults.

- Error guessing [1*, c9s8]

In error guessing, test cases are specifically designed by software engineers trying to figure out the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the software engineer's expertise.

- Mutation testing [1*, c3s5]

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 sub-topic 4.2. *Evaluation of the tests performed*), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving 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, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically derived in a systematic way [11].

3.5. *Usage-based techniques*

- Operational profile [1*. c15s5]

In testing for reliability evaluation, the test environment must reproduce the operational environment of the software as closely as possible. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their frequency of occurrence in actual operation. Operational profiles can be used during the system test for designing and guiding test case derivation. The purpose is to meet the reliability objectives and exercise relative usage and criticality of different functions in the field [5].

- User observation heuristics [9*, c5, c7].

Usability principles can be used as a guideline for checking and discovering a good proportion of problems in the user interface design [9*, c1s4])(see *User Interface Design* in the Software Design KA). Specialized heuristics, also called usability inspection methods, are applied for the systematic observation of system usage under controlled conditions in order to determine how people can use the system and its interfaces. Usability heuristics include cognitive walkthroughs, claims analysis, field observations, thinking-aloud, and even indirect approaches such as user's questionnaires and interviews.

3.6. *Model-based testing techniques*

Model-based testing refers to an abstract (formal) representation of the software under test or of its requirements (see *Modeling* in the Software Engineering Models and Methods KA). This model is used for validating requirements, checking their consistency, and generating test cases focused on the behavioral aspect of the software. The key components of these techniques are [12]: the notation used for representing the model of the software, the test strategy, or algorithm for test case generation; and the supporting infrastructure for the test execution, including the evaluation of the expected outputs. Due to the complexity of the adopted techniques, model-based testing approaches are often used in conjunction with test automation harnesses. Main techniques are listed in the following points.

- Decision table [1*, c9s6]

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 technique is *cause-effect graphing*. [1*, c13s6].

- Finite-state machine-based [1*, c10]

By modeling a program as a finite state machine, tests can be selected in order to cover states and transitions on it.

- Testing from formal specifications [1*, c10s11, 2*, c15]

Giving the specifications in a formal language (see also *Formal Methods* in the Software Engineering Models and Methods KA) allows for automatic derivation of functional test cases, and, at the same time, provides an oracle for checking test results.

TTCN3 (Testing and Test Control Notation version 3) is a language specifically developed for writing test cases. The notation was conceived for specific needs of testing telecommunication systems, so it is particularly suitable to test complex communication protocols.

3.7. *Techniques based on the nature of the application*

The above techniques apply to all types of software. However, for some kinds of applications, some additional know-how is required for test derivation. A list of a few specialized testing fields is provided here, based on the nature of the application under test:

- Object-oriented testing
- Component-based testing
- Web-based testing
- Testing of concurrent programs
- Protocol conformance testing
- Testing of real-time systems
- Testing of safety-critical systems
- Testing of service-oriented systems
- Testing of open-source systems
- Testing of embedded systems

3.8. *Selecting and combining techniques*

- Functional and structural [1*, c9]

Model-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 could be used in combination, depending on budgetary considerations.

- Deterministic vs. random [1*, c9s6]

Test cases can be selected in a deterministic way, according to one of the various techniques listed, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing. Several analytical and empirical comparisons have been conducted to analyze the conditions that make one approach more effective than the other.

4. Test-Related Measures

Sometimes test techniques are confused with test objectives. 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 the objective of testing per se: 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-related measures that provide an evaluation of the program under test based on the observed test outputs and those that evaluate the thoroughness of the test set. (See *Software engineering measurement* in the Software Engineering Management KA for information on measurement programs. See *Process and product measurement* in the Software Engineering Process KA for information on measures).

Measurement is usually considered instrumental to quality analysis. Measurement may also be used to optimize the planning and execution of the tests. Test management can use several process measures to monitor progress. Measures relative to the test process for management purposes are considered in topic 5.1 *Practical considerations*.

4.1. Evaluation of the program under test

- Program measurements to aid in planning and designing testing [13*, c11]

Measures based on program size (for example, source lines of code or function points (see *Measuring Requirements* in the Software Requirements KA)) or on program structure (like complexity) are used to guide testing. Structural measures can also include measurements among program modules in terms of the frequency with which modules call each other.

- Fault types, classification, and statistics [13*, c4]

The testing literature is rich in classifications and taxonomies of faults. To make testing more effective, it is important to know which types of faults could be found in the software under test and the relative frequency with which these faults have occurred in the past. This information can be very useful in making quality predictions as well as in process improvement (see *Defect characterization* in the Software Quality KA).

- Fault density [1*, c13s4, 13*, c4]

A program under test can be assessed by counting and classifying the discovered faults by their types. For each fault class, fault density is measured as the ratio between the number of faults found and the size of the program.

- Life test, reliability evaluation [1*, c15, 13*, c3]

A statistical estimate of software reliability, which can be obtained by reliability achievement and evaluation (see sub-topic 2.2), can be used to evaluate a product and decide whether or not testing can be stopped.

- Reliability growth models [1*, c15, 13*, c8]

Reliability growth models provide a prediction of reliability based on failures. They assume, in general, that when the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), the estimated product's reliability exhibits, on average, an increasing trend. There now exist dozens of published models. Many are laid down on some common assumptions while others differ. Notably, these models are divided into *failure-count* and *time-between-failure* models.

4.2. Evaluation of the tests performed

- Coverage/thoroughness measures [13*, c11]

Several test adequacy criteria require that the test cases systematically exercise a set of elements identified in the program or in the specifications (see subarea 3 *Test Techniques*). To evaluate the thoroughness of the executed tests, testers can monitor the elements covered so that they can dynamically measure the ratio between covered elements and their total number. For example, it is possible to measure the percentage of covered branches in the program flowgraph or that of the functional requirements exercised among those listed in the specifications document. Code-based adequacy criteria require appropriate instrumentation of the program under test.

- Fault seeding [1*, c2s5, 13*, c6]

Some faults are artificially introduced into the program before testing. When the tests are executed, some of these seeded faults will be revealed as well as, possibly, some faults that were already. In theory, depending on which and how many of the artificial faults are discovered, testing effectiveness can be evaluated and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of seeded faults relative to genuine faults and the small sample size on which any extrapolations are based. Some also argue that this technique should be used with great care since inserting faults into software involves the obvious risk of leaving them there.

- Mutation score [1*, c3s5]

In mutation testing (see sub-topic 3.4 *Fault-based techniques*), the ratio of killed mutants to 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

Several studies have been conducted to compare the relative effectiveness of different test techniques. It is important to be precise as to the property against which the techniques are being assessed; what, for instance, is the exact meaning given to the term "effectiveness"? Possible interpretations include the number of tests needed to find the first failure, the ratio of the number of faults found through testing to all the faults found during and after testing, and how much reliability was improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the notions of effectiveness specified above.

5. Test Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process that is run by people. The test process supports testing activities and provides guidance to testing teams, from test planning to test output evaluation, in such a way as to provide justified assurance that the test objectives will be met in a cost –effective way.

5.1. Practical considerations

- Attitudes/Egoless programming [1*c16, 13*, c15]

A very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development and maintenance; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel responsible for failures revealed by their code.

- Test guides [1*, c12s1, 13*, c15s1]

The testing phases could be guided by various aims—for example, risk-based testing uses the product risks to prioritize and focus the test strategy, and scenario-based testing defines test cases based on specified software scenarios.

- Test process management [1*, c12, 13*, c15]

Test activities conducted at different levels (see subarea 2 *Test Levels*) must be organized—together with people, tools, policies, and measurements—into a well-defined process that is an integral part of the life cycle.

- Test documentation and work products [1*, c8s12, 13*, c4s5]

Documentation is an integral part of the formalization of the test process. Test documents may include, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log, and Test Incident or Problem Report. The software under test is documented as the Test Item. Test documentation should be produced and continually updated to the same level of quality as other types of documentation in software engineering.

- Internal vs. independent test team [1*, c16]

Formalization of the test process may involve formalizing the test team organization as well. The test team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members (in the hope of bringing an unbiased, independent perspective), or, finally, of both internal and external members. Considerations of cost, schedule, maturity levels of the involved organizations, and criticality of the application may determine the decision.

- Cost/effort estimation and other process measures [1*, c18s3, 13*, c5s7]

Several measures related to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various 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, and number of test cases failed, among others.

Evaluation of test phase reports can be combined with root-cause analysis to evaluate test-process effectiveness in finding faults as early as possible. Such an evaluation could be associated with the analysis of risks. Moreover, the resources that are worth spending on testing should be commensurate with the use/criticality of the application: different techniques have different costs and yield different levels of confidence in product reliability.

- Termination [13*, c10s4]

A decision must be made as to 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 in themselves. The decision also involves considerations about the costs and risks incurred by possible remaining failures, as opposed to the costs incurred by continuing to test. (See "Test selection criteria/Test adequacy criteria" in 1.2 *Key issues*).

- Test reuse and test patterns [13*, c2s5]

To carry out testing or maintenance in an organized and cost-effective way, the means used to test each part of the software should be reused systematically. This repository of test materials must be under the control of software configuration management so that changes to software requirements or design can be reflected in changes to the tests conducted.

The test solutions adopted for testing some application types 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.

5.2. *Test activities*

Under this topic, a brief overview of test activities is given; as often implied by the following description, successful management of test activities strongly depends on the software-configuration management process (see the Software Configuration Management KA).

- Planning [1*, c12s1, c12s8]

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

- Test-case generation [1*, c12s1, c12s3]

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

- Test environment development [1*, c12s6]

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

- Execution [1*, c12s7]

Execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the results. Hence, testing should be performed in accordance with documented procedures using a clearly defined version of the software under test.

- Test results evaluation [13*, c15]

The results of testing must be evaluated to determine whether or not the test has been successful. In most cases, “successful” means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults, however, but could be judged as simply noise. Before a fault can be removed, an 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 them.

- Problem reporting/Test log [1*, c13s9]

Testing activities can be entered into a test log to identify when a test was conducted, who performed the test, what software configuration was the basis for testing, and other relevant identification information. Unexpected or incorrect test results can be recorded in a problem-reporting system, the data of which form the basis for later debugging and fixing the problems that 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 first thought. Test reports are also an input to the change-management request process (see *Software configuration control* in the Software Configuration Management KA).

- Defect tracking [13*, c9]

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

6. Software Testing Tools

6.1. *Testing tool support* [1*, c12s11, 13*, c5]

Testing requires fulfilling many labor-intensive tasks, running numerous executions, and handling a great amount of information. Appropriate tools can alleviate the burden of clerical, tedious operations and make them less error-prone. Sophisticated tools can support test design, making it more effective.

- Selecting tools [1*, c12s11]

Guidance to managers and testers on how to select those tools that will be most useful to their organization and processes is a very important topic, as tool selection greatly affects testing efficiency and effectiveness. Tool selection depends on diverse evidence, such as development choices, evaluation objectives, execution facilities, and so on. In general, there may not be a unique tool satisfying all needs and a suite of tools could be the most appropriate choice.

6.2. *Categories of tools*

We categorize the available tools according to their functionality. In particular:

- Test harnesses (drivers, stubs) [1*, c3s9] provide a controlled environment in which tests can be launched and the test outputs can be logged. In order to execute parts of a software, drivers and stubs are provided to simulate caller and called modules, respectively.
- Test generators [1*, c12s11] provides assistance in the generation of tests. The generation can be random, pathwise (based on the flowgraph), model-based, or a mix thereof.
- Capture/Replay tools [1*, c12s11] automatically re-execute, or replay, previously run tests, which have recorded inputs and outputs (e.g., screens).

- Oracle/File comparators/Assertion checking [1*, c9s7] assist in deciding whether a test outcome is successful or faulty.
- Coverage analyzer & Instrumenter [1*, c4] work together. Coverage analyzers assess which and how many entities of the program flowgraph have been exercised amongst all those required by the selected coverage-testing criterion. The analysis can be done thanks to program instrumenters, which insert probes into the code.
- Tracers [1*, c1s7] trace the history of a program's execution.
- Regression testing tools [1*, c12s16] support the re-execution of a test suite after a software has been modified. They can also help to select a subset according to the change.
- Reliability evaluation tools [13*, c8] support test results analysis and graphical visualization in order to assess reliability-related measures according to selected models.

	[1*] Naik and Tripathy, 2008	[2*] Sommerville, 2011	[9*] Nielsen, 1993	[13*] Kan, 2003
1. **Software Testing Fundamentals**				
1.1 Testing-Related Terminology				
Definitions of testing and related terminology	c1,c2	c8		
Faults vs. failures	c1s5	c11		
1.2 Key Issues				
Test selection criteria/Test adequacy criteria (or stopping rules)	c1s14, c6s6, c12s7			
Testing effectiveness/Objectives for testing	c13s11, c11s4			
Testing for defect identification	c1s14			
The oracle problem	c1s9, c9s7			
Theoretical and practical limitations of testing	c2s7			
The problem of infeasible paths	c4s7			
Testability	c17s2			
1.3 Relationship of testing to other activities				
Testing vs. Static Software Quality Management Techniques	c12			
Testing vs. Correctness Proofs and Formal Verification	c17s2			
Testing vs. Debugging	c3s6			
Testing vs. Programming	c3s2			
2. Test Levels	c1s13	c8s1		
2.1 The Target of the Test	c1s13	c8s1		
Unit testing	c3	c8		
Integration testing	c7	c8		
System testing	c8	c8		
2.2 Objectives of Testing	c1s7			

	[1*] Naik and Tripathy, 2008	[2*] Sommerville, 2011	[9*] Nielsen, 1993	[13*] Kan, 2003
Acceptance/qualification	c1s7	c8s4		
Installation testing	c12s2			
Alpha and Beta testing	c13s7, c16s6	c8s4		
Reliability and evaluation achievement	c15	c15s2		
Regression testing	c8s11, c13s3			
Performance testing	c8s6			
Security testing	c8s3	c11s4		
Stress testing	c8s8			
Back-to-back testing				
Recovery testing	c14s2			
Configuration testing	c8s5			
Usability and human computer interaction testing			c6	
Test-driven development	c1s16			
3. Test Techniques				
3.1 Based on the software engineer's intuition and experience				
Ad hoc				
Exploratory testing				
3.2 Input domain-based techniques				
Equivalence partitioning	c9s4			
Pairwise testing	c9s3			
Boundary-value analysis	c9s5			
Random testing	c9s7			
3.3 Code-based techniques				
Control-flow-based criteria	c4			
Data flow-based criteria	c5			
Reference models for code-based testing (flowgraph, call graph)	c4			
3.4 Fault-based techniques	c1s14			
Error guessing	c9s8			
Mutation testing	c3s5			
3.5 Usage-based techniques				
Operational profile	c15s5			
User observation heuristics			c5, c7	
3.6 Model-based testing techniques				
Decision table	c9s6			
Finite-state machine-based	c10			

	[1*] Naik and Tripathy, 2008	[2*] Sommerville, 2011	[9*] Nielsen, 1993	[13*] Kan, 2003
Testing from formal specifications	c10s11	c15		
3.7 Techniques based on the nature of the application				
3.8 Selecting and combining techniques				
Functional and structural	c9			
Deterministic vs. random	c9s6			
4. Test-related measures				
4.1 Evaluation of the program under test				
Program measurements to aid in planning and designing testing	c12s8			c11
Fault types, classification, and statistics				c4
Fault density	c13s3			c4
Life test, reliability evaluation	c15			c3
Reliability growth models	c15			c8
4.2 Evaluation of the tests performed				
Coverage/thoroughness measures				c11
Fault seeding	c2s5			c6
Mutation score	c3s5			
Comparison and relative effectiveness of different techniques				
5 Test Process				
5.1 Practical considerations				
Attitudes/Egoless programming	c16			c15
Test guides	c12s1			c15s1
Test process management	c12			c15
Test documentation and work products	c8s12			c4s5
Internal vs. independent test team	c16			
Cost/effort estimation and other process measures	c18s3			c5s7
Termination				c10s4
Test reuse and patterns				c2s5
5.2 Test Activities				
Planning	c12s1 c12s8			
Test-case generation	c12s1 c12s3			
Test environment development	c12s6			
Execution	c12s7			
Test results evaluation				c15
Problem reporting/Test log	c13s9			
Defect tracking				c9

	[1*] Naik and Tripathy, 2008	[2*] Sommerville, 2011	[9*] Nielsen, 1993	[13*] Kan, 2003
6. Software Testing Tools				
6.1 Testing tool support	c12s11			c5
Selecting Tools	c12s11			
6.2 Categories of Tools				
Test harness	c3s9			
Test generators	c12s11			
Capture/Replay	c12s11			
Oracle/file comparators/assertion checking	c9s7			
Coverage analyzer/Instrumenter	c4			
Tracers	c1s7			
Regression testing tools	c12s16			
Reliability evaluation tools				c8

[1*] S. Naik and P. Tripathy, "Software Testing and Quality Assurance: Theory and Practice," ed: Wiley, 2008, p. 648.

[2*] I. Sommerville, *Software Engineering*, 9th ed. New York: Addison-Wesley, 2010.

[3] IEEE/ISO/IEC, "IEEE/ISO/IEC 24765: Systems and Software Engineering - Vocabulary," 1st ed, 2010.

[4] ISO/IEC/IEEE, "Draft Standard P29119-1/DIS for Software and Systems Engineering--Software Testing--Part 1: Concepts and Definitions," ed, 2012.

[5] M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*. IEEE Computer Society Press, McGraw-Hill, 1996.

[6] H. Zhu*, et al.*, "Software unit test coverage and adequacy," *Acm Computing Surveys,* vol. 29, pp. 366-427, Dec 1997.

[7] E. W. Dijkstra, "Notes on Structured Programming," Technological University, Eindhoven1970.

[8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing Verification & Reliability,* vol. 22, pp. 67-120, Mar 2012.

[9*] J. Nielsen, *Usability Engineering*, 1st ed. Boston: Morgan Kaufmann, 1993.

[10] T. Y. Chen*, et al.*, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software,* vol. 83, pp. 60-66, Jan 2010.

[11] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *Ieee Transactions on Software Engineering,* vol. 37, pp. 649-678, Sep-Oct 2011.

[12] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*: Morgan Kaufmann, 2007.

[13*] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston: Addison-Wesley, 2002.