INTERNATIONAL STANDARD

ISO/IEC/IEEE 29119-1

Second edition
2022-01

Software and systems engineering — Software testing —

Part 1: General concepts

Ingénierie du logiciel et des systèmes — Essais du logiciel —

Partie 1: Concepts généraux

Reference number
ISO/IEC/IEEE 29119-1:2022(E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO/IEC documents should be noted. This document was drafted in accordance with the rules given in the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see https://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

ISO/IEC/IEEE 29119-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 7, *Software and systems engineering*, in cooperation with the Systems and Software Engineering Standards Committee of the IEEE Computer Society, under the Partner Standards Development Organization cooperation agreement between ISO and IEEE.

This second edition cancels and replaces the first edition (ISO/IEC/IEEE 29119-1:2013), which has been technically revised.

The main changes are as follows:

— Testing terms and their definitions that are not covered within this document have been removed. This has led to this document being renamed from ‘Concepts and definitions’ to ‘General concepts’.

— The coverage of test concepts has been made more concise and re-ordered.

— The concept of test sub-processes has been removed due to its complexity and replaced with additional coverage of the instantiation of test processes.

— The expected content of a test strategy has been clarified.

— A simplified test design process is described, with the derivation of test cases now based on test models rather than on test conditions.

— The coverage of metrics and measures has been moved from an annex into the body of the document.

— The annex explaining how testing fits into different life cycle models has been removed.

— A new annex providing examples of how systems from different domains are associated with certain characteristics and test approaches has been added.

A list of all parts in the ISO/IEC/IEEE 29119 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The purpose of the ISO/IEC/IEEE 29119 series is to define an internationally agreed set of standards for software testing that can be used by any organization when performing any form of software testing and using any life cycle.

It is recognized that there are many different types of software, software organizations, and methodologies. Software domains include information technology (IT), personal computers (PC), embedded, mobile, scientific and many other classifications. Software organizations range from small to large, co-located to world-wide, and commercial to those providing a public service. Software development methodologies include object-oriented, traditional, agile and DevOps. These and other factors influence software testing. The ISO/IEC/IEEE 29119 series can support testing in many different contexts.

This document facilitates the use of other parts in the ISO/IEC/IEEE 29119 series by introducing the general concepts on which the ISO/IEC/IEEE 29119 series is built.

A general introduction to software testing is provided. The role of software testing in quality management and as part of verification and validation is described; and its implementation in the form of both static and dynamic testing is defined. The impracticality of exhaustive testing and the need for sampling are explained; and the importance of the test basis and test oracle are described. The benefits of test independence are introduced.

Test plans and test strategies are described in the context of risk-based testing, which is the recommended approach to strategizing and managing testing that underlies the ISO/IEC/IEEE 29119 series and provides the basis for test prioritization and focus. Test levels, test types and test design techniques (and corresponding measures) are described in the context of their inclusion as part of the test strategy.

Various test frameworks are presented, including test processes (and test process improvement), test metrics, test documentation, configuration management and tool support.

The performance of test design and execution based on the use of a test model is described. Several of the most important test design and execution choices are considered, including scripted and exploratory testing approaches, the importance of test design techniques for the creation of test cases, test patterns, retesting and regression testing, manual and automated testing, back-to-back and A/B testing.

Several activities that directly support test design and executions are introduced, including test environments, test data management, communications and reporting and defect and incident management.

Annex A briefly describes a number of system characteristics and suggested associated test approaches. If a tester can identify which of the system characteristics apply to the system they are testing, then they should consider whether the specialized testing listed for the characteristic is appropriate for inclusion in their test strategy.

Annex B introduces several generic testing roles and briefly describes their responsibilities.

The test process model that the ISO/IEC/IEEE 29119 series is based on is defined in detail in ISO/IEC/IEEE 29119-2. ISO/IEC/IEEE 29119-2 covers the software testing processes at the organizational level, test management level and for dynamic test levels. Testing is the primary approach to risk treatment in software development. This document defines a risk-based approach to testing.

Templates and examples of test documentation that are produced during the testing process are defined in ISO/IEC/IEEE 29119-3. Software testing techniques that can be used during testing are defined in ISO/IEC/IEEE 29119-4.

While this document is informative, ISO/IEC/IEEE 29119-2, ISO/IEC/IEEE 29119-3 and ISO/IEC/IEEE 29119-4 are normative, meaning that they include requirements for anyone wanting to claim conformance to these standards. Users who want to use the standards but have good reasons for not following every requirement (e.g. for someone following an agile approach to development and testing) can claim tailored conformance as long as the level of tailoring and its rationale are described and agreed. Specific details of conformance are provided in the relevant conformance clause in each of the standards.

The ISO/IEC/IEEE 29119 series can be used in isolation or can be used as part of a larger set of standards that cover other aspects of the software life cycle. For instance, some users use ISO/IEC/IEEE 12207 to define software system life cycle models appropriate to their products and services (and some may use the corresponding systems engineering standard, ISO/IEC/IEEE 15288), and reference the ISO/IEC/IEEE 29119 series for their software testing needs.

Together, the ISO/IEC/IEEE 29119 series aims to provide stakeholders with the ability to manage and perform software testing in any organization.

Software and systems engineering — Software testing —

Part 1: General concepts

1 Scope

This document specifies general concepts in software testing and presents key concepts for the ISO/IEC/IEEE 29119 series.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO, IEC and IEEE maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

— IEEE Standards Dictionary Online: available at https://dictionary.ieee.org

NOTE For additional terms and definitions in the field of systems and software engineering, see ISO/IEC/IEEE 24765, which is published periodically as a "snapshot" of the SEVOCAB (Systems and software Engineering Vocabulary) and is publicly accessible at https://www.computer.org/sevocab.

3.1
A/B testing
split-run testing
statistical *testing* (3.131) approach that allows testers to determine which of two systems or components performs better

3.2
accessibility testing
type of usability *testing* (3.131) used to measure the degree to which a *test item* (3.107) can be operated by users with the widest possible range of characteristics and abilities

3.3
activation function
transfer function
<*artificial intelligence* (3.7)> formula associated with a node in a *neural network* (3.50) that determines the output of the node (*activation value* (3.4)) from the inputs to the neuron

3.4
activation value
<*artificial intelligence* (3.7)> output of an *activation function* (3.3) of a node in a *neural network* (3.50)

3.5
actual result
set of behaviours or conditions of a *test item* (3.107), or set of conditions of associated data or the *test environment* (3.95), observed as a result of *test execution* (3.99)

EXAMPLE Outputs to screen, outputs to hardware, changes to data, reports, and communication messages sent.

3.6
AI-based system
system including one or more components implementing *AI* (3.7)

3.7
artificial intelligence
AI
capability of an engineered system to acquire, process and apply knowledge and skills

3.8
autonomous system
<*artificial intelligence* (3.7)> system capable of working without human intervention for sustained periods

3.9
autonomy
<*artificial intelligence* (3.7)> ability of a system to work for sustained periods without human intervention

3.10
back-to-back testing
differential testing
approach to *testing* (3.131) whereby an alternative version of the system is used to generate *expected results* (3.35) for comparison from the same test inputs

EXAMPLE The alternative version can be a system that already exists, a system developed by an independent team or a system implemented using a different programming language.

3.11
base choice
base value
input parameter value that is normally selected based on being a representative or typical value for the parameter

3.12
boundary value analysis
specification-based *test design technique* (3.94) based on exercising the boundaries of *equivalence partitions* (3.30)

3.13
branch testing
structure-based *test case* (3.85) design technique based on exercising branches in the *control flow* (3.22) of the *test item* (3.107)

3.14
cause-effect graph
graphical representation of *decision rules* (3.25) between causes (inputs described as Boolean *conditions* (3.21)) and effects (outputs described as Boolean expressions)

3.15
cause-effect graphing
specification-based *test design technique* (3.94) based on exercising *decision rules* (3.25) in a *cause-effect graph* (3.14)

3.16
classification
<*artificial intelligence* (3.7)> *machine learning* (3.44) function that predicts the output class for a given input

3.17
classification tree
hierarchical tree model of the input data to a program in which the inputs are represented by distinct classifications (relevant test aspects) and classes (input values)

3.18
combinatorial testing
combinatorial test design techniques
class of specification-based *test design techniques* (3.94) based on exercising combinations of *P-V pairs* (3.56)

EXAMPLE *Pairwise testing* (3.57), *base choice* (3.11) testing.

3.19
compatibility testing
type of *testing* (3.131) that measures the degree to which a *test item* (3.107) can function satisfactorily alongside other independent products in a shared environment (co-existence), and where necessary, exchanges information with other systems or components (interoperability)

3.20
completion criteria
conditions under which the *testing* (3.131) activities are considered complete

3.21
condition
Boolean expression containing no Boolean operators

EXAMPLE "A < B" is a condition but "A and B" is not.

3.22
control flow
sequence in which operations are performed during the execution of a *test item* (3.107)

3.23
decision
type of statement in which a choice between two or more possible outcomes controls which set of actions will result

Note 1 to entry: Typical decisions are simple selections (e.g. if-then-else), to decide when to exit loops (e.g. while-loop), and in case (switch) statements (e.g. case-1–2-3-...-N).

3.24
decision outcome
result of a *decision* (3.23) that determines the branch to be executed

3.25
decision rule
combination of *conditions* (3.21) (also known as causes) and actions (also known as effects) that produce a specific outcome in *decision table testing* (3.27) and *cause-effect graphing* (3.15)

3.26
decision table
tabular representation of *decision rules* (3.25) between causes (inputs described as Boolean *conditions* (3.21)) and effects (outputs described as Boolean expressions)

3.27
decision table testing
specification-based *test design technique* (3.94) based on exercising *decision rules* (3.25) in a *decision table* (3.26)

3.28
decision testing
structure-based *test case* (3.85) design technique based on exercising *decision outcomes* (3.24) in the *control flow* (3.22) of the *test item* (3.107)

3.29
dynamic testing
testing (3.131) in which a *test item* (3.107) is evaluated by executing it

3.30
equivalence partition
equivalence class
class of inputs or outputs that are expected to be treated similarly by the *test item* (3.107)

3.31
equivalence partitioning
test design technique (3.94) in which *test cases* (3.85) are designed to exercise *equivalence partitions* (3.30) by using one or more representative members of each partition

3.32
error guessing
test design technique (3.94) in which *test cases* (3.85) are derived on the basis of the tester's knowledge of past failures, or general knowledge of failure modes

Note 1 to entry: The relevant knowledge can be gained from personal experience, or can be encapsulated in, for example, a defects database or a "bug taxonomy".

3.33
executable statement
statement which, when compiled, is translated into object code, which will be executed procedurally when the *test item* (3.107) is running and may perform an action on program data

3.34
exhaustive testing
test approach (3.83) in which all combinations of input values and preconditions are tested

Note 1 to entry: In nearly all non-trivial situations, exhaustive testing is impossible, due to the large number of possible tests.

3.35
expected result
observable predicted behaviour of the *test item* (3.107) under specified conditions based on its specification or another source

3.36
experience-based testing
class of *test case* (3.85) design techniques based on using the experience of testers to generate test cases

EXAMPLE *Error guessing* (3.32).

Note 1 to entry: Experience-based testing can include concepts such as test attacks, tours, and error taxonomies which target potential problems such as security, performance, and other quality areas.

3.37
exploratory testing
experience-based testing (3.36) in which the tester spontaneously designs and executes tests based on the tester's existing relevant knowledge, prior exploration of the *test item* (3.107) (including the results of previous tests), and heuristic "rules of thumb" regarding common software behaviours and types of failure

Note 1 to entry: Exploratory testing hunts for hidden properties (including hidden behaviours) that, while quite possibly benign by themselves, can interfere with other properties of the software under test, and so constitute a risk that the software will fail.

3.38
fuzz testing
<*artificial intelligence* (3.7)> software *testing* (3.131) approach in which high volumes of random (or near random) data, called fuzz, are used to generate inputs to the *test item* (3.107)

3.39
incident
anomalous or unexpected event, set of events, condition, or situation at any time during the life cycle of a project, product, service, or system

3.40
incident report
documentation of the occurrence, nature, and status of an *incident* (3.39)

Note 1 to entry: Incident reports are also known as anomaly reports, bug reports, defect reports, error reports, issues, problem reports and trouble reports, amongst other terms.

3.41
keyword
<*keyword-driven testing* (3.42)> one or more words used as a reference to a specific set of actions intended to be performed during the execution of one or more *test cases* (3.85)

Note 1 to entry: The actions include interactions with the User Interface during the test, verification, and specific actions to set up a *test scenario* (3.123).

Note 2 to entry: Keywords are named using at least one verb.

Note 3 to entry: Composite keywords can be constructed based on other keywords.

3.42
keyword-driven testing
testing (3.131) using *test cases* (3.85) composed from *keywords* (3.41)

3.43
load testing
type of *performance testing* (3.58) conducted to evaluate the behaviour of a *test item* (3.107) under anticipated conditions of varying load, usually between anticipated conditions of low, typical, and peak usage

3.44
machine learning
ML
process using computational techniques to enable systems to learn from data or experience

3.45
maintainability testing
test type (3.130) conducted to evaluate the degree of effectiveness and efficiency with which a *test item* (3.107) may be modified

3.46
manual testing
humans performing tests by entering information into a *test item* (3.107) and verifying the results

Note 1 to entry: Automated *testing* (3.131) uses tools, *robots* (3.70), and other *test execution engines* (3.100) to perform tests. Manual testing does not use these items.

3.47
MC/DC testing
modified condition decision testing
structure-based *test case* (3.85) design technique based on demonstrating that a single Boolean *condition* (3.21) within a *decision* (3.23) can independently affect the outcome of the decision

3.48
metamorphic relation
description of how changes to the test inputs for a *test case* (3.85*)* affect the expected outputs based on the required behaviour of a *test item* (3.107)

3.49
metamorphic testing
specification-based *test case* (3.85) design technique based on generating test cases based on existing test cases and *metamorphic relations* (3.48)

3.50
neural network
artificial neural network
<*artificial intelligence* (3.7)> network of primitive processing elements connected by weighted links with adjustable weights, in which each element produces a value by applying a nonlinear function to its input values, and transmits it to other elements or presents it as an output value

Note 1 to entry: Whereas some neural networks are intended to simulate the functioning of neurons in the nervous system, most neural networks are used in artificial intelligence as realizations of the connectionist model.

Note 2 to entry: Examples of nonlinear functions are a threshold function, a sigmoid function, and a polynomial function.

[SOURCE: ISO/IEC 2382:2015, 2120625, modified — The admitted term "neural net" has been removed; notes 3 to 5 to entry have been removed.]

3.51
neuron coverage
<*artificial intelligence* (3.7)> proportion of activated neurons divided by the total number of neurons in the *neural network* (3.50) (normally expressed as a percentage) for a set of tests

Note 1 to entry: A neuron is considered to be activated if its *activation value* (3.4) exceeds zero.

3.52
non-deterministic system
system which, given a particular set of inputs and starting state, will not always produce the same set of outputs and final state

3.53
organizational test practices
documentation that expresses the recommended approaches or methods for the *testing* (3.131) to be performed within an organization, providing detail on how the testing is to be performed

Note 1 to entry: The organizational test practices are aligned with the *organizational test policy* (3.118).

Note 2 to entry: An organization can have more than one organizational test practices document to cover markedly different contexts, such one for mobile apps and one for *safety* (3.71) critical systems.

Note 3 to entry: The organizational test practices can incorporate the context of the test policy where no separate test policy is available

3.54
organizational test process
test process (3.121) for developing and managing *organizational test specifications* (3.55)

3.55
organizational test specification
document that provides information about *testing* (3.131) for an organization, i.e. information that is not project specific

EXAMPLE The most common examples of organizational test specifications are the *organizational test policy* (3.118) and the *organizational test practices* (3.53).

3.56
P-V pair
parameter-value pair
combination of a *test item* (3.107) parameter with a value assigned to that parameter, used as a *test coverage item* (3.90)

3.57
pairwise testing
black-box *test design technique* (3.94) in which *test cases* (3.85) are designed to execute all possible discrete combinations of each pair of input parameters

Note 1 to entry: Pairwise testing is the most popular form of *combinatorial testing* (3.18).

3.58
performance testing
type of *testing* (3.131) conducted to evaluate the degree to which a *test item* (3.107) accomplishes its designated functions within given constraints of time and other resources

3.59
portability testing
type of *testing* (3.131) conducted to evaluate the ease with which a *test item* (3.107) can be transferred from one hardware or software environment to another, including the level of modification needed for it to be executed in various types of environment

3.60
procedure testing
type of functional suitability *testing* (3.131) conducted to evaluate whether procedural instructions for interacting with a *test item* (3.107) or using its outputs meet user requirements and support the purpose of their use

3.61
product risk
risk that a product can be defective in some specific aspect of its function, quality, or structure

3.62
project risk
risk related to the management of a project

EXAMPLE Lack of staffing, strict deadlines, changing requirements.

3.63
random testing
specification-based *test design technique* (3.94) based on generating *test cases* (3.85) to exercise randomly selected *test item* (3.107) inputs

3.64
regression testing
testing (3.131) performed following modifications to a *test item* (3.107) or to its operational environment, to identify whether failures in unmodified parts of the test item occur

Note 1 to entry: Regression testing differs from *retesting* (3.68) in that it does not test that the modification works correctly, but that other parts of the system have not been accidentally affected by the change.

Note 2 to entry: The adequacy of a set of regression *test cases* (3.85) depends on the item under test and on the modifications to that item or its operational environment.

3.65
regulatory standard
standard promulgated by a regulatory agency

3.66
reliability testing
type of *testing* (3.131) conducted to evaluate the ability of a *test item* (3.107) to perform its required functions, including evaluating the frequency with which failures occur, when it is used under stated conditions for a specified period of time

3.67
requirements-based testing
specification-based *test case* (3.85) design technique based on exercising atomic requirements

EXAMPLE An atomic requirement can be 'The system shall collect and store the date of birth of all registered users.'

3.68
retesting
confirmation testing
testing (3.131) performed to check that modifications made to correct a fault have successfully removed the fault

Note 1 to entry: When retesting is performed often it is also complemented by *regression testing* (3.64), to help ensure that other unmodified parts of the *test item* (3.107) have not been accidentally adversely affected by the modifications.

3.69
risk-based testing
testing (3.131) in which the management, selection, prioritization, and use of testing activities and resources are consciously based on corresponding types and levels of analysed risk

3.70
robot
<*artificial intelligence* (3.7)> programmed actuated mechanism with a degree of *autonomy* (3.9), moving within its environment, to perform intended tasks

Note 1 to entry: A robot includes the control system and interface of the control system.

Note 2 to entry: The classification of robot into industrial robot or service robot is done according to its intended application.

3.71
safety
expectation that a system does not, under defined conditions, lead to a state in which human life, health, property, or the environment is endangered

[SOURCE: ISO/IEC/IEEE 12207:2017, 3.1.48]

3.72
scenario testing
specification-based *test case* (3.85) design technique based on exercising sequences of interactions between the *test item* (3.107) and other systems

Note 1 to entry: Users are considered to be other systems in this context.

3.73
scripted testing
testing (3.131) performed based on a documented *test script* (3.124)

Note 1 to entry: This term normally applies to manually executed testing, rather than the execution of an automated script.

3.74
security testing
test type (3.130) conducted to evaluate the degree to which a *test item* (3.107), and associated data and information, are protected so that unauthorized persons or systems cannot use, read, or modify them, and authorized persons or systems are not denied access to them

3.75
specification-based testing
black-box testing
closed box testing
testing (3.131) in which the principal *test basis* (3.84) is the external inputs and outputs of the *test item* (3.107), commonly based on a specification, rather than its implementation in source code or executable software

3.76
state transition testing
specification-based *test case* (3.85) design technique based on exercising transitions in a state model

EXAMPLE Example state models are state transition diagram and state table.

3.77
statement testing
structure-based *test case* (3.85) design technique based on exercising *executable statements* (3.33) in the source code of the *test item* (3.107)

3.78
static testing
testing (3.131) in which a *test item* (3.107) is examined against a set of quality or other criteria without the test item being executed

EXAMPLE Reviews, static analysis.

3.79
stress testing
type of performance efficiency *testing* (3.131) conducted to evaluate a *test item's* (3.107) behaviour under conditions of loading above anticipated or specified capacity requirements, or of resource availability below minimum specified requirements

3.80
structure-based testing
white box testing
glass-box testing
structural testing
dynamic testing (3.29) in which the tests are derived from an examination of the structure of the *test item* (3.107)

Note 1 to entry: Structure-based testing is not restricted to use at component level and can be used at all levels, e.g. menu item coverage as part of a system test.

Note 2 to entry: Techniques include *branch testing* (3.13), *decision testing* (3.28), and *statement testing* (3.77).

3.81
suspension criteria
criteria used to (temporarily) stop all or a portion of the *testing* (3.131) activities

3.82
test
activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component

3.83
test approach
high-level test implementation choice, typically made as part of the *test strategy* (3.127) design activity

EXAMPLE 1 The use of model-based *testing* (3.131) for the functional system testing.

EXAMPLE 2 Typical choices made as test approaches are *test level* (3.108), *test type* (3.130), *test technique* (3.94), *test practice* (3.119) and the form of *static testing* (3.78) to be used.

3.84
test basis
information used as the basis for designing and implementing *test cases* (3.85)

Note 1 to entry: The test basis may take the form of documentation, such as a requirements specification, design specification, or module specification, but may also be an undocumented understanding of the required behaviour.

3.85
test case
set of preconditions, inputs and *expected results* (3.35), developed to drive the execution of a *test item* (3.107) to meet *test objectives* (3.114)

Note 1 to entry: A test case is the lowest level of test implementation documentation (i.e. test cases are not made up of test cases) for the *test level* (3.108) or *test type* (3.130) for which it is intended.

Note 2 to entry: Test case preconditions include the required state of the *test environment* (3.95), data (e.g. databases) used by the test item, and the test item itself.

Note 3 to entry: Inputs are the data information and actions, where applicable, used to drive *test execution* (3.99).

3.86
test completion process
test management process (3.110) that aims to ensure that useful test assets are made available for later use, *test environments* (3.95) are left in a satisfactory condition, and the results of *testing* (3.131) are recorded and communicated to relevant stakeholders

3.87
test completion report
test summary report
report that provides a summary of the *testing* (3.131) that was performed

3.88
test condition
testable aspect of a component or system, such as a function, transaction, feature, quality attribute, or structural element identified as a basis for *testing* (3.131)

Note 1 to entry: The ISO/IEC/IEEE 29119 series does not use the concept of test conditions, but instead uses the concept of a *test model* (3.111) for test design.

3.89
test coverage
degree, expressed as a percentage, to which specified *test coverage items* (3.90) have been exercised by a *test case* (3.85) or test cases

3.90
test coverage item
coverage item
measurable attribute of a *test item* (3.107) that is the focus of *testing* (3.131)

EXAMPLE *Equivalence partitions* (3.30), transitions between states, *executable statements* (3.33).

3.91
test data
data created or selected to satisfy the input requirements for executing one or more *test cases* (3.85)

Note 1 to entry: Test data can be stored within the *test item* (3.107) (e.g. in arrays or flat files), or can come from external sources, such as other systems, hardware devices, or human operators.

3.92
test data readiness report
document describing the status of each *test data* (3.91) requirement

3.93
test design and implementation process
test process (3.121) for deriving and specifying *test cases* (3.85) and *test procedures* (3.120)

3.94
test design technique
test technique
procedure used to create or select a *test model* (3.111), identify *test coverage items* (3.90), and derive corresponding *test cases* (3.85)

EXAMPLE *Equivalence partitioning* (3.31), *boundary value analysis* (3.12), *decision table testing* (3.27), *branch testing* (3.13).

Note 1 to entry: The test design technique is typically used to achieve a required level of *test coverage* (3.89).

Note 2 to entry: Some *test practices* (3.119), such as *exploratory testing* (3.37) or model-based *testing* (3.131) are sometimes referred to as "test techniques". Following the definition in the ISO/IEC/IEEE 29119 series, they are not test design techniques as they are not themselves providing a way to create test cases, but instead use test design techniques to achieve that.

3.95
test environment
environment containing facilities, hardware, software, firmware, procedures, needed to conduct a test

Note 1 to entry: A test environment can contain multiple environments to accommodate specific *test level* (3.108) or *test types* (3.130) (e.g. a unit test environment, a performance test environment).

Note 2 to entry: A test environment can comprise several interconnected systems or virtual environments.

3.96
test environment and data management process
test process (3.121) for establishing and maintaining a required *test environment* (3.95) and corresponding *test data* (3.91)

3.97
test environment readiness report
document that describes the status of the *test environment* (3.95)

Note 1 to entry: This can list the status of each of the *test environment requirements* (3.98).

3.98
test environment requirements
description of the necessary properties of the *test environment* (3.95)

Note 1 to entry: All or parts of the test environment requirements can reference where the information can be found, e.g. in the *organizational test practices* (3.53) document, *test plan* (3.117), and *test specification* (3.125).

3.99
test execution
process of running a *test* (3.82) on the *test item* (3.107), producing *actual results* (3.5)

3.100
test execution engine
<*keyword-driven testing* (3.42)> tool implemented in software and sometimes in hardware that can manipulate the *test item* (3.107) to execute *test cases* (3.85)

Note 1 to entry: A typical test execution engine includes unit test tool frameworks, stimulation-command systems, capture and playback tools or hardware *robots* (3.70) along with the software to control them.

3.101
test execution log
record of the execution of one or more *test procedures* (3.120)

3.102
test execution process
dynamic *test process* (3.121) for executing *test procedures* (3.120) created in the *test design and implementation process* (3.93) in the prepared *test environment* (3.95), and recording the results

3.103
test framework
structured set of principles, guidelines, and practices used for organizing, selecting and communicating *testing* (3.131)

3.104
test incident
event occurring during the execution of a *test* (3.82) that requires investigation

3.105
test incident reporting process
dynamic *test process* (3.121) for reporting *incidents* (3.39) requiring further action that were identified during the *test execution process* (3.102) to the relevant stakeholders

3.106
test independence
degree to which those performing *testing* (3.131) have separate responsibilities from those developing the *test item* (3.107)

3.107
test item
test object
work product to be tested

EXAMPLE Software component, system, requirements document, design specification, user guide.

3.108
test level
one of a sequence of test stages each of which is typically associated with the achievement of particular objectives and used to treat particular risks

EXAMPLE The following are common test levels, listed sequentially: unit/component *testing* (3.131), integration testing, system testing, system integration testing, acceptance testing.

Note 1 to entry: It is not always necessary for a *test item* (3.107) to be tested at all test levels, but the sequence of test levels generally stays the same.

Note 2 to entry: Typical objectives can include consideration of basic functionality for unit/component testing, interaction between integrated components for integration testing, acceptability to end users for acceptance testing.

3.109
test management
planning, scheduling, estimating, monitoring, reporting, control, and completion of test activities

3.110
test management process
process used to coordinate, monitor, and control *testing* (3.131)

Note 1 to entry: See *test strategy and planning process* (3.128), *test monitoring and control process* (3.113), *test completion process* (3.86).

3.111
test model
representation of the *test item* (3.107), which allows the *testing* (3.131) to be focused on particular characteristics or qualities

EXAMPLE Requirements statements, of *equivalence partitions* (3.30), state transition diagram, use case description, *decision table* (3.26), input syntax description, source code, *control flow* (3.22) graph, parameters and values, *classification tree* (3.17), natural language.

Note 1 to entry: The test model and the required *test coverage* (3.89) are used to identify *test coverage items* (3.90).

Note 2 to entry: A separate test model can be required for each different type of required test coverage included in the test *completion criteria* (3.20).

Note 3 to entry: A test model can include one or more *test conditions* (3.88).

Note 4 to entry: Test models are commonly used to support test design (e.g. they are used to support test design in ISO/IEC/IEEE 29119-4, and they are used in model-based testing). Other types of models exist to support other aspects of testing, such as *test environment* (3.95) models, test maturity models and test architecture models.

3.112
test model specification
document specifying the *test model* (3.111)

3.113
test monitoring and control process
test management process (3.110) that aims to ensure that *testing* (3.131) is performed in line with a *test plan* (3.117) and with *organizational test specifications* (3.55)

3.114
test objective
reason for performing *testing* (3.131)

EXAMPLE Checking for correct implementation, identification of defects, measuring quality.

3.115
test oracle
source of information for determining whether a test has passed or failed

Note 1 to entry: The test oracle is often a specification used to generate *expected results* (3.35) for individual *test cases* (3.85), but other sources may be used, such as comparing *actual results* (3.5) with those of another similar program or system or asking a human expert.

3.116
test oracle problem
challenge of determining whether a *test* (3.82) has passed or failed for a given set of test inputs and state

3.117
test plan
detailed description of *test objectives* (3.114) to be achieved and the means and schedule for achieving them, organized to coordinate *testing* (3.131) activities for some *test item* (3.107) or set of test items

Note 1 to entry: A project can have more than one test plan, for example there can be a project test plan (also known as a master test plan) that encompasses all testing activities on the project; further detail of particular test activities can be defined in one or more *test level* (3.108)/*test type* (3.130) plans (e.g. a system test plan or a performance test plan).

Note 2 to entry: A test plan is typically a written document, although other formats can be possible as defined locally within an organization or project.

Note 3 to entry: Test plans can also be written for non-project activities, for example a maintenance test plan.

3.118
test policy
organizational test policy
executive-level document that describes the purpose, goals, principles, and scope of *testing* (3.131) within an organization

Note 1 to entry: The test policy defines what testing is performed and what it is expected to achieve but does not detail how testing is to be performed.

Note 2 to entry: The test policy can provide a framework for establishing, reviewing, and continually improving the organization's testing.

3.119
test practice
conceptual framework that can be applied to the *organizational test process* (3.54), the *test management process* (3.110), and/or the dynamic *test process* (3.121) to facilitate *testing* (3.131)

3.120
test procedure
sequence of *test cases* (3.85) in execution order, with any associated actions required to set up preconditions and perform wrap up activities post execution

3.121
test process
set of *testing* (3.131) activities performed to achieve a *test objective* (3.114)

Note 1 to entry: The test process for a particular project can consist of multiple *test levels* (3.108) and *test types* (3.130).

3.122
test result
indication of whether or not a specific *test case* (3.85) has passed or failed, i.e. if the *actual results* (3.5) corresponds to the *expected results* (3.35) or if deviations were observed

3.123
test scenario
situation or setting for a *test item* (3.107) used as the basis for generating *test cases* (3.85)

3.124
test script
test procedure specification
document specifying one or more *test procedures* (3.120)

3.125
test specification
complete documentation of the test design, *test cases* (3.85) and *test procedures* (3.120) for a specific *test item* (3.107)

Note 1 to entry: A test specification can be detailed in one document, in a set of documents, or in other ways, for example in a mixture of documents and database entries.

3.126
test status report
report that provides information about the status of the *testing* (3.131) that is being performed in a specified reporting period

3.127
test strategy
part of the *test plan* (3.117) that describes the approach to *testing* (3.131) for a specific project, *test level* (3.108) or *test type* (3.130)

Note 1 to entry: The test strategy usually describes some or all of the following: the test levels and test types to be implemented; the *retesting* (3.68) and *regression testing* (3.64) to be employed; the *test design techniques* (3.94) and corresponding test *completion criteria* (3.20) to be used; *test data* (3.91); *test environment* (3.95) and testing tool requirements; and expectations for test deliverables.

3.128
test strategy and planning process
test management process (3.110) used to design the *test strategy* (3.127), complete test planning, and create and maintain *test plans* (3.117)

3.129
test suite
set of *test cases* (3.85) or *test procedures* (3.120)

Note 1 to entry: The grouping into a test suite is typically based on when tests are executed.

3.130
test type
testing (3.131) that is focused on specific quality characteristics

EXAMPLE *Security testing* (3.74), functional testing, usability testing, and *performance testing* (3.58).

Note 1 to entry: A test type can be performed at a single *test level* (3.108) or across several test levels (e.g. performance testing performed at a unit test level and at a system test level).

3.131
testing
set of activities conducted to facilitate discovery and evaluation of properties of *test items* (3.107)

Note 1 to entry: Testing activities include planning, preparation, execution, reporting, and management activities, insofar as they are directed towards testing.

3.132
testware
artefacts produced during the *test process* (3.121) required to plan, design, and execute tests

Note 1 to entry: Testware can include such things as documentation, scripts, inputs, *expected results* (3.35), files, databases, environment, and any additional software or utilities used in the course of *testing* (3.131).

3.133
unscripted testing
dynamic testing (3.29) in which the tester's actions are not prescribed by written instructions in a *test case* (3.85)

4 Software testing concepts

4.1 Introduction to software testing

4.1.1 Overview

The ISO/IEC/IEEE 29119 series is based on a common understanding of software testing across a large part of the industry. To facilitate understanding of the ISO/IEC/IEEE 29119 series by all readers, this clause documents this common understanding. More detailed information on these fundamental topics can be found in textbooks, some of which are referenced in the bibliography.

4.1.2 Relationship to quality management

Quality management includes quality assurance (QA) and quality control (QC). Quality assurance is focused on the proper implementation of processes and process improvement, while quality control focuses on those activities supporting the achievement of appropriate levels of quality; testing is one form of quality control. Test results are used by QA and QC.

4.1.3 Verification and validation

Verification and validation are separate processes which both employ testing as one of their principal practices. Verification focuses on the conformance of a test item with specifications, specified requirements, or other documents, while validation focuses on the acceptability of the test item to meet the needs of the stakeholders, when used as intended. Testing (static and dynamic) supports both verification and validation, as shown in Figure 1. More detailed information on verification and validation can be found in IEEE 1012.

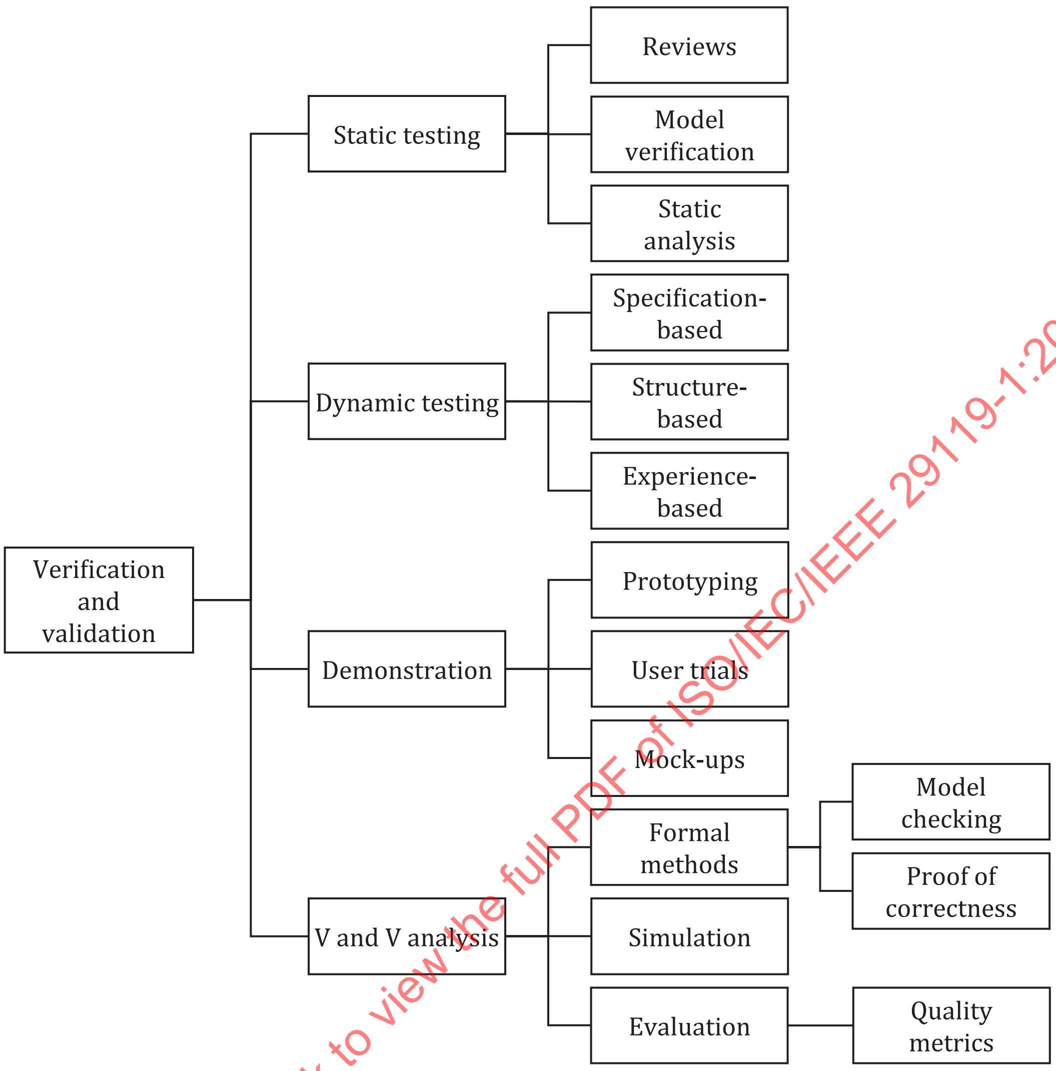

Figure 1 — Verification and validation methods or practices

4.1.4 Test item

The term “test item” describes a work product that is an object of testing. Other terms used for “test item” are “test object”, “software under test” and “system under test”.

A test item can be either a complete system or a part of a larger system, including hardware, software, and related documentation. A test item can be either executable (e.g. binary code, some models) or not executable (e.g. a documented specification). If the test item is executable, then dynamic testing can be performed, otherwise (it is not executable) only static testing can be performed by means of reviews, static analysis and model verification, as shown in Figure 1.

While the focus of the ISO/IEC/IEEE 29119 series is on software and related documents, many of the described concepts are also applicable to other test items, such as complete systems comprising both hardware and software.

4.1.5 Static and dynamic testing

Testing can take two forms: static and dynamic, as shown in Figure 1.

Static testing is evaluation of a test item where no execution of the code takes place and can be performed manually (e.g. reviews) or by using tools (e.g. static analysis). For static testing, the test item can take the form of documentation or source code, and it can be performed anywhere in the life cycle. Reviews and example review types are described in detail in ISO/IEC 20246. Reviews range in formality and include inspections, technical reviews, walkthroughs, and informal reviews. Static analysis involves the use of tools to detect anomalies in code or documents without execution (e.g. a compiler, a cyclomatic complexity analyser, or a security analyser for code).

Dynamic testing involves executing code and running test cases. Dynamic testing can thus only occur in the parts of the life cycle when executable code is available.

4.1.6 Exhaustive testing and sampling

To prove – by dynamic testing – that a specific test item meets all requirements under all given circumstances, then all possible combinations of input values in all possible states would need to be tested. This impractical activity is referred to as "exhaustive testing". For a simple software program, the number of combinations of input values is very large (e.g. a program processing two 64-bit numbers would require 2^{128} tests just to cover all possible input combinations). In practice, test items tend to be more and more complex, and so the application of exhaustive testing is not possible.

For that reason, in practice software testing derives test suites by sampling from the (extremely large) set of possible input combinations and states. Choosing the subset of possible tests that are most likely to uncover issues of interest is one of the most demanding tasks of a tester. The ISO/IEC/IEEE 29119 series recommends the use of risk-based testing (see 4.2.2) as the basis for selecting the tests to be used.

4.1.7 Testing as a heuristic

In engineering (and software engineering) a heuristic is an experience-based (trial and error) method that can be used as an aid to problem solving and design. However, while heuristics can be used to solve problems, they are fallible in that sometimes they may not solve, or only partially solve, a problem. Much of systems and software testing is based on heuristics. For instance, they are useful when creating models of the system to be tested; however, they may fail to fully model the system and so defects in the system may not be found even though the testing appears to be complete. Recognition that the manner in which testing is undertaken may be fallible makes it possible to partially treat the risk of an ineffective test approach by employing multiple approaches in the test strategy.

4.1.8 Purpose of testing

Testing usually serves more than one purpose. Typical purposes include, but are not restricted to:

a) detecting defects - this allows for their subsequent removal thus increasing software quality;

b) gathering information on the test item - testing generates information; this information can serve different purposes, such as:

- developers can use the information to remove defects, increase the code quality or learn to create better code in the future;
- testers can use the information to create better test cases;
- managers can use the information to decide when to stop testing;
- users eventually benefit from a higher product quality.

c) creating confidence and taking decisions - by providing evidence that the test item performs correctly under specific circumstances, the stakeholders' confidence that the test item will perform correctly operationally increases; with sufficient confidence, stakeholders can decide to release the test item.

Testing may be performed for some or all of the above purposes; and additional purposes not listed may also exist; these purposes should be identified and agreed as a starting point to any testing activity.

4.1.9 Test basis

The term "test basis" refers to that information used when developing tests and test cases. Typically, it is used to decide the focus of a test case, the test inputs, and the expected results; thus, the test basis defines the expected behaviour of the test item.

It is useful if the test basis is available as a single document or a set of documents to provide a common reference for test creation and review. In practice, the required information may only be available in an unstructured or informal way, such as verbally.

If there is missing or inadequate information test case analysis and design cannot be completed.

4.1.10 Test oracle

A test oracle is a source of information used to determine whether a test has passed or failed. A test oracle may be, but is not limited to:

— a requirements or design specification;

— another similar system (e.g. the system that is being replaced because it is too slow);

— a human expert or group of experts.

Test oracles do not always provide an expected result. In some situations, the test oracle comprises a constraint that defines the pass/fail criterion (e.g. actual result is greater than 10).

Test oracles can be complete (they provide all the information needed for a test item) or partial (they only provide the information needed for a subset of all test cases).

If test oracles are complete and formal, it should be possible to create an automated test oracle and subsequently automate the testing. In practice, this is rarely possible and human experts are needed as well; this is often known as the test oracle problem. A second part of the test oracle problem is when the complexity of the test item (e.g. an AI-based system analysing big data) makes it impossible for expert humans to know when a test item has passed or failed a test.

4.1.11 Test independence

Test independence describes the degree of separation between those performing testing and those developing the test item. Test independence is considered to be high if the person who performs testing is different from the person who performs the development and is considered to be low if the same person both develops and tests.

In general, a high degree of test independence is considered desirable for two reasons. The first reason is that an independent tester is less likely to make the same (wrong) assumptions as the developer and so is more likely to detect defects in the developer's work. The second reason is that a developer testing their own work may suffer from confirmation bias where they are likely to look for results that confirm that decisions they made earlier during development were correct.

On the other hand, some people argue that developers testing their own work is more efficient due to their familiarity with it. A compromise where testers and developers work together is also advocated, as is a test-first approach where developers write tests before they write code.

4.2 Test plans and test strategies

4.2.1 General

A test plan describes the objectives of the testing, and the activities to be performed to achieve those objectives. The choice of activities is described in the test strategy, which is part of the test plan (although it may be documented separately). The selection of activities is based on the risks associated with the objectives, which are largely concerned with meeting product and project requirements. Much of the test plan other than the test strategy describes the implementation of the test strategy, such as providing scheduling and staffing details.

The test plan and its test strategy can be applied to a whole project, one or more test levels, or one or more test types.

The activities in the test plan implement various test approaches, and these are selected to manage the perceived risks. The selection of these test approaches is not simple. In an ideal world, the higher exposure risks would simply be treated until an acceptable overall risk exposure level is reached. However, in practice the challenge is that risks are often interrelated. For instance, if it's decided to treat a product risk by performing more rigorous testing, the risk that the testing will overrun (perhaps in terms of both time and budget) is likely to be introduced (or increased). Thus, the selection of test approaches in the test strategy is typically a complex activity.

4.2.2 Risks and risk management

4.2.2.1 Risk categories

Risks can be categorized as either product or project risks.

Product risks are concerned with the deliverable product (the test item) and include possible harm to users or other stakeholders due to the product not performing as required.

Project risks are concerned with how the product is developed and include the danger that developers and other project members lack the necessary skills, or that the product will be delivered late or over budget.

4.2.2.2 Risk management process

The process for managing risks by testing (often called risk-based testing) is similar to most other risk management processes. Initially potential risks are identified, sometimes using checklists based on quality characteristics, such as those defined in the ISO/IEC 25000 SQuaRE family of standards. Next, they are analysed to determine the potential impact (severity) they would have (on a delivered product or the project) if they were to occur. The likelihood of each risk is determined, which can be based on factors such as requirement quality, staff capabilities, system complexity and historical information. A risk exposure level is then established, based on combining the impact and likelihood of each risk. Risks can then be prioritized accordingly, and mitigating actions decided, if appropriate (or possible) – always remembering that a treatment for one risk can be the cause, or increased exposure, of another risk.

4.2.3 Risks and requirements as the basis of a test strategy

The risk that the product does not meet stakeholder requirements is often a major driver in the selection of test approaches. There is sometimes confusion that the choice of test approaches is either based on risk or based on requirements. This is not a choice between alternatives – not meeting stakeholder requirements is typically one of the highest risks on any project – therefore choosing test approaches that help ensure requirements are met is a valid and vital part of a risk-based test strategy.

NOTE The requirements that can be considered include (and are not limited to) test item requirements, project requirements, regulatory requirements, organizational requirements, contractual requirements.

EXAMPLE A test strategy can be based on several test approaches and test practices. In a specific situation, test managers can decide to use unit testing, integration testing and system testing as levels of testing. Functional testing and load testing are the types of testing used. For functional testing, the test design techniques equivalence partitioning and boundary value analysis are chosen, along with the requirement to achieve a structural test coverage measure. Additionally, the test strategy specifies that the test practices scripted testing, exploratory testing and test automation are user in this situation. All these decisions are recorded in the test strategy which is intended to achieve the test goals identified by the test manager, and to address the perceived risks.

The selection of functional testing is often used to check the implementation of functional requirements, while non-functional types of testing (see Figure 2), such as performance and penetration testing are used to check that requirements in areas of performance and security are met.

4.2.4 Test approaches

4.2.4.1 General

Test approaches are selected to be part of the test strategy based on their ability to treat perceived risks. A wide range of the most common test approaches are shown in Figure 2. Each test strategy includes at least some of these categories of test approaches. For instance, in a project test strategy, a selection of test levels and test types is made. If dynamic testing is identified as a necessary risk treatment (it cannot be selected for managing risks earlier in the life cycle when no executables are available), then test design techniques and (normally) corresponding test completion measures (defined in terms of test coverage measures) are included in the test strategy.

4.2.4.2 Test levels

The testing for a project is performed at various distinct stages of the life cycle, typically referred to as test levels. Each test level is associated with a type of test item (e.g. module, system), has specific objectives and is used to treat specific risks. For instance, integration testing has the objective of getting software parts to work together and is used to treat risks associated with interfaces between test items and attempts to help ensure that they communicate correctly.

Test levels may be associated with static and dynamic testing and are closely related to development activities. For dynamic testing, each of the test levels typically requires a specific test environment. Common test levels, presented in the order they are typically performed and in order of the size of the test items from individual units or modules to complete systems:

— unit testing;

— integration testing;

— system testing;

— acceptance testing.

More test levels are shown in Figure 2.

4.2.4.3 Types of testing

The different types of testing are shown in Figure 2 and described in detail in ISO/IEC/IEEE 29119-4.

4.2.4.4 Test design techniques / measures

Test design techniques for creating test cases and test coverage measures used to measure the completeness of using the test design techniques are shown in Figure 2 and described in detail in ISO/IEC/IEEE 29119-4.

4.2.4.5 Test practices

A variety of test practices can be implemented as part of a test strategy. A list of more common test practices is given in Figure 2, although others are possible.

4.2.4.6 Static testing

In test plans, use of static testing helps form an optimal test strategy. Static analysis and reviews can be applied prior to dynamic testing and can find defects before test execution becomes possible. Most software projects use reviews (see ISO/IEC 20246) as an approach to identifying issues early in the lifecycle.

4.2.5 Testing in development and maintenance life cycles

The test processes, documentation and techniques defined in the ISO/IEC/IEEE 29119 series are generic and apply equally to both development and maintenance of software, and to agile and tradition life cycles. It is possible to tailor use of these standards dependent on the situation, for instance, a lean level of documentation may be specified on agile projects.

Testing in both development and maintenance should be based on the perceived risks. During maintenance, the risks will additionally need to take account of the changes made and the expected impact of those changes.

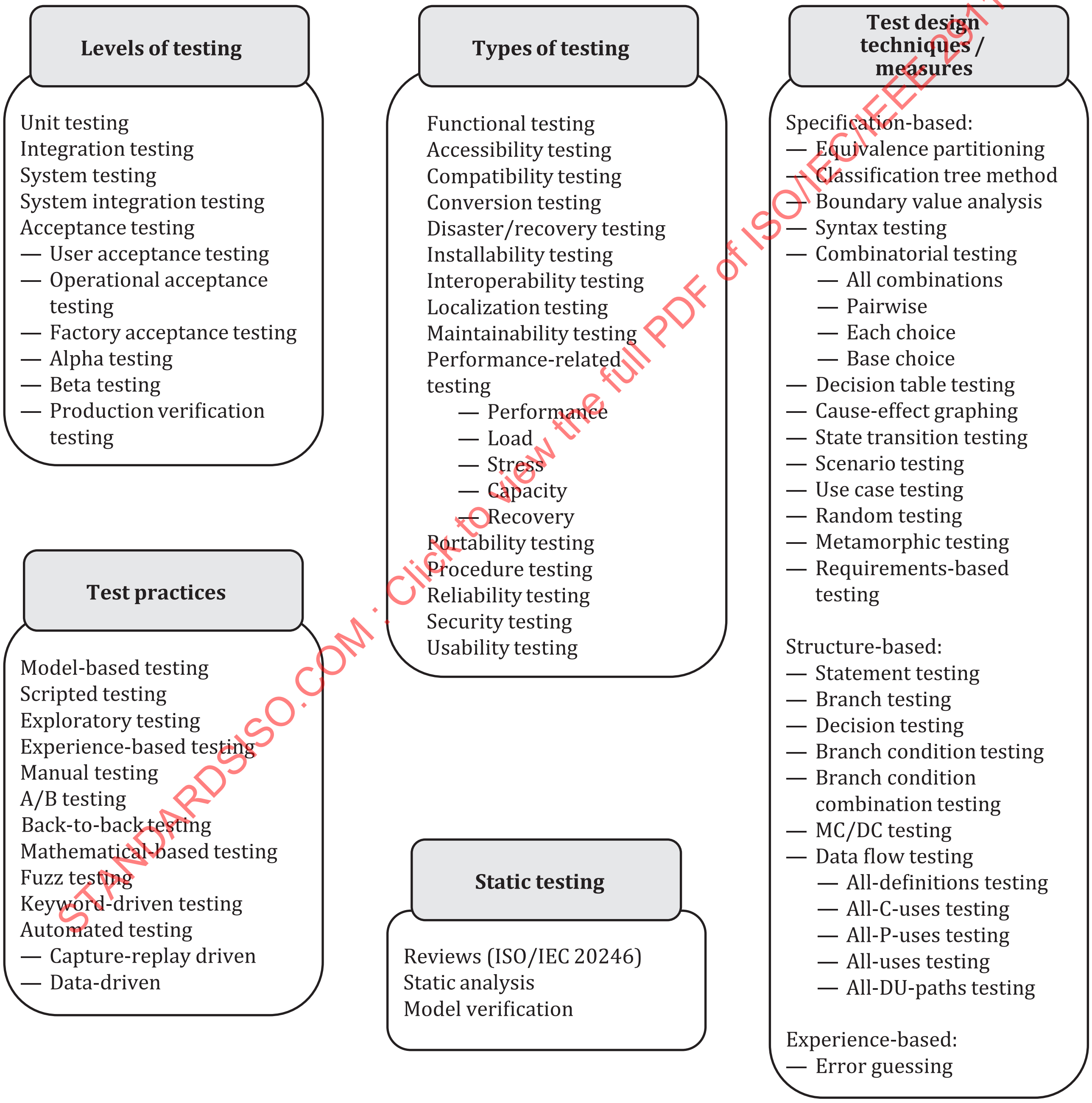

Figure 2 — Example test approach choices

4.2.6 Domains and system characteristics

The ISO/IEC/IEEE 29119 series is applicable to software from all domains, including, but not limited to, medical devices, commercial systems, defence systems, telecoms, and health. Systems from a specific domain share some similarities and these similarities can extend to the testing performed on them. However, given the wide range of systems from some domains (and that some systems cross two or more domains), it can be difficult to identify an appropriate set of testing approaches simply based on the domain.

Systems from a specific domain will typically share a mix of system characteristics. For instance, a future avionics system can include the following characteristics – embedded, real-time, autonomous and AI. As system characteristics are less abstract than domains, it is easier to associate specific testing approaches with these characteristics.

If the system characteristics associated with a given system are known, that suggests the types of testing that should be considered for inclusion in the test strategy (alongside those identified as a result of performing risk-based testing – see 4.2.2 for more on risk-based testing).

Annex A provides more detail on various system characteristics and common associated test approaches.

4.2.7 Test strategy contents

The expected contents of the test strategy are shown below:

— test approach choices (see Figure 2):

  — test levels (as part of a project test plan);

  — test types (sometimes only one test type if this part of a specialized test plan focused on a single system characteristic, such as usability);

  — selected test practices

  — selected static testing options

  — test design techniques (often selected based on team skills and familiarity, on the format of the test basis and perceptions of expected defect types based on experience);

— test deliverables (may be driven by regulatory requirements, where the risk being treated is partially not meeting mandated standards);

— test completion criteria (should match the test techniques, where selected, as these provide a means of knowing when enough tests have been executed);

— entry and exit criteria (for each of the test activities; these are typically used to manage the test process);

— degree of independence (see 4.1.11);

— metrics to be collected (often related to the test completion criteria, but may also be included to support test process improvement);

— test data requirements (more realistic data may be used to treat risks associated with data validity);

— test environment requirements (often selected by balancing the risks of using unrepresentative test environments with the costs of such environments, the required level of test automation should be specified here);

— retesting and regression testing (levels need to be based on a knowledge of the risks associated with developers making changes);

— suspension and resumption criteria (often defined to provide testers with an agreed set of conditions that developers need to meet to make testing worthwhile);

— deviations from the organizational test practices (and perhaps the test policy).

4.3 Test frameworks

4.3.1 Test processes

4.3.1.1 General

Processes describe a set of interrelated or interacting activities which transform inputs into outputs. It is often useful to consider the testing activities separately from the development (and other) activities, and these activities are described by test processes. In ISO/IEC/IEEE 29119-2, test processes are defined at three levels – at the organizational level, management level and dynamic testing level. The test processes defined in this document exhibit generic characteristics that can be used in a wide range of situations. Organizations and projects may adopt these generic test processes and supplement them with additional activities, procedures, and practices, as necessary.

4.3.1.2 Organizational level testing

Testing at the organizational level is concerned with defining and maintaining policy and test practices that apply across the whole organization, rather than to specific projects. The rules and guidance defined in the test policy and one or more organizational test practices documents serve as the basis for all the software testing being performed within the organization. The existence of such rules and guidance provide constraints on projects that remove the need for duplicated decision-making (e.g. the documented organizational test practices may mandate that a specific test automation framework will be used on all projects) and encourage the reuse of skills and communication between projects. The test policy and organizational test practices documents are usually seen in more mature organizations and larger organizations running multiple projects. Testing can be, and is, performed without a test policy and documented organizational test practices in organizations of lower maturity, but this gives less coherence to the testing within the organization and can make the testing less effective and efficient.

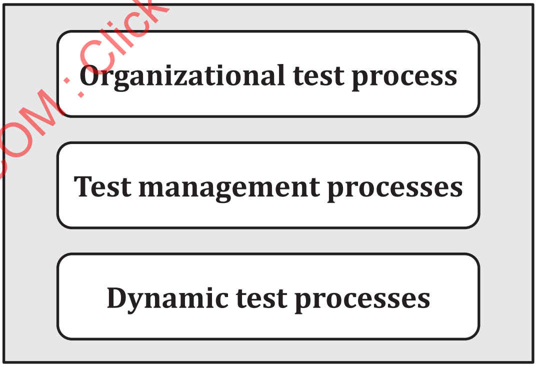

Figure 3 — The three levels of test processes

4.3.1.3 Project-based testing

When software is developed and maintained the stakeholders perform a set of interrelated processes typically defined by a life cycle model. The life cycle model acts as a common reference for communication and understanding for both the initial development and subsequent maintenance of the software.

The test processes used on a project can be mandated by regulatory standards (e.g. for safety-related software), by organizational constraints (e.g. in the organizational test practices document) and by the specific needs of the project.

The ISO/IEC/IEEE 29119 series defines project-based test processes at two levels: to support the management of testing and to support dynamic testing. Static testing techniques, such as reviews and static analysis are defined in separate standards (e.g. ISO/IEC 20246). Thus, overall, three levels of test processes are defined, as shown in Figure 3.

4.3.1.4 Test management processes

The management of the testing to be performed is described by the test management processes. Initially a test plan is created in the test strategy and planning process based on an analysis of identified risks and project constraints, and taking account of the organizational test practices document, where one exists. The test plan includes the project-specific test strategy, the staffing, and scheduling of the testing. The test plan drives the testing, which is managed through the test monitoring and control process. When testing is complete, the test completion process includes activities to generates a test completion report, clean up the test environment, archive testware and perform lessons learned, among others. The test management processes are shown in Figure 4.

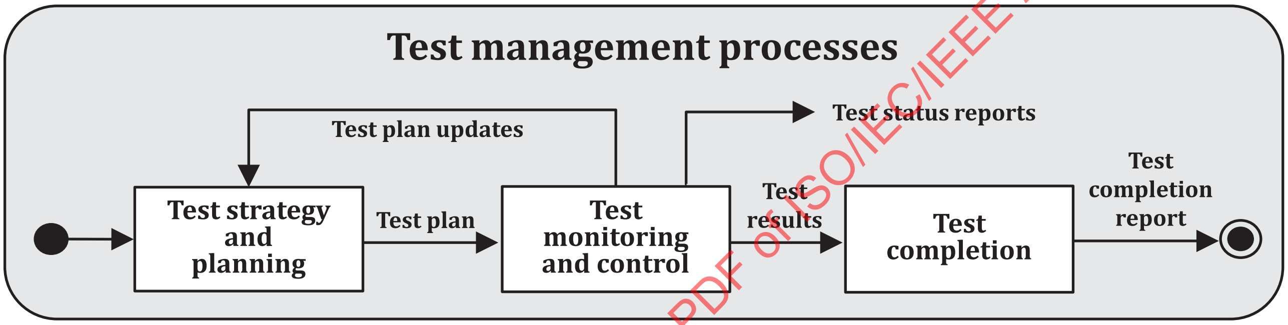

Figure 4 — Test management processes

4.3.1.5 Dynamic testing processes

The dynamic testing processes describe how the test cases are developed, the test environment is set-up, the test cases are executed, and any issues reported. The interaction of the dynamic testing processes is shown in Figure 5. Note that this figure only shows a logical ordering for a single test (these processes will be run many times for the testing on a project) and some processes may run more often than others (e.g. environment set-up will normally occur less frequently than test execution). Also, in practice, some of these processes may run in parallel (e.g. test design may be done at the same time as test environment set-up).

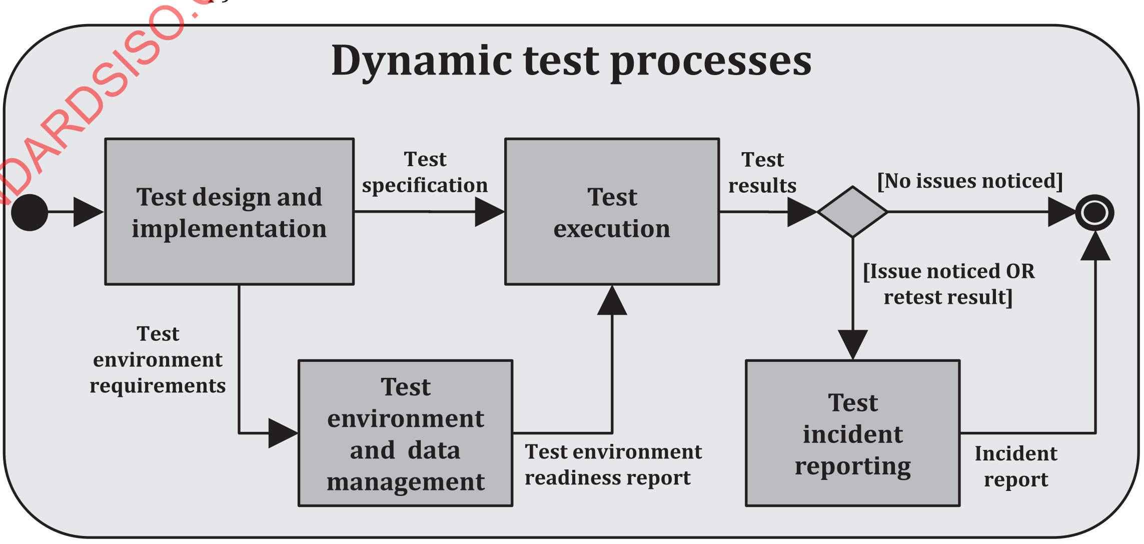
Dynamic test processes
Test design and implementation
Test specification
Test execution
Test results
[No issues noticed]
[Issue noticed OR retest result]
Test environment requirements
Test environment and data management
Test environment readiness report
Test incident reporting
Incident report

Figure 5 — Dynamic test processes

4.3.1.6 Instantiation of test processes

4.3.1.6.1 General

The eight test processes defined in ISO/IEC/IEEE 29119-2 need to be applied based on the situation, which will vary between organizations, between projects and within projects.

4.3.1.6.2 Organization level instantiation

At the organizational level, there is a single organizational test process, which can be applied for the development and maintenance of any organization-level test specification. Therefore, this process will be instantiated for each separate organization-level test specification that is developed and maintained. Typically, this will mean that the process is instantiated twice; once for the test policy and once for the organizational test practices document (organizations may have multiple organizational test practices documents where projects within the organization are dissimilar enough to warrant separate rules and guidelines, in which case each separate organizational test practices document will require a separate instantiation of the organizational test process).

4.3.1.6.3 Test management level instantiation

At the test management level, the three management level test processes (test strategy and planning, test monitoring and control, and test completion) can be applied to manage different levels and types of testing. For instance, they can be used to manage project level testing, component testing, integration testing, system testing, acceptance testing, usability testing, performance testing and security testing. Thus, on a single large project, it may be necessary to instantiate the three management level test processes (they are nearly always instantiated as a complete set of three) several times. On a small project it may only be necessary to instantiate them once.

For all projects, it is necessary to instantiate them to manage the project level testing. If it is decided that lower levels of testing (e.g. component testing, system testing) will be managed separately, then the test management processes will need to be instantiated for each of these levels that are managed separately. Similarly, if it is decided that a test type should be managed separately (e.g. usability testing, security testing) then, again, the test management processes will need to be instantiated for each of these types. One way of deciding if it is necessary to instantiate them is when it is decided that a separate test manager role is required for a specific test level or test type.

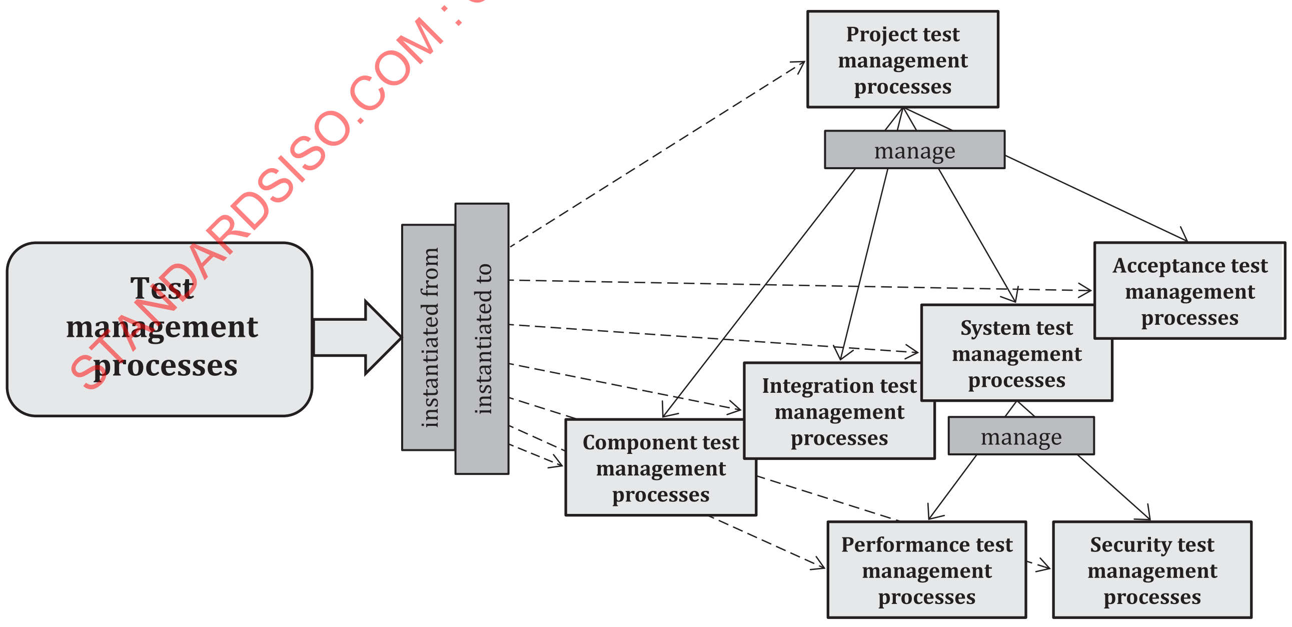
Test management processes
instantiated from
instantiated to
Project test management processes
manage
Acceptance test management processes
System test management processes
Integration test management processes
Component test management processes
manage
Performance test management processes
Security test management processes

Figure 6 — Hierarchy of instantiated test management processes

Where several test management processes are instantiated for a project, the processes instantiated for the project level testing are used to manage the other test management processes. It is also possible for lower level test processes to manage other test processes, so creating a hierarchy of test management, but this would only be applicable for very large projects. Such a hierarchy of test management processes is shown in Figure 6. All the test management processes in the hierarchy on the right-hand side are instantiated from the three base test management processes. The project test management processes would be the responsibility of the project test manager. Lower level sets of test management processes would typically be the responsibility of lower level test managers (e.g. a performance test manager would be responsible for the performance test management processes).

4.3.1.6.4 Dynamic testing level instantiation

At the dynamic testing level, the set of dynamic test processes will normally need to be instantiated many times. It is usual that they are separately instantiated for every different test level and test type (e.g. for component testing, integration testing, system testing, acceptance testing, performance testing, usability testing, security testing).

In practice, it is also common for them to be separately instantiated for the testing of different parts of a system (e.g. they can be separately instantiated for the different modules for component testing, or for the acceptance testing of the functions associated with a specific stakeholder). In this way the design and execution of the tests associated with a particular function can be performed separately from the design and execution of other functions. Figure 7 shows the dynamic test processes instantiated separately for each test level and separately for the component testing of three different modules.

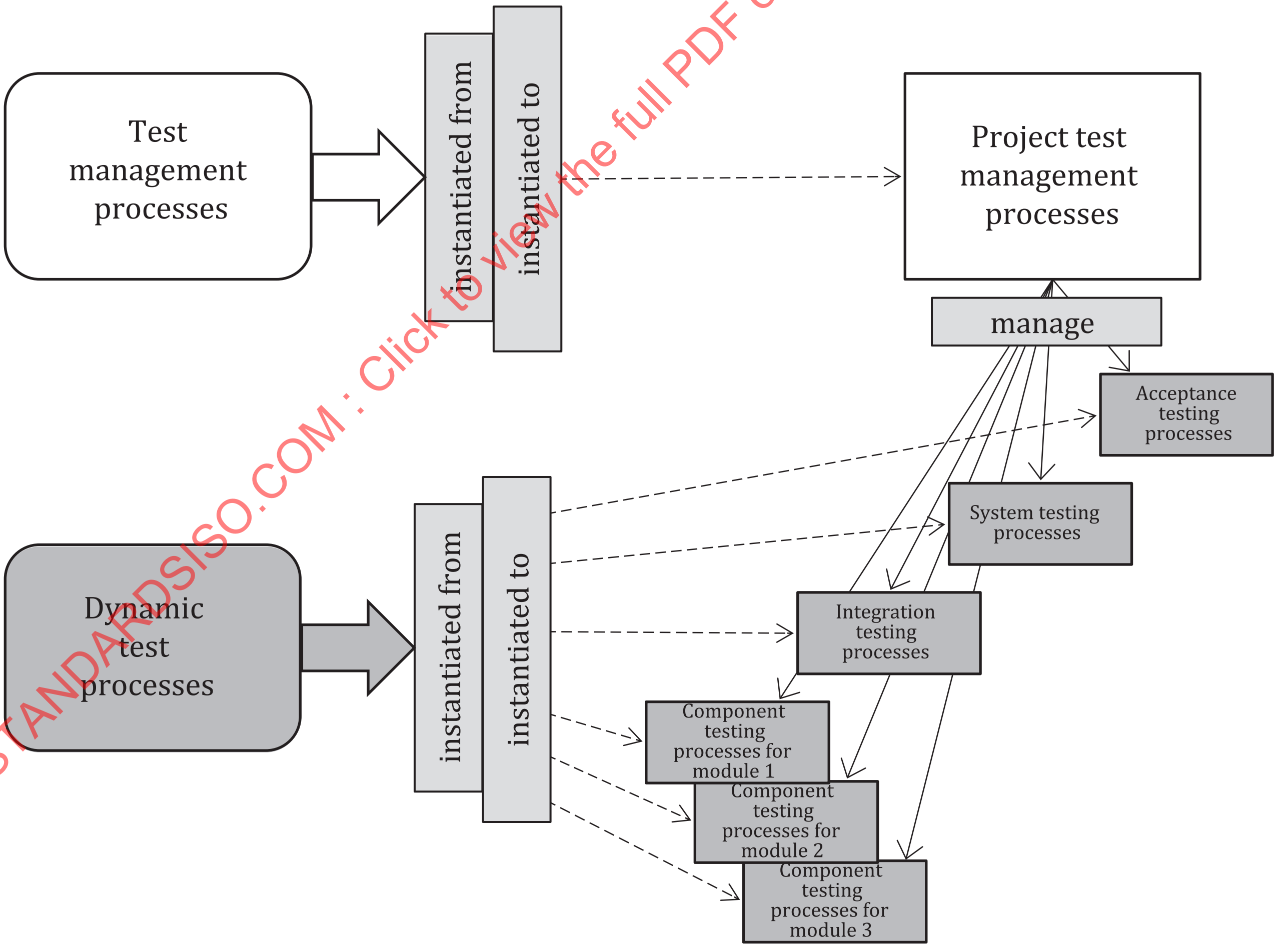
Test management processes
instantiated from
instantiated to
Project test management processes
manage
Acceptance testing processes
System testing processes
Integration testing processes
Dynamic test processes
instantiated from
instantiated to
Component testing processes for module 1
Component testing processes for module 2
Component testing processes for module 3

Figure 7 — Instantiation of multiple sets of dynamic test processes

4.3.2 Test documentation

4.3.2.1 General

Test documentation is produced as a result of performing the test processes and is defined in detail in ISO/IEC/IEEE 29119-3. The destination of these outputs is either a stakeholder (e.g. test status reports to a project manager) or another process (e.g. a test plan to direct the performance of dynamic testing). As the test documentation comes from the processes, it can be categorized in terms of outputs at the three test process levels.

4.3.2.2 Organization level documentation

4.3.2.2.1 General

The organizational test process develops and maintains organizational test specifications. These specifications typically take the form of either a test policy or an organizational test practices document.

4.3.2.2.2 Test policy

The test policy expresses the organization’s expectations and approach to software testing in business terms. Although it would also be useful to anyone involved with testing, it is aimed at executives and senior managers. The approach described in the test policy (what the organization wants to achieve) guides the content of the organizational test practices document, which describes how the organization will achieve it.

4.3.2.2.3 Organizational test practices

The organizational test practices document is aligned with the test policy and expresses requirements and constraints on how the testing should be performed on all the projects run within the organization (unless they are too dissimilar in character, in which case multiple organizational test practices documents may be generated).

4.3.2.3 Test management level documentation

4.3.2.3.1 General

Each of the three test management processes generates one main piece of test documentation. The test plan is generated as a result of test strategy and planning; test status reports are generated to document the progress of the testing effort by the test monitoring and control process; and the test completion report is generated to summarize the test effort from the test completion process.

4.3.2.3.2 Test plan

The test plan provides a detailed description of how the testing for the associated test management processes should be done. As test management processes can be instantiated to achieve a variety of goals (e.g. project test management, system test management, performance test management) the corresponding test plans also necessarily have a different focus depending on the reason for performing the testing. However, the types of information held in the test plan tend to remain the same whatever the form of testing that is being planned.

The test plan is used by the test monitoring and control process as the basis for managing the testing, but it is also used by the dynamic testing processes as it specifies how the dynamic testing should be performed (e.g. it specifies the required test techniques, test completion criteria, who should be performing the testing and when). The test plan is also used to specify how static testing (e.g. reviews and static analysis) should be performed. The test plan may be updated by the test monitoring and control process, for instance, due to changing requirements (e.g. test completion date moved forward) or when the specified testing cannot be done by the testers.

4.3.2.3.3 Test status report

Test status reports are generated by the test monitoring and control process to provide information on the current progress of the testing against the test plan. These reports may be used by the project manager to gauge progress or, if the test management is being performed at a lower level, they may be used to inform the project test manager of progress at this lower test level. test status reports are normally produced on a regular basis (as specified in the test plan), although the frequency of reporting may increase towards the end of the testing.

4.3.2.3.4 Test completion report

The test completion report summarizes the testing that was performed during the testing, details the testware that was archived and the state of the test environment, and describes the lessons learned from performing the testing. Where the test completion report is for a higher level of testing, it should also incorporate details of the testing performed at lower test levels (e.g. if it is a project test completion report and there was a separate system test plan then it should include details of the system testing, which should be available in the system test completion report). One test completion report is normally generated for the testing associated with each corresponding test plan.

4.3.2.4 Dynamic testing level documentation

4.3.2.4.1 General

The amount of test documentation produced by the dynamic testing processes can vary greatly as the requirements for documenting test case design, test environments and test results can be quite different for different forms of testing. The test documentation requirements should be specified in the test plan and, depending on the level of test automation, some of the documentation may be held in a test management tool.

4.3.2.4.2 Test specification

Test specifications should provide sufficient detail to determine how the test cases in the test procedures were derived from the test basis and that they achieve the required level of test coverage. Test specifications may take the form of a test procedure (supporting manual test execution) or an automated test script (supporting automated test execution). Traceability from the test cases back to the test basis is required to confirm coverage and to support the selection of tests to re-run when changes are made.

4.3.2.4.3 Test environment requirements and test data requirements

Where these are not already included in the test plan, any requirements for the test environment and test data are specified.

4.3.2.4.4 Test environment report and test data readiness report

The status of the test environment (e.g. availability) and test data are reported to the relevant stakeholders.

4.3.2.4.5 Test execution log

The execution of tests, including actual results, are recorded in the test logs.

4.3.2.4.6 Test incident report

Where necessary, incident reports (also known as defect reports) are generated to allow, when considered cost-effective, defects to be removed.

4.3.3 Documentation requirements

ISO/IEC/IEEE 29119-3 is extremely flexible in its documentation requirements. ISO/IEC/IEEE 29119-3 specifies the required information to be recorded, but it does not require specific naming or terminology to be used; nor does it require specific documents to be created (the information can be stored in multiple documents or combined into a single document). Documents do not have to paper-based, and the information can be held electronically or within test automation tools.

4.3.4 Configuration management and testing

Configuration management (defined in ISO 10007) is a set of activities used by the test processes at all three levels so that configuration items (e.g. test procedures, test scripts) are uniquely identified, changes to these items are recorded and the current status of configuration items can be reported. Configuration items of particular interest to testers are the inputs and outputs of the test processes and include all the test documentation mentioned in the previous subsections, along with items that are tested (e.g. systems and their components) and the corresponding test basis (e.g. requirements specification).

Where possible, for the purpose of being able to recreate a problem so it can be analysed further, the configuration management system should support the ability to repeat a test at any point in the future under the same conditions as before. It is acceptable practice to exclude certain types of testing - for example, unit testing - from this repeatability requirement as long as the exception is documented.

4.3.5 Tool support

Tool support (usually referred to as test tools) is available for many of the tasks and activities described in the test management and dynamic testing processes defined in ISO/IEC/IEEE 29119-2, as well as aspects of the testing techniques described in ISO/IEC/IEEE 29119-4. The following list provides examples of some of the areas covered by test tools:

— test case management;

— test monitoring and control;

— test data generation;

— static analysis;

— test case generation;

— test case execution;

— test environment implementation and maintenance.

There are a wide variety of test automation tools available. They can be developed in-house, obtained commercially, or obtained from the open-source community.

4.3.6 Process improvement and testing

Process improvement includes the identification and implementation of changes to an organization's processes so that they meet the organization's business goals more effectively and efficiently.

The test processes and process improvement interact in two ways:

a) the test processes deliver information on which process improvement changes can be based;

b) the test processes can themselves be subject to process improvement.

When testing delivers information to process improvement it typically comes from the lessons learned activity, which is part of the test completion process.

When test processes are improved, this can be part of a wider initiative covering all development, test and support processes or a more localized initiative targeted at the test processes.

4.3.7 Test metrics

To effectively control the test processes, metrics can be used to monitor them. Metrics should be selected with care, so that the information that is needed to control the processes is collected, and no measures are collected unnecessarily.

Some examples of metrics that can be used in testing are:

— residual risk: number of risks identified compared to number of risks treated;

— cumulative defects: number of defects opened compared to number of defects closed;

— test coverage: number of requirements covered by executed tests;

— defect detection percentage: number of defects found in testing compared to number of defects found overall.

4.4 Test design and execution

4.4.1 Test model

The test model is used as the basis of test design – it models the required behaviour of the test item and is used to generate test cases that cover the model to the required level of coverage. Figure 8 shows the relationships between the major test artefacts involved in test design, showing the test model as a key part.

The notation used for the test model is selected by considering the required test coverage; this indicates the form of the test coverage items – and these are incorporated into the test model. For instance, imagine the required test coverage is to exercise all valid transitions between states that the test item can be in. The test coverage items are these valid transitions and so a test model needs to be chosen that clearly shows these transitions. At this point the tester chooses between the different notations available for state models (e.g. they can choose a state transition diagram or a state table).

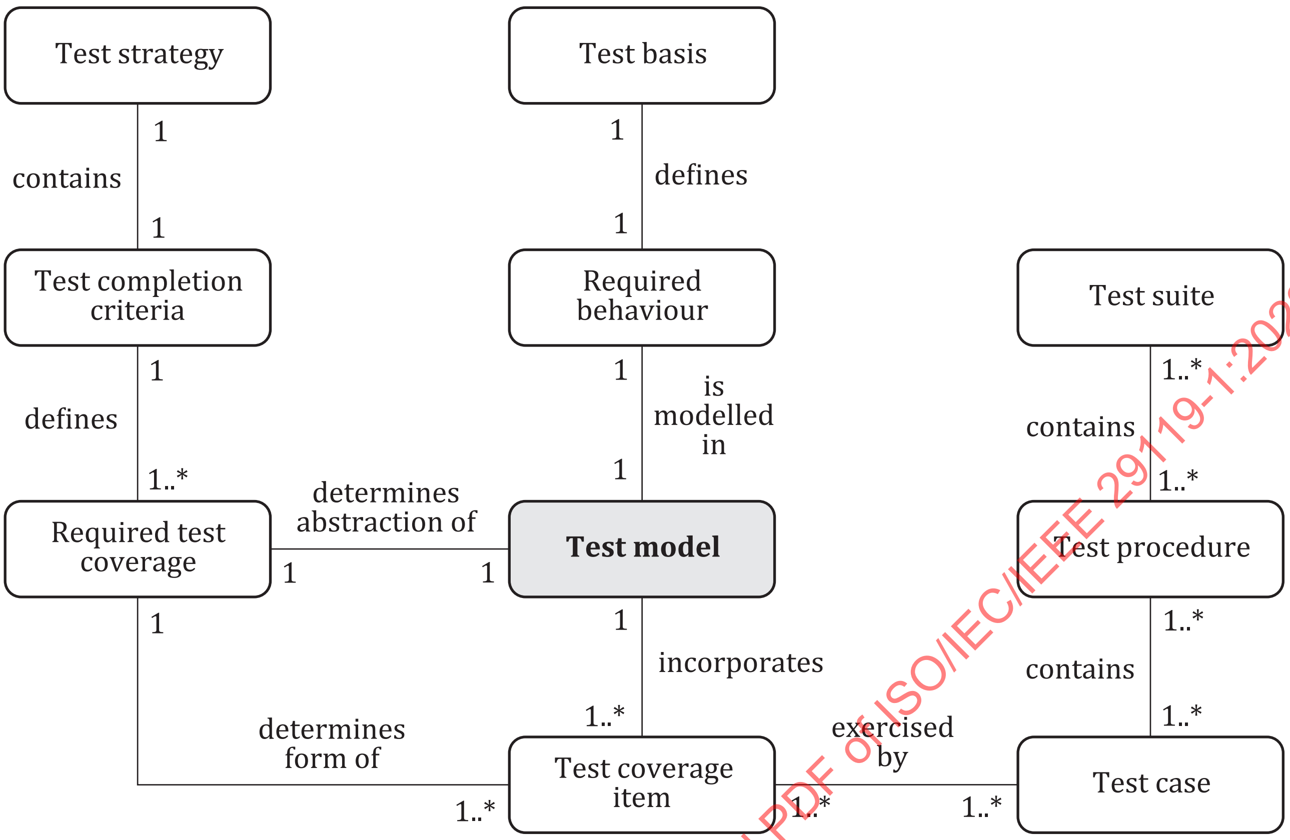

Figure 8 — Test model in perspective

As shown in Figure 8, the required test coverage is specified in the test strategy as one of the test completion criteria. The required behaviour is defined in the test basis and subsequently modelled in the test model. The test model is used to identify explicit test coverage items, which are used to generate test cases to cover them. Test procedures (test scripts) contain sequences of test cases, and several test procedures comprise a test suite.

The use of the test model to generate test cases (and, subsequently, test procedures) is defined in ISO/IEC/IEEE 29119-2. The documentation of the test model (in the test model specification) is defined in ISO/IEC/IEEE 29119-3. The use of the test model by test design techniques is defined in ISO/IEC/IEEE 29119-4.

4.4.2 Model-based testing

Model-based testing (MBT) uses models to generate test cases systematically and automatically. For MBT models are formal or semi-formal representations of the required behaviour of an item under test. Models can be developed for a complete software system or parts of a system. Use of MBT allows testers to design and implement tests at various levels of abstraction. MBT languages and notation can be formally defined syntactically and, in some cases, semantically. By using MBT tool support environments, test cases can quickly be generated from the model and automatically executed. Thus, MBT can support improved testing beyond that supported by natural languages and manual execution.

4.4.3 Scripted and exploratory testing

4.4.3.1 Overview

Test design and execution can be conducted in a number of ways, depending on the needs of each project:

- scripted, which can be:
  - manual;
  - automated;
- exploratory.

In practice, a combination is typically used, as the scripted testing helps achieve required test coverage levels, while the exploratory testing allows for creativity and the rapid execution of tests, when needed.

4.4.3.2 Scripted testing

In scripted testing, test cases are recorded (e.g. documented in a test management tool or in a spreadsheet) and can then be executed manually or executed automatically using an automated testing tool. The level of detail required within each test case (such as the granularity of test steps and expected results) often depends on the documentation requirements of the project, which are typically decided during test planning (see 4.2). The level of detail also depends on whether the testing is automated (which typically requires more detail). For manual tests, the level of detail also depends on the knowledge, experience, and capability of the test executor (including their system and testing domain knowledge).

4.4.3.3 Exploratory testing

In exploratory testing[16], tests are designed and executed on the fly, as the tester interacts with and learns about the test item. Session sheets are often used to structure exploratory testing sessions (e.g. by setting a focus and time limits on each test session). These same session sheets are also used to capture information about what was tested, and any anomalous behaviour observed. Exploratory tests are often not wholly unscripted, as high-level test scenarios (sometimes called “test ideas”) are often documented in the session sheets to provide a focus for the exploratory testing session.

4.4.4 Test design techniques

In both scripted and exploratory testing, tests are often created by using test design techniques, such as those defined in ISO/IEC/IEEE 29119-4. In scripted testing, tests are typically designed in systematic and methodical ways, with one technique being used at a time, such as is demonstrated within the examples of ISO/IEC/IEEE 29119-4:2021, Annexes B and C. During exploratory testing, the same test design techniques are typically used, but their application is usually much more informal, with a variety of techniques being interleaved, with less test documentation.

The aim of applying test design techniques during exploratory testing is often to target specific types of defects, “cover” specific aspects of requirements, specifications, or source code. The main difference with how test techniques are used during scripted testing is that various techniques are used to generate test cases to test a specific aspect of the test item and so a technique may be used to generate a single test cases before another technique is subsequently used for the next test case, which explores a different aspect of the test item.

A variety of test design techniques is typically required to suitably cover any system, regardless of whether the testing is scripted or exploratory.

4.4.5 Experience-based testing

4.4.5.1 General

Experience-based testing is based on:

— previous testing experience;

— knowledge of particular software and systems;

— domain knowledge;

— metrics from previous projects (within the organization and from industry).

Experience-based test practices typically do not rely on large amounts of documentation (e.g. test procedures) to execute testing. On a continuum from scripted to unscripted testing, these experience-based test practices are primarily unscripted. Using such practices may mean that only tailored conformance to ISO/IEC/IEEE 29119-2 can be achieved.

ISO/IEC/IEEE 29119-4 describes the experience-based test design technique of error guessing. Other experience-based test practices include (but are not limited to) exploratory testing (see 4.4.3.3), tours, attacks, and checklist-based testing.

4.4.5.2 Tours

Tours provide generalized testing advice that guides testers through the paths of an application like a tour guide leads a tourist through the landmarks of a big city. Tours are not aimed at providing guidance on which test input values to select, rather they are aimed at providing higher level guidance for test design. For instance, a tour can suggest which areas of an application should be the focus of a two-hour exploratory testing session.

An example of a tour used for exploratory testing would be the ‘Landmark tour’. In this tour, testers choose key features, decide on a sequence to visit them, and then explore the application going from landmark to landmark until all of them have been visited.

4.4.5.3 Attacks

Attacks are identified based on their potential for exploiting a specific fault model. A fault model is a way of thinking about how and why software fails[15]. Fault models can be behaviour-based, which allows testers to look for effective tests (attacks) that will identify failures in software. Specific fault models can be used, for instance, to design attacks that may result in security compromises (e.g. for security testing), if a specialist type of testing is performed.

An example of a security attack would be to block access to software libraries. This attack is based on the fault model that an application will behave insecurely if software libraries that it relies on fail to load – and developers assume that calls to these libraries will work each time and make no provision for when this does not happen.

4.4.5.4 Checklist-based testing

Checklist-based testing is based on the tester generating tests based on a list of pre-determined items. The items on the checklist can be based on personal experience, commonly found defects, and perceived risks, among others. Checklist-based testing is often focused on a particular quality characteristic (e.g. a user interface checklist). Checklists should be reviewed and updated regularly so that that they do not become stale and that they continue to target the most important defects. Checklists can be focused on different levels of detail and the tests generated from checklists may be scripted or unscripted.

Checklists can come from various sources; for instance, they may be specific to an individual tester, an organization, may be shared on the web or may form part of a regulatory standard.

4.4.6 Retesting and regression testing

When a developer fixes a defect, tests are run to check that the fix was made successfully; this is known as retesting or confirmation testing. In most cases, the original test cases associated with the fixed code are used for retesting, but they are sometimes supplemented by new test cases that provide improved coverage.

When a developer makes changes to existing software (e.g. to fix a defect, add functionality, change non-functional characteristics), tests are run to check that the original behaviour of the software, which is expected to be unchanged by the modification, has not been adversely affected by the changes; this is known as regression testing.

4.4.7 Manual and automated testing

Test cases can either be run manually by a human test executor, or they can be executed by a test automation tool.

The decision to conduct manual or automated testing depends on various factors, including the following.

— The number of times a test case is likely to be re-executed. A common heuristic is that if a set of test cases is going to be executed 5 or more times (e.g. during multiple cycles of regression testing), then it is typically cost-effective to automate the tests. The number of times a test case is going to be rerun can depend on the life cycle methodology in use (e.g. traditional life cycles can require infrequent regression testing, whereas agile projects typically require regression testing at least every sprint, while projects utilizing continuous integration (CI) can rerun a test case every build or once per day). Thus, within agile or CI methodologies, regression test cases are highly likely to be re-run more than 5 times.

— The capability of the testers on the project, as most test automation tools typically require testers with some programming capabilities.

— The budget and time available for acquiring and piloting test automation tools.

— The budget available for hiring and/or training testers in test automation tools.

There are three common approaches to the automation of test design and execution. These are:

— capture-replay;

— data-driven;

— keyword-driven (for a detailed description, see ISO/IEC/IEEE 29119-5).

4.4.8 Continuous testing

In continuous testing, test execution is started via an automated process that can occur on-demand, such as whenever code is checked into a central repository, on a nightly basis, or whenever builds are deployed to test or production environments. Continuous testing typically occurs in the context of continuous integration (CI) and continuous delivery (CD).

The application of continuous testing and continuous integration allow early testing.

4.4.9 Back-to-back testing

In back-to-back testing, an alternative version of the system (e.g. already existing, developed by a different team, implemented using a different programming language, an executable design model) is used as a comparison system to generate expected results for comparison from the same test inputs.

As such, back-to-back testing is not a test case generation technique as test inputs are not generated. Only the expected results are generated automatically by the functionally equivalent system. When used in partnership with tools for generating test inputs (random or otherwise) it becomes a powerful way to perform high-volume automated testing.

4.4.10 A/B testing

A/B testing is a statistical testing approach that allows testers to determine which of two systems performs better – it is sometimes known as split-run testing. It is often used for digital marketing (e.g. finding the email that gets the best response) in client-facing situations. As an example, A/B testing is often used to optimize user interface design. For instance, the user interface designer hypothesizes that by changing the colour of the 'buy' button from the current red to blue, that sales will increase. A new variant of the interface is created with a blue button and the two interfaces are assigned to different users. The sales rates for the two variants are compared and, given a statistically significant number of uses, it is possible to determine if the hypothesis was correct. This form of A/B testing requires a statistically significant number of uses and can be time-consuming, although tools can be used to support it.

A/B testing is not a test case generation technique as test inputs are not generated. A/B testing is a means of solving the test oracle problem by using the existing system as a partial oracle.

4.4.11 Mathematical-based and fuzz testing

Mathematical-based test practices can be used to plan, design, select data and set up input conditions when the test item's required behaviour, input space or output space can be described in sufficient detail, for instance using a formal language. Formal specification languages can be used to support model-based testing, and formal analysis of source code can be used to identify potential risks and inform the test strategy.

Mathematics form the basis for some of the test case design techniques described in ISO/IEC/IEEE 29119-4, such as random test case selection, and they can also be used to support combinatorial testing approaches. Similarly, fuzz testing uses a tool to generate high volumes of test input data, which is then used to test a program using a simple test oracle (e.g. checking that the program does not crash). Fuzz testing can be surprisingly effective at finding defects, but, as with random testing, test coverage is difficult to measure. When measures of confidence are needed from fuzz testing a statistical approach can be used to determine the significance of the test results based on the number of tests run.

Due to the large numbers of inputs that can typically be generated using mathematical practices, automated tools are usually needed.

4.4.12 Test environments

Test environment requirements should be considered as part of test planning, to allow all necessary components to be acquired, set up, configured, and validated prior to test execution, and cleaned up after execution. Test environment requirements can include:

— software – such as operating systems, programs, and other apps;

— services – such as virtual or cloud services;

— hardware – such as servers and end-user desktop computers, laptops, and mobile devices;

— network – such as switches, routers, network connections at specific speeds;

— interfaces – such as to internal, virtual, or third-party systems;

— peripherals – such as printers, scanners, and card scanners;

— data – including data stored in databases;

— test tools – such as test management and execution tools;

— availability – such as hours per day and times to set-up.

Test environment requirements should be recorded so that they can be communicated and understood. If a separate team is setting up a test environment, then that team should produce a test environment readiness report, to confirm that the environment has been set up, configured, and validated as required. With a separate team, there should be a service level agreement (SLA) specifying when and how problems with the environment will be addressed.

4.4.13 Test data management

Test data setup and clean-up requirements should be identified to help ensure the system is in a state that enables each test case to be executed and re-executed. For instance, when testing a credit rating system, a customer's historical banking data are required to determine the customer's credit score.

Private data may need to be anonymized or sanitized to help meet data privacy regulations.

Test data setup, clean-up and sanitization may be carried out manually or automated using scripts. Test data and automated scripts should be versioned and placed under source control.

4.5 Project management and testing

Project management refers to the support processes that are used to plan and control the course of a project including the management of the testing within the overall project. Regardless of who has responsibility for the individual processes, the project management and test management processes are closely related, as shown in Figure 9.

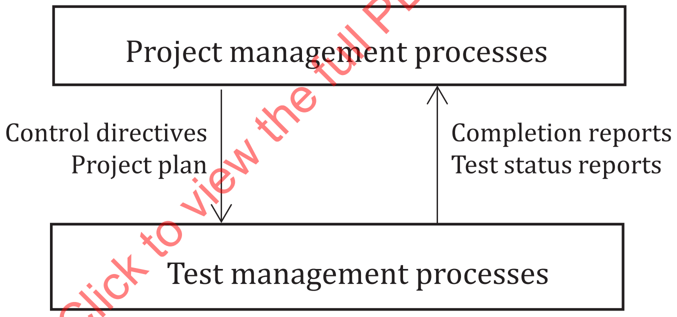

Figure 9 — Relationship between the overall project and the testing

The estimation, risk analysis, and scheduling of the test activities should be consolidated with the overall project planning. The project plan, which is an information item from the project management process, is therefore an input to the test management process.

During the course of the testing, measurements collected from detailed test activities are analysed by the test manager and communicated to the project manager for analysis in the project context. This may result in changes to the project plan, updated project plans and appropriate directives, which will need to be issued to the testers to help ensure that the testing is aligned with the project plan.

4.6 Communication and reporting

Communication by testers should provide stakeholders with relevant and timely information that is appropriate for the audience. This information may provide developers with details on needed modifications, alert project managers about progress against schedule and budget, or inform external parties on the conclusion of testing.

Test plans will specify the nature and timing of reporting on test tasks. Tasks with an extended duration may require periodic status reporting. Status reports can indicate the need to adjust previously planned priorities, schedules, or resource allocations. Completion reports describe the performed testing and the outcome.

4.7 Defects and incident management

Failed tests, as indications of possible defects, need to be investigated systematically to determine if the defects are in the system under test or in the tests themselves (which may indicate a flawed or missing requirement). A test result that differs from the predicted behaviour is regarded as an incident to be recorded, investigated, and possibly resolved. Defects can be found by dynamic and static testing as well as other activities such as analysis or verification and validation.

Annex A
(informative)

System characteristics and testing – examples

A.1 Overview

Each system can be categorized by a set of characteristics that describe certain aspects of the system. If these system characteristics are chosen carefully, then specific test approaches can be associated with these characteristics that are known to be appropriate in many (but not all) situations.

This annex briefly describes a number of system characteristics (many more can be identified) and the associated test approaches. If a tester can identify which of the system characteristics apply to the system they are testing, then they should consider whether the specialized testing listed for the characteristic is appropriate for inclusion in their test strategy. This should supplement the input of risk-based testing (see 4.2) and should in no way be considered a replacement for it.

Where a system characteristic has 'associated characteristics' identified (e.g. real-time is an associated characteristic of closed loop), then the testing applicable to the associated characteristic is also likely to be applicable (e.g. the testing suggested for real-time systems should also be applicable to closed loop systems).

A.2 Artificial intelligence (AI)

A.2.1 Description

Artificial intelligence comes in a variety of forms (e.g. classification, optimization, planning) and is used to solve a range of problems (e.g. search engines, image recognition, speech recognition, understanding and synthesis). Whichever form an AI-based system takes, it is normally complex and difficult to understand. They are also typically probabilistic in nature, meaning that most AI-based systems are non-deterministic.

A.2.2 Associated characteristics

Typical associated characteristics are scientific/technical (see A.14).

A.2.3 Specialized testing

Back-to-back and metamorphic testing both address the problem of testing non-deterministic systems.

The unique computing architectures of AI-based systems (e.g. massively concurrent neural networks) means that if structure-based testing is used then specific structural coverage criteria will need to be used (e.g. neuron coverage).

The different architectures and development process associated with AI mean that specialist skills are needed by testers (or data scientists in the testing role).

For more details on the testing of AI-based systems refer to ISO/IEC TR 29119-11.

A.3 Autonomous

A.3.1 Description

Autonomous systems address a wide range of application areas and can vary widely in terms of their implementation technology. An autonomous system is defined as a system that works for sustained periods independent of human control. This definition means that automatic systems and automated systems are also forms of autonomous system. Autonomous systems can be distinguished by their varying levels of flexibility, as shown in Table A.1.

Table A.1 — Levels of flexibility in autonomous systems

Flexibility (later levels may include previous levels)	**Example systems**
Fixed rules	Simple thermostat
Fixed rules with feedback	Anti-lock brake system (ABS)
Fixed neural network that does not change its behaviour based on its experience	Traffic jam system, e.g. SAE level 3
	Production line robot (using ML-based image processing)
System changes its behaviour based on its past experiences	Self-learning thermostat
	Financial trading system
	Smart building

A.3.2 Associated characteristics

Typical associated characteristics are real-time (see A.11).

A.3.3 Specialized testing

One approach to testing for autonomy is to attempt to force the system out of its autonomous behaviour and request intervention in unspecified circumstances (a form of negative testing). Negative testing can also be used to attempt to 'fool' the system into thinking it is in control when it should request intervention (e.g. by creating test scenarios at the boundary of its operational envelope – suggesting the application of boundary value concepts to scenario testing).

A.4 Bespoke

A.4.1 Description

Bespoke systems are built for a specific customer and purpose (i.e. they are not COTS systems – see A.6).

A.4.2 Associated characteristics

There are no associated characteristics.

A.4.3 Specialized testing

Acceptance testing is performed for a specific customer (and set of users).