Certified Tester
Advanced Level
Test Analyst (CTAL-TA)
Syllabus

v4.0

International Software Testing Qualifications Board

Copyright Notice

Revision History

Version	Date	Remarks
v4.0	2025/05/02	Major update with overall revision and scope update
v3.1.2	2022/01/31	Errata: Minor formatting, grammar, and wording defects fixed
v3.1.1	2021/05/15	Errata: The copyright notice has been adapted to the current ISTQB® standards
v3.1	2021/03/03	Minor update with section 3.2.3 rewritten and various wording improvements
v3.0 (2019)	2019/10/19	Major update with overall revision and scope reduction
v2.0 (2012)	2012/10/19	First version as a separate CTAL-TA Syllabus

Table of Contents

Acknowledgments

The General Assembly of the ISTQB® formally released this document on May 2nd, 2025.

It was produced by a team from the International Software Testing Qualifications Board: Armin Born, Filipe Carlos, Wim Decoutere, István Forgács, Matthias Hamburg (Product Owner), Attila Kovács, Sandy Liu, François Martin, Stuart Reid, Adam Roman, Jan Sabak, Murian Song, Tanja Tremmel, Marc-Florian Wendland, Tao Xian Feng.

The team thanks Julia Sabatine for her proofreading, Gary Mogyorodi for his technical review, Daniel Pol'an for his constant technical support, and the review team and the Member Boards for their suggestions and input.

The following persons participated in the reviewing, commenting, and balloting of this syllabus:

Current edition (v4.0): Gergely Ágnecz, Laura Albert, Dani Almog, Somying Atiporntham, Andre Baumann, Lars Bjørstrup, Ralf Bongard, Earl Burba, Filipe Carlos, Renzo Cerquozzi, Kanitta Chantaramanon, Alessandro Collino, Nicola De Rosa, Aneta Derkova, Dingguofu, Ole Chr. Hansen, Zheng Dandan, Karol Frühauf, Dinu Gamage, Chen Geng, Sabine Gschwandtner, Ole Chr. Hansen, Zsolt Hargitai, Tharushi Hettiarachchi, Ágota Horváth, Arnika Hryszko, Caroline Julien, Beata Karpinska, Ramit Manohar Kaul, Mattijs Kemmink, László Kvintovics, Thomas Letzkus, Marek Majerník, Donald Marcotte, Dénes Medzihradszky, Imre Mészáros, Krisztián Miskó, Gary Mogyorodi, Ebbe Munk, Maysinee Nakmanee, Ingvar Nordström, Tal Pe'er, Kunlanan Peetijade, Sahani Pinidiya, Lukáš Piška, Nishan Portoyan, Meile Posthuma, Yasassri Ratnayake, Randall Rice, Adam Roman, Marc Rutz, Jan Sabak, Vimukthi Saranga, Sumuduni Sasikala, Gil Shekel, Radoslaw Smilgin, Péter Sótér , Helder Sousa, Richard Taylor, Benjamin Timmermans, Giancarlo Tomasig, Robert Treffny, Shun Tsunoda, Stephanie Ulrich, François Vaillancourt, Linda Vreeswijk, Carsten Weise, Marc-Florian Wendland, Paul Weymouth, Elżbieta Wiśniewska, John Young, Claude Zhang.

Edition 2021-2022 (v3.1): Gery Ágnecz, Armin Born, Chenyifan, Klaudia Dussa-Zieger, Chen Geng (Kevin), Istvan Gercsák, Richard Green, Ole Chr. Hansen, Zsolt Hargitai, Andreas Hetz, Tobias Horn, Joan Killeen, Attila Kovacs, Rik Marselis, Marton Matyas, Blair Mo, Gary Mogyorodi, Ingvar Nordström, Tal Pe'er, Palma Polyak, Nishan Portoyan, Meile Posthuma, Stuart Reid, Murian Song, Péter Sótér, Lucjan Stapp, Benjamin Timmermans, Chris van Bael, Stephanie van Dijck, Paul Weymouth.

Subsequently, Tal Pe'er, Stuart Reid, Marc-Florian Wendland, and Matthias Hamburg suggested formal, grammar, and wording improvements that have been implemented and published in Errata 3.1.1 and 3.1.2.

Edition 2019 (v3.0): Laura Albert, Markus Beck, Henriett Braunné Bokor, Francisca Cano Ortiz, Guo Chaonian, Wim Decoutere, Milena Donato, Klaudia Dussa-Zieger, Melinda Eckrich-Brajer, Péter Földházi Jr, David Frei, Chen Geng, Matthias Hamburg, Zsolt Hargitai, Zhai Hongbao, Tobias Horn, Ágota Horváth, Beata Karpinska, Attila Kovács, József Kreisz, Dietrich Leimsner, Ren Liang, Claire Lohr, Ramit Manohar Kaul, Rik Marselis, Marton Matyas, Don Mills, Blair Mo, Gary Mogyorodi, Ingvar Nordström, Tal Peer, Pálma Polyák, Meile Posthuma, Lloyd Roden, Adam Roman, Abhishek Sharma, Péter Sótér, Lucjan Stapp, Andrea Szabó, Jan te Kock, Benjamin Timmermans, Chris Van Bael, Erik van Veenendaal, Jan Versmissen, Carsten Weise, Robert Werkhoven, Paul Weymouth.

Edition 2012 (v2.0): Graham Bath, Arne Becher, Rex Black, Piet de Roo, Frans Dijkman, Mats Grindal, Kobi Halperin, Bernard Homès, Maria Jönsson, Junfei Ma, Eli Margolin, Rik Marselis, Don Mills, Gary Mogyorodi, Stefan Mohacsi, Reto Mueller, Thomas Mueller, Ingvar Nordstrom, Tal Pe'er, Raluca Madalina

Popescu, Stuart Reid, Jan Sabak, Hans Schaefer, Marco Sogliani, Yaron Tsubery, Hans Weiberg, Paul Weymouth, Chris van Bael, Jurian van der Laar, Stephanie van Dijk, Erik van Veenendaal, Wenqiang Zheng, Debi Zylbermann.

0 Introduction

0.1 Purpose of this Syllabus

This syllabus forms the basis for the International Software Testing Qualification for the Advanced Level Test Analyst. The ISTQB® provides this syllabus as follows:

1. To member boards, to translate into their local language, and to accredit training providers. Member boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.
2. To certification bodies, to derive examination questions in their local language adapted to the learning objectives for this syllabus.
3. To training providers, to produce training material and determine appropriate teaching methods.
4. To certification candidates, to prepare for the certification exam (either as part of a training course or independently).
5. To the international software and systems engineering community to advance the software and systems testing profession and as a basis for books and articles.

0.2 The Advanced Level Test Analyst in Software Testing

The ISTQB® Advanced Level Test Analyst (CTAL-TA) certification provides the skills needed to perform structured and thorough software testing across the entire software development lifecycle. It details the test analyst's role and responsibilities at every step of a standard test process and expands on important test techniques. The Advanced Level Test Analyst certification is aimed at people holding a Foundation Level certificate who wish to further develop their expertise in test analysis and test techniques.

In this syllabus, the test analyst is understood as a role which:

- focuses more on the business needs of the customer than on technical aspects of testing
- performs mainly functional testing but also contributes to user-focused, non-functional testing, such as usability, adaptability, installability, or interoperability testing
- uses black-box test techniques and experience-based testing rather than white-box test techniques
- improves the effectiveness of testing using defect prevention techniques

0.3 Career Path for Testers

The ISTQB® scheme supports testing professionals at all stages of their careers. Individuals who achieve the ISTQB® Advanced Level Test Analyst certification may also be interested in the other Core Advanced Levels (Technical Test Analyst and Test Management) and, thereafter, Expert Level (Test Management or Improving the Test Process). Anyone seeking to develop skills in testing practices in an Agile environment area could consider the Agile Technical Tester or Agile Test Leadership at Scale certifications. In addition, specialist streams offer certification products focusing on specific test technologies and approaches, specific quality characteristics and test levels, or testing within particular industry domains. Please visit www.istqb.org for the latest information on ISTQB®´s Certified Tester Scheme.

0.4 Business Outcomes

This section lists the Business Outcomes expected of a candidate who has achieved the Advanced Level Test Analyst certification.

An Advanced Level Test Analyst Certified Tester can…

Code	Description
TA-BO1	Support and perform appropriate testing based on the software development lifecycle followed
TA-BO2	Apply the principles of risk-based testing
TA-BO3	Select and apply appropriate test techniques to support the achievement of test objectives
TA-BO4	Provide documentation with appropriate levels of detail and quality
TA-BO5	Determine the appropriate types of functional testing to be performed
TA-BO6	Contribute to non-functional testing
TA-BO7	Contribute to defect prevention
TA-BO8	Improve the efficiency of the test process with the use of tools
TA-BO9	Specify the requirements for test environments and test data

0.5 Learning Objectives and Cognitive Level of Knowledge

Learning objectives support the business outcomes and are used to create the Certified Tester Advanced Level Test Analyst exams.

In general, all contents of this syllabus are examinable, except for the Introduction and Appendices. The exam questions will confirm knowledge of keywords at K1 level (see below) or learning objectives at the respective level of knowledge.

The specific learning objectives and their levels of knowledge are shown at the beginning of each chapter and classified as follows:

- K2: Understand
- K3: Apply
- K4: Analyze

For all terms listed as keywords just below chapter headings, the correct name and definition from the ISTQB® glossary shall be remembered (K1), even if not explicitly mentioned in any learning objective.

Further details and examples of learning objectives are given in *Section 7*.

0.6 The Advanced Level Test Analyst Certificate Exam

The Advanced Level Test Analyst certificate exam will be based on this syllabus. Answers to exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable except for the Introduction and Appendices. Standards and books are included as references, but their content is not examinable beyond what is summarized in the syllabus itself from such standards and books.

Refer to the Exam Structures and Rules document compatible with the Advanced Level Test Analyst v4.0 for further details.

The entry criterion for taking the ISTQB® Certified Tester Advanced Level Test Analyst is that candidates are interested in software testing. However, it is strongly recommended that candidates also:

- Have at least a minimal background in either software development or software testing, such as six months experience as a system or user acceptance tester or as a software developer
- Take a course that has been accredited to ISTQB® standards (by one of the ISTQB-recognized member boards).

Entry Requirement Note: The ISTQB® Foundation Level certificate shall be obtained before taking the Advanced Level Test Analyst certification exam.

0.7 Accreditation

An ISTQB® Member Board may accredit training providers whose course material follows this syllabus. Training providers should obtain accreditation guidelines from the Member Board or the body that performs the accreditation. An accredited course is recognized as conforming to this syllabus and allows an ISTQB® exam to be included in the course. The accreditation guidelines for this syllabus follow the general Accreditation Guidelines published by the ISTQB® Processes Management and Compliance Working Group.

0.8 Handling of Standards

International standardization organizations like IEEE and ISO have issued standards associated with quality characteristics and software testing. Such standards are referenced in this syllabus. The purpose of these references is to provide a framework (as in the references to ISO/IEC 25010 regarding quality characteristics) or to provide a source of additional information if desired by the reader. Please note that the ISTQB® syllabi use standard documents as a reference. Standards documents are not intended for examination. Refer to section 6 - References for more information on standards.

0.9 Level of Detail

The level of detail in this syllabus allows internationally consistent courses and exams. To achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the Advanced Level Test Analyst
- A list of terms that students must be able to recall
- Learning objectives for each knowledge area, describing the cognitive learning outcomes to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content does not describe the entire knowledge area of software testing; it reflects the level of detail to be covered in Advanced Level Test Analyst training courses.

The syllabus uses the terminology (i.e. the name and meaning) of the terms used in software testing and quality assurance according to the ISTQB® Glossary (ISTQB-Glossary, 2024).

For the terminology in related disciplines please refer to the respective glossaries: IREB-CPRE for requirements engineering (IREB-Glossary, 2024), and IEEE-Pascal for software engineering (Computer Society et al., 2024).

0.10 How this Syllabus is Organized

There are five chapters with examinable content. The top-level heading for each chapter specifies the time allotted for the chapter; timing is not provided below the chapter level. For accredited training courses, the syllabus requires a minimum of **20.25 hours** of instruction (**1215 minutes**), distributed over the five chapters as follows:

- Chapter 1 (225 minutes): The Tasks of the Test Analyst in the Test Process
  - The student learns how the test analyst is involved in various software development lifecycles.
  - The student learns how the test analyst is involved in various test activities.
  - The student learns about tasks performed by the test analyst related to work products.
- Chapter 2 (90 minutes) The Tasks of the Test Analyst in Risk-Based Testing
  - The student learns how the test analyst contributes to product risk analysis.
  - The student learns how to analyze the impact of changes to determine the scope of regression testing.
- Chapter 3 (615 minutes) Test Analysis and Test Design
  - The student learns about data-based test techniques, such as domain testing, combinatorial testing, and random testing.
  - The student learns about behavior-based test techniques, such as CRUD testing, state transition testing, and scenario-based testing.
  - The student learns about rule-based test techniques, such as decision table testing and metamorphic testing.
  - The student learns about experience-based testing, such as session-based testing and crowd testing.
  - The student learns how to select appropriate test techniques to mitigate product risks.
- Chapter 4 (60 minutes) Testing Quality Characteristics
  - The student learns how to perform several types of functional testing.
  - The student learns how to use the specific knowledge of functionality to contribute to non-functional test types, such as usability testing, flexibility testing, and compatibility testing.
- Chapter 5 (225 minutes) Software Defect Prevention
  - The student learns about various defect prevention practices.
  - The student learns about various approaches that support phase containment.

- The student learns how to mitigate the recurrence of defects.

1 The Tasks of the Test Analyst in the Test Process – 225 minutes

Keywords

high-level test case, keyword, keyword-driven testing, low-level test case, software development lifecycle, test analysis, test analyst, test case, test condition, test data, test design, test environment, test execution, test implementation, test oracle, test script, testware

Learning Objectives for Chapter 1:

1.1 Testing in the Software Development Lifecycle

TA-1.1.1 (K2) Summarize the involvement of the test analyst in various software development lifecycles

1.2 Involvement in Test Activities

TA-1.2.1 (K2) Summarize the tasks performed by the test analyst as part of test analysis

TA-1.2.2 (K2) Summarize the tasks performed by the test analyst as part of test design

TA-1.2.3 (K2) Summarize the tasks performed by the test analyst as part of test implementation

TA-1.2.4 (K2) Summarize the tasks performed by the test analyst as part of test execution

1.3 Tasks Related to Work Products

TA-1.3.1 (K2) Differentiate between high-level test cases and low-level test cases

TA-1.3.2 (K2) Explain the quality criteria for test cases

TA-1.3.3 (K2) Give examples of test environment requirements

TA-1.3.4 (K2) Explain the test oracle problem and potential solutions

TA-1.3.5 (K2) Give examples of test data requirements

TA-1.3.6 (K3) Use keyword-driven testing to develop test scripts

TA-1.3.7 (K2) Summarize the types of tools to manage the testware

Introduction to the Tasks of the Test Analyst in the Test Process

The Foundation Level Syllabus (ISTQB-CTFL, v4.0.1) describes two principal roles in testing: a test management role and a testing role. In this syllabus, the person in a testing role responsible for testing the software's business aspects is called a test analyst (TA). Although this responsibility is rarely assigned to a dedicated position or role, the TA has clearly defined tasks and required competencies. In terms of test levels, the TA focuses on system testing, acceptance testing, and system integration testing. In terms of quality characteristics, the TA's competencies focus on functional suitability and also cover certain user-facing non-functional quality characteristics such as usability, adaptability, installability, and interoperability.

1.1 Testing in the Software Development Lifecycle

1.1.1 Involvement of the Test Analyst in Various Software Development Lifecycles

The organization of test activities can vary depending on the software development lifecycle (SDLC) being followed. Therefore, the TA's involvement in test activities may vary depending on the adopted SDLC model.

In **sequential development models**, development activities are done in phases, and begin when the previous phase is completed. Typically, there is little overlap between these activities. Therefore, the TA's tasks usually change over time. In the early phases of the SDLC, the TA focuses on supporting test planning. The TA begins test analysis when the test basis is being produced. Test design and test implementation follow in parallel with software design and implementation. Finally, the TA executes the tests and supports test completion during the late SDLC phases.

Incremental development models divide the software into smaller, manageable increments. Each increment is developed and tested independently. Therefore, the TA performs the same activities for each increment (i.e., test analysis, test design, test implementation, test execution, and test management support). However, the work of the TA may be organized differently for each increment. Testing focuses on the new or modified features. In addition, due to the increased risk of regression, the TA must pay particular attention to the refactoring and the assembly of regression test suites.

In **iterative development models**, the development process is cyclical. The project undergoes repeated prototyping, testing, refining, and deployment cycles. The TA role is dynamic and adaptive. The TA collaborates closely with developers and business representatives, adapting to the evolving product. The TA adapts and modifies test conditions and test cases as the software evolves and provides feedback to improve the test process at each iteration. The more frequent the iterations are, the more critical the ongoing maintenance and development of the regression tests by the TA.

An SDLC may combine elements of various models and specific techniques and approaches (e.g., Agile software development combines aspects of both iterative and incremental models). In such cases, the TA's involvement will depend on the SDLC's specific characteristics and how they are combined. A good practice common to all SDLC models is that the TA should be involved from the initial phases of the SDLC.

1.2 Involvement in Test Activities

The Foundation Level Syllabus (ISTQB-CTFL, v4.0.1) describes seven activities within the test process. The TA is mainly focused on four of them: test analysis, test design, test implementation, and test execution.

The TA's tasks in these four activities are described in detail in the following sections.

1.2.1 Test Analysis

During test analysis, the TA checks the completeness of the test basis and collects any additional information relevant for testing. This includes not only documentation but also verbal information, e.g., conversations in collaborative user story writing (see ISTQB-CTFL (v4.0.1), Section 4.5.1). Changes to the test basis can lead to adjustments to the test scope in coordination with test management.

To proceed effectively with test analysis, the TA checks the following entry criteria:

- Test planning has been performed, and the test scope, test objectives, and test approach are clear.
- The test basis (containing information such as requirements or user stories) is defined.
- The product risks already identified have been evaluated and documented if required.

The TA evaluates the test basis to identify any defects it may contain and assess its testability, thus providing early feedback to the product owners. This may include modeling the system behavior according to the test techniques to be applied (see *Section 3*, *Section 5.2.1*). Review techniques are also applied as part of the process (see *Section 5.2.2*). If not directly fixed, defects regarding the test basis must be documented. Moreover, the TA determines the test oracles needed (see *Section 1.3.4*).

The TA defines and prioritizes test conditions for each test item in scope. The test conditions address the test objectives (see ISTQB-CTFL (v4.0.1), Section 1.1.1) and must be traceable to the elements of the test basis. The scope and focus of the test conditions take the product risks into consideration. In incremental or iterative development models, this includes determining the scope of regression testing based on an impact analysis. In Agile software development, test conditions can be expressed as acceptance criteria that reflect the risks of the user stories.

The TA can proceed in stages, starting with high-level test conditions such as 'functionality of screen x'. Next, the TA defines more detailed test conditions such as 'Screen x rejects account numbers that are one digit too short'. This approach supports sufficient coverage and enables an early start to the test design, e.g. for user stories that still need to be refined.

The TA involves the stakeholders in reviewing the test conditions to ensure the test basis is clearly understood and that testing is aligned with the test objectives.

1.2.2 Test Design

Test design describes how to perform testing to achieve the stated test objectives. This is typically done with test cases. The way the test design is performed depends on many factors, including required coverage, the test basis, the SDLC, project constraints, and the knowledge and experience of the testers.

During test design, the TA determines in which areas low-level test cases or high-level test cases are appropriate (see *Section 1.3.1*). In both cases, the TA must identify clear pass/fail criteria. The TA designs the test cases for the new or changed test conditions according to the quality criteria (see *Section 1.3.2*). For regression tests, the selection of existing high-level test cases or the adaptation of existing low-level test cases based on their prioritization is usually sufficient.

The TA captures traceability between the test basis, test conditions, and test cases. In experience-based testing, test cases are not always documented; instead, the test conditions (among others) might guide the test execution. TA can design some test cases based on high-level test objectives.

In addition to these tasks, the TA defines the test environment requirements (see *Section 1.3.3*) and identifies, creates, and specifies the requirements for test data (see *Section 1.3.5*).

The TA uses the exit criteria defined in test planning to determine when enough test cases have been designed. However, other exit criteria such as residual risk levels or project constraints (e.g., budget or time) also indicate when the test design may end.

Test design can be supported by tools but should be tool- and technology-agnostic to remain tool-independent. The test design applies a systematic approach using test techniques or is ad hoc otherwise.

Test cases have a communicative role and should be understandable by the relevant stakeholders. As a test case may not always be executed by its author, other testers need to understand how to execute it, its test objectives (i.e., its underlying test conditions), and its importance. Test cases must also be understandable by developers who may implement the tests or rerun them in case of a failure and auditors who may have to approve them.

1.2.3 Test Implementation

During test implementation, the TA provides the testware that is needed for test execution. The TA may organize test procedures and test scripts into test suites or suggest test cases for automation. Defining test procedures requires carefully identifying constraints and dependencies that may influence the test execution order. In addition to the steps contained in test cases, the test procedures include steps for setting up any initial preconditions (e.g., loading test data from a data repository), verifying expected results and postconditions, and resetting steps following execution (e.g., resetting database, environment, and system).

The TA prioritizes the test procedures and test scripts for test execution based on the prioritization criteria identified during risk analysis and test planning and identifies the test procedures or test scripts that should be executed on the current version of the test object. This enables related tests (e.g., for new features or regression testing) to be executed together in a specific test run. The TA updates the traceability between the test basis and other testware such as test procedures, test scripts, and test suites.

The TA can assist the TM in defining a test execution schedule, including resource allocation, to enable efficient test execution by defining the test execution order (see ISTQB-CTFL, v4.0.1, Section 5.1.5).

The TA creates input and environment data to load into databases and other repositories (see *Section 1.3.5*). This data must be “fit for purpose” to support the specific test objectives.

The TA should also verify that the test environment is fully set up and ready for test execution (see *Section 1.3.3*). This is best carried out by designing and running a smoke test. The test environment should reveal defects in the test object through test execution, operate normally when failures do not occur, and adequately replicate, if required, the production or end-user environment.

The level of detail and the associated complexity of work carried out during test implementation may be influenced by the level of detail of the test conditions and test cases. In some cases, regulatory rules apply, and testware should provide evidence of compliance with applicable standards (e.g., *RTCA DO-178C*, 2011).

1.2.4 Test Execution

Test execution is carried out according to the test execution schedule. The typical tasks performed by the TA are executing tests, comparing actual results to expected results, analyzing anomalies, reporting defects, and logging the test results.

The TA executes tests manually. This includes exploratory testing, executing test procedures, regression testing, and confirmation testing. For exploratory testing, the TA can use session-based testing with test charters (see *Section 3.4.1*). The TA can also run automated test scripts, but it may be the task of developers, test automation engineers (TAEs), or technical test analysts (TTAs) to run automated test scripts and analyze them in case of failures.

The TA analyzes anomalies that occur in manual or automated test executions to establish their likely causes. An anomaly may be the consequence of a defect in a test object. Still, other reasons may also exist, including missing preconditions, incorrect test data, defects in test scripts or the test environment, or misunderstanding of specifications. The TA logs the actual results of the test execution, communicates defects based on the observed failures, and reports on them if needed.

The TA updates the traceability between the test basis and other testware by considering test results. This information enables the transformation of test results to high-level risk or coverage information, which allows the stakeholders to make informed decisions. For example, this clarifies to stakeholders how many test cases related to a test condition have passed or failed.

In addition to these typical tasks, the TA evaluates the test results, including the following tasks:

- Recognizing defect clusters, which may indicate the need for more testing of a particular part of the test object (see *Section 5.3.1*).
- Manually re-executing automated tests that have failed to make sure that the test automation did not produce a false-positive result.
- Suggesting additional tests based on what was learned during previous tests.
- Identifying new risks from information obtained when performing test execution.
- Suggesting improvements to test design or test implementation (e.g., improvements to test procedures) or even to the system under test.
- Suggesting improvements to the regression test suites, including refactoring, scope adjustments, and test automation (see *Section 2.2.1*).

1.3 Tasks Related to Work Products

The TA must ensure the quality of the work products for which they are responsible. This includes test cases, test environments, test data, test oracles, and test scripts. In this section, the syllabus discusses the TA's tasks related to testware and the types of tools for managing testware.

1.3.1 High-Level Test Cases and Low-Level Test Cases

A high-level test case (also called an *abstract test case* or a *logical test case*) describes the circumstances in which the test object is examined by indicating which test conditions are covered by the test case. High-level test cases are, therefore, suitable for ensuring that tests cover all relevant test conditions. High-level test cases do not contain concrete information for preconditions, input data, expected outputs, or postconditions. These are all expressed at an abstract level (e.g., "order more than one book, with the order price resulting in a discount; expected result: discount is assigned").

A low-level test case (also called a *concrete test case* or a *physical test case*) is the detailed refinement of a high-level test case. Low-level test cases describe what data needs to be prepared, what actions the tester must do (if necessary), and what the concrete expected result is. Low-level test cases contain

specific preconditions, input data, expected results, and postconditions (e.g., “order books B1 ($10) and B2 ($20), with total order price $30; expected result: 10% discount assigned, total price: $27”).

Typically, a TA initially designs high-level test cases, which are the foundation for developing low-level test cases. One high-level test case can be implemented in one or more low-level test cases. Sometimes, the test case may remain high-level, with the concrete information determined by the TA during test execution. For instance, high-level test cases can guide the TA when creating test objectives within a test charter for session-based testing, allowing TAs to elaborate on these objectives during the test execution (see *Section 3.4.1*).

The transition from high-level to low-level test cases is more than just filling in concrete values. It is also a step from conceptual to technical. It is often deferred from test design to test implementation, especially if specific test data is needed. The TA must ensure that everything necessary to execute the low-level test cases is known (Koomen et al., 2006). In practice, many test cases are hybrid, being concrete in some aspects while abstract in others. This is often due to a trade-off between maintainability and comprehensibility of test cases.

1.3.2 Quality Criteria for Test Cases

Neglecting test case quality can lead to many problems, such as high maintainability costs, reduced comprehensibility, or execution delays. Quality criteria for test cases are a first step towards more maintainable test cases. They include:

- **Correctness**. A test case must facilitate accurate verification of the test conditions on which it is based.
- **Feasibility**. It must be possible to execute a test case.
- **Necessity**. Every test case should cover a clear test objective, as expressed in its title or summary. Duplicates should be avoided. Things that should not be tested should not have test cases designed.
- **Understandability**. Test cases may be reviewed, modified, and executed by people other than the author. The TA should write test cases in a language and format understandable to all stakeholders involved without explaining the obvious. Complex test cases should be simplified or split up.
- **Traceability**. Test cases should be traceable to test conditions, requirements, and risks to enable the TA to keep them up to date (see ISTQB-CTFL (v4.0.1), Section 1.4.4).
- **Consistency**. Consistency in language, formatting, and structure makes the test cases easier to understand and maintain. The TA may use a glossary for this purpose.
- **Precision**. There should be only one interpretation of a test case to avoid false-negative and false-positive test results. Ambiguous terms like ‘suitable’, ‘as needed’, or ‘several’ should be avoided.
- **Completeness**. All necessary attributes (e.g., see *ISO/IEC/IEEE 29119-3*, 2021) should be present, including the required test data (see *Section 1.3.5*) and a clear expected result to avoid doubt when comparing with the actual result.
- **Conciseness**. The granularity of test cases (i.e., one large test case with many test actions versus several smaller test cases) should correspond to the test basis and test conditions. Smaller test cases focused on a few coverage items are preferable, as they ease finding the causes of failures, can be flexibly combined into test procedures and test suites, and a failure during test execution would not block any further tests.

The format and level of detail of the test cases depend on the project and product context and should be agreed upon within the test team.

1.3.3 Test Environment Requirements

The test environment is a critical success factor for both manual and automated test execution. The implementation of the test environment impacts testability, defect detection, overall testing costs, and the reliability of test results. A test case that passes or fails in the test environment should have the same test result when executed in production. Ideally, a test environment is robust, predictable, and integrated with the test automation framework, if required. The TA may define the test environment requirements during test design based on the analysis of:

- test conditions, test cases, and test data requirements, such that test environment requirements describe the conditions necessary to set up and maintain the test environment to ensure the preconditions of the test execution are met
- test levels and test types, which influence the trade-off between test environment flexibility and similarity to the production environment
- availability and independence of components and systems, which may indicate the need to use test doubles (e.g., stubs or drivers)

The test environment requirements describe the test environment items, which can be categorized into several types, such as hardware, middleware, software, virtualized services, network, interfaces, tools, security, configuration, and venue. For each test environment item, the requirements should include the following information (*ISO/IEC/IEEE 29119-3*, 2021):

- unique identifier – used for traceability purposes
- description – in sufficient detail to implement it as required
- responsibility – describes who is responsible for making it available
- period needed – identifies when and for how long the item is needed
- fidelity - the degree to which this item represents or deviates from the production environment

The requirements should also address the test environment's overarching needs, including setup, backup and restore, security needs, the ability to change the test environment, and roles and authorization (Koomen et al., 2006).

The TA should document test environment requirements clearly, compactly, and coherently. The TA can use diagrams or tables. To avoid redundancies and unnecessary documentation, the TA can reference existing test environments and focus on the specific needs of the test level. The relevant stakeholders (e.g., developers, TTAs, test automation engineers, business analysts, sponsors, and product owners) should also review, approve, and update test environment requirements.

1.3.4 Determining Test Oracles

A test oracle is needed to determine the expected results in dynamic testing. Ideally, the test basis will provide the test oracles (e.g., in a textual or formal specification). Human experience or knowledge about the test object can also serve as a test oracle. An automated test oracle may be required for cost-effectiveness reasons (e.g., if inputs are automatically generated or human oracles are costly).

Depending on the quality and completeness of the test basis or system characteristics, a cost-effective test oracle may not be available. This is known as the *test oracle problem*. Typical factors contributing to the test oracle problem include data-related complexity, non-determinism (e.g., AI-based systems), probabilistic behavior, and missing or ambiguous requirements.

Some known solutions to the test oracle problem are (Barr et al., 2014):

- Pseudo-oracles are independently developed systems that fulfill the same specification as the test object (e.g., legacy systems, simplified versions of test objects). Developing a dedicated pseudo-oracle for a complex test object may be expensive but is typical for critical systems.
- Model-based testing may formalize the test oracle as part of the test model. It enables the generation of expected outputs and the derivation of tests from the model. Behavior-based test techniques often involve the implementation of an automated test oracle (see *Section 3.2.2* and *Section 3.2.3*).
- Property-based testing uses specified properties of the test object to verify relations between the input and the expected result of individual test cases. If such a relation is not met, the test case fails. This solution is well suited for test automation, but its effectiveness depends on the relations, which may be difficult to determine.
- Metamorphic testing (see *Section 3.3.2*).
- Human oracles use the capability of humans to determine the expected results. Human resources may be costly and scarce. However, human oracles are preferred in some test approaches (e.g., exploratory testing).

Assertions may be built into test automation code or in the test object itself to implement an automated test oracle. They are executable statements that verify the state or behavior of the test object. When built in the test object, they usually only verify what is necessary for the continuation of the task.

1.3.5 Test Data Requirements

During test design, the TA identifies and requests test data that may need preparation or provisioning for test execution. Considerations include purpose, format, and context of use. (*ISO/IEC/IEEE 29119-3*, 2021, Section 8.5) also describes test data requirements. Key aspects are:

- **Similarity with production data.** Production data reflects real-world data but may lack variability. Synthetic data allows for controlled variability. It should reflect key aspects of production data patterns, distributions, and outliers but may overlook certain defects that only occur with real-world data. For systems that lack production data, synthetic data must reflect realistic business and technical scenarios. Personas help by providing realistic, user-centered profiles that guide the creation of data reflecting diverse user scenarios and behaviors.
- **Confidentiality.** Sensitive test data (e.g., personal information) requires protection. Pseudonymized data replaces personal information with artificial identifiers. Anonymized data removes identifiable information about individuals. If necessary, the TA must observe data protection regulations like the GDPR in the European Union (Commission, 2016) or the HIPAA in the U.S. (Health et al., 2024).
- **Purpose.** Test data is crucial for determining preconditions and expected results that impact the system state and its configuration (e.g., system time and date). It also includes specifying user permissions and establishing relations between products, departments, and categories.

- **Coverage criteria.** Test data must align with the coverage criteria for the chosen test technique. In addition to valid test data, this may also require invalid test data, such as for negative tests.
- **Data format.** Systematic data management (e.g., in API testing) may require structured data (e.g., CSV, JSON, XML, or database).
- **Traceability.** Traceability ensures test data maintainability when changes are made to test cases.
- **Maintainability.** Hard-coded test data in low-level test cases should be avoided to facilitate defect detection and maintenance. Test cases should separate test logic from test data (see *Section 1.3.2*).
- **Dependencies.** Dependent data requires a series of steps to create them.
- **Availability.** Service virtualization can address missing data by simulating absent or inaccessible services to interact with external systems or services.
- **Time sensitivity and data aging.** Test cases involving outdated or time-sensitive data may impact system behavior in unexpected or inaccurate ways.

1.3.6 Developing Test Scripts Using Keyword-Driven Testing

If keyword-driven testing is used, the TA creates test scripts using keywords. The implementation of the test scripts is the task of the TTA, the TAE, or a developer.

The TA identifies and specifies keywords by analyzing the test basis or by collaborating with the stakeholders. Keywords can be classified into two categories: action and verification. Action keywords must interact with the test object (e.g., executing functions, submitting data, navigating within the test object), the test environment (e.g., setting up configurations and activating simulators), or other components or systems (e.g., triggering an interface of the test object). Verification keywords represent assertions to evaluate whether the actual result produced by the test object matches the expected result.

Keywords reside on at least two abstraction layers: the domain layer and the test interface layer. Domain layer keywords correspond to business-related actions and reflect the terminology of the application domain. They abstract from the technical details of the test object's interface. Test interface layer keywords communicate with the test objects, test environment items, or other components or systems via their test interface. They reside on the lowest abstraction layer. Additional intermediate layers may support the maintainability of keywords.

Keywords can be atomic or composed out of other keywords. A keyword's structure and abstraction layer are independent attributes. However, composite keywords often reside on a higher level of abstraction, while atomic keywords tend to reside on the test interface layer.

The tasks of the TA in keyword-driven testing include:

- Specifying keywords and their parameters
- Specifying the keyword test cases, i.e., test scripts using keywords
- Specifying the additional steps to the test scripts using keywords such as preconditions, verification actions, and cleaning up the test environment after the test
- Maintaining keyword test cases to reflect changes to the test object
- Executing keyword test scripts, either automated or manually
- Analyzing failed keyword test cases to determine the cause of the failure

When analyzing the test basis, the TA looks for interactions between the test object and its environment (e.g., users, other systems, and devices). Consider, for example, the user story 'As a member, I want to authenticate myself, such that I get access to the facilities', with the acceptance criteria 'Valid member cards can be used for authentication'. The TA may specify an (action) keyword 'Authenticate Member' with a parameter 'member card' and a verification keyword 'Verify Access'. The TA checks whether the identified keywords reside on the appropriate abstraction layer.

When specifying keywords, the TA must keep in mind that keywords must:

- contain a verb (+ noun)
- use the imperative form of the verb (+ noun)
- be unique in their meaning
- be adequately documented
- reflect the vocabulary of the application domain
- be reusable

When composing keyword test cases, the TA may recognize some missing keywords and immediately specify them. Keyword test cases may be written in various formats, such as lists or tables.

Keywords may change over a project's lifetime and are prone to being redundantly specified (Rwemalika et al., 2019). The recommendations mentioned in this section aim to avoid redundancy and reduce maintenance efforts.

Although keyword-driven testing is a test automation approach, manual testing can also benefit from it. If applied for manual testing, it effectively supports a later transition from manual to automated testing.

More details on test execution automation and keyword-driven testing are available in (ISTQB-TTA, v4.0), (ISTQB-TAE, v2.0), and (*ISO/IEC/IEEE 29119-5*, 2016).

1.3.7 Tools Applied in Managing the Testware

Proper testware management using tools can help the TA support the overall test process. Tools can provide the status of work products and support test monitoring and test control. In case of failures in the system under test, they allow the TA to check test results from previous test runs and analyze the point in time where defects have occurred.

Some key types of tools in managing the testware include:

- **Test management tools** to provide a repository of all relevant testware (e.g., test conditions, test cases, test scripts, test suites, and test runs). The tools facilitate traceability (e.g., via traceability matrix), retrieval of test cases, scheduling test runs, recording test results, and overall reporting of test progress and quality.
- **Defect management tools** to log, prioritize, and monitor the process of defect resolution.
- **Test data management tools** to create and maintain test data, including the protection of sensitive data (see *Section 1.3.5*).
- **Configuration management tools** to facilitate test-related activities within the development, release, and operation processes, including managing the configuration and availability of test environments.

- **Requirement management tools** to contain and track high-level requirements, ensuring these are clearly defined, versioned, and traced throughout the SDLC.

The TA supports the management of testware by:

- analyzing the project and release to select the correct subset of testware (e.g., a specification identified by its version to match the version of SUT)
- defining an appropriate functional (e.g., by features or modules) or technical structure (e.g., by test type or environment) for the organization of test cases in the test management tool
- adding metadata to the test cases (e.g., information on effort that is related to test execution or the specific test environment required)
- ensuring the traceability between requirements, test conditions, tests, test runs, and defects
- selecting the correct test suites for regression testing (for manual/automated test execution)
- configuration management of the test cases, including the identification of outdated test cases.

Tools help to organize, track, and ensure the quality of the test process. The TA supports this effort by selecting, structuring, and maintaining test cases, ensuring traceability, and managing test case versions for efficient testing and test control.

2 The Tasks of the Test Analyst in Risk-Based Testing – 90 minutes

Keywords

impact analysis, product risk, regression testing, risk analysis, risk assessment, risk control, risk identification, risk mitigation, risk monitoring, risk-based testing

Learning Objectives for Chapter 2:

2.1 Risk Analysis

TA-2.1.1 (K2) Summarize the test analyst's contribution to product risk analysis

2.2 Risk Control

TA-2.2.1 (K4) Analyze the impact of changes to determine the scope of regression testing

Introduction to the Tasks of the Test Analyst in Risk-Based Testing

Risk-based testing is a test approach that prioritizes test efforts based on the risk levels of the test items. The test manager determines this approach. The TA plays an active role in implementing it. Risk-based testing is covered in more detail in (ISTQB-TM, v3.0).

2.1 Risk Analysis

According to (ISTQB-CTFL, v4.0.1, Section 5.2), risk analysis involves risk identification and risk assessment. This syllabus focuses on how the TA should participate in risk analysis activities to ensure risk-based testing is implemented correctly.

2.1.1 The Contribution of the Test Analyst to Product Risk Analysis

The TA often possesses unique knowledge of the system, as well as experience and intuitive knowledge of what usually goes wrong, what impact it has, and how testing can mitigate the risk. This makes the TA a valuable stakeholder for product risk analysis.

In **risk identification**, the TA contributes with their own experience and knowledge to retrospectives, risk workshops, brainstorming, and creating checklists. The TA can also conduct interviews with stakeholders to better understand what they consider to be the most significant risks from their perspective.

During **risk assessment**, the TA, together with other stakeholders, contributes to the determination of the risk level by estimating several factors such as:

- frequency of use and criticality of the affected features
- criticality of the affected business objectives
- financial, environmental, and reputational damage
- quality of the test basis
- legal or safety needs

The TA also helps categorize product risks by the quality characteristics impacted, for example, using the product quality model of *ISO/IEC 25010* (2023). The risk level is often not uniformly distributed across the test object. In such cases, the TA should break down the test object into test items (e.g., components, interfaces, and features) and assess a given risk for each test item separately.

Finally, as part of the product risk assessment, the TA proposes suitable test activities to mitigate each identified product risk. These activities may include static testing and dynamic testing. Depending on factors such as the risk level, the associated test item type, and the quality characteristic affected, the TA indicates the required test levels, test types, test techniques, levels of independence of testing, and test thoroughness. In the spirit of shift left, the TA indicates which test activities can mitigate the risk earliest to minimize the testing effort.

2.2 Risk Control

According to (ISTQB-CTFL, v4.0.1, Section 5.2.4), risk control involves risk mitigation and risk monitoring. The TA understands the system's functionalities and potential risks related to them. Hence, the TA's role is crucial in several risk mitigation actions mentioned in (ISTQB-CTFL, v4.0.1, Section 5.2.4):

- Performing reviews is discussed in *Section 5.2.2* of this syllabus.
- Applying the appropriate test techniques and coverage levels is discussed in *Section 3.5*.
- Applying the appropriate test types is discussed in Chapter 4.
- Performing regression testing is discussed below.

In addition, since risks and risk levels are not static and change over time, risk-based testing involves regular risk monitoring. In an iterative lifecycle, this is performed at a frequency determined by the team (likely to be once per iteration). In other lifecycles, its frequency is set by the person responsible for product risk management, often the test manager. The TA contributes by updating the risk register based on the changes made and adjusting the risk mitigation actions mentioned above.

2.2.1 Determining the Scope of Regression Testing

The main objective of regression testing is to ensure confidence in the quality of the test object after a change is made. However, it might be impossible to execute all regression tests during regression testing due to constraints (e.g., restrictions on time, budget, test environment, or test data). This is primarily a concern with manually executed tests but also applies to automated tests, for example, when the test cycles are short and there are many long-running automated tests. This makes it necessary to select appropriate regression tests based on specific criteria. It is essential to review the scope of regression testing with every test cycle. The review may conclude that the existing regression test suite needs to be adjusted.

The most reliable technique for selecting automated tests is an impact analysis, which tools can support. Such tools are based on an automated configuration management system. They register which configuration items are activated during the execution of each test case. When a change occurs, they track which configuration items have been modified and select regression tests that interact with the changed items. By doing this, the tools ensure that after a change to a configuration item, the tests focus on those areas in which they are most likely to find failures (Juergens et al., 2018).

When it comes to manual test execution, there has not been conclusive evidence of a technique in regression test selection that is clearly superior, as the results depend on various factors (Engström et al., 2010). Therefore, the TA must decide which technique to use based on the given situation. Commonly used techniques include:

- **Risk-based test selection**, where the TA maintains the traceability of the regression test suite to a risk register. When a change is made, and the risk register is updated accordingly, the TA adjusts the regression test suite to cover the highest risk levels.
- **History-based testing**, where the TA evaluates past test executions and determines which have exposed defects or were sensitive to similar changes to the one made since the last test execution. Executing the corresponding tests again as part of the regression test increases the likelihood of exposing similar defects. The TA can also include some tests that have not been executed for a long time to ensure they still pass.
- **Coverage-based testing**, where the TA selects a small number of tests that achieve as much coverage as possible based on the test technique(s) chosen. The amount of tests must be carefully balanced with the coverage increase per test.
- **The requirement traceability matrix**, which is used to assess the impact of requirements changes on the associated tests. It can be particularly valuable when new or changed requirements indirectly

impact existing features. The TA selects regression tests for the directly affected and for related features to cover possible unintended side effects. In Agile software development, this can be done similarly by selecting tests that cover the acceptance criteria impacted by new or changed user stories.

- **Testing based on operational profiles**, where the TA selects the regression test cases to be executed based on the patterns of use of the test object. For example, when testing an online store, one such test can include the user logging in, searching for products, adding them to the cart, and placing the order. When an application is changed significantly, this technique provides a quick overview of the overall system functionality. If this results in too many tests, the TA prioritizes critical patterns of use that occur often and cover critical functionalities and business processes.
- **Impact analysis**, which can also be used for selecting regression test cases for manual execution if the TA knows which of them interact with the changed configuration items.

It is often necessary to use a combination of selection techniques for a more comprehensive and effective regression test suite. However, the TA must carefully balance the need for coverage with a manageable size of the test suite. After each test cycle, the TA analyzes the test results to determine the effectiveness of the techniques applied (see *Section 5.3.1*). Next time, the TA retains effective techniques and replaces ineffective ones. This continuously improves regression test selection over time. It is essential in iterative and incremental development models, where changes are frequent.

3 Test Analysis and Test Design – 615 minutes

Keywords

checklist-based testing, behavior-based test technique, combinatorial testing, crowd testing, CRUD testing, data-based test technique, decision table testing, domain testing, equivalence partition, experience-based testing, metamorphic relation, metamorphic testing, random testing, rule-based test technique, scenario-based testing, session-based testing, state transition testing, test charter

Learning Objectives for Chapter 3:

3.1 Data-Based Test Techniques

TA-3.1.1 (K3) Apply domain testing

TA-3.1.2 (K3) Apply combinatorial testing

TA-3.1.3 (K2) Summarize the benefits and limitations of random testing

3.2 Behavior-Based Test Techniques

TA-3.2.1 (K2) Explain CRUD testing

TA-3.2.2 (K3) Apply state transition testing

TA-3.2.3 (K3) Apply scenario-based testing

3.3 Rule-Based Test Techniques

TA-3.3.1 (K3) Apply decision table testing

TA-3.3.2 (K3) Apply metamorphic testing

3.4 Experience-Based Testing

TA-3.4.1 (K3) Prepare test charters for session-based testing

TA-3.4.2 (K3) Prepare checklists that support experience-based testing

TA-3.4.3 (K2) Give examples of the benefits and limitations of crowd testing

3.5 Applying the Most Appropriate Test Techniques

TA-3.5.1 (K4) Select appropriate test techniques to mitigate product risks for a given situation

TA-3.5.2 (K2) Explain the benefits and risks of automating the test design

Introduction to Test Analysis and Test Design

Test techniques are mainly used in test analysis and test design. The test techniques discussed in this chapter cover black-box test techniques and experience-based test techniques. White-box test techniques are discussed in (ISTQB-TTA, v4.0).

This syllabus sub-divides black-box test techniques into three categories based on the underlying test conditions that model:

- elements of the data (data-based)
- elements of the dynamic behavior (behavior-based)
- elements of static behavior rules (rule-based)

3.1 Data-Based Test Techniques

In this section, the term “domain” is used to represent the set of input data of a test item. Input data from various areas of the domain leads to various behaviors of the test item. Data-based test techniques aim to verify that the implementation handles specific domain areas correctly. The Foundation Level Syllabus (ISTQB-CTFL, v4.0.1, Section 4.2) covers two test techniques that can be categorized as data-based: equivalence partitioning (EP) and boundary value analysis (BVA). This syllabus introduces three further data-based test techniques:

- domain testing - extends EP and BVA to domains with multiple parameters and complex partitions
- combinatorial testing - focuses on interactions of multiple parameters in a multidimensional domain
- random testing - selects random inputs from the domain based on a specified probability distribution

Note that random testing can generally refer to both random input data and random events. This syllabus only deals with testing with random input data.

3.1.1 Domain Testing

Domain testing verifies whether the test item behaves as specified on the domain’s equivalence partitions and at their borders. In this case, equivalence partitions are defined using expressions that combine atomic conditions by Boolean operators (e.g., AND, OR, and NOT) and involve one or more interacting variables. Each atomic condition defines a border of the equivalence partition. Closed borders are formed by relational expressions with operators $\leq$, $\geq$ or =, and open borders are formed by relational expressions with operators $<$, $>$, or $\neq$.

For example, the expression height $>$ 1.29 meters AND weight / height^2 $\geq$ 30 defines an equivalence partition of a two-dimensional domain of two variables that has two borders, an open and a closed one.

Domain testing seeks to identify defects in equivalence partition implementations (e.g., wrong operator or constant) by selecting appropriate coverage items that can reveal those defects. Coverage criteria in domain testing refer to the ON, OFF, IN, and OUT points:

- For a closed border, an ON point lies on this border. For an open border, an ON point lies inside the equivalence partition and is closest to the border according to the given precision.
- For a closed border, an OFF point lies outside the equivalence partition and is closest to the border according to the given precision. For an open border, an OFF point lies on this border.

- An IN point belongs to the equivalence partition and is not an ON point.
- An OUT point lies outside the equivalence partition and is not an OFF point.

Each of these types of points is related to the equivalence partition's border. The same point can have different types for different borders.

Coverage criteria include:

Simplified domain coverage (Jeng et al., 1994), which requires the following coverage items:

- One ON and one OFF point for each border defined by one of the operators $<$, $\leq$, $>$ or $\geq$.
- One ON point and two OFF points located on different sides of the border for the = operator.
- One OFF point and two ON points located on different sides of the border for the $\neq$ operator.

Each OFF point shall be as close as possible to the corresponding ON point according to the given precision.

Reliable domain coverage (Forgács et al., 2024), which requires the following coverage items:

- One ON, one OFF, one IN, and one OUT point for each border defined by one of the operators $<$, $\leq$, $>$, or $\geq$.
- One ON point and two OFF points located on different sides of the border for the = operator.
- One OFF point and two ON points located on different sides of the border for the $\neq$ operator.

For both coverage criteria discussed, the number of coverage items depends linearly on the number of borders. It can be optimized for both coverage criteria by using the same coverage item for different borders. For example, all borders of an equivalence partition can use a common IN point or a pair of ON and OFF points of a border can be used as OFF and ON points for the adjacent equivalence partition's border. For domains with many borders, an advanced optimization algorithm is available (Forgács et al., 2024).

Reliable domain coverage results in a slightly larger amount of coverage items than simplified domain coverage but can detect considerably more domain defects (*Site of Software Test Design*, 2020).

Domain testing can be applied at any test level. It generalizes BVA and EP (see ISTQB-CTFL, v4.0.1, Section 4.2) to more complex domains. Various approaches to domain testing can be found in textbooks like (Beizer, 1990, Ch. 6; Binder, 2000, Section 10.2.4; Kaner; Padmanabhan, et al., 2013; Jorgensen, 2014, Ch. 5).

3.1.2 Combinatorial Testing

Some software failures arise from specific combinations of parameter values, known as interaction failures. Combinatorial testing aims to reveal such failures by exploring these parameter value combinations.

In combinatorial testing, the test conditions usually combine configuration parameters or input data values. Hence, there are two main approaches to combinatorial testing. The first approach uses combinations of configuration parameters, and each of these combinations can be tested using the same test case. The second approach uses combinations of input data values, which then become part of complete test cases, creating a test suite for the system under test (D. R. Kuhn et al., 2013).

A specific parameter-value pair consists of a parameter and its value (e.g., '(color, red)').

The combinatorial coverage criteria include the following (Ammann et al., 2008), (Forgács et al., 2019):

- **Base choice coverage** assumes that some parameter-value pairs are more important than others. A base parameter-value pair is chosen for each parameter, and a base coverage item is the combination of base parameter-value pairs. Subsequent coverage items are created from the base coverage item for each parameter by replacing its base parameter-value pair with each non-base value.
- **Pairwise coverage**, in which coverage items are pairs of parameter-value pairs for any two parameters. Tools are available for generating coverage items. However, finding a minimal set of test cases achieving pairwise coverage is generally difficult.

For parameters with many values, EP may first be applied to reduce this number and the set of resulting combinations. Capturing the parameters and their values in a classification tree or a feature model (see IREB-Glossary, 2024) supports this activity. The final number of test cases may be affected by constraints between parameter-value pairs, by manually added combinations known to be problematic, or because of invalid or infeasible combinations.

The critical insight for combinatorial testing is that not every parameter contributes to a failure and that most failures are triggered by a single parameter value or interactions between a relatively small number of parameters (Cohen et al., 1994). This aligns with the coupling effect hypothesis, indicating that detecting simple defects in a program can often uncover complex defects as well (Offutt, 1992). In a limited study (D. Kuhn et al., 2004), the results showed that about 97% of failures are caused by only one or two interacting conditions, which indicates that pairwise testing is an effective test technique.

More information on combinatorial testing, including other coverage criteria like diff-pair-t or n-wise testing, can be found in (Ammann et al., 2008), (Forgács et al., 2019). Tools supporting combinatorial testing are also available at (Czerwonka, 2004).

3.1.3 Random Testing

Random testing involves selecting test data randomly from the input domain of the test item based on a specified probability distribution. For validation purposes, a distribution based on operational profiles is recommended. For verification purposes, the distribution should be usage-agnostic to avoid biases. Expected results might be added to the test cases using a test oracle, which most often requires an automated test oracle.

Random testing can be guided or unguided. In unguided random testing, the probability distribution remains fixed throughout the process. Guided random testing, exemplified by techniques like adaptive random testing (Huang et al., 2019), adjusts the distribution based on previously selected values, evolving over time. Guided random testing aims to cover the input domain effectively, considering defects often cluster in specific domain regions.

Random testing lacks recognized coverage criteria. Therefore, the exit criteria can only rely on the number of tests executed, testing time, or similar measures of completion.

Random testing is especially valuable when domain knowledge is limited or there is a need for a large volume of test data. It is cost-effective and provides, in probabilistic terms, insights into test object reliability. Random testing helps to avoid biases such as overlooking defects in manual testing due to misplaced trust in some code or functionality. However, random testing also has several challenges and limitations, including neglecting data semantics, potentially missing defects related to data meaning, overlooking certain defects, generating redundant tests, dependency on an automated test oracle, and

random outputs leading to inconsistent test results. Balancing the advantages and limitations of random testing is essential in each testing context.

Traditionally, random testing is considered less effective than other test techniques. In recent years, this hypothesis has been investigated in many empirical studies. The finding is that under the circumstances mentioned above, random testing can be more effective and efficient than other data-based test techniques (Arcuri et al., 2012), (Wu et al., 2020). Random testing is also applied in fuzz testing and chaos engineering.

3.2 Behavior-Based Test Techniques

Behavior-based test techniques derive test cases from specifications of the dynamic (i.e., state-dependent) behavior of the test item. This syllabus discusses three behavior-based test techniques:

- CRUD testing
- state transition testing
- scenario-based testing

CRUD testing and scenario-based testing extend the range of black-box test techniques known from (ISTQB-CTFL, v4.0.1, Section 4.2). This syllabus supplements state transition testing with additional coverage criteria.

3.2.1 CRUD Testing

CRUD testing verifies the lifecycle of data entities processed by the test item. CRUD stands for *create, read, update, and delete*. These are the four basic operations that functions can perform on entities.

The CRUD matrix gives an overview of the lifecycle of the data entities. Its columns represent the entities, and its rows represent functions. Suppose a function executes one or more particular create, read, update, or delete operations on a given entity. This is shown in the matrix using the initials of these operations: C, R, U, or D. To create a CRUD matrix, the TA determines for each function which of the four operations it carries out on which entities. Special attention should be given to the read operation, often implicitly linked to the C-U-D operations.

CRUD testing consists of two parts (Koomen et al., 2006):

- **Completeness testing** is static testing that verifies if all possible operations (i.e., C, R, U, and D) occur with every entity (i.e., if the entire lifecycle has been implemented for every entity). The absence of an operation is an anomaly that requires investigation.
- **Consistency testing** is dynamic testing aimed at integrating the various functions and checking whether the entity is used consistently. It verifies that the functions interact correctly when handling an entity. The test cases should cover all operations in the CRUD matrix. In addition, negative testing should also be included (e.g., reading an entity that has not yet been created). Test cases are designed per entity by combining functions to cover its entire lifecycle.

CRUD coverage is measured by the number of operations executed by the test suite divided by the total number of operations in the CRUD matrix. A more rigorous CRUD coverage can consider specific combinations of operations as coverage items (e.g., after each U, all possible Rs should be covered).

CRUD testing is primarily used at the system level. It focuses on defects in the behavior of the test item in treating entities, such as data integrity violations, access control defects, or data inconsistencies.

3.2.2 State Transition Testing

Many complex systems are stateful (i.e., the reaction of the system to an event depends on the current state of the system). Examples of stateful systems are embedded systems, dialog-based systems, control systems, or systems that deal with entities and their lifecycles.

A state simplifies complex internal details, which makes it easier for stakeholders to understand the expected behavior. During test analysis, the TA must ensure that the state-based model represents the expected behavior of the test item at the level of detail needed for testing. If the test basis already contains such a model, the TA must check whether it contains the test conditions and, if necessary, adapt it or design a new model.

In state-based models, the nodes represent states, and the edges represent state transitions. State transitions are triggered by events and may include guard conditions and actions (see ISTQB-CTFL, v4.0.1, Section 4.2.4). Several variants of these models are in use, such as extended finite state machines (Bochmann et al., 1994), Harel state charts (Harel, 1987), or UML state machines (*OMG® UML*, 2017).

In addition to the state transition coverage criteria discussed in (ISTQB-CTFL, v4.0.1), two further criteria are discussed below, for which a high defect detection effectiveness has been empirically proven:

- **N-switch coverage** applies to valid sequences of N+1 consecutive transitions, also called N-switches (Chow, 1978). 0-switch coverage equals the *valid transitions coverage* (see ISTQB-CTFL, v4.0.1, Section 4.2.4). A 1-switch is a pair of incoming and outgoing transitions at a state. Extending N-switches at their end by valid subsequent state transitions results in N+1 switches. 0- and 1-switch coverage are frequently used in practice. A 100% 2-switch or higher coverage is only indicated for a high risk of failure due to unexpected sequences of events, as the number of N-switches can grow exponentially with N.
- **Round-trip coverage** (Ammann et al., 2008) applies to paths in a state-based model that form loops. A round trip is a loop in which the start and end states are the same, and no other state in this loop occurs twice. The coverage items are the round trips. (Antoniol et al., 2002) found this criterion highly effective in detecting defects.

State transition testing is well suited for automation using model-based testing tools. As with most black-box test techniques, state transition testing contributes to defect prevention by detecting defects in the specification during modeling. State transition testing can be applied at any test level.

Further state transition coverage criteria are discussed in (Rechtberger et al., 2022). A recent state-based model is the action-state model discussed in (Forgács et al., 2024).

3.2.3 Scenario-Based Testing

Scenario-based testing evaluates the test item's behavior in realistic scenarios. Techniques such as user research, user stories, personas, and user journey maps (see *Section 11*) may help to identify valuable scenarios. In scenario-based testing, the TA creates a scenario model based on sequences of actions that constitute workflows through the test item (cf. *ISO/IEC/IEEE 29119-4*, 2021). This syllabus discusses the following two models: activity diagrams and use cases. Other models include flowcharts, business process model and notation diagrams (*OMG® BPMN*, 2013), sequence diagrams, or collaboration

diagrams (*OMG® UML*, 2017). These models are not discussed in this syllabus. Please refer to (ISTQB-AcT, v1.0) for more details.

An **activity diagram** is a graphical representation of the workflow within a system. Activity diagrams are particularly useful for modeling business processes but can also model control flow. Activity diagrams extend the notation of flowcharts, allowing for the modeling of concurrency. The main elements of activity diagrams are start and end nodes, actions, transitions, decision nodes, merge nodes, fork nodes, join nodes, and swim lanes.

A **use case** is a textual or graphical description of actions representing the interactions between a user and a system or between systems. Three types of scenarios are distinguished in the model:

- **Main scenario** (so-called “happy path”) – a typical, expected sequence of actions leading to the achievement of a specific goal from the user’s perspective. A use case can only have one main scenario.
- **Extension** (or alternative scenario) – a sequence of actions, other than the main scenario, that eventually leads to achieving the goal of the main scenario.
- **Exception** - a sequence of actions that does not allow the achievement of the goal of the main scenario due to an unexpected action (e.g., abnormal use or invalid input).

In scenario-based testing, the TA designs test cases to cover the scenarios (i.e., coverage items), often following a risk-based prioritization. When the scenario model does not contain loops, each scenario can be tested with a separate test case (i.e., all possible scenarios in the model can be exercised with a test suite). The number of potential scenarios (paths) may be infinite when loops exist. In this case, the simple loop coverage can be applied to the scenario model. Simple loop coverage requires testing when each loop is executed zero times (i.e., skipped), with exactly one iteration, with more than one iteration (i.e., a typical number of iterations), and with the maximum number of iterations (if possible).

Scenario-based coverage is measured by the number of executed scenarios divided by the number of all identified scenarios. Scenarios can be identified by applying various coverage criteria to the scenario model (see Koomen et al., 2006). Coverage criteria may require testing each scenario more than once. For example, a scenario may require additional EP or BVA coverage of variables occurring in the scenario. In such cases, one scenario may need to be tested by more than one test case.

Scenario-based testing is often performed in system testing or acceptance testing. It takes the form of end-to-end testing focused on a system’s functional suitability (see *Section 4.1*) from the user’s perspective. However, scenario-based testing can also be used at other test levels (e.g., component integration testing with scenarios based on the interaction protocols or component testing of stateful, object-oriented classes with scenarios invoking various methods) and in non-functional testing (e.g., scenarios may constitute the elements of the operational profiles used in reliability, flexibility, or compatibility testing).

3.3 Rule-Based Test Techniques

Rule-based test techniques verify the implementation of the stateless behavior of a test item, specified by rules that are valid regardless of its state (e.g., business rules). In this syllabus, two rule-based test techniques are discussed, namely:

- decision table testing

- metamorphic testing

The Foundation Level Syllabus (ISTQB-CTFL, v4.0.1) covers the basics of decision table testing. This syllabus discusses more advanced topics. The terminology and notation used follow the standard (*OMG® DMN*, 2024).

3.3.1 Decision Table Testing

In decision table testing, the TA usually begins by creating a full decision table or analyzing an existing one obtained from the test basis. The number of rules of a full decision table is the product of the numbers of values of its conditions. This number grows exponentially with the number of conditions and their values and motivates the minimization of the decision tables.

Decision tables can be minimized by merging rules using the don't care operator, '–'. It is recommended to disregard infeasible rules (i.e., rules with a combination of condition values that can never occur) when merging rules. Often, infeasible rules are removed from the decision table before merging. A possible systematic minimization algorithm looks out for action-equivalent rules that only differ in one condition and cover all its possible values. These rules are merged, and the differing condition value is replaced by '–'. The result of systematic minimization can depend on the order in which the columns are minimized, not always leading to an optimal solution. The TA has to check whether any further minimization is possible.

It is the task of the TA to review the decision table, possibly with other stakeholders, with the following criteria:

- consistency (i.e., if two different rules apply to the same combination of condition values, then they are action-equivalent)
- feasibility (i.e., contains no infeasible rules)
- completeness (i.e., no feasible combination of condition values is missing)
- correctness (i.e., the rules model the system's intended behavior)

In addition, it is advisable that rules do not overlap (i.e., for any combination of condition values, at most one rule applies). Overlapping rules may happen when the original decision table is already minimized or if rules are merged incorrectly.

The *checksum procedure* uses the number of rules in a minimized decision table to indicate overlaps and gaps. For each rule in the minimized table, the number of rules it represents in the original decision table is calculated. A rule without a '–' in the conditions (i.e., with individual values for all conditions) scores 1. Each '–' value multiplies the rule's score by the number of individual values for the respective condition. The sum of the rule scores is the checksum of the minimized decision table. If the checksum is less than the checksum of the original decision table, the minimized decision table is incomplete. If the checksum is higher, some rules overlap, or additional rules exist (e.g., infeasible combinations of conditions). If minimization was performed correctly, the checksums are equal. Equal checksums alone do not guarantee that the minimized decision table is equivalent to the original one.

Decision table coverage is measured by dividing the number of columns exercised by the total number of feasible columns in the decision table. When implementing test cases resulting from a decision table rule, the TA must implement the conditions and actions. The TA has to decide how to implement the '–' condition values for a given rule because '–' represents at least two values. However, if the risk level associated with the decision table is high, the TA should refrain from minimizing and should measure the decision table coverage of the feasible columns in the full decision table.

3.3.2 Metamorphic Testing

Metamorphic testing (MT) is a technique aimed at generating test cases that are based on an existing source test case. One or more follow-up test cases are generated by changing (metamorphizing) the source test case based on a metamorphic relation (MR). The MR defines a property of the test item and describes how a change in a test case's inputs is reflected in the test case's expected results.

The TA combines the source and follow-up test cases of an MR to a test procedure, with a joint result evaluation. If they fulfill the metamorphic relation, the test passes, otherwise it fails. In case of failure, the subsequent debugging must determine which of the individual test cases involved has failed.

For example, consider a function that determines the average of a series of numbers. A source test case is created with a series of numbers and an expected average, and the test case is run to confirm that it passes. An MR could state that any permutation of the series of numbers results in the same average. Using this MR, the TA can create several follow-up test cases, each with the same set of numbers in the input but in a different order. The expected result remains the same.

For the same average function, the TA can also use another MR, which states that if each number of the series is multiplied by the same number, x, then the expected result will also be multiplied by x. With this MR, the TA can create any number of follow-up test cases from a source test case by choosing different values of x. This can be particularly useful when test design and test execution are automated. The TA can also combine two or more MRs to create follow-up test cases (e.g., permute and multiply by 2).

MT can also be used in the presence of a test oracle problem (see *Section 1.3.4*). In such a situation, the expected results of the source test case and the follow-up test cases are not available, so their test results cannot be evaluated individually. An example can be an AI-based actuarial program that predicts the age at death based on a large dataset. The MR might state that if the number of cigarettes smoked is increased, then the predicted age of death should decrease.

Currently, there are no recognized coverage measures for MT that provide useful exit criteria. Covering each MR once is insufficient because only partial verification of the expected results can be obtained. The TA can combine MT with random testing to generate many low-level source test cases and follow-up test cases for the same MR. For example, in the average calculation mentioned above, a random number generator can be used to generate various inputs for the source test case, as well as random permutations and multipliers for follow-up test cases.

MT can be used for most test items and applied to functional and non-functional testing (e.g., load testing, generating load by follow-up test cases using MRs, or installability testing with various installation parameters that can be selected in multiple sequences). It is a preferred test technique for AI-based systems (ISTQB-AI, v1.0).

For more details, see also the standard (*ISO/IEC/IEEE 29119-4*, 2021) or the survey articles (Segura; Towey, et al., 2020) and (Segura; Fraser, et al., 2016).

3.4 Experience-Based Testing

A TA employs experience-based test approaches and test techniques, leveraging their expertise and past encounters to guide testing. This chapter details the TA's use of documentation in session-based testing and checklist-based testing (see ISTQB-CTFL, v4.0.1, Sections 4.4.2 and 4.4.3). In addition, crowd testing is discussed. All experience-based test techniques can be applied in collaboration-based testing, like acceptance test-driven development. For more details, see (ISTQB-CTFL, v4.0.1, Section 4.5).

3.4.1 Test Charters Supporting Session-Based Testing

In exploratory testing, a test charter provides the mission of a test session, which outlines its scope and objectives and information such as limitations, timelines, and risks. It serves as a roadmap for testing, providing structure to the test sessions. Test charters help a TA stay focused on specific areas or features to be tested while allowing them the flexibility to explore the system when necessary. The test charter does not specify the test suites that will be executed in each test session.

When preparing a test charter for session-based testing, the TA must consider certain factors that influence the test charter design, in particular:

- customer and requirement factors (e.g., requirements elicited from clients, business use cases for the system, quality requirements), and user journey maps (i.e., user interaction with the system over time)
- product factors (e.g., functional flows, principal goals of the product, product features, software design, and interfaces)
- project management factors (e.g., time constraints, project purpose, estimated effort, and business value)

A test charter consists of a mission that describes the test objective and various additional information.

A popular lightweight format of a mission is “Explore [target] With [resources] To discover [information]” (Hendrickson, 2013), where [target] describes what is to be explored (e.g., area, feature, risk, component, and requirement), [resources] describe what the TA will use (e.g., test data, configurations, tools, restrictions, heuristics, and dependencies), and [information] explains which type of information the TA is aiming to find (e.g., quality characteristic evaluations, expected type of defects, and violations of standards).

Test charters may contain but are not limited to the following additional information (Ghazi et al., 2017):

- organizational information (e.g., duration of the test session, start date and time, and tester’s name)
- test objectives (e.g., motivation of the test and mission of the test charter)
- test scope (e.g., specific areas of interest within the system under test, test level, test techniques to be used, test ideas, exit criteria, priorities, what the test charter is supposed to cover, and a description of what will not be tested)
- entry criteria (i.e., preconditions that must be met to be able to start the test session)
- product-related information (e.g., definition, data, and workflow among components, and system architecture)
- limitations (i.e., what the product must never do)
- description of the test environment
- existing data sources, product information, and test tools that would aid testing
- historical information (e.g., previously found defects such as compatibility and interoperability defects, current open questions that refer to existing anomalies, and test-related failure patterns of the past)
- constraints and risks (e.g., regulations, rules, and standards used)

During a test session, the TA records test logs that include questions, observations, or ideas for future testing along with the test results. This data is documented in a session sheet.

The scope and detail of information included in specific test charters may vary and affect the degree of flexibility of the TA. For example, defining only the general test objectives provides ample room for exploration, while adding the information on test techniques to be used may constrain the TA. On the other hand, adding information (e.g., what the system definitely cannot do) can help lower the likelihood of reporting false-positive results.

3.4.2 Checklists Supporting Experience-Based Test Techniques

Checklist-based testing is a widely used test technique due to its adaptability, simplicity, and effectiveness in ensuring software quality. By using checklists, the TA ensures that they cover all known essential aspects of a test item, which prevents overlooking critical areas. They also introduce consistency across test cycles and among various TAs. When using a checklist, the TA focuses on important aspects. The checklists are a form of recording the TA's past experiences with failures and defects. They serve as a reminder or a source of inspiration if the TA runs out of ideas (e.g., during exploratory testing). The TA saves time by reusing a standard checklist or a checklist created by one of their peers, but only if these checklists are relevant to the test item. Checklists help reduce the work needed to document test cases, which can be a great asset when requirements and software are constantly changing.

Preparing checklists is often the TA's responsibility. Checklists support experience-based testing, as they help organize, structure, and guide the testing. Creating a reusable, maintainable, clear, and efficient checklist requires effort.

The following steps can be a guideline for setting up a proper checklist for experience-based testing:

First, the TA determines the checklist's scope, objectives, and format because they influence the testing depth and required level of detail. A read-do checklist contains the major elements to consider for a certain process (e.g., checklist items contain concrete invalid inputs for an input field to be checked). A do-confirm checklist serves as an aid to guide the thought process, providing experience-based testing ideas to explore an application further (e.g., a checklist item might require checking whether the search results are relevant).

Next, the TA collects the information necessary to define the checklist items. This can include gathering insights from experienced professionals, browsing defect libraries and defect taxonomies (Beizer, 1990), (Kaner; Falk, et al., 1999), reviewing relevant documentation, and analyzing risks, test cases, and potential scenarios. Checklist items should be clear, specific, unambiguous, consistent, relevant, maintainable, actionable, and measurable. They should be formulated as questions that can be answered with 'yes', 'no', or 'not applicable'. They require assigned priorities based on their importance, potential impact, and risk level. Checklists are not comprehensive how-to guides. Instead, they are quick and simple tools for expert professionals.

Finally, the TA structures and organizes the checklist by categorizing checklist items into logical groups based on functional areas, user roles, test levels, or other relevant criteria. Creating separate categories can be especially useful for a long checklist.

Where possible, templates and standards should be used. The TA can save effort by utilizing existing templates or predefined checklists aligned with industry standards and best practices.

A checklist is never finalized. The TA continually reviews and refines it, adapting it to reflect new findings, changed priorities, feedback from other testers, or lessons learned from previous test cycles. By sharing

the checklist with other testers, the TA promotes consistency and collaboration and helps them better understand the test items and the critical areas to focus on during testing.

3.4.3 Crowd Testing

Crowd testing distributes tests among a group of internal or external testers of diverse backgrounds and locations. It can be a cost-effective way to validate usability and covers both functional and non-functional quality characteristics (Alyahya, 2020), (Leicht et al., 2017).

Some of the benefits of crowd testing include the following:

- **Diverse test environments**. Testers can be in various geographical locations and use various environment configurations with a wide variety of devices, browsers, and network conditions.
- **More flexibility**. Easily scalable to handle many tests in a short time.
- **Cost-effectiveness**. Crowd testing is typically less expensive than maintaining a large and diverse in-house test team or supplementing with external testing services.
- **Rapid feedback**. Testers can provide quick feedback, helping to identify and fix failures early.
- **Real user perspective**. Testers can be actual users of the application and can better provide insights into its user experience and usability. This can be especially valuable in user acceptance testing.
- **Variability**. As tests are executed by a wide variety of testers every time, they are not as repeatable. While this could be a limitation, it also results in wider coverage, which increases the chances of finding defects.

Some of the limitations of crowd testing include the following:

- **Unreliable quality of testing**. Testing quality can vary significantly depending on the skills of individual testers, although this may not be relevant, for example, when the goal is feedback on user experience.
- **Communication challenges**. Coordinating with many testers from various locations with varying time zones, cultural differences, and language barriers can be challenging.
- **Security**. Sharing software with external testers poses data security and confidentiality risks. Proper measures can mitigate these risks, allowing for the responsible use of crowd testing without disclosing sensitive details or facilitating plagiarism.
- **Documentation and reporting**. Ensuring comprehensive test documentation and managing a large number of findings, including duplicates and false positives, can be challenging when dealing with a large and diverse group of testers.

Crowd testing is an approach that does not replace TAs applying test techniques but can increase the coverage of diverse test environments.

3.5 Applying the Most Appropriate Test Techniques

Testing should be as effective and efficient as possible in the given context. To this end, the TA supports the test manager in selecting the most appropriate test technique(s). In addition, the TA can use

automation to support the test activities. This includes automating the test design, described in this section, and supporting test execution automation, described in *Section 1.3.6*.

3.5.1 Selecting Test Techniques to Mitigate Product Risks

Selecting the most appropriate test technique(s) is crucial to effective and efficient product risk mitigation and is influenced by many factors, including the following.

Test objectives (see ISTQB-CTFL, v4.0.1, Section 1.1.1) specify which aspects of the test object to evaluate. They guide the choice of test techniques and depend on the system type (e.g., domain-based testing for numerical calculations in engineering versus decision table testing in credit risk management).

Product risks are associated with potential defects. These defects can best be detected using particular test techniques, as most test techniques focus on detecting specific types of defects (see *Section 3.1*, *Section 3.2*, *Section 3.3* and *Section 3.4* in this syllabus). For example:

- Data-based test techniques can detect defects in data handling, domain implementation, user interfaces, calculations, and parameter combinations.
- Behavior-based test techniques can detect user requirement defects, such as missing features, communication defects, and processing defects.
- Rule-based test techniques can detect defects in logic and control flows.

Risk analysis also helps determine the appropriate test approach, for example:

- Coverage-based exit criteria. The higher the risk level, the more rigorous coverage may be required (e.g., all combinations in combinatorial coverage instead of pairwise coverage). However, the TA must always take into account the trade-off between coverage strength and required test effort.
- Experience-based testing can be used when defining coverage is difficult, when a risk level is low, or when the project schedule is tight.

Test basis. If the specification of the test object is using models, the TA can use test techniques based on these models. If it is impossible or difficult to derive a test oracle from the test basis, test techniques like metamorphic testing or experience-based test techniques can be applied.

Knowledge of recurring defect types may indicate selecting the test techniques that focus on detecting such defects (e.g., checklist-based testing). If the expectation is to discover similar defects as in previous iterations or projects, using the same successful test techniques may be reasonable.

Tester's knowledge and experience. If the TA is not familiar with a given test technique, it is not recommended to use it in critical test assignments. Domain knowledge may also impact the selection of test techniques. For example, little or no domain knowledge indicates that test techniques like exploratory testing may be ineffective.

Software development lifecycle used. A sequential development model is conducive to employing more formal techniques. In contrast, an iterative development model might be more appropriate for adopting lightweight test techniques (e.g., experience-based test techniques) or when the test design can be automated.

Customer and contract requirements. Contracts can explicitly require performing specific testing (e.g., in terms of certain test levels or test types), which influences the selection of test techniques (e.g., acceptance criteria with a set of scenarios provided by the client suggest the use of a scenario-based test technique).

Regulatory requirements. When a project follows a standard, it may require the use of specific test techniques. For example, the standard (*ISO 26262*, 2018) requires the use of test techniques such as equivalence partitioning, boundary value analysis, or error guessing, depending on the ASIL (Automotive Safety Integrity Level) assigned to a test object.

Project constraints, such as time and budget, may affect the use of time-consuming techniques or those requiring expensive resources.

Test techniques are often combined to increase the efficiency and effectiveness of defect detection. For example:

- BVA can be used for guard conditions in state transition testing.
- Domain testing can be used in scenario-based testing to determine the value of a condition from a decision table or a variable occurring in a system under test.
- Scenario-based testing can be combined with decision coverage, a white-box test technique discussed in (ISTQB-TTA, v4.0), to rigorously cover decisions in the business process. For an example, see process cycle testing in (Koomen et al., 2006).
- Scenario-based testing can be combined with round-trip coverage to address the specific risks of cyclical business process activities.

3.5.2 Benefits and Risks of Automating the Test Design

The TA may use tools to apply test techniques, especially black-box test techniques. When automating the test design, the TA creates a test model and automatically generates testware from that model. For example, the TA designs a state transition model and lets a model-based testing tool generate test cases for round-trip coverage.

Automating the test design often improves the efficiency and effectiveness of testing. Its benefits include:

- **Defect prevention**. Modeling for testing is an effective way of evaluating the quality of the test basis (see *Section 5.2.1*).
- **Extended capability**. Automation allows for the application of more complex test techniques and coverage criteria such as combinatorial testing, random testing, or N-switch coverage, reducing the risk of untested code.
- **Improved comprehensibility**. Test selection criteria specified in a tool refer more clearly and visibly to the test conditions and justify the generated coverage more comprehensibly.
- **Less repetitive work**. Testware can be generated from the test model, reducing manual, repetitive work such as specifying tests.
- **Less maintenance efforts**. The test model is the single source of truth used to derive the testware. As a result, only the test model needs to be maintained.
- **Less defective testware**. Manual work is prone to errors. Testware generated by tools has a higher quality and consistency.
- **Enhanced team collaboration**. Stakeholders may review the test model to find defects or to better understand the test conditions.

- **Enhanced traceability**. It is easier to link the elements of a test model to the test conditions than the test cases themselves. If supported by the tool, the generated test cases will inherit those links, improving overall traceability in testing.
- **Various output formats**. Testware can be generated in various output formats as required for other tools and subsequent activities.

The TA must also consider the risks of automating the test design. They include overlooking test conditions that are not shown in the model, underestimating the maintenance effort of the test model, stakeholders finding the model difficult to understand, and the generic risks of test automation (see ISTQB-CTFL, v4.0.1, Section 6.2).

4 Testing Quality Characteristics – 60 minutes

Keywords

adaptability, compatibility, flexibility, functional appropriateness, functional completeness, functional correctness, functional suitability, functional testing, installability, interaction capability, interoperability, usability, user experience

Learning Objectives for Chapter 4:

4.1 Functional Testing

TA-4.1.1 (K2) Differentiate between functional correctness, functional appropriateness, and functional completeness testing

4.2 Usability Testing

TA-4.2.1 (K2) Explain how the test analyst contributes to usability testing

4.3 Flexibility Testing

TA-4.3.1 (K2) Explain how the test analyst contributes to adaptability and installability testing

4.4 Compatibility Testing

TA-4.4.1 (K2) Explain how the test analyst contributes to interoperability testing

Introduction to Testing Quality Characteristics

This syllabus uses the software quality model provided in ISO 25010 (*ISO/IEC 25010*, 2023) as a guide and discusses the quality characteristics in focus for a TA. However, the usability terminology used differs from this standard to be consistent with the Usability Testing Syllabus (ISTQB-UT, v1.0).

Appendix F provides an overview of the quality model of the current ISO 25010 standard, the changes compared to the previous version, and the related ISTQB® syllabi that focus on testing specific quality characteristics.

4.1 Functional Testing

Functional testing is one of the core tasks of the TA. While (ISTQB-CTFL, v4.0.1) briefly summarizes functional testing as a test type, this syllabus goes into more detail, discussing the sub-characteristics of functional suitability.

4.1.1 Sub-characteristics of Functional Suitability

The product quality model of ISO 25010 (*ISO/IEC 25010*, 2023) differentiates between three sub-characteristics of functional suitability. Although all of them can be assessed by functional testing, not every functional test activity addresses them equally well. The TA should be able to select the appropriate test levels and test techniques to address product risks related to a specific sub-characteristic of functional suitability.

Functional completeness testing should cover all specified tasks of the software and the intended users' objectives. The key question is whether everything that is asked for is implemented.

Functional completeness should be addressed as early as possible by reviewing the requirements specification in sequential development models and by discussing user stories, including acceptance criteria, during collaborative user story writing in Agile software development. In system testing, system integration testing, and acceptance testing, functional completeness can be tested dynamically.

Behavior-based test techniques, such as scenario-based testing, are a good fit, although other black-box test techniques are also suitable. Traceability between test basis, test conditions, and test cases is essential when determining the achieved level of functional completeness.

Functional correctness testing answers the question of whether the actual results are correct (e.g., accurate, precise, and consistent) for valid and invalid inputs. It is crucial to find an effective test oracle that provides the expected results in detail.

Functional correctness can be tested at any test level. In terms of shift left, most functional correctness testing should occur in component testing and component integration testing. Even if the TA is not responsible for these test levels, the TA should contribute to them to best achieve the test objectives.

All black-box test techniques, experience-based test techniques, and collaboration-based testing are suitable.

Functional appropriateness testing verifies that the functions facilitate accomplishing specified tasks and objectives. The focus is on whether everything implemented fulfills the users' needs.

Functional appropriateness testing may include user interface design reviews, especially for interactive applications. Dynamic testing starts with system testing and acceptance testing in sequential development

models, as well as demo sessions in Agile software development.

Exploratory testing and collaboration-based testing are most appropriate. In addition, behavior-based black-box test techniques are also suitable.

4.2 Usability Testing

Usability refers to a broad concept of user-related quality characteristics, covering interaction capability from the product quality model (*ISO/IEC 25010*, 2023) and beneficialness from the ISO 25019 quality-in-use model (*ISO/IEC 25019*, 2023). More on usability testing can be found in (ISTQB-UT, v1.0), (*ISO 9241-210*, 2019), and (UXQB-FL, v4.01).

4.2.1 Contribution of the Test Analyst to Usability Testing

Usability testing usually focuses on evaluating the following aspects:

- interaction capability - enabling users to complete tasks in specific contexts of use (*ISO/IEC 25010*, 2023) effectively, efficiently, and satisfactorily
- user experience - addressing the users' perceptions before, during, and after interacting with the test object
- accessibility - ensuring that users with disabilities, diverse cultural backgrounds, or language barriers can use the system both effectively and efficiently

The TA can collaborate in usability testing from an early phase by using their knowledge of the target user groups, their goals, their context of use, potential difficulties using the system, or negative user experience.

The TA can contribute to the principal usability evaluation techniques as follows:

- **Usability reviews** are performed by usability experts to identify potential usability problems and deviations from established criteria. Usability reviews may vary from informal reviews to inspections. The TA can adapt the review criteria to the specific needs of the user groups, the particular business objectives and priorities, and the context of use (e.g., tailoring a generic usability checklist).
- **Usability test sessions** involve future users or their representatives trying to solve predefined tasks to evaluate if tasks can be completed effectively, efficiently, and satisfactorily. The TA may contribute to designing scenarios for the usability test sessions according to personas, user groups, or operational profiles.
- **User questionnaires or surveys** including rating and feedback measure user satisfaction. Examples of user questionnaires are SUMI (Software Usability Measurement Inventory SUMI, 1991) and WAMMI (Website Analysis and Measurement Inventory WAMMI, 1999). The TA can help design a questionnaire and evaluate responses to address the specific goals of the targeted users within their context of use.

In accessibility testing, a common test objective is to verify compliance with standards. The international standard Web Content Accessibility Guidelines (WCAG, 2023) defines three levels of conformance (A, AA, and AAA) that represent increasing degrees of web content accessibility. National standards include the United Kingdom's Equality Act (UK Government, 2010) and the United States' Americans Disabilities Act (U.S. Department of Justice, 2010). The TA can identify the required compliance level and the specific needs of the intended target group by analyzing the context of use.

4.3 Flexibility Testing

Flexibility testing (also known as portability testing) verifies that the test object can be adapted to changes in its contexts of use or system environment. The ISO 25010 product quality model (*ISO/IEC 25010*, 2023) differentiates between the following sub-characteristics of flexibility:

- adaptability
- scalability
- installability
- replaceability

Flexibility has both technical and non-technical aspects. This section focuses on how the TA can contribute to adaptability and installability testing. Scalability and replaceability are related to technical aspects and discussed in (ISTQB-PT, v1.0) and (ISTQB-TTA, v4.0), respectively.

4.3.1 Contribution of the Test Analyst to Adaptability Testing and Installability Testing

Adaptability testing verifies that the test object can be adapted for or transferred to the intended target hardware, software, or other operational or usage environments.

The TA supports adaptability testing by identifying the intended target environments (e.g., versions of mobile operating systems supported and versions of browsers that may be used) and designing tests that cover combinations of these environments. Because this requires test data representing various environment parameter configurations, test techniques such as combinatorial testing are often applied (see *Section 3.1.2*). Another example is integrating various platform components into customer projects to ensure compatibility across target environments. Depending on product risk, the TA designs and executes smoke tests or a more comprehensive test suite to verify that the test object is adapted to the target environment correctly.

By following adaptability testing good practices (e.g., defining target environments early, using combinatorial testing, smoke testing on new environments, and monitoring environment-specific defects), the TA can uncover defects that could limit the software's lifespan and usability (e.g., ease of use on different screen sizes). The TA's work in adaptability testing should also be supported by automated cross-platform testing implemented by test automation engineers. Rigorous adaptability testing can detect environment-specific issues early, helping to avoid frequent major updates or redesigns after deployment and lowering maintenance costs.

Installability testing verifies that the test object can be installed, uninstalled, updated, and reconfigured correctly in specified environments. Installability testing extends beyond merely checking if the installation procedure runs to completion.

The typical installability test objectives in focus for the TA are:

- To verify that the installation procedures are executed correctly under various environment parameter configurations. This makes test techniques like combinatorial testing helpful, similar to adaptability testing.
- To design and execute tests to determine whether the test object works properly after installation or update

- To check how easy it is for users to install, uninstall, or update the software. This includes reviewing the installation documentation.
- To test permissions-related behavior, particularly for mobile applications (see ISTQB-MAT, v1.0).

4.4 Compatibility Testing

Compatibility testing verifies whether a test object is compatible with other components or systems when it is used. The ISO 25010 product quality model (*ISO/IEC 25010*, 2023) differentiates between two sub-characteristics of compatibility: interoperability and coexistence. Therefore, compatibility testing can be subdivided into the following test types:

- **Interoperability testing**, which verifies compatibility with components or systems with which the test object is intended to interact. These tests are typically black-box functional tests, so the TA is usually responsible for them.
- **Coexistence testing**, which verifies that the test object can share its target environment with other components or systems without interference. This technical test type is discussed in the Technical Test Analyst Syllabus (ISTQB-TTA, v4.0).

4.4.1 Contribution of the Test Analyst to Interoperability Testing

The goal of interoperability testing is to verify that two or more components or systems can exchange information and mutually use the information that has been exchanged. When the information exchange involves a data transformation, interoperability testing must include verification of that transformation. Interoperability testing is particularly important when multiple systems need to collaborate, share data, or perform tasks together. This is especially the case in modern software architectures such as cloud solutions, web services, microservices, containerization, and the Internet of Things.

Interoperability usually happens on various architectural levels. The TA must understand the possible interactions to define proper test conditions to cover them. Not all interactions may be documented. The TA may indirectly retrieve information about the interactions from architecture and design documentation. Hence, it is crucial to understand this documentation to ensure that all important aspects of the interactions will be tested.

Interoperability testing can detect defects in:

- data transformations for the exchange of data
- interpretation or use of exchanged data
- communication flows and protocols
- compliance with standards
- end-to-end functionality
- design documentation

Interoperability testing is usually applied during integration testing. Black-box test techniques, such as data-based test techniques that focus on the exchanged data, behavior-based test techniques for interpreting or using the data, and end-to-end functionality or rule-based test techniques for data transformation, are well suited to interoperability testing. Experience-based test techniques can usefully

complement black-box testing. Test cases from functional testing or adaptability testing may be reused for interoperability testing.

Examples of general standards in interoperability are (*ISO 15745*, 2003) and (*ISO 16100*, 2009). ETSI (*ETSI EG 202 237 v1.2.1*, 2010) provides an example of a concrete interoperability testing methodology in the telecommunications domain.

5 Software Defect Prevention – 225 minutes

Keywords

ad hoc reviewing, checklist-based reviewing, defect prevention, model-based testing, perspective-based reading, review technique, role-based reviewing, root cause analysis, scenario-based reviewing, test result

Learning Objectives for Chapter 5:

5.1 Defect Prevention Practices

TA-5.1.1 (K2) Explain how the test analyst can contribute to defect prevention

5.2 Supporting Phase Containment

TA-5.2.1 (K3) Use a model of the test object to detect defects in a specification

TA-5.2.2 (K3) Apply a review technique to a test basis to find defects

5.3 Mitigating the Recurrence of Defects

TA-5.3.1 (K4) Analyze test results to identify potential improvements to defect detection

TA-5.3.2 (K2) Explain how defect classification supports root cause analysis

Introduction to Software Defect Prevention

The goal of defect prevention is to implement actions that reduce the likelihood of (re)occurrence of defects in work products and mitigate the propagation of defects to subsequent phases of the SDLC. These efforts result in a variety of important benefits, including reduced costs and labor, increased productivity, and improved product quality. Defect prevention is the responsibility of the whole team. The TA can utilize their specific knowledge and experience to contribute to it.

Defect prevention practices include:

- prevent defect introduction, which is a part of quality assurance activities
- prevent defects from escaping to subsequent phases of the SDLC (see *Section 5.2*)
- prevent defects from recurring (see *Section 5.3*)

5.1 Defect Prevention Practices

5.1.1 Contribution of the Test Analyst to Defect Prevention

The TA can contribute to defect prevention in several ways, using their domain knowledge, test expertise, and analytical skills. Examples include:

- Participating in risk analysis – ensuring that identified risks are properly mitigated (e.g., selecting the most adequate test techniques).
- Reviewing requirements, models, and specifications – allowing early detection of defects in the test basis preventing them from escaping into the code, thus significantly decreasing the cost of fixing them.
- Participating in retrospectives – allowing the identification of potential improvements in test analysis, test design, test implementation, and test execution (e.g., using more effective test techniques, targeting testing of specific areas of risk, or improving test data and test environments to reduce both false-positive results and false-negative results) to better prevent defects from escaping.
- Defect data collection and evaluation – supporting root cause analysis and process improvement by collecting detailed data on defects to enable their classification and statistical analysis and thus facilitate root cause analysis.
- Participating in root cause analysis – preventing defects from reoccurrence by proposing corrective actions to address the identified root causes.

In addition to participating in defect prevention, the TA assesses (usually in consultation with the test manager) whether the proposed measures have resulted in the desired effect. Examples of metrics that help to evaluate the effectiveness of these measures include:

- Defect removal efficiency (DRE) – measures the ratio of defects removed before the release and the total number of defects. The (unknown) denominator can be estimated or replaced by the total number of defects found up to a certain point in time (e.g., up to 6 months after the release). High DRE indicates fewer defects escape to production, which might imply better defect prevention practices. However, DRE does not distinguish between defects caught through prevention and detection.

- Phase containment effectiveness (PCE) – measures the number of defects introduced and removed in the same phase relative to the total number of defects introduced in that phase. High PCE indicates that fewer defects escape to later phases.
- Cost of quality – illustrates the relation between defect prevention, defect detection, and removal costs (see ISTQB-TM, v3.0, Section 3.2.1).

5.2 Supporting Phase Containment

The objective of phase containment is to detect and remove defects in the same phase of the SDLC in which they were introduced. In Agile software development, the shift left approach can be used similarly. This policy reduces the cost of quality. Because the test basis is an important input to test analysis and test design, the TA can best contribute to phase containment by evaluating the test basis quality. Early attention to test basis quality will minimize later effort and prevent the propagation of defects to subsequent phases. This syllabus discusses two common options for the TA to find defects in the test basis: modeling for testing purposes and reviewing the test basis using various review techniques (*ISO/IEC 20246*, 2017).

5.2.1 Using Models to Detect Defects

Modeling provides abstractions of a system at a certain level of precision and detail. It helps stakeholders better understand the system being developed. As discussed below, modeling can support phase containment in at least three ways:

- detecting defects in specifications
- detecting defects in models
- detecting defects in test objects by using model-based testing

Detecting defects in specifications. Specifications are often provided as informally written text. The TA can formally represent these specifications using models. When creating a model, the test conditions (e.g., requirements) are mapped to model elements and linked to them for traceability. Formalization and visualization of the test conditions in a model efficiently reveal defects such as incompleteness, inconsistencies, or ambiguities. The strength of modeling is that the specification is transformed while reviews check it in its original form. As a result, the TA also contributes to finding appropriate solutions for the defects detected.

Data-based models include domain models and combinatorial testing models. They allow the TA to detect domain defects such as overlapping partitions, gaps in the domain coverage, empty partitions, or missing or incorrect combinations of parameters.

Behavior-based models include CRUD matrices, state-based models, or scenario models. They allow the TA to detect incomplete or inconsistent entity lifecycles, missing or faulty state transitions, deadlocks, endless loops, ambiguous or inconsistent system behavior, or missing exception handling.

Rule-based models like decision tables or metamorphic relations allow the TA to detect defects in business rules, such as omissions, inconsistencies, ambiguities, redundancies, error-prone scenarios, or complex business logic.

Modeling can also detect defects such as inconsistencies in naming, data values (e.g., boundaries), and inputs or outputs. It can also identify missing, incomplete, ambiguous, or unnecessary information.

Detecting defects in models. If the test basis contains models, the TA can analyze these models to detect defects. This syllabus discusses defect detection in three typical models used in software testing: state transition diagrams (see *Section 3.2.2*), activity diagrams (see *Section 3.2.3*), and decision tables (see *Section 3.3.1*). Examples of defects in the models mentioned above include:

- state transition diagrams – missing/wrong states, improper transitions, incorrect guard conditions or actions, redundant or unreachable states, and nondeterministic behavior
- activity diagrams – missing, unreachable, or dead-end actions, incorrect order of actions, wrong, non-exclusive, or incomplete guard conditions in decision nodes, missing synchronization points, or improperly synchronized parallel flows that can lead to unintended behavior
- decision tables – overlapping, inconsistent, or infeasible rules, incompleteness (e.g., missing combinations of conditions or missing actions for a given combination of conditions)

All models can also contain defects such as syntax errors, typos, duplicates, and inconsistent naming of model elements.

Detecting defects by using model-based testing (MBT). In this test approach, an MBT tool designs and generates test cases and other test design testware based on an MBT model and test selection criteria defined by the TA. MBT is effective in finding anomalies in the specification because it allows for comprehensive coverage and systematic exploration of the expected behavior of the test object according to the MBT model. MBT models, especially the graphical ones, foster communication with stakeholders, creating a common perception and understanding of the test basis. More details on MBT can be found in the Model-Based Tester Syllabus (ISTQB-MBT, v1.1).

5.2.2 Applying Review Techniques

A well-defined test basis serves as the cornerstone for ensuring the quality of work products and the success of projects. Reviewing the test basis helps identify and address defects early, preventing defects from escaping to subsequent phases.

The TA can employ various review techniques during the individual review to identify defects in the test basis. Selecting the most appropriate review technique can enhance the efficiency and effectiveness of the review. This selection should consider factors such as reviewing goals, project objectives, available resources, test basis type, associated risks, business domain, and company culture.

Below, this syllabus discusses five review techniques commonly used by the TA.

Ad hoc reviewing is carried out by reviewers informally, without a structured process. Reviewers are provided with little or no guidance on how the task should be performed. Ad hoc reviewing needs little preparation and is highly dependent on the reviewers' skills. During the review, reviewers read the test basis and document the anomalies as they encounter them. This technique, left unmanaged, can lead to a high volume of duplicate anomaly reports from multiple reviewers.

Checklist-based reviewing involves evaluating the test basis against a predefined checklist. Checklists remind reviewers to check specific points and can de-personalize the review. Checklists can be generic or specific to quality characteristics, test objectives, or test basis type. The TA tailors the checklist to the test basis type, risk level, or test condition. This ensures that the review will focus on the most relevant aspects of the test basis. Checklists should be regularly updated with previously missed defects. Keeping checklists current prevents overlooking newly identified anomalies. Checklists are not all-inclusive. Therefore, the TA is not limited to checking the listed items. This maximizes defect detection and allows the TA to capture anomalies that checklists may not explicitly cover.

Scenario-based reviewing involves simulating a process or activity to identify anomalies and refine the test basis. This review technique is most effective when the test basis has a scenario-based format, such as a use case or activity diagram. In such cases, reviewers can perform "dry runs" based on the expected usage of the work product. Scenarios provide valuable guidelines, but the reviewers are not constrained to documented scenarios and should also think beyond the scenarios to identify more anomalies.

Role-based reviewing involves assigning specific roles or responsibilities to reviewers. Typical roles are based on specific end-user types (e.g., experienced, inexperienced, senior, or child user) or specific roles in the organization (e.g., administrator or regular user). Each role can be described by a persona (i.e., the concrete but fictional character designed to represent the characteristics, needs, goals, and preferences of a particular group of users). By distributing responsibilities among reviewers based on their roles, role-based reviews allow individuals to focus on specific aspects of the test basis, ensuring comprehensive coverage and, at the same time, avoiding duplication of anomalies.

Perspective-based reading involves reviewing the test basis from various perspectives or viewpoints (e.g., designer, tester, marketer, administrator, and end user). This leads to more in-depth individual reviewing with less duplication of anomalies among reviewers. In addition, perspective-based reading requires the reviewers to attempt to use the test basis under review to generate the work product they would derive from it. For example, a tester would attempt to generate draft acceptance tests based on requirements specifications to see if all the necessary information is included.

5.3 Mitigating the Recurrence of Defects

A TA can proactively help minimize the recurrence of defects into the software. This syllabus discusses two approaches related to mitigating the recurrence of defects: analyzing test results to improve test analysis and test design and supporting root cause analysis with defect classification.

5.3.1 Analyzing Test Results to Improve Defect Detection

Test results allow the TA to identify failures, but they also provide feedback to the TA to help improve defect detection effectiveness. Some commonly used techniques to analyze test results are described below.

Predicted versus actual defect cluster analysis. A few components usually contain most defects (see ISTQB-CTFL, v4.0.1, Section 1.3). After testing, the TA can predict defect-prone areas and compare predicted versus actual defect clusters. In the case of discrepancies, more rigorous testing may be applied to areas where more defects were found than expected. When determining the clusters, measurable criteria should ensure clarity and consistency (e.g., defect density and defect severity). Among these criteria, the severity of the defects should play a significant role. Small clusters of critical or major defects are usually more important (i.e., need more rigorous testing) than larger clusters of minor or cosmetic defects.

Defect detection percentage (DDP) analysis. DDP is one of the most important test effectiveness measures for a test level. When calculating DDP, the number of escaped defects should be limited to those that the test level under consideration could have detected. Clear boundaries for defect counting, such as temporal limits (e.g., defects found within a defined time frame after release) and exclusion criteria (e.g., defects in third-party components or specific customer environments), should be established to ensure consistency. A low DDP for a given test level indicates a high percentage of escaped defects, meaning defect detection is ineffective. In such a case, the TA should analyze the reasons and propose

measures to improve it, making the test level more focused and rigorous. The DDP is best divided by severity levels, as the priority of reducing escaped defects usually depends on their severity.

Structural coverage analysis assesses the extent to which tests have exercised specific areas of the test object. Identifying low-coverage areas allows the TA to target test efforts in those areas. Increasing their coverage helps to discover new, previously escaped defects. Structural coverage such as statement coverage, branch coverage, or neuron coverage is usually measured with test tools. When selecting the additional areas to be covered, their risk levels should be considered.

Test gap analysis assesses the extent to which tests have exercised recent code changes. This allows the TA to focus additional test effort in areas that are especially error prone (i.e., new changes that have not been tested at all) instead of targeting all areas that have low coverage (e.g., code that has not changed in a long time and has been tested for previous releases).

Defect arrival pattern analysis. The number or density of defects found in successive phases of a project (e.g., iterations) can be compared with patterns that describe the theoretical distribution of these values over time. A classic example of a defect arrival pattern is the *Rayleigh* model (Elsayed, 2021). It has a single peak and is skewed to the right, showing that the expected number of defects found first increases in time and, after reaching its maximum value, drops down slowly towards zero. Based on the analysis of such a pattern, it is possible to infer the strength of existing test cases and the potential for their improvement. For example, suppose defect detection remains at a constant, low level when the pattern suggests it should be increasing. In that case, it may mean that the existing tests are too weak and unable to detect additional defects.

The analytical methods described above use various defect-related metrics, such as the number of defects or DDP. These metrics are calculated based on the test results (e.g., number of tests passed/failed), defect reports, and structural metrics (e.g., code coverage or defect density). Note, however, that these metrics are not always as easy to read from the test results as they may seem. For example, the number of failed tests is not necessarily the same as the number of defects detected by these tests. To calculate the actual number of defects detected, the results of the debugging process must be carefully analyzed because the relation between test results and defects can be many-to-many. Several tests may detect the same defect or one test may detect several defects. Moreover, the severity of the defects may differ from the criticality of the tests, as a critical test case may fail due to a cosmetic defect.

5.3.2 Supporting Root Cause Analysis with Defect Classification

Root cause analysis (RCA) is a technique for identifying and addressing the underlying or fundamental causes of a defect rather than only its symptoms. RCA supports a structured approach to quality improvement, and its primary objective is to prevent the recurrence of defects. The TA uses various techniques to identify root causes of defects and failures (e.g., defect taxonomies, the five whys technique, cause-effect diagrams, and Pareto analysis).

The classical RCA involves subject matter experts studying a defect in considerable detail after resolving it. However, there are usually many defects that need to be analyzed. Therefore, it would be very inefficient and time-consuming to have preventive action planning for each defect. One way to approach this problem is to classify defects and then perform the RCA for the defect types occurring.

Defect classification is based on the recognition that individual defects capture a great deal of information about the development process and the system under test. Defect classification allows the TA to extract information about various aspects of the development process from the defect and turn it into a process measurement. This, in turn, gives an insight into the types of errors made during development, which

is helpful for process improvement. Defect classification bridges the gap between quantitative defect statistics and qualitative RCA. To effectively support RCA, the defects should be uniformly classified throughout the entire SDLC, from early testing to production.

The TA should support their organization in standardizing software defect classification. This will improve communication and the exchange of information regarding defects among developers and organizations, facilitating the RCA.

Examples of defect classification methods are:

- orthogonal defect classification (ODC) (Chillarege, 1992), which classifies each defect into orthogonal (i.e., mutually exclusive) attributes, collected both when a defect is reported and when it is fixed
- IEEE 1044 (*IEEE 1044*, 2009), a standard classification for software anomalies providing a core set of attributes for the classification of failures and defects
- severity-based classification, which classifies defects based on their severity (e.g., critical, major, minor, trivial)
- defect taxonomy models, such as (Beizer, 1990) or (Catolino et al., 2019)

Defects can also be mapped to quality attributes using software quality models such as (*ISO/IEC 25010*, 2023) or the FURPS model (Grady et al., 1987).

More information on RCA can be found in (ISTQB-ITP, v1.0).

6 References

Standards

ETSI EG 202 237 v1.2.1: Internet Protocol Testing (IPT), 2010. Standard. European Telecommunications Standards Institute.

IEEE 1044: Standard Classification for Software Anomalies, 2009. Standard. Institute of Electrical and Electronics Engineers.

ISO 15745: Industrial automation systems and integration – Open systems application integration framework, 2003. Standard. International Organization for Standardization.

ISO 16100: Industrial automation systems and integration – Manufacturing software capability profiling for interoperability, 2009. Standard. International Organization for Standardization.

ISO 26262: Road vehicles – functional safety – Part 6: Product development at the software level, 2018. Standard. International Organization for Standardization.

ISO 9241-210: Ergonomics of human-system interaction, part 210: Human-centred design for interactive systems, 2019. Standard. International Organization for Standardization.

ISO/IEC 20246: Software and systems engineering – Work product reviews, 2017. Standard. International Organization for Standardization.

ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model, 2023. Standard. International Organization for Standardization.

ISO/IEC 25019: Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality-in-use model, 2023. Standard. International Organization for Standardization.

ISO/IEC/IEEE 29119-3: Software testing, Part 3: Test documentation, 2021. Standard. International Organization for Standardization.

ISO/IEC/IEEE 29119-4: Software testing, Part 4: Test techniques, 2021. Standard. International Organization for Standardization.

ISO/IEC/IEEE 29119-5: Software testing, Part 5: Keyword-Driven Testing, 2016. Standard. International Organization for Standardization.

OMG® BPMN: Business Process Model and Notation, version 2.0, 2013. Standard. Object Management Group, OMG®.

OMG® DMN: Decision Model and Notation, version 1.5, 2024. 2024-01. Standard. Object Management Group.

OMG® UML: Unified Modeling Language, version 2.5, 2017. Standard. Object Management Group.

RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification, 2011. Standard. Radio Technical Commission for Aeronautics, RTCA Inc.

ISTQB® Documents

ISTQB-ACT, ISTQB®, 2019. *Certified Tester Specialist Mobile Acceptance Testing: Syllabus*. Version 1.0.

ISTQB-AI, ISTQB®, 2021. *Certified Tester AI Testing: Syllabus*. Version 1.0.

ISTQB-CTFL, ISTQB®, 2024. *Certified Tester Foundation Level: Syllabus*. Version 4.0.1.

ISTQB-ITP, ISTQB®, 2011. *Certified Tester Expert Level Improving the Test Process: Syllabus*. Version 1.0.

ISTQB-MAT, ISTQB®, 2019. *Certified Tester Specialist Mobile Application Testing: Syllabus*. Version 1.0.

ISTQB-MBT, ISTQB®, 2024. *Certified Tester Model-Based Tester: Syllabus*. Version 1.1.

ISTQB-PT, ISTQB®, 2018. *Certified Tester Performance Testing: Syllabus*. Version 1.0.

ISTQB-TAE, ISTQB®, 2024. *Certified Tester Test Automation Engineering: Syllabus*. Version 2.0.

ISTQB-TM, ISTQB®, 2024. *Certified Tester Advanced Level Test Management: Syllabus*. Version 3.0.

ISTQB-TTA, ISTQB®, 2021. *Certified Tester Advanced Level Technical Test Analyst: Syllabus*. Version 4.0.

ISTQB-UT, ISTQB®, 2018. *Certified Tester Usability Testing: Syllabus*. Version 1.0.

Books

AMMANN, Paul; OFFUTT, Jeff, 2008. *Introduction to Software Testing*. Cambridge University Press.

BEIZER, Boris, 1990. *Software Testing Techniques (2nd Ed.)* International Thomson Computer Press.

BINDER, Robert V., 2000. *Testing Object-Oriented Systems*. Addison-Wesley.

ELSAYED, Elsayed A., 2021. *Reliability Engineering*. Wiley.

FORGÁCS, Istvan; KOVÁCS, Attila, 2019. *Practical Test Design: Selection of traditional and automated test design techniques*. BCS, The Chartered Institute for IT.

FORGÁCS, Istvan; KOVÁCS, Attila, 2024. *Modern Software Testing Techniques*. Apress.

GRADY, Robert; CASWELL, Deborah, 1987. *Software Metrics: Establishing a Company-wide Program*. Prentice Hall.

HENDRICKSON, Elisabeth, 2013. *Explore It!* Pragmatic Bookshelf.

JORGENSEN, Paul, 2014. *Software Testing. A Craftsman's Approach*. CRC Press.

KANER, Cem; FALK, Jack; NGUYEN, Hung Q., 1999. *Testing Computer Software*. Wiley.

KANER, Cem; PADMANABHAN, Sowmya; HOFFMAN, Douglas, 2013. *The Domain Testing Workbook*. Context Driven Press.

KOOMEN, Tim; AALST, Leo van der; BROEKMAN, Bart; VROON, Michiel, 2006. *TMap Next for result-driven testing*. UTN Publishers.

KUHN, D. Richard; YU, Lei; KACKER, Raghu N., 2013. *Introduction to Combinatorial Testing*. CRC Press.

Articles

ALYAHYA, Sultan, 2020. Crowdsourced Software Testing: A Systematic Literature Review. *Information and Software Technology*. Vol. 127, p. 106363. Available from DOI: 10.1016/j.infsof.2020.106363.

ANTONIOL, Giuliano; BRIAND, Lionel; DI PENTA, Massimiliano; LABICHE, Yvan, 2002. A case study using the round-trip strategy for state-based class testing. *Proceedings 13th International Symposium*

on Software Reliability Engineering, pp. 269–279. ISBN 0-7695-1763-3. Available from DOI: 10.1109/ISSRE.2002.1173268.

ARCURI, Andrea; IQBAL, Muhammad Zohaib; BRIAND, Lionel, 2012. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering*. Vol. 38, pp. 258–277. Available from DOI: 10.1109/TSE.2011.121.

BARR, Earl; HARMAN, Mark; MCMINN, Phil; SHAHBAZ, Muzammil; YOO, Shin, 2014. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*. Vol. 41, no. 5, pp. 507–525. Available from DOI: 10.1109/TSE.2014.2372785.

BOCHMANN, Gregor; PETRENKO, Alexandre, 1994. Protocol Testing: Review of Methods and Relevance for Software Testing. *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 109–124. Available from DOI: 10.1145/186258.187153.

CATOLINO, Gemma; PALOMBA, Fabio; ZAIDMAN, Andy; FERRUCCI, Filomena, 2019. Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types. *Journal of Systems and Software*. Vol. 152, pp. 165–181. Available from DOI: 10.1016/j.jss.2019.03.002.

CHILLAREGE, R. et al., 1992. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*. Vol. 18, pp. 943–956. Available from DOI: 10.1109/32.177364.

CHOW, Tsun, 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*. Vol. 4, pp. 178–187. Available from DOI: 10.1109/TSE.1978.231496.

COHEN, D.M.; DALAL, Siddhartha; KAJLA, A.; PATTON, G.C., 1994. The Automatic Efficient Test Generator (AETG) system. *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pp. 303–309. ISBN 0-8186-6665-X. Available from DOI: 10.1109/ISSRE.1994.341392.

ENGSTRÖM, Emelie; RUNESON, Per; SKOGLUND, Mats, 2010. A systematic review on regression test selection techniques. *Information and Software Technology*. Vol. 52, pp. 14–30. Available from DOI: 10.1016/j.infsof.2009.07.001.

GHAZI, Ahmad Nauman; GARIGAPATI, Ratna; PETERSEN, Kai, 2017. Checklists to Support Test Charter Design in Exploratory Testing. *Agile Processes in Software Engineering and Extreme Programming. XP 2017*. ISBN 978-3-319-57632-9. Available from DOI: 10.1007/978-3-319-57633-6_17.

HAREL, David, 1987. Statecharts: A Visual Formalism For Complex Systems. *Science of Computer Programming*. Vol. 8, pp. 231–274. Available from DOI: 10.1016/0167-6423(87)90035-9.

HUANG, Rubing; SUN, Weifeng; XU, Yinyin; CHEN, Haibo; TOWEY, Dave; XIA, Xin, 2019. A Survey on Adaptive Random Testing. *IEEE Transactions on Software Engineering*. Vol. 47, no. 10, pp. 2052–2083. Available from DOI: 10.1109/TSE.2019.2942921.

JENG, Bingchiang; WEYUKER, Elaine, 1994. A Simplified Domain-Testing Strategy. *ACM Transactions on Software Engineering and Methodology*. Vol. 3, pp. 254–270. Available from DOI: 10.1145/196092.193171.

JUERGENS, Elmar; PAGANO, Dennis; GOEB, Andreas, 2018. Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites. *Whitepaper, CQSE GmbH*.

KUHN, D.; WALLACE, Dolores; JR, A.M., 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*. Vol. 30, pp. 418–421. Available from DOI: 10.1109/TSE.2004.24.

LEICHT, Niklas; BLOHM, Ivo; LEIMEISTER, Jan Marco, 2017. Leveraging the Power of the Crowd for Software Testing. *IEEE Software*. Vol. 34, pp. 62–69. Available from DOI: 10.1109/MS.2017.37.

OFFUTT, A. Jefferson, 1992. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*. Vol. 1, pp. 5–20.

RECHTBERGER, Vaclav; BURES, Miroslav; AHMED, Bestoun, 2022. Overview of Test Coverage Criteria for Test Case Generation from Finite State Machines Modelled as Directed Graphs. *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Valencia, Spain*, pp. 207–214.

RWEMALIKA, Renaud; KINTIS, Marinos; PAPADAKIS, Mike; LE TRAON, Yves; LORRACH, Pierre, 2019. On the Evolution of Keyword-Driven Test Suites. *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 335–345.

SEGURA, Sergio; FRASER, Gordon; SANCHEZ, Ana; RUIZ-CORTÉS, Antonio, 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering*, pp. 46–53.

SEGURA, Sergio; TOWEY, Dave; ZHOU, Zhi Quan; CHEN, Tsong Yueh, 2020. Metamorphic Testing: Testing the Untestable. *IEEE Software*. Vol. 42, no. 9, pp. 805–824.

WU, Huayao; NIE, Changhai; PETKE, Justyna; JIA, Yue; HARMAN, Mark, 2020. An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing. *IEEE Transactions on Software Engineering*. Vol. 46, no. 3, pp. 302–320.

Web Pages

COMMISSION, European, 2016. *General Data Protection Regulation: Regulation (EU) 2016/679* [online]. [visited on 2024-09-15]. Available from: https://commission.europa.eu/law/law-topic/data-protection/legal-framework-eu-data-protection_en?.

COMPUTER SOCIETY, IEEE; JTC 1/SC7, ISO/IEC, 2024. *SE VOCAB: Software and Systems Engineering Vocabulary* [online]. [visited on 2024-09-15]. Available from: https://pascal.computer.org/.

CZERWONKA, Jacek, 2004. *Pairwise Testing: Combinatorial Test Case Generation* [online]. [visited on 2024-09-15]. Available from: www.pairwise.org.

HEALTH, U.S. Department of; SERVICES, Human, 2024. *Health Insurance Portability and Accountability Act: Standards for Privacy of Individually Identifiable Health Information (Privacy Rules)* [online]. [visited on 2024-09-15]. Available from: https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html.

IREB-GLOSSARY, IREB®, 2024. *The CPRE Glossary: Core terms of Requirements Engineering* [online]. [visited on 2024-09-15]. Available from: https://glossary.istqb.org.

ISTQB-GLOSSARY, ISTQB®, 2024. *Standard Glossary of Terms Used in Software Testing* [online]. [visited on 2024-09-15]. Available from: https://glossary.istqb.org.

Site of Software Test Design, 2020 [online]. [visited on 2024-09-15]. Available from: https://test-design.org/.

SUMI, 1991. *Software Usability Measurement Inventory: Standard evaluation questionnaire for assessing quality of use* [online]. [visited on 2024-09-15]. Available from: sumi.uxp.ie.

U.S. DEPARTMENT OF JUSTICE, Civil Rights Division, 2010. *Americans with Disabilities Act: Guidance & Resource Materials* [online]. [visited on 2024-09-15]. Available from: www.ada.gov/resources/.

UK GOVERNMENT, Equalities Office, 2010. *Equality Act 2010: guidance: Information and guidance on the Equality Act 2010, including age discrimination and public sector Equality Duty.* [online]. [visited on 2024-09-15]. Available from: www.gov.uk/guidance/equality-act-2010-guidance.

WAMMI, 1999. *Website Analysis and Measurement Inventory: Professional service for analyzing user experience* [online]. [visited on 2024-09-15]. Available from: www.wammi.com.

WCAG, 2023. *Web Content Accessibility Guidelines 2.2: W3C Recommendation* [online]. [visited on 2024-09-15]. Available from: www.w3.org/TR/WCAG22/.

The previous references point to information available on the Internet and elsewhere. Even though those references were checked at the time of publication of this syllabus, the ISTQB® cannot be held responsible if the references are unavailable anymore.

7 Appendix A – Learning Objectives/Cognitive Level of Knowledge

The specific learning objectives for this syllabus are shown at the beginning of each chapter. Each topic in the syllabus will be examined according to its learning objective.

The learning objectives begin with an action verb corresponding to its cognitive level of knowledge, as listed below.

Level 1: Remember (K1)

The candidate will remember, recognize, and recall a term or concept.

Action verbs: Recall, recognize.

Note: The advanced-level syllabi do not have specific K1-level learning objectives. However, the syllabus content in chapters 1 – 5 and the definitions of all terms listed as keywords just below the chapter headings shall be remembered (K1), even if not explicitly mentioned in the learning objectives.

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify, and give examples for the testing concept.

Action verbs: Classify, compare, differentiate, distinguish, explain, give examples, interpret, summarize

Examples	Notes
Differentiate between functional correctness, functional appropriateness, and functional completeness testing	Looks for differences between concepts.
Explain CRUD testing	
Give examples of test environment requirements	
Summarize the involvement of the test analyst in various software development lifecycles	

Level 3: Apply (K3)

The candidate can carry out a procedure when confronted with a familiar task or select the correct procedure and apply it to a given context.

Action verbs: Apply, implement, prepare, use

Examples	Notes
Apply domain testing	Should refer to a procedure / technique / process etc.
Prepare test charters for session-based testing	
Use keyword-driven testing to develop test scripts	Can be used in a LO that wants the candidate to be able to use a technique or procedure. Similar to 'apply'.

Level 4: Analyze (K4)

The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. A typical application is to analyze a document, software or project situation and propose appropriate actions to solve a problem or task.

Action verbs: Analyze, deconstruct, outline, prioritize, select.

Examples	Notes
Analyze the impact of changes to determine the scope of regression testing	Examinable only in combination with a measurable goal of the analysis. Should be of form “Analyze X to Y” (or similar).
Select appropriate test techniques to mitigate product risks for a given situation	Needed where the selection requires analysis.

Reference

(For the cognitive levels of learning objectives)

Anderson, L. W. and Krathwohl, D. R. (eds) (2001) A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom’s Taxonomy of Educational Objectives, Allyn & Bacon

8 Appendix B – Business Outcomes traceability matrix with Learning Objectives

This section lists the traceability between the Business Outcomes and the Learning Objectives of Advanced Level Test Analyst.

Business Outcomes: Advanced Level Test Analyst		TA-BO1	TA-BO2	TA-BO3	TA-BO4	TA-BO5	TA-BO6	TA-BO7	TA-BO8	TA-BO9
TA-BO1	Support and perform appropriate testing based on the software development lifecycle followed	6								
TA-BO2	Apply the principles of risk-based testing		3							
TA-BO3	Select and apply appropriate test techniques to support the achievement of test objectives			13						
TA-BO4	Provide documentation with appropriate levels of detail and quality				5					
TA-BO5	Determine the appropriate types of functional testing to be performed					1				
TA-BO6	Contribute to non-functional testing						3			
TA-BO7	Contribute to defect prevention							5		
TA-BO8	Improve the efficiency of the test process with the use of tools								4	
TA-BO9	Specify the requirements for test environments and test data									2

Business Outcomes: Advanced Level Test Analyst		TA-BO1	TA-BO2	TA-BO3	TA-BO4	TA-BO5	TA-BO6	TA-BO7	TA-BO8	TA-BO9
LO Number	Learning Objective (K-Level)									
1	The Tasks of the Test Analyst in the Test Process – 225 minutes									
1.1	Testing in the Software Development Lifecycle									
TA-1.1.1	Summarize the involvement of the test analyst in various software development lifecycles (K2)	X								
1.2	Involvement in Test Activities									
TA-1.2.1	Summarize the tasks performed by the test analyst as part of test analysis (K2)	X								
TA-1.2.2	Summarize the tasks performed by the test analyst as part of test design (K2)	X								
TA-1.2.3	Summarize the tasks performed by the test analyst as part of test implementation (K2)	X								
TA-1.2.4	Summarize the tasks performed by the test analyst as part of test execution (K2)	X								
1.3	Tasks Related to Work Products									
TA-1.3.1	Differentiate between high-level test cases and low-level test cases (K2)				X					
TA-1.3.2	Explain the quality criteria for test cases (K2)				X					
TA-1.3.3	Give examples of test environment requirements (K2)				X					X
TA-1.3.4	Explain the test oracle problem and potential solutions (K2)			X	X					
TA-1.3.5	Give examples of test data requirements (K2)				X					X
TA-1.3.6	Use keyword-driven testing to develop test scripts (K3)	X							X	
TA-1.3.7	Summarize the types of tools to manage the testware (K2)								X	
2	The Tasks of the Test Analyst in Risk-Based Testing – 90 minutes									

Business Outcomes: Advanced Level Test Analyst		TA-BO1	TA-BO2	TA-BO3	TA-BO4	TA-BO5	TA-BO6	TA-BO7	TA-BO8	TA-BO9
2.1	Risk Analysis									
TA-2.1.1	Summarize the test analyst's contribution to product risk analysis (K2)		X							
2.2	Risk Control									
TA-2.2.1	Analyze the impact of changes to determine the scope of regression testing (K4)		X						X	
3	Test Analysis and Test Design – 615 minutes									
3.1	Data-Based Test Techniques									
TA-3.1.1	Apply domain testing (K3)			X						
TA-3.1.2	Apply combinatorial testing (K3)			X						
TA-3.1.3	Summarize the benefits and limitations of random testing (K2)			X						
3.2	Behavior-Based Test Techniques									
TA-3.2.1	Explain CRUD testing (K2)			X						
TA-3.2.2	Apply state transition testing (K3)			X						
TA-3.2.3	Apply scenario-based testing (K3)			X						
3.3	Rule-Based Test Techniques									
TA-3.3.1	Apply decision table testing (K3)			X						
TA-3.3.2	Apply metamorphic testing (K3)			X						
3.4	Experience-Based Testing									
TA-3.4.1	Prepare test charters for session-based testing (K3)			X						
TA-3.4.2	Prepare checklists that support experience-based testing (K3)			X						

Business Outcomes: Advanced Level Test Analyst		TA-BO1	TA-BO2	TA-BO3	TA-BO4	TA-BO5	TA-BO6	TA-BO7	TA-BO8	TA-BO9
TA-3.4.3	Give examples of the benefits and limitations of crowd testing (K2)			X						
3.5	Applying the Most Appropriate Test Techniques									
TA-3.5.1	Select appropriate test techniques to mitigate product risks for a given situation (K4)		X	X						
TA-3.5.2	Explain the benefits and risks of automating the test design (K2)								X	
4	Testing Quality Characteristics – 60 minutes									
4.1	Functional Testing									
TA-4.1.1	Differentiate between functional correctness, functional appropriateness, and functional completeness testing (K2)					X				
4.2	Usability Testing									
TA-4.2.1	Explain how the test analyst contributes to usability testing (K2)						X			
4.3	Flexibility Testing									
TA-4.3.1	Explain how the test analyst contributes to adaptability and installability testing (K2)						X			
4.4	Compatibility Testing									
TA-4.4.1	Explain how the test analyst contributes to interoperability testing (K2)						X			
5	Software Defect Prevention – 225 minutes									
5.1	Defect Prevention Practices									
TA-5.1.1	Explain how the test analyst can contribute to defect prevention (K2)							X		
5.2	Supporting Phase Containment									
TA-5.2.1	Use a model of the test object to detect defects in a specification (K3)							X		

Business Outcomes: Advanced Level Test Analyst		TA-BO1	TA-BO2	TA-BO3	TA-BO4	TA-BO5	TA-BO6	TA-BO7	TA-BO8	TA-BO9
TA-5.2.2	Apply a review technique to a test basis to find defects (K3)							X		
5.3	Mitigating the Recurrence of Defects									
TA-5.3.1	Analyze test results to identify potential improvements to defect detection (K4)							X		
TA-5.3.2	Explain how defect classification supports root cause analysis (K2)							X		

9 Appendix C – Release Notes

ISTQB® Advanced Level Test Analyst Syllabus v4.0 is a major update. For this reason, there are no detailed release notes per chapter and section. However, a summary of principal changes is provided below. Additionally, in a separate Release Notes document, ISTQB® provides detailed traceability between the learning objectives in the v3.1.2 and v4.0 versions of the Advanced Level Test Analyst syllabus.

This major release has made the following changes:

Syllabus structure

All learning objectives have been edited to make them atomic and to create one-to-one traceability from learning objectives to content to avoid having content without a corresponding learning objective. Each learning objective is described in a section with the same number (e.g., TA-1.1.1 is discussed in Section 1.1.1). The goal was to make this version easier to read, understand, learn, and translate, focusing on increasing practical usefulness and the balance between knowledge and skills.

There are 36 learning objectives in v4.0 compared to 31 in v3.1.2. Several K4 learning objectives were reduced to K2 or K3. In the case of the learning objectives related to test techniques, the motivation was that the focus should be on applying the test techniques and not on analyzing the documentation for further test design. Some learning objectives were merged into a single learning objective. The table below shows detailed statistics on the number of learning objectives and the training time.

Syllabus version	#K2 LO (time)	#K3 LO (time)	#K4 LO (time)	#Total LO (time)
current (4.0)	22 (330 min)	11 (660 min)	3 (225 min)	36 (1215 min)
previous (3.1.2)	16 (240 min)	5 (300 min)	10 (750 min)	31 (1290 min)

Classification of test techniques

The structure of Chapter 3 reflects a more detailed classification of test techniques compared to v3.1.2. Black-box test techniques are now categorized as data-based, behavior-based, and rule-based, according to the type of the underlying test object model.

Expanding the scope of Chapter 5

The Old Chapter 5 (devoted solely to reviews) was expanded to discuss other essential forms of defect prevention practices used by the TA, such as using models to detect defects in specifications, analyzing test results to improve defect detection, and using defect classification to support root cause analysis.

Changes in learning objectives

- Chapter 1 was reorganized into a section about test activities and one about testware.
- The topic on high-level test cases and low-level test cases (old TA-1.4.2) was downgraded from K4 to K2 (TA-1.3.1).
- The K3 LO on risk-based testing (old TA-2.1.1) was split into two, K2 TA-2.1.1 related to risk analysis and K4 TA-2.2.1 related to risk control.
- Equivalence partitioning (old K4 TA-3.2.1) and boundary value analysis (old K4 TA-3.2.2) were replaced with the more general K3 TA-3.1.1 on domain testing.

- Pairwise testing (old K4 TA-3.2.6) and classification tree diagrams (old K2 TA-3.2.5) were merged into one, more general K3 TA-3.1.2 on combinatorial testing.
- State transition testing (old K4 TA-3.2.4) was replaced with K3 TA-3.2.2, which focuses on N-switch and round-trip coverage. Redundancies with CTFL have been removed.
- Use case testing (old K4 TA-3.2.7) was replaced with the more general K3 TA-3.2.3 on scenario-based testing.
- Decision table testing (old K4 TA-3.2.3) was downgraded to K3 (TA-3.3.1).
- Exploratory testing (old K3 TA-3.3.2) was replaced with two separate LOs: on preparing test charters (K3 TA-3.4.1) and on preparing checklists supporting experience-based testing (K3 TA-3.4.2). Redundancies with the current CTFL v4.0 have been removed.
- Four LOs related to comparing test techniques and applying the most appropriate technique (old: K4 TA-3.2.8, K2 TA-3.3.3, K2 TA-3.4.1, K2 TA-3.3.1) were combined into one K4 TA-3.5.1.
- Four LOs about functional testing (old: K2 TA-4.2.1, K2 TA-4.2.2, K2 TA-4.2.3 and K4 TA-4.2.7) were combined into one K2 TA-4.1.1. The topic was simplified because functional testing is already largely described in the context of test techniques in Chapter 3.
- Topics from Chapter 6 (Test Tools) were moved to Section 1.3, focusing on more practical issues. The old K3 TA-6.2.1 'For a given scenario determine the appropriate activities for a Test Analyst in a keyword-driven testing project' was slightly changed to K3 TA-1.3.6 'Use keyword-driven testing to develop test scripts'. The old K2 TA-6.3.1 'Explain the usage and types of test tools applied in test design, test data preparation and test execution' was slightly changed to K2 TA-1.3.7 'Summarize the types of tools applied in managing the testware'.
- Two similar LOs on reviews (old K3 TA-5.2.1 and K3 TA-5.2.2) were combined into one K3 TA-5.2.2.

New topics

- Quality criteria for test cases (K2 TA-1.3.2)
- Test environment requirements (K2 TA-1.3.3)
- Determining test oracles (K2 TA-1.3.4)
- Test data requirements (K2 TA-1.3.5)
- Random testing (K2 TA-3.1.3)
- CRUD testing (K2 TA-3.2.1)
- Metamorphic testing (K3 TA-3.3.2)
- Crowd testing (K2 TA-3.4.3)
- Benefits and risks of automating the test design (K2 TA-3.5.2)
- Contributions of the test analyst to defect prevention (K2 TA-5.1.1)
- Using models to detect defects in specifications (K3 TA-5.2.1)
- Analyzing test results to improve defect detection (K4 TA-5.3.1)
- Supporting root cause analysis with defect classification (K2 TA-5.3.2)

Standard updates

Changes in the latest versions of the international standards for software quality (*ISO/IEC 25010*, 2023) and test techniques (*ISO/IEC/IEEE 29119-4*, 2021) have led to the adaptation of the syllabus's corresponding content.

10 Appendix D – List of Abbreviations

Abbreviation	Meaning
AI	artificial intelligence
BPMN	Business Process Model and Notation
BVA	boundary value analysis
CRUD	create, read, update, and delete
CSV	comma-separated value
DDP	defect detection percentage
DRE	defect removal efficiency
EP	equivalence partitioning
GDPR	General Data Protection Regulation
JSON	JavaScript Object Notation
MBT	model-based testing
MR	metamorphic relation
MT	metamorphic testing
ODC	orthogonal defect classification
PCE	phase containment effectiveness
RCA	root cause analysis
SDLC	software development lifecycle
TA	test analyst
TTA	technical test analyst
UML	Unified Modeling Language
UX	user experience
XML	Extensible Markup Language

11 Appendix E – Domain-Specific Terms

Term	Definition
CRUD matrix	A matrix that indicates the action types of the functions on the entities within a system.
data semantics	The meaning and interpretation of data.
five whys technique	An iterative interrogative technique used to determine the root cause of a defect or problem by repeating the question 'why?' five times, each time directing the current 'why' to the answer of the previous 'why'.
Pareto analysis	An approach to identify problem areas or tasks that will have the biggest payoff.
user journey map	User experience visualization documentation that shows the steps that a user takes in a process to accomplish a goal.
user research	A discipline of learning about users' needs and thought processes by studying how they perform tasks, observing how they interact with a component or system, or by data analysis and interpretation.

12 Appendix F – Software Quality Model

The table below shows the quality characteristics of the ISO/IEC 25010 product quality model (*ISO/IEC 25010*, 2023). It indicates which characteristics/sub-characteristics are addressed within this syllabus (TA) and which are covered in other ISTQB® syllabi (Technical Test Analyst (TTA), Performance Testing (PT), Usability Testing (UT), Automotive Software Tester (AuT), and Security Tester (SEC)). If several syllabi cover a quality characteristic, the one covering it in the most detail is listed first. The table also compares the current ISO/IEC 25010 model with the 2011 version used in the previous version of this syllabus.

ISO/IEC 25010:2023 (current)	**ISO/IEC 25010:2011 (previous)**	**Notes**	**ISTQB® syllabi**
Functional suitability	**Functional suitability**		**TA**
Functional completeness	Functional completeness		
Functional correctness	Functional correctness		
Functional appropriateness	Functional appropriateness		
Performance efficiency	**Performance efficiency**		PT, TTA
Time behavior	Time behavior		
Resource utilization	Resource utilization		
Capacity	Capacity		
Compatibility	**Compatibility**		**TA**, TTA
Co-existence	Co-existence		TTA
Interoperability	Interoperability		**TA**
Interaction capability	**Usability**	**Renamed**	UT, **TA**
Appropriateness recognizability	Appropriateness recognizability		
Learnability	Learnability		
Operability	Operability		
User error protection	User error protection		
User engagement	User interface aesthetics	Renamed	
Inclusivity	Accessibility	Split and renamed	
User assistance			
Self-descriptiveness		New	

ISO/IEC 25010:2023 (current)	ISO/IEC 25010:2011 (previous)	Notes	ISTQB® syllabi
Reliability	**Reliability**		TTA
Faultlessness	Maturity	Renamed	
Availability	Availability		
Fault tolerance	Fault tolerance		
Recoverability	Recoverability		
Security	**Security**		SEC, TTA
Confidentiality	Confidentiality		
Integrity	Integrity		
Non-repudiation	Non-repudiation		
Accountability	Accountability		
Authenticity	Authenticity		
Resistance		New	
Maintainability	**Maintainability**		TTA
Modularity	Modularity		
Reusability	Reusability		
Analysability	Analysability		
Modifiability	Modifiability		
Testability	Testability		
Flexibility	**Portability**	**Renamed**	TTA, **TA**, PT
Adaptability	Adaptability		**TA**, TTA
Scalability		New	PT
Installability	Installability		**TA**, TTA
Replaceability	Replaceability		TTA
Safety		**New**	AuT
Operational constraint		New	
Risk identification		New	
Fail safe		New	
Hazard warning		New	
Safe integration		New	

13 Appendix G – Trademarks

ISTQB® is a registered trademark of the International Software Testing Qualifications Board.

UML® is a registered trademark of the Object Management Group (OMG).

BPMN™ is a trademark of the Object Management Group (OMG).

14 Index