Syllabus

SDET

A4Q Software Development Engineer in Test

Foundation Level

Version 2022 EN

GTB German Testing Board

Software. Testing. Excellence.

A4Q Alliance for Qualification

English edition.
Published by
Alliance for Qualification & German Testing Board e.V.

Composition of learning objectives from the syllabi of the International Software Testing Qualifications Board (ISTQB®) original title: Certified Tester Foundation Level Syllabus Version 2018 V3.1 as well as Certified Tester Advanced Level Syllabus Technical Test Analyst Version V4.0 extended by hands-on Levels of competency.

Copyright Notice

Authors

For authors of, and contributors to the original edition CTFL 2018 V3.1 see [ISTQB_FL_SYL].

For authors of, and contributors to the original edition CTAL-TTA V4.0 see [ISTQB_ATTA_SYL].

Authors of the German & English hands-on levels of competency for A4Q-SDET Foundation Level as well as the selection of learning objectives from the above syllabi: Members of the GTB Working Group SDET Foundation Level: Jürgen Beniermann, Daniel Fröhlich, Thorsten Geiselhart, Matthias Hamburg, Armin Metzger, Andreas Reuys, Erhardt Wunderlich.

Revision History of this document

Version	Date	Remarks
A4Q-TF4D 2021 V1.0	19.03.2021	Testing Foundations for Developers
A4Q-SDET Foundation Level 2022	28.04.2022	Software Development Engineer in Test

Revision History of the original documents

For a revision history of the original edition CTFL 2018 V3.1 see [ISTQB_FL_SYL].

For a revision history of the original edition CTAL-TTA V4.0 see [ISTQB_ATTA_SYL].

Table of Contents

Acknowledgements

This document A4Q Software Development Engineer in Test (A4Q-SDET Foundation Level) was formally published on 03/19/2021 by A4Q and GTB. On 04/15/2022, the document was renamed to A4Q Software Development Engineer in Test - Foundation Level (A4Q SDET Foundation Level).

The A4Q and the GTB would like to thank the members of the GTB working group SDET: Jürgen Beniermann, Daniel Fröhlich, Thorsten Geiselhart, Matthias Hamburg, Armin Metzger, Andreas Reuys, Erhardt Wunderlich for their commitment and participation in the realization of this syllabus.

The team thanks the reviewers for their review findings on the composition of that syllabus:

Arne Becher, Florian Fieber, Dietrich Leimsner, Elke Mai, Carsten Weise.

The team thanks the back office for the technical assembly of the syllabi from the original documents.

The team, A4Q and the GTB would like to thank Andreas Spillner, the "father to that thought" of this syllabus with " Testing Foundations for Developers". He had the idea to establish a syllabus tailored to the demands of developers by a targeted selection of learning objectives from the ISTQB Certified Tester Foundation Level and additions. With the appropriate training, developers can gain or deepen the knowledge of the testing foundations required for their work without the fear of being turned into a tester. The acquired knowledge bridges the gap between developers and testers and significantly improves communication between them. His book "Lean Testing für C++-Programmierer" (Lean Testing for C++-Programmers), published by dpunkt.verlag together with Ulrich Breymann, was written with the idea of illustrating the advantages of systematic testing for developers. It describes all the fundamental test techniques for developer testing and explains them in concrete examples, up to the creation of the test cases (www.leantesting.de).

0 Introduction to this Syllabus

0.1 Purpose of this Syllabus

This syllabus defines the testing foundations for developers, based on the Foundation Level and the Advanced Level Technical Test Analyst of the Software Testing Qualifications Program of the International Software Testing Qualifications Board (hereinafter referred to as ISTQB®).

The GTB provides this syllabus as follows:

1. To training providers, to produce courseware and determine appropriate teaching methods.
2. To certification candidates, to prepare for the certification exam (either as part of a training course or independently).
3. To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and professional articles.

The GTB may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission from the GTB.

0.2 Software Development Engineer in Test (A4Q-SDET Foundation Level)

The goal of this syllabus is to teach testing skills to developers, not to transform developers into testers.

Software Development Engineer in Test (SDET Foundation Level) is based on the Certified Tester training program. They are intended to address the necessary testing skills in all lifecycles, regardless of whether there is a specific "tester" role.

This syllabus is aimed especially at software developers, but also at people in the role of product owner, project manager, quality manager, software development manager, system analyst (business analyst), IT manager or management consultant who want to acquire the basic testing skills for developers.

0.3 Business Outcomes

This section lists the business outcomes that can be expected of candidates with A4Q-SDET Foundation Level certification.

An A4Q-SDET Foundation Level is able to ...

SDET-BO1	…promote efficient and effective communication by using a common vocabulary for software testing.[1]
SDET-BO2	…understand fundamental concepts of software testing.[2]
SDET-BO3	…use established black-box techniques for designing tests.[3]
SDET-BO4	…use established white-box techniques for designing tests.[4]
SDET-BO5	…implement tests according to given test designs.
SDET-BO6	…execute implemented tests and report results.
SDET-BO7	…improve the quality characteristics of code and architecture by making use of different analysis techniques.[5]

0.4 Examinable Learning Objectives and Cognitive Levels of Knowledge

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

In general, all contents of this syllabus are examinable at a K1 level, except for the Introduction and Appendices. That is, the candidate may be asked to recognize, remember, or recall a keyword or concept mentioned in any of the chapters. The knowledge levels of the specific learning objectives are shown at the beginning of each chapter, and classified as follows:

- K1: remember
- K2: understand
- K3: apply

The definitions of all terms listed as keywords just below chapter headings shall be remembered (K1), even if not explicitly mentioned in the learning objectives.

The syllabus contains learning objectives that are explicitly marked as *non-exam relevant* [6]. These learning objectives include further knowledge required to complete the testing foundations for developers. The core elements of these learning objectives are provided in an overview at K1 level.

[1] Taken over from FL-BO1 [ISTQB_FL_OVW]
[2] Taken over from FL-BO2 [ISTQB_FL_OVW]
[3] Adapted from FL-BO5 [ISTQB_FL_OVW]
[4] Adapted from TTA3 [ISTQB_AL_OVW]
[5] Adapted from TTA3 [ISTQB_AL_OVW]
[6] *Note: Below, all parts of the syllabus that are non-exam relevant, but set a meaningful context for various reasons, are shown in blue italics.*

0.5 Hands-on Levels of Competency

In Software Development Engineer in Test, the concept of Hands-On Objectives which focus on practical skills and competencies is applied.

Competencies can be achieved by performing hands-on exercises, such as those shown in the following non-exhaustive list:

- Exercises for K3 level learning objectives performed using paper and pen or word processing software.
- Setting up and using test environments.
- Testing applications on virtual and physical devices.
- Using tools to test or assist in testing related tasks.

The following levels apply to hands-on objectives:

- H0: This can include a live demo of an exercise or recorded video. Since this is not performed by the trainee, it is not strictly an exercise.
- H1: Guided exercise. The trainees follow a sequence of steps performed by the trainer.
- H2: Exercise with hints. The trainee is given an exercise with relevant hints to enable the exercise to be solved within the given timeframe.
- H3: Unguided exercises without hints.

0.6 The Examination

The exam Software Development Engineer in Test will be based on this syllabus. Answers to exam questions may require knowledge based on more than one section of this syllabus. The used key terms according to the glossary and all sections of the syllabus, unless marked as *non-exam relevant* [7] are examinable, except for the introduction and appendices. Standards, books, and other ISTQB® syllabi are included as references, but their content is not examinable, beyond what is summarized in this syllabus itself.

The exam is multiple choice. There are 40 questions. To pass the exam, at least 65% of the questions (i.e., 26 questions) must be answered correctly. Hands on objectives and exercises will not be examined.

Exams may be taken as part of an accredited training course or taken independently (e.g., at an exam center or in a public exam). Completion of an accredited training course is not a prerequisite for the exam.

0.7 Gender-neutral wording

For reasons of simpler readability, gender-neutral differentiation is omitted. In the interest of equal treatment, all role designations apply to all genders.

[7] *Note: Below, all parts of the syllabus that are non-exam relevant, but set a meaningful context for various reasons, are shown in blue italics.*

0.8 Accreditation

The GTB or A4Q may accredit training providers whose course material follows this syllabus. The accreditation guidelines can be obtained from Alliance for Qualification or from the German Testing Board. An accredited course is recognized as conforming to this syllabus and may include an A4Q-SDET Foundation Level examination.

0.9 How this Syllabus is Organized

There are four chapters with examinable content. The main heading for each chapter specifies the time allocated for the chapter. There is no time reference for the subchapters. For accredited training courses, the syllabus requires a minimum of 14,25 hours of instruction, distributed across the chapters as follows:

- Chapter 1: Fundamentals of Testing - 110 minutes
- Chapter 2: Testing Throughout the Software Development Lifecycle - 70 minutes
- Chapter 3: Static Testing - 225 minutes
- Chapter 4: Test Techniques - 450 minutes

1 Fundamentals of Testing – 110 minutes

Keywords

coverage, debugging, defect, error, failure, quality, quality assurance, root cause, test analysis, test basis, test case, test condition, test data, test design, test execution, test implementation, test object, test objective, test oracle, test procedure, test process, test suite, testing, traceability, validation, verification

Learning Objectives for Fundamentals of Testing

1.1 What is Testing?

FL-1.1.1	(K1) Identify typical objectives of testing
FL-1.1.2	(K2) Differentiate testing from debugging

1.2 Why is Testing necessary?

FL-1.2.1	(K2) Give examples of why testing is necessary
FL-1.2.2	(K2) Describe the relationship between testing and quality assurance and give examples of how testing contributes to higher quality
FL-1.2.3	(K2) Distinguish between error, defect, and failure
FL-1.2.4	(K2) Distinguish between the root cause of a defect and its effects

1.3 Seven Testing Principles

FL-1.3.1	(K2) Explain the seven testing principles

1.4 Test Process (non-exam relevant)

FL-1.4.1	*(K2) Explain the impact of context on the test process (non-exam relevant)*
FL-1.4.2	*(K2) Describe the test activities and respective tasks within the test process (non-exam relevant)*
FL-1.4.3	*(K2) Differentiate the work products that support the test process (non-exam relevant)*
FL-1.4.4	*(K2) Explain the value of maintaining traceability between the test basis and test work products (non-exam relevant)*

1.1 What is Testing?

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death. Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.

A common misperception of testing is that it only consists of running tests, i.e., executing the software and checking the results. As described in section 1.4, software testing is a process which includes many different activities; test execution (including checking of results) is only one of these activities. The test process also includes activities such as test planning, analyzing, designing, and implementing tests, reporting test progress and results, and evaluating the quality of a test object.

Some testing does involve the execution of the component or system being tested; such testing is called dynamic testing. Other testing does not involve the execution of the component or system being tested; such testing is called static testing. So, testing also includes reviewing work products such as requirements, user stories, and source code.

Another common misperception of testing is that it focuses entirely on verification of requirements, user stories, or other specifications. While testing does involve checking whether the system meets specified requirements, it also involves validation, which is checking whether the system will meet user and other stakeholder needs in its operational environment(s).

Test activities are organized and carried out differently in different lifecycles (see section 2.1).

1.1.1 Typical Objectives of Testing

For any given project, the objectives of testing may include:

- To prevent defects by evaluate work products such as requirements, user stories, design, and code
- To verify whether all specified requirements have been fulfilled
- To check whether the test object is complete and validate if it works as the users and other stakeholders expect
- To build confidence in the level of quality of the test object
- To find defects and failures by reducing the level of risk of inadequate software quality
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object
- To comply with contractual, legal, or regulatory requirements or standards, and/or to verify the test object's compliance with such requirements or standards

The objectives of testing can vary, depending upon the context of the component or system being tested, the test level, and the software development lifecycle model. These differences may include, for example:

- During component testing, one objective may be to find as many failures as possible so that the underlying defects are identified and fixed early. Another objective may be to increase code coverage of the component tests.

- During acceptance testing, one objective may be to confirm that the system works as expected and satisfies requirements. Another objective of this testing may be to give information to stakeholders about the risk of releasing the system at a given time.

1.1.2 Testing and Debugging

Testing and debugging are different. Executing tests can show failures that are caused by defects in the software. Debugging is the development activity that finds, analyzes, and fixes such defects. Subsequent confirmation testing checks whether the fixes resolved the defects. In some cases, testers are responsible for the initial test and the final confirmation test, while developers do the debugging, associated component and component integration testing (continues integration). However, in Agile development and in some other software development lifecycles, testers may be involved in debugging and component testing.

ISO standard (ISO/IEC/IEEE 29119-1) has further information about software testing concepts.

1.2 Why is Testing necessary?

Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation. When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems. In addition, software testing may also be required to meet contractual or legal requirements or industry-specific standards.

1.2.1 Testing's Contributions to Success

Throughout the history of computing, it is quite common for software and systems to be delivered into operation and, due to the presence of defects, to subsequently cause failures or otherwise not meet the stakeholders' needs. However, using appropriate test techniques can reduce the frequency of such problematic deliveries, when those techniques are applied with the appropriate level of test expertise, in the appropriate test levels, and at the appropriate points in the software development lifecycle. Examples include:

- Having testers involved in requirements reviews or user story refinement could detect defects in these work products. The identification and removal of requirements defects reduces the risk of incorrect or untestable features being developed.
- Having testers work closely with system designers while the system is being designed can increase each party's understanding of the design and how to test it. This increased understanding can reduce the risk of fundamental design defects and enable tests to be identified at an early stage.
- Having testers work closely with developers while the code is under development can increase each party's understanding of the code and how to test it. This increased understanding can reduce the risk of defects within the code and the tests.
- Having testers verify and validate the software prior to release can detect failures that might otherwise have been missed, and support the process of removing the defects that caused the failures (i.e., debugging). This increases the likelihood that the software meets stakeholder needs and satisfies requirements.

In addition to these examples, the achievement of defined test objectives (see section 1.1.1) contributes to overall software development and maintenance success.

1.2.2 Quality Assurance and Testing

While people often use the phrase *quality assurance* (or just *QA*) to refer to testing, quality assurance and testing are not the same, but they are related. A larger concept, quality management, ties them together. Quality management includes all activities that direct and control an organization with regard to quality. Among other activities, quality management includes both quality assurance and quality control. Quality assurance is typically focused on adherence to proper processes, in order to provide confidence that the appropriate levels of quality will be achieved. When processes are carried out properly, the work products created by those processes are generally of higher quality, which contributes to defect prevention. In addition, the use of root cause analysis to detect and remove the causes of defects, along with the proper application of the findings of retrospective meetings to improve processes, are important for effective quality assurance.

Quality control involves various activities, including test activities, that support the achievement of appropriate levels of quality. Test activities are part of the overall software development or maintenance process. Since quality assurance is concerned with the proper execution of the entire process, quality assurance supports proper testing. As described in sections 1.1.1 and 1.2.1, testing contributes to the achievement of quality in a variety of ways.

1.2.3 Errors, Defects, and Failures

A person can make an error (mistake), which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product. An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product. For example, a requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code.

If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances. For example, some defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.

Errors may occur for many reasons, such as:

- Time pressure
- Human fallibility
- Inexperienced or insufficiently skilled project participants
- Miscommunication between project participants, including miscommunication about requirements and design
- Complexity of the code, design, architecture, the underlying problem to be solved, and/or the technologies used
- Misunderstandings about intra-system and inter-system interfaces, especially when such intra-system and inter-system interactions are large in number
- New, unfamiliar technologies

In addition to failures caused due to defects in the code, failures can also be caused by environmental conditions. For example, radiation, electromagnetic fields, and pollution can cause defects in firmware or influence the execution of software by changing hardware conditions.

Not all unexpected test results are failures. False positives may occur due to errors in the way tests were executed, or due to defects in the test data, the test environment, or other testware, or for other reasons. The inverse situation can also occur, where similar errors or defects lead to false negatives.

False negatives are tests that do not detect defects that they should have detected; false positives are reported as defects, but aren’t actually defects.

1.2.4 Defects, Root Causes and Effects

The root causes of defects are the earliest actions or conditions that contributed to creating the defects. Defects can be analyzed to identify their root causes, so as to reduce the occurrence of similar defects in the future. By focusing on the most significant root causes, root cause analysis can lead to process improvements that prevent a significant number of future defects from being introduced.

For example, suppose incorrect interest payments, due to a single line of incorrect code, result in customer complaints. The defective code was written for a user story which was ambiguous, due to the product owner’s misunderstanding of how to calculate interest. If a large percentage of defects exist in interest calculations, and these defects have their root cause in similar misunderstandings, the product owners could be trained in the topic of interest calculations to reduce such defects in the future.

In this example, the customer complaints are effects. The incorrect interest payments are failures. The improper calculation in the code is a defect, and it resulted from the original defect, the ambiguity in the user story. The root cause of the original defect was a lack of knowledge on the part of the product owner, which resulted in the product owner making an error while writing the user story. The process of root cause analysis is discussed in ISTQB-CTEL-TM and ISTQB-CTEL-ITP.

1.3 Seven Testing Principles

A number of testing principles have been suggested over the past 50 years and offer general guidelines common for all testing.

1. Testing shows the presence of defects, not their absence

Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, testing is not a proof of correctness.

2. Exhaustive testing is impossible

Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Rather than attempting to test exhaustively, risk analysis, test techniques, and priorities should be used to focus test efforts.

3. Early testing saves time and money

To find defects early, both static and dynamic test activities should be started as early as possible in the software development lifecycle. Early testing is sometimes referred to as *shift left*. Testing early in the software development lifecycle helps reduce or eliminate costly changes (see section 3.1).

4. Defects cluster together

A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort (as mentioned in principle 2).

5. Beware of the pesticide paradox

If the same tests are repeated over and over again, eventually these tests no longer find any new defects. To detect new defects, existing tests and test data may need changing, and new tests may need to be written. (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while.) In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.

6. Testing is context dependent

Testing is done differently in different contexts. For example, safety-critical industrial control software is tested differently from an e-commerce mobile app. As another example, testing in an Agile project is done differently than testing in a sequential software development lifecycle project (see section 2.1).

7. Absence-of-errors is a fallacy

Some organizations expect that testers can run all possible tests and find all possible defects, but principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy (i.e., a mistaken belief) to expect that just finding and fixing a large number of defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users' needs and expectations, or that is inferior compared to other competing systems.

See Myers 2011, Kaner 2002, Weinberg 2008, and Beizer 1990 for examples of these and other testing principles.

1.4 Test Process (non-exam relevant)

There is no one universal software test process, but there are common sets of test activities without which testing will be less likely to achieve its established objectives. These sets of test activities are a test process. The proper, specific software test process in any given situation depends on many factors. Which test activities are involved in this test process, how these activities are implemented, and when these activities occur may be discussed in an organization's test strategy.

1.4.1 Test Process in Context (non-exam relevant)

Contextual factors that influence the test process for an organization, include, but are not limited to:

- *Software development lifecycle model and project methodologies being used*
- *Test levels and test types being considered*
- *Product and project risks*
- *Business domain*
- *Operational constraints, including but not limited to:*
    - *Budgets and resources*
    - *Timescales*
    - *Complexity*
    - *Contractual and regulatory requirements*
- *Organizational policies and practices*
- *Required internal and external standards*

The following sections describe general aspects of organizational test processes in terms of the following:

- *Test activities and tasks*
- *Test work products*
- *Traceability between the test basis and test work products*

It is very useful if the test basis (for any level or type of testing that is being considered) has measurable coverage criteria defined. The coverage criteria can act effectively as key performance indicators (KPIs) to drive the activities that demonstrate achievement of software test objectives (see section 1.1.1).

For example, for a mobile application, the test basis may include a list of requirements and a list of supported mobile devices. Each requirement is an element of the test basis. Each supported device is also an element of the test basis. The coverage criteria may require at least one test case for each element of the test basis. Once executed, the results of these tests tell stakeholders whether specified requirements are fulfilled and whether failures were observed on supported devices.

ISO standard (ISO/IEC/IEEE 29119-2) has further information about test processes.

1.4.2 Test Activities and Tasks (non-exam relevant)

A test process consists of the following main groups of activities:

- *Test planning*
- *Test monitoring and control*
- *Test analysis*
- *Test design*
- *Test implementation*
- *Test execution*
- *Test completion*

Each main group of activities is composed of constituent activities, which will be described in the subsections below. Each constituent activity consists of multiple individual tasks, which would vary from one project or release to another.

Further, although many of these main activity groups may appear logically sequential, they are often implemented iteratively. For example, Agile development involves small iterations of software design, build, and test that happen on a continuous basis, supported by on-going planning. So test activities are also happening on an iterative, continuous basis within this software development approach. Even in sequential software development, the stepped logical sequence of main groups of activities will involve overlap, combination, concurrency, or omission, so tailoring these main groups of activities within the context of the system and the project is usually required.

Test planning

Test planning involves activities that define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context (e.g., specifying suitable test techniques and tasks, and formulating a test schedule for meeting a deadline). Test plans may be revisited based on feedback from monitoring and control activities. Test planning is further explained in section 5.2.

Test monitoring and control

Test monitoring involves the on-going comparison of actual progress against planned progress using any test monitoring metrics defined in the test plan. Test control involves taking actions necessary to meet the objectives of the test plan (which may be updated over time). Test monitoring and control are *supported by the evaluation of exit criteria, which are referred to as the definition of done in some software development lifecycle models (see ISTQB-CTFL-AT). For example, the evaluation of exit criteria for test execution as part of a given test level may include:*

- *Checking test results and logs against specified coverage criteria*
- *Assessing the level of component or system quality based on test results and logs*
- *Determining if more tests are needed (e.g., if tests originally intended to achieve a certain level of product risk coverage failed to do so, requiring additional tests to be written and executed)*

Test progress against the plan is communicated to stakeholders in test progress reports, including deviations from the plan and information to support any decision to stop testing.

Test monitoring and control are further explained in section 5.3.

Test analysis

During test analysis, the test basis is analyzed to identify testable features and define associated test conditions. In other words, test analysis determines "what to test" in terms of measurable coverage criteria.

Test analysis includes the following major activities:

- *Analyzing the test basis appropriate to the test level being considered, for example:*
  - *Requirement specifications, such as business requirements, functional requirements, system requirements, user stories, epics, use cases, or similar work products that specify desired functional and non-functional component or system behavior*
  - *Design and implementation information, such as system or software architecture diagrams or documents, design specifications, call flow graphs, modelling diagrams (e.g., UML or entity-relationship diagrams), interface specifications, or similar work products that specify component or system structure*
  - *The implementation of the component or system itself, including code, database metadata and queries, and interfaces*
  - *Risk analysis reports, which may consider functional, non-functional, and structural aspects of the component or system*
- *Evaluating the test basis and test items to identify defects of various types, such as:*
  - *Ambiguities*
  - *Omissions*
  - *Inconsistencies*
  - *Inaccuracies*
  - *Contradictions*
  - *Superfluous statements*
- *Identifying features and sets of features to be tested*
- *Defining and prioritizing test conditions for each feature based on analysis of the test basis, and considering functional, non-functional, and structural characteristics, other business and technical factors, and levels of risks*
- *Capturing bi-directional traceability between each element of the test basis and the associated test conditions (see sections 1.4.3 and 1.4.4)*

The application of black-box, white-box, and experience-based test techniques can be useful in the process of test analysis (see chapter 4) to reduce the likelihood of omitting important test conditions and to define more precise and accurate test conditions.

In some cases, test analysis produces test conditions which are to be used as test objectives in test charters. Test charters are typical work products in some types of experience-based testing (see section 4.4.2). When these test objectives are traceable to the test basis, coverage achieved during such experience-based testing can be measured.

The identification of defects during test analysis is an important potential benefit, especially where no other review process is being used and/or the test process is closely connected with the review process. Such test analysis activities not only verify whether the requirements are consistent, properly expressed, and complete, but also validate whether the requirements properly capture customer, user, and other stakeholder needs. For example, techniques such as behavior driven development (BDD) and acceptance test driven development (ATDD), which involve generating test conditions and test cases from user stories and acceptance criteria prior to coding. These techniques also verify, validate, and detect defects in the user stories and acceptance criteria (see ISTQB-CTFL-AT).

Test design

During test design, the test conditions are elaborated into high-level test cases, sets of high-level test cases, and other testware. So, test analysis answers the question "what to test?" while test design answers the question "how to test?"

Test design includes the following major activities:

- *Designing and prioritizing test cases and sets of test cases*
- *Identifying necessary test data to support test conditions and test cases*
- *Designing the test environment and identifying any required infrastructure and tools*
- *Capturing bi-directional traceability between the test basis, test conditions, and test cases (see section 1.4.4)*

The elaboration of test conditions into test cases and sets of test cases during test design often involves using test techniques (see chapter 4).

As with test analysis, test design may also result in the identification of similar types of defects in the test basis. Also, as with test analysis, the identification of defects during test design is an important potential benefit.

Test implementation

During test implementation, the testware necessary for test execution is created and/or completed, including sequencing the test cases into test procedures. So, test design answers the question "how to test?" while test implementation answers the question "do we now have everything in place to run the tests?"

Test implementation includes the following major activities:

- *Developing and prioritizing test procedures, and, potentially, creating automated test scripts*
- *Creating test suites from the test procedures and (if any) automated test scripts*
- *Arranging the test suites within a test execution schedule in a way that results in efficient test execution (see section 5.2.4)*

- *Building the test environment (including, potentially, test harnesses, service virtualization, simulators, and other infrastructure items) and verifying that everything needed has been set up correctly*
- *Preparing test data and ensuring it is properly loaded in the test environment*
- *Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test suites (see section 1.4.4)*

Test design and test implementation tasks are often combined.

In exploratory testing and other types of experience-based testing, test design and implementation may occur, and may be documented, as part of test execution. Exploratory testing may be based on test charters (produced as part of test analysis), and exploratory tests are executed immediately as they are designed and implemented (see section 4.4.2).

Test execution

During test execution, test suites are run in accordance with the test execution schedule.

Test execution includes the following major activities:

- *Recording the IDs and versions of the test item(s) or test object, test tool(s), and testware*
- *Executing tests either manually or by using test execution tools*
- *Comparing actual results with expected results*
- *Analyzing anomalies to establish their likely causes (e.g., failures may occur due to defects in the code, but false positives also may occur (see section 1.2.3)*
- *Reporting defects based on the failures observed see [ISTQB_FL_SYL – section 5.6 Defect Management]*
- *Logging the outcome of test execution (e.g., pass, fail, blocked)*
- *Repeating test activities either as a result of action taken for an anomaly, or as part of the planned testing (e.g., execution of a corrected test, confirmation testing, and/or regression testing)*
- *Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test results.*

Test completion

Test completion activities collect data from completed test activities to consolidate experience, testware, and any other relevant information. Test completion activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), an Agile project iteration is finished, a test level is completed, or a maintenance release has been completed.

Test completion includes the following major activities:

- *Checking whether all defect reports are closed, entering change requests or product backlog items for any defects that remain unresolved at the end of test execution*
- *Creating a test summary report to be communicated to stakeholders*
- *Finalizing and archiving the test environment, the test data, the test infrastructure, and other testware for later reuse*
- *Handing over the testware to the maintenance teams, other project teams, and/or other stakeholders who could benefit from its use*

- *Analyzing lessons learned from the completed test activities to determine changes needed for future iterations, releases, and projects*
- *Using the information gathered to improve test process maturity*

1.4.3 Test Work Products (non-exam relevant)

Test work products are created as part of the test process. Just as there is significant variation in the way that organizations implement the test process, there is also significant variation in the types of work products created during that process, in the ways those work products are organized and managed, and in the names used for those work products. This syllabus adheres to the test process outlined above, and the work products described in this syllabus and in the ISTQB® Glossary. ISO standard (ISO/IEC/IEEE 29119-3) may also serve as a guideline for test work products.

Many of the test work products described in this section can be captured and managed using test management tools and defect management tools [ISTQB_FL_SYL – section 6 Tool Support for Testing].

Test planning work products

Test planning work products typically include one or more test plans. The test plan includes information about the test basis, to which the other test work products will be related via traceability information (see below and section 1.4.4), as well as exit criteria (or definition of done) which will be used during test monitoring and control. Test plans are described in [ISTQB_FL_SYL – section 5.2 Test Planing and Estimation].

Test monitoring and control work products

Test monitoring and control work products typically include various types of test reports, including test progress reports produced on an ongoing and/or a regular basis, and test summary reports produced at various completion milestones. All test reports should provide audience-relevant details about the test progress as of the date of the report, including summarizing the test execution results once those become available.

Test monitoring and control work products should also address project management concerns, such as task completion, resource allocation and usage, and effort.

Test monitoring and control, and the work products created during these activities, are further explained in [ISTQB_FL_SYL – section 5.3 Test Monitoring and Control].

Test analysis work products

Test analysis work products include defined and prioritized test conditions, each of which is ideally bi-directionally traceable to the specific element(s) of the test basis it covers. For exploratory testing, test analysis may involve the creation of test charters. Test analysis may also result in the discovery and reporting of defects in the test basis.

Test design work products

Test design results in test cases and sets of test cases to exercise the test conditions defined in test analysis. It is often a good practice to design high-level test cases, without concrete values for input data and expected results. Such high-level test cases are reusable across multiple test cycles with different concrete data, while still adequately documenting the scope of the test case. Ideally, each test case is bi-directionally traceable to the test condition(s) it covers.

Test design also results in:

- *the design and/or identification of the necessary test data*
- *the design of the test environment*
- *the identification of infrastructure and tools*

Though the extent to which these results are documented varies significantly.

Test implementation work products

Test implementation work products include:

- *Test procedures and the sequencing of those test procedures*
- *Test suites*
- *A test execution schedule*

Ideally, once test implementation is complete, achievement of coverage criteria established in the test plan can be demonstrated via bi-directional traceability between test procedures and specific elements of the test basis, through the test cases and test conditions.

In some cases, test implementation involves creating work products using or used by tools, such as service virtualization and automated test scripts.

Test implementation also may result in the creation and verification of test data and the test environment. The completeness of the documentation of the data and/or environment verification results may vary significantly.

The test data serve to assign concrete values to the inputs and expected results of test cases. Such concrete values, together with explicit directions about the use of the concrete values, turn high-level test cases into executable low-level test cases. The same high-level test case may use different test data when executed on different releases of the test object. The concrete expected results which are associated with concrete test data are identified by using a test oracle.

In exploratory testing, some test design and implementation work products may be created during test execution, though the extent to which exploratory tests (and their traceability to specific elements of the test basis) are documented may vary significantly.

Test conditions defined in test analysis may be further refined in test implementation.

Test execution work products

Test execution work products include:

- *Documentation of the status of individual test cases or test procedures (e.g., ready to run, pass, fail, blocked, deliberately skipped, etc.)*
- *Defect reports see [ISTQB_FL_SYL – section 5.6 Defect Management]*
- *Documentation about which test item(s), test object(s), test tools, and testware were involved in the testing*

Ideally, once test execution is complete, the status of each element of the test basis can be determined and reported via bi-directional traceability with the associated the test procedure(s). For example, we can say which requirements have passed all planned tests, which requirements have failed tests and/or have defects associated with them, and which requirements have planned tests still waiting to be run. This enables verification that the coverage criteria have been met, and enables the reporting of test results in terms that are understandable to stakeholders.

Test completion work products

Test completion work products include test summary reports, action items for improvement of subsequent projects or iterations, change requests or product backlog items, and finalized testware.

1.4.4 Traceability between the Test Basis and Test Work Products (non-exam relevant)

As mentioned in section 1.4.3, test work products and the names of those work products vary significantly. Regardless of these variations, in order to implement effective test monitoring and control, it is important to establish and maintain traceability throughout the test process between each element of the test basis and the various test work products associated with that element, as described above. In addition to the evaluation of test coverage, good traceability supports:

- *Analyzing the impact of changes*
- *Making testing auditable*
- *Meeting IT governance criteria*
- *Improving the understandability of test progress reports and test summary reports to include the status of elements of the test basis (e.g., requirements that passed their tests, requirements that failed their tests, and requirements that have pending tests)*
- *Relating the technical aspects of testing to stakeholders in terms that they can understand*
- *Providing information to assess product quality, process capability, and project progress against business goals*

Some test management tools provide test work product models that match part or all of the test work products outlined in this section. Some organizations build their own management systems to organize the work products and provide the information traceability they require.

2 Testing Throughout the Software Development Lifecycle – 70 minutes

Keywords

change-related testing, component integration testing, component testing, confirmation testing, functional testing, impact analysis, integration testing, maintenance testing, non-functional testing, regression testing, test basis, test case, test environment, test level, test object, test objective, test type, white-box testing

Learning Objectives for Testing Throughout the Software Development Lifecycle

2.1 Test Levels (non-exam relevant)

FL-2.2.1	*(K2) Compare the different test levels from the perspective of objectives, test basis, test objects, typical defects and failures, and approaches and responsibilities*

2.2 Test Types

FL-2.3.1	(K2) Compare functional, non-functional, and white-box testing
FL-2.3.3	(K2) Compare the purposes of confirmation testing and regression testing

2.3 Maintenance Testing

FL-2.4.1	(K2) Summarize triggers for maintenance testing
FL-2.4.2	(K2) Describe the role of impact analysis in maintenance testing

2.1 Test Levels (non-exam relevant)

Test levels are groups of test activities that are organized and managed together. Each test level is an instance of the test process, consisting of the activities described in section 1.4, performed in relation to software at a given level of development, from individual units or components to complete systems or, where applicable, systems of systems. Test levels are related to other activities within the software development lifecycle. The test levels used in this syllabus are:

- *Component testing*
- *Integration testing*
- *System testing*
- *Acceptance testing*

Test levels are characterized by the following attributes:

- *Specific objectives*
- *Test basis, referenced to derive test cases*
- *Test object (i.e., what is being tested)*
- *Typical defects and failures*
- *Specific approaches and responsibilities*

For every test level, a suitable test environment is required. In acceptance testing, for example, a production-like test environment is ideal, while in component testing the developers typically use their own development environment.

2.1.1 Component Testing (non-exam relevant)

Objectives of component testing

Component testing (also known as unit or module testing) focuses on components that are separately testable. Objectives of component testing include:

- *Reducing risk*
- *Verifying whether the functional and non-functional behaviors of the component are as designed and specified*
- *Building confidence in the component's quality*
- *Finding defects in the component*
- *Preventing defects from escaping to higher test levels*

In some cases, especially in incremental and iterative development models (e.g., Agile) where code changes are ongoing, automated component regression tests play a key role in building confidence that changes have not broken existing components.

Component testing is often done in isolation from the rest of the system, depending on the software development lifecycle model and the system, which may require mock objects, service virtualization, harnesses, stubs, and drivers. Component testing may cover functionality (e.g., correctness of calculations), non-functional characteristics (e.g., searching for memory leaks), and structural properties (e.g., decision testing).

Test basis

Examples of work products that can be used as a test basis for component testing include:

- *Detailed design*
- *Code*
- *Data model*
- *Component specifications*

Test objects

Typical test objects for component testing include:

- *Components, units or modules*
- *Code and data structures*
- *Classes*
- *Database modules*

Typical defects and failures

Examples of typical defects and failures for component testing include:

- *Incorrect functionality (e.g., not as described in design specifications)*
- *Data flow problems*
- *Incorrect code and logic*

Defects are typically fixed as soon as they are found, often with no formal defect management. However, when developers do report defects, this provides important information for root cause analysis and process improvement.

Specific approaches and responsibilities

Component testing is usually performed by the developer who wrote the code, but it at least requires access to the code being tested. Developers may alternate component development with finding and fixing defects. Developers will often write and execute tests after having written the code for a component. However, in Agile development especially, writing automated component test cases may precede writing application code.

For example, consider test driven development (TDD). Test driven development is highly iterative and is based on cycles of developing automated test cases, then building and integrating small pieces of code, then executing the component tests, correcting any issues, and re-factoring the code. This process continues until the component has been completely built and all component tests are passing. Test driven development is an example of a test-first approach. While test driven development originated in eXtreme Programming (XP), it has spread to other forms of Agile and also to sequential lifecycles (see ISTQB-CTFL-AT).

2.1.2 Integration Testing (non-exam relevant)

Objectives of integration testing

Integration testing focuses on interactions between components or systems. Objectives of integration testing include:

- *Reducing risk*

- *Verifying whether the functional and non-functional behaviors of the interfaces are as designed and specified*
- *Building confidence in the quality of the interfaces*
- *Finding defects (which may be in the interfaces themselves or within the components or systems)*
- *Preventing defects from escaping to higher test levels*

As with component testing, in some cases automated integration regression tests provide confidence that changes have not broken existing interfaces, components, or systems.

There are two different levels of integration testing described in this syllabus, which may be carried out on test objects of varying size as follows:

- *Component integration testing focuses on the interactions and interfaces between integrated components. Component integration testing is performed after component testing, and is generally automated. In iterative and incremental development, component integration tests are usually part of the continuous integration process.*
- *System integration testing focuses on the interactions and interfaces between systems, packages, and microservices. System integration testing can also cover interactions with, and interfaces provided by, external organizations (e.g., web services). In this case, the developing organization does not control the external interfaces, which can create various challenges for testing (e.g., ensuring that test-blocking defects in the external organization's code are resolved, arranging for test environments, etc.). System integration testing may be done after system testing or in parallel with ongoing system test activities (in both sequential development and iterative and incremental development).*

Test basis

Examples of work products that can be used as a test basis for integration testing include:

- *Software and system design*
- *Sequence diagrams*
- *Interface and communication protocol specifications*
- *Use cases*
- *Architecture at component or system level*
- *Workflows*
- *External interface definitions*

Test objects

Typical test objects for integration testing include:

- *Subsystems*
- *Databases*
- *Infrastructure*
- *Interfaces*
- *APIs*
- *Microservices*

Typical defects and failures

Examples of typical defects and failures for component integration testing include:

- *Incorrect data, missing data, or incorrect data encoding*
- *Incorrect sequencing or timing of interface calls*
- *Interface mismatch*
- *Failures in communication between components*
- *Unhandled or improperly handled communication failures between components*
- *Incorrect assumptions about the meaning, units, or boundaries of the data being passed between components*

Examples of typical defects and failures for system integration testing include:

- *Inconsistent message structures between systems*
- *Incorrect data, missing data, or incorrect data encoding*
- *Interface mismatch*
- *Failures in communication between systems*
- *Unhandled or improperly handled communication failures between systems*
- *Incorrect assumptions about the meaning, units, or boundaries of the data being passed between systems*
- *Failure to comply with mandatory security regulations*

Specific approaches and responsibilities

Component integration tests and system integration tests should concentrate on the integration itself. For example, if integrating module A with module B, tests should focus on the communication between the modules, not the functionality of the individual modules, as that should have been covered during component testing. If integrating system X with system Y, tests should focus on the communication between the systems, not the functionality of the individual systems, as that should have been covered during system testing. Functional, non-functional, and structural test types are applicable.

Component integration testing is often the responsibility of developers. System integration testing is generally the responsibility of testers. Ideally, testers performing system integration testing should understand the system architecture, and should have influenced integration planning.

If integration tests and the integration strategy are planned before components or systems are built, those components or systems can be built in the order required for most efficient testing. Systematic integration strategies may be based on the system architecture (e.g., top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components. In order to simplify defect isolation and detect defects early, integration should normally be incremental (i.e., a small number of additional components or systems at a time) rather than "big bang" (i.e., integrating all components or systems in one single step). A risk analysis of the most complex interfaces can help to focus the integration testing.

The greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting. This is one reason that continuous integration, where software is integrated on a component-by-component basis (i.e., functional integration), has become common practice. Such continuous integration often includes automated regression testing, ideally at multiple test levels.

2.1.3 System Testing (non-exam relevant)

Objectives of system testing

System testing focuses on the behavior and capabilities of a whole system or product, often considering the end-to-end tasks the system can perform and the non-functional behaviors it exhibits while performing those tasks. Objectives of system testing include:

- *Reducing risk*
- *Verifying whether the functional and non-functional behaviors of the system are as designed and specified*
- *Validating that the system is complete and will work as expected*
- *Building confidence in the quality of the system as a whole*
- *Finding defects*
- *Preventing defects from escaping to higher test levels or production*

For certain systems, verifying data quality may also be an objective. As with component testing and integration testing, in some cases automated system regression tests provide confidence that changes have not broken existing features or end-to-end capabilities. System testing often produces information that is used by stakeholders to make release decisions. System testing may also satisfy legal or regulatory requirements or standards.

The test environment should ideally correspond to the final target or production environment.

Test basis

Examples of work products that can be used as a test basis for system testing include:

- *System and software requirement specifications (functional and non-functional)*
- *Risk analysis reports*
- *Use cases*
- *Epics and user stories*
- *Models of system behavior*
- *State diagrams*
- *System and user manuals*

Test objects

Typical test objects for system testing include:

- *Applications*
- *Hardware/software systems*
- *Operating systems*
- *System under test (SUT)*
- *System configuration and configuration data*

Typical defects and failures

Examples of typical defects and failures for system testing include:

- *Incorrect calculations*
- *Incorrect or unexpected system functional or non-functional behavior*
- *Incorrect control and/or data flows within the system*
- *Failure to properly and completely carry out end-to-end functional tasks*
- *Failure of the system to work properly in the system environment(s)*
- *Failure of the system to work as described in system and user manuals*

Specific approaches and responsibilities

System testing should focus on the overall, end-to-end behavior of the system as a whole, both functional and non-functional. System testing should use the most appropriate techniques (see chapter 4) for the aspect(s) of the system to be tested. For example, a decision table may be created to verify whether functional behavior is as described in business rules.

System testing is typically carried out by independent testers who rely heavily on specifications. Defects in specifications (e.g., missing user stories, incorrectly stated business requirements, etc.) can lead to a lack of understanding of, or disagreements about, expected system behavior. Such situations can cause false positives and false negatives, which waste time and reduce defect detection effectiveness, respectively. Early involvement of testers in user story refinement or static testing activities, such as reviews, helps to reduce the incidence of such situations.

2.1.4 Acceptance Testing (non-exam relevant)

Objectives of acceptance testing

Acceptance testing, like system testing, typically focuses on the behavior and capabilities of a whole system or product. Objectives of acceptance testing include:

- *Establishing confidence in the quality of the system as a whole*
- *Validating that the system is complete and will work as expected*
- *Verifying that functional and non-functional behaviors of the system are as specified*

Acceptance testing may produce information to assess the system's readiness for deployment and use by the customer (end-user). Defects may be found during acceptance testing, but finding defects is often not an objective, and finding a significant number of defects during acceptance testing may in some cases be considered a major project risk. Acceptance testing may also satisfy legal or regulatory requirements or standards.

Common forms of acceptance testing include the following:

- *User acceptance testing*
- *Operational acceptance testing*
- *Contractual and regulatory acceptance testing*
- *Alpha and beta testing.*

Each is described in the following four subsections.

User acceptance testing (UAT)

User acceptance testing of the system is typically focused on validating the fitness for use of the system by intended users in a real or simulated operational environment. The main objective is building *confidence that the users can use the system to meet their needs, fulfill requirements, and perform business processes with minimum difficulty, cost, and risk.*

Operational acceptance testing (OAT)

The acceptance testing of the system by operations or systems administration staff is usually performed in a (simulated) production environment. The tests focus on operational aspects, and may include:

- *Testing of backup and restore*
- *Installing, uninstalling and upgrading*
- *Disaster recovery*
- *User management*
- *Maintenance tasks*
- *Data load and migration tasks*
- *Checks for security vulnerabilities*
- *Performance testing*

The main objective of operational acceptance testing is building confidence that the operators or system administrators can keep the system working properly for the users in the operational environment, even under exceptional or difficult conditions.

Contractual and regulatory acceptance testing

Contractual acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance criteria should be defined when the parties agree to the contract. Contractual acceptance testing is often performed by users or by independent testers.

Regulatory acceptance testing is performed against any regulations that must be adhered to, such as government, legal, or safety regulations. Regulatory acceptance testing is often performed by users or by independent testers, sometimes with the results being witnessed or audited by regulatory agencies.

The main objective of contractual and regulatory acceptance testing is building confidence that contractual or regulatory compliance has been achieved.

Alpha and beta testing

Alpha and beta testing are typically used by developers of commercial off-the-shelf (COTS) software who want to get feedback from potential or existing users, customers, and/or operators before the software product is put on the market. Alpha testing is performed at the developing organization's site, not by the development team, but by potential or existing customers, and/or operators or an independent test team. Beta testing is performed by potential or existing customers, and/or operators at their own locations. Beta testing may come after alpha testing, or may occur without any preceding alpha testing having occurred.

One objective of alpha and beta testing is building confidence among potential or existing customers, and/or operators that they can use the system under normal, everyday conditions, and in the operational environment(s) to achieve their objectives with minimum difficulty, cost, and risk. Another objective may be the detection of defects related to the conditions and environment(s) in which the system will be used, especially when those conditions and environment(s) are difficult to replicate by the development team.

Test basis

Examples of work products that can be used as a test basis for any form of acceptance testing include:

- *Business processes*
- *User or business requirements*
- *Regulations, legal contracts and standards*
- *Use cases and/or user stories*
- *System requirements*
- *System or user documentation*
- *Installation procedures*
- *Risk analysis reports*

In addition, as a test basis for deriving test cases for operational acceptance testing, one or more of the following work products can be used:

- *Backup and restore procedures*
- *Disaster recovery procedures*
- *Non-functional requirements*
- *Operations documentation*
- *Deployment and installation instructions*
- *Performance targets*
- *Database packages*
- *Security standards or regulations*

Typical test objects

Typical test objects for any form of acceptance testing include:

- *System under test*
- *System configuration and configuration data*
- *Business processes for a fully integrated system*
- *Recovery systems and hot sites (for business continuity and disaster recovery testing)*
- *Operational and maintenance processes*
- *Forms*
- *Reports*
- *Existing and converted production data*

Typical defects and failures

Examples of typical defects for any form of acceptance testing include:

- *System workflows do not meet business or user requirements*
- *Business rules are not implemented correctly*
- *System does not satisfy contractual or regulatory requirements*
- *Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform*

Specific approaches and responsibilities

Acceptance testing is often the responsibility of the customers, business users, product owners, or operators of a system, and other stakeholders may be involved as well.

Acceptance testing is often thought of as the last test level in a sequential development lifecycle, but it may also occur at other times, for example:

- *Acceptance testing of a COTS software product may occur when it is installed or integrated*
- *Acceptance testing of a new functional enhancement may occur before system testing*

In iterative development, project teams can employ various forms of acceptance testing during and at the end of each iteration, such as those focused on verifying a new feature against its acceptance criteria and those focused on validating that a new feature satisfies the users' needs. In addition, alpha tests and beta tests may occur, either at the end of each iteration, after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contractual acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.

2.2 Test Types

A test type is a group of test activities aimed at testing specific characteristics of a software system, or a part of a system, based on specific test objectives. Such objectives may include:

- Evaluating functional quality characteristics, such as completeness, correctness, and appropriateness
- Evaluating non-functional quality characteristics, such as reliability, performance efficiency, security, compatibility, and usability
- Evaluating whether the structure or architecture of the component or system is correct, complete, and as specified
- Evaluating the effects of changes, such as confirming that defects have been fixed (confirmation testing) and looking for unintended changes in behavior resulting from software or environment changes (regression testing)

2.2.1 Functional Testing

Functional testing of a system involves tests that evaluate functions that the system should perform. Functional requirements may be described in work products such as business requirements specifications, epics, user stories, use cases, or functional specifications, or they may be undocumented. The functions are "what" the system should do.

Functional tests should be performed at all test levels (e.g., tests for components may be based on a component specification), though the focus is different at each level (see section 2.2).

Functional testing considers the behavior of the software, so black-box techniques may be used to derive test conditions and test cases for the functionality of the component or system (see section 4.2).

The thoroughness of functional testing can be measured through functional coverage. Functional coverage is the extent to which some functionality has been exercised by tests, and is expressed as a percentage of the type(s) of element being covered. For example, using traceability between tests and functional requirements, the percentage of these requirements which are addressed by testing can be calculated, potentially identifying coverage gaps.

Functional test design and execution may involve special skills or knowledge, such as knowledge of the particular business problem the software solves (e.g., geological modelling software for the oil and gas industries).

2.2.2 Non-functional Testing

Non-functional testing of a system evaluates characteristics of systems and software such as usability, performance efficiency or security. Refer to ISO standard (ISO/IEC 25010) for a classification of software product quality characteristics. Non-functional testing is the testing of “how well” the system behaves.

Contrary to common misperceptions, non-functional testing can and often should be performed at all test levels, and done as early as possible. The late discovery of non-functional defects can be extremely dangerous to the success of a project.

Black-box techniques (see section 4.2) may be used to derive test conditions and test cases for non-functional testing. For example, boundary value analysis can be used to define the stress conditions for performance tests.

The thoroughness of non-functional testing can be measured through non-functional coverage. Non-functional coverage is the extent to which some type of non-functional element has been exercised by tests, and is expressed as a percentage of the type(s) of element being covered. For example, using traceability between tests and supported devices for a mobile application, the percentage of devices which are addressed by compatibility testing can be calculated, potentially identifying coverage gaps.

Non-functional test design and execution may involve special skills or knowledge, such as knowledge of the inherent weaknesses of a design or technology (e.g., security vulnerabilities associated with particular programming languages) or the particular user base (e.g., the personas of users of healthcare facility management systems).

Refer to ISTQB-CTAL-TA, ISTQB-CTAL-TTA, ISTQB-CTAL-SEC, and other ISTQB® specialist modules for more details regarding the testing of non-functional quality characteristics.

2.2.3 White-box Testing

White-box testing derives tests based on the system’s internal structure or implementation. Internal structure may include code, architecture, work flows, and/or data flows within the system (see section 4.3).

The thoroughness of white-box testing can be measured through structural coverage. Structural coverage is the extent to which some type of structural element has been exercised by tests, and is expressed as a percentage of the type of element being covered.

At the component testing level, code coverage is based on the percentage of component code that has been tested, and may be measured in terms of different aspects of code (coverage items) such as the percentage of executable statements tested in the component, or the percentage of decision outcomes tested. These types of coverage are collectively called code coverage. At the component integration testing level, white-box testing may be based on the architecture of the system, such as interfaces between components, and structural coverage may be measured in terms of the percentage of interfaces exercised by tests.

White-box test design and execution may involve special skills or knowledge, such as the way the code is built, how data is stored (e.g., to evaluate possible database queries), and how to use coverage tools and to correctly interpret their results.

2.2.4 Change-related Testing

When changes are made to a system, either to correct a defect or because of new or changing functionality, testing should be done to confirm that the changes have corrected the defect or implemented the functionality correctly, and have not caused any unforeseen adverse consequences.

- Confirmation testing: After a defect is fixed, the software may be tested with all test cases that failed due to the defect, which should be re-executed on the new software version. The software may also be tested with new tests to cover changes needed to fix the defect. At the very least, the steps to reproduce the failure(s) caused by the defect must be re-executed on the new software version. The purpose of a confirmation test is to confirm whether the original defect has been successfully fixed.
- Regression testing: It is possible that a change made in one part of the code, whether a fix or another type of change, may accidentally affect the behavior of other parts of the code, whether within the same component, in other components of the same system, or even in other systems. Changes may include changes to the environment, such as a new version of an operating system or database management system. Such unintended side-effects are called regressions. Regression testing involves running tests to detect such unintended side-effects.

Confirmation testing and regression testing are performed at all test levels.

Especially in iterative and incremental development lifecycles (e.g., Agile), new features, changes to existing features, and code refactoring result in frequent changes to the code, which also requires change-related testing. Due to the evolving nature of the system, confirmation and regression testing are very important. This is particularly relevant for Internet of Things systems where individual objects (e.g., devices) are frequently updated or replaced.

Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation. Automation of these tests should start early in the project [see ISTQB_FL_SYL – Chapter 6 Tool Support for Testing].

2.3 Maintenance Testing

Once deployed to production environments, software and systems need to be maintained. Changes of various sorts are almost inevitable in delivered software and systems, either to fix defects discovered in operational use, to add new functionality, or to delete or alter already-delivered functionality. Maintenance is also needed to preserve or improve non-functional quality characteristics of the component or system over its lifetime, especially performance efficiency, compatibility, reliability, security, , and portability.

When any changes are made as part of maintenance, maintenance testing should be performed, both to evaluate the success with which the changes were made and to check for possible side-effects (e.g., regressions) in parts of the system that remain unchanged (which is usually most of the system). Maintenance can involve planned releases and unplanned releases (hot fixes).

A maintenance release may require maintenance testing at multiple test levels, using various test types, based on its scope. The scope of maintenance testing depends on:

- The degree of risk of the change, for example, the degree to which the changed area of software communicates with other components or systems
- The size of the existing system
- The size of the change

2.3.1 Triggers for Maintenance

There are several reasons why software maintenance, and thus maintenance testing, takes place, both for planned and unplanned changes.

We can classify the triggers for maintenance as follows:

- Modification, such as planned enhancements (e.g., release-based), corrective and emergency changes, changes of the operational environment (such as planned operating system or database upgrades), upgrades of COTS software, and patches for defects and vulnerabilities
- Migration, such as from one platform to another, which can require operational tests of the new environment as well as of the changed software, or tests of data conversion when data from another application will be migrated into the system being maintained
    - Retirement, such as when an application reaches the end of its life. When an application or system is retired, this can require testing of data migration or archiving if long data-retention periods are required.
    - Testing restore/retrieve procedures after archiving for long retention periods may also be needed.
    - regression testing may be needed to ensure that any functionality that remains in service still works.

For Internet of Things systems, maintenance testing may be triggered by the introduction of completely new or modified things, such as hardware devices and software services, into the overall system. The maintenance testing for such systems places particular emphasis on integration testing at different levels (e.g., network level, application level) and on security aspects, in particular those relating to personal data.

2.3.2 Impact Analysis for Maintenance

Impact analysis evaluates the changes that were made for a maintenance release to identify the intended consequences as well as expected and possible side effects of a change, and to identify the areas in the system that will be affected by the change. Impact analysis can also help to identify the impact of a change on existing tests. The side effects and affected areas in the system need to be tested for regressions, possibly after updating any existing tests affected by the change.

Impact analysis may be done before a change is made, to help decide if the change should be made, based on the potential consequences in other areas of the system.

Impact analysis can be difficult if:

- Specifications (e.g., business requirements, user stories, architecture) are out of date or missing
- Test cases are not documented or are out of date
- Bi-directional traceability between tests and the test basis has not been maintained
- Tool support is weak or non-existent
- The people involved do not have domain and/or system knowledge
- Insufficient attention has been paid to the software's maintainability during development

3 Static Testing – 225 minutes

Keywords

ad hoc review, checklist-based review, control flow analysis, cyclomatic complexity, data flow analysis, definition-use pair, dynamic testing, perspective-based reading, review, role-based reviewing, scenario-based review, static analysis, static testing

Learning Objectives for Static Testing

3.1 Static Testing Basics (non-exam relevant)

FL-3.1.1	*(K1) Recognize types of software work product that can be examined by the different static testing techniques*
FL-3.1.2	*(K2) Use examples to describe the value of static testing*
FL-3.1.3	*(K2) Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software lifecycle*

3.2 Applying review techniques

FL-3.2.4	(K3) Apply a review technique to a work product to find defects
HO-3.2.4	(H2) Review a piece of code with a given checklist. Document the findings.

Guideline for hands-on objective:

Provide a typical code review checklist with various anomalies.

Provide a piece of code that contains several of the anomalies from the checklist. Provide a findings list template to participants to document findings. Review the findings list with the participants.

3.3 Static Analysis

TTA-3.2.1	(K3) Use control flow analysis to detect if code has any control flow anomalies and to measure cyclomatic complexity
HO-3.2.1	(H1) For a piece of code, use a static analysis tool to find typical control flow anomalies. Understand the report of the tool and how the anomalies affect the product quality characteristics.

Guideline for hands-on objective:

Provide one or more pieces of code that are syntactically correct and contain different types of control flow anomalies mentioned in the Syllabus.

Guide the participants in running the static analysis tool and displaying the reports on control flow anomalies. Participants shall discuss the defects found and indicate the quality characteristic affected (functional correctness, maintainability, security etc.).

TTA-3.2.2	(K3) Use data flow analysis to detect if code has any data flow anomalies
HO-3.2.2	(H1) For a piece of code, understand the report of a static analysis tool concerning data flow anomalies and how those anomalies affect functional correctness and maintainability.

Guideline for hands-on objective:

Provide a piece of code that is syntactically correct and contains the main types of data flow anomalies for some variables.

Run static code analysis, explain the data flow anomalies reported to the participants, and discuss their impact on functional correctness or on maintainability.

TTA-3.2.3	(K3) Propose ways to improve the maintainability of code by applying static analysis
HO-3.2.3	(H2) For a piece of code violating a given set of coding standards and guidelines, fix the maintainability defects reported by static code analysis. Subsequently confirm by re-testing that the defects are resolved and verify that no new issues have been introduced.

Guideline for hands-on objective:

Provide a set of coding standards and guidelines out of the ones mentioned in the Syllabus. Provide a piece of code that is syntactically correct and contains violations against this set. Run a static analysis tool testing the code against this given set and provide the report on deviations to the participants.

The participants shall fix the maintainability defects reported by the tool. They shall rerun the static analysis to confirm that the defects are resolved and verify that no new issues have been introduced.

Note: TTA-3.2.4 has been removed from Advanced Level Syllabus – Technical Test Analyst v4.0

3.1 Static Testing Basics (non-exam relevant)

In contrast to dynamic testing, which requires the execution of the software being tested, static testing relies on the manual examination of work products (i.e., reviews) or tool-driven evaluation of the code or other work products (i.e., static analysis). Both types of static testing assess the code or other work product being tested without actually executing the code or work product being tested.

Static analysis is important for safety-critical computer systems (e.g., aviation, medical, or nuclear software), but static analysis has also become important and common in other settings. For example, static analysis is an important part of security testing. Static analysis is also often incorporated into automated software build and distribution tools, for example in Agile development, continuous delivery, and continuous deployment.

3.1.1 Work Products that Can Be Examined by Static Testing (non-exam relevant)

Almost any work product can be examined using static testing (reviews and/or static analysis), for example:

- *Specifications, including business requirements, functional requirements, and security requirements*
- *Epics, user stories, and acceptance criteria*
- *Architecture and design specifications*
- *Code*
- *Testware, including test plans, test cases, test procedures, and automated test scripts*
- *User guides*
- *Web pages*
- *Contracts, project plans, schedules, and budget planning*
- *Configuration set up and infrastructure set up*
- *Models, such as activity diagrams, which may be used for Model-Based testing (see ISTQB-CTFL-MBT and Kramer 2016)*

Reviews can be applied to any work product that the participants know how to read and understand. Static analysis can be applied efficiently to any work product with a formal structure (typically code or models) for which an appropriate static analysis tool exists. Static analysis can even be applied with tools that evaluate work products written in natural language such as requirements (e.g., checking for spelling, grammar, and readability).

3.1.2 Benefits of Static Testing (non-exam relevant)

Static testing techniques provide a variety of benefits. When applied early in the software development lifecycle, static testing enables the early detection of defects before dynamic testing is performed (e.g., in requirements or design specifications reviews, backlog refinement, etc.). Defects found early are often much cheaper to remove than defects found later in the lifecycle, especially compared to defects found after the software is deployed and in active use. Using static testing techniques to find defects and then fixing those defects promptly is almost always much cheaper for the organization than using dynamic testing to find defects in the test object and then fixing them, especially when considering the additional costs associated with updating other work products and performing confirmation and regression testing.

Additional benefits of static testing may include:

- *Detecting and correcting defects more efficiently, and prior to dynamic test execution*
- *Identifying defects which are not easily found by dynamic testing*
- *Preventing defects in design or coding by uncovering inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies in requirements*
- *Increasing development productivity (e.g., due to improved design, more maintainable code)*
- *Reducing development cost and time*
- *Reducing testing cost and time*
- *Reducing total cost of quality over the software's lifetime, due to fewer failures later in the lifecycle or after delivery into operation*
- *Improving communication between team members in the course of participating in reviews*

3.1.3 Differences between Static and Dynamic Testing (non-exam relevant)

Static testing and dynamic testing can have the same objectives (see section 1.1.1), such as providing an assessment of the quality of the work products and identifying defects as early as possible. Static and dynamic testing complement each other by finding different types of defects.

One main distinction is that static testing finds defects in work products directly rather than identifying failures caused by defects when the software is run. A defect can reside in a work product for a very long time without causing a failure. The path where the defect lies may be rarely exercised or hard to reach, so it will not be easy to construct and execute a dynamic test that encounters it. Static testing may be able to find the defect with much less effort.

Another distinction is that static testing can be used to improve the consistency and internal quality of work products, while dynamic testing typically focuses on externally visible behaviors.

Compared with dynamic testing, typical defects that are easier and cheaper to find and fix through static testing include:

- *Requirement defects (e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies)*
- *Design defects (e.g., inefficient algorithms or database structures, high coupling, low cohesion)*
- *Coding defects (e.g., variables with undefined values, variables that are declared but never used, unreachable code, duplicate code)*
- *Deviations from standards (e.g., lack of adherence to coding standards)*
- *Incorrect interface specifications (e.g., different units of measurement used by the calling system than by the called system)*
- *Security vulnerabilities (e.g., susceptibility to buffer overflows)*
- *Gaps or inaccuracies in test basis traceability or coverage (e.g., missing tests for an acceptance criterion)*

Moreover, most types of maintainability defects can only be found by static testing (e.g., improper modularization, poor reusability of components, code that is difficult to analyze and modify without introducing new defects).

3.2 Applying Review Techniques

Reviews vary from informal to formal. Informal reviews are characterized by not following a defined process and not having formal documented output. Formal reviews are characterized by team participation, documented results of the review, and documented procedures for conducting the review. The formality of a review process is related to factors such as the software development lifecycle model, the maturity of the development process, the complexity of the work product to be reviewed, any legal or regulatory requirements, and/or the need for an audit trail.

The focus of a review depends on the agreed objectives of the review (e.g., finding defects, gaining understanding, educating participants such as testers and new team members, or discussing and deciding by consensus).

ISO standard (ISO/IEC 20246) contains more in-depth descriptions of the review process for work products, including roles and review techniques.

3.2.1 Applying Review Techniques

There are a number of review techniques that can be applied during the individual review (i.e., individual preparation) activity to uncover defects. These techniques can be used across the review types described above. The effectiveness of the techniques may differ depending on the type of review used. Examples of different individual review techniques for various review types are listed below.

Ad hoc

In an ad hoc review, reviewers are provided with little or no guidance on how this task should be performed. Reviewers often read the work product sequentially, identifying and documenting issues as they encounter them. Ad hoc reviewing is a commonly used technique needing little preparation. This technique is highly dependent on reviewer skills and may lead to many duplicate issues being reported by different reviewers.

Checklist-based

A checklist-based review is a systematic technique, whereby the reviewers detect issues based on checklists that are distributed at review initiation (e.g., by the facilitator). A review checklist consists of a set of questions based on potential defects, which may be derived from experience. Checklists should be specific to the type of work product under review and should be maintained regularly to cover issue types missed in previous reviews. The main advantage of the checklist-based technique is a systematic coverage of typical defect types. Care should be taken not to simply follow the checklist in individual reviewing, but also to look for defects outside the checklist.

Scenarios and dry runs

In a scenario-based review, reviewers are provided with structured guidelines on how to read through the work product. A scenario-based review supports reviewers in performing “dry runs” on the work product based on expected usage of the work product (if the work product is documented in a suitable format such as use cases). These scenarios provide reviewers with better guidelines on how to identify specific defect types than simple checklist entries. As with checklist-based reviews, in order not to miss other defect types (e.g., missing features), reviewers should not be constrained to the documented scenarios.

Perspective-based

In perspective-based reading, similar to a role-based review, reviewers take on different stakeholder viewpoints in individual reviewing. Typical stakeholder viewpoints include end user, marketing, designer, tester, or operations. Using different stakeholder viewpoints leads to more depth in individual reviewing with less duplication of issues across reviewers.

In addition, perspective-based reading also requires the reviewers to attempt to use the work product under review to generate the product they would derive from it. For example, a tester would attempt to generate draft acceptance tests if performing a perspective-based reading on a requirements specification to see if all the necessary information was included. Further, in perspective-based reading, checklists are expected to be used.

Empirical studies have shown perspective-based reading to be the most effective general technique for reviewing requirements and technical work products. A key success factor is including and weighing different stakeholder viewpoints appropriately, based on risks. See Shul 2000 for details on perspective-based reading, and Sauer 2000 for the effectiveness of different review techniques.

Role-based

A role-based review is a technique in which the reviewers evaluate the work product from the perspective of individual stakeholder roles. Typical roles include specific end user types (experienced, inexperienced, senior, child, etc.), and specific roles in the organization (user administrator, system administrator, performance tester, etc.). The same principles apply as in perspective-based reading because the roles are similar.

3.3 Static Analysis

The objective of static analysis is to detect actual or potential defects in code and system architecture and to improve their maintainability.

3.3.1 Control Flow Analysis

Control flow analysis is the static technique where the steps followed through a program are analyzed through the use of a control flow graph , usually with the use of a tool. There are a number of anomalies which can be found in a system using this technique, including loops that are badly designed (e.g., having multiple entry points or that do not terminate), ambiguous targets of function calls in certain languages, incorrect sequencing of operations, code that cannot be reached, uncalled functions, etc.

Control flow analysis can be used to determine cyclomatic complexity. The cyclomatic complexity is a positive integer which represents the number of independent paths in a strongly connected graph.

The cyclomatic complexity is generally used as an indicator of the complexity of a component. Thomas McCabe's theory [McCabe76] was that the more complex the system, the harder it would be to maintain and the more defects it would contain. Many studies have noted this correlation between complexity and the number of contained defects. Any component that is measured with a higher complexity should be reviewed for possible refactoring, for example division into multiple components.

3.3.2 Data Flow Analysis

Data flow analysis covers a variety of techniques which gather information about the use of variables in a system. The lifecycle of each variable along a control flow path is investigated, (i.e., where it is declared, defined, used, and destroyed), since potential anomalies can be identified if these actions are used out of sequence [Beizer90].

One common technique classifies the use of a variable as one of three atomic actions:

- when the variable is defined, declared, or initialized (e.g., x:=3)

- when the variable is used or read (e.g., if x > temp)
- when the variable is killed, destroyed, or goes out of scope (e.g., text_file_1.close, loop control variable (i) on exit from loop)

Sequences of such actions that indicate potential anomalies include:
- definition followed by another definition or kill with no intervening use
- definition with no subsequent kill (e.g., leading to a possible memory leak for dynamically allocated variables)
- use or kill before definition
- use or kill after a kill

Depending on the programming language, some of these anomalies may be identified by the compiler, but a separate static analysis tool might be needed to identify the data flow anomalies. For instance, re-definition with no intervening use is allowed in most programming languages, and may be deliberately programmed, but it would be flagged by a data flow analysis tool as being a possible anomaly that should be checked.

The use of control flow paths to determine the sequence of actions for a variable can lead to the reporting of potential anomalies that cannot occur in practice. For instance, static analysis tools cannot always identify if a control flow path is feasible, as some paths are only determined based on values assigned to variables at run time. There is also a class of data flow analysis problems that are difficult for tools to identify, when the analyzed data are part of data structures with dynamically assigned variables, such as records and arrays. Static analysis tools also struggle with identifying potential data flow anomalies when variables are shared between concurrent threads of control in a program as the sequence of actions on data becomes difficult to predict.

In contrast to data flow analysis, which is static testing, data flow testing is dynamic testing in which test cases are generated to exercise ‘definition-use pairs’ in program code. Data flow testing uses some of the same concepts as data flow analysis as these definition-use pairs are control flow paths between a definition and a subsequent use of a variable in a program.

3.3.3 Using Static Analysis for Improving Maintainability

Static analysis can be applied in several ways to improve the maintainability of code, architecture and websites.

Poorly written, uncommented and unstructured code tends to be harder to maintain. It may require more effort for developers to locate and analyze defects in the code, and the modification of the code to correct a defect or add a new feature may result in further defects being introduced.

Static analysis is used to verify compliance with coding standards and guidelines; where non-compliant code is identified, it can be updated to improve its maintainability. These standards and guidelines describe required coding and design practices such as conventions for naming, commenting, indentation and modularization. Note that static analysis tools generally raise warnings rather than detect defects. These warnings (e.g., on level of complexity) may be provided even though the code may be syntactically correct.

Modular designs generally result in more maintainable code. Static analysis tools support the development of modular code in the following ways:
- They search for repeated code. These sections of code may be candidates for refactoring into modules (although the runtime overhead imposed by module calls may be an issue for real-time systems).
- They generate metrics which are valuable indicators of code modularization. These include measures of coupling and cohesion. A system that is to have good maintainability is more likely

to have a low measure of coupling (the degree to which modules rely on each other during execution) and a high measure of cohesion (the degree to which a module is self-contained and focused on a single task).

- They indicate, in object-oriented code, where derived objects may have too much or too little visibility into parent classes.
- They highlight areas in code or architecture with a high level of structural complexity.

The maintenance of a web site can also be supported using static analysis tools. Here the objective is to check if the tree-like structure of the site is well-balanced or if there is an imbalance that will lead to:

- More difficult testing tasks
- Increased maintenance workload

In addition to evaluating maintainability, static analysis tools can also be applied to the code used for implementing websites to check for possible exposure to security vulnerabilities such as code injection, cookie security, cross-site scripting, resource tampering, and SQL code injection. Further details are provided in [ISTQB_ATTA_SYL – Section 4.3 Security Testing] and in the Advanced Level Security Testing syllabus [ISTQB_ ALSEC_SYL].

4 Test Techniques – 450 minutes

Keywords

atomic condition, black-box test technique, boundary value analysis, coverage, decision coverage, decision table testing, decision testing, equivalence partitioning, experience-based test technique, multiple condition testing, modified condition / decision testing, state transition testing, statement coverage, statement testing, short-circuiting, test technique, use case testing, white-box test technique

Learning Objectives for Test Techniques

4.1 Test Techniques

FL-4.1.1	(K2) Explain the characteristics, commonalities, and differences between black-box test techniques, white-box test techniques, and experience-based test techniques

4.2 Black-box Test Techniques

FL-4.2.1	(K3) Apply equivalence partitioning to derive test cases from given requirements
HO-4.2.1	(H2) For a given specification item, design and implement a test suite, applying equivalence partitioning. Execute the test suite with the corresponding software.

Guideline for hands-on objective:

Provide a specification item and the corresponding software as test item. The test item shall contain defects that can be detected by equivalence partitioning.

The participants shall design, implement and execute the test cases and verify that all partitions are covered. If not, they shall add test cases until the test goal is reached and all defects are detected.

Optional: Fix the defects in the software and rerun the test cases to confirm that the defects are resolved and verify that no new issues have been introduced.

The example shall lead to test cases for at least 2 valid and at least 1 invalid equivalence classes.

FL-4.2.2	(K3) Apply boundary value analysis to derive test cases from given requirements
HO-4.2.2	(H2) For a given specification item, design and implement a test suite, applying boundary value analysis. Execute the test suite with the corresponding software.

Guideline for hands-on objective:

Provide a specification item and the corresponding software as test item. The test item shall contain defects that can be detected by boundary value analysis.

The participants shall design, implement and execute the test cases and verify that all boundary values are covered. If not, they shall add test cases until the test goal is reached and all defects are detected.

Optional: Fix the defects in the software and rerun the test cases to confirm that the defects are resolved and verify that no new issues have been introduced.

The example shall lead to test cases for at least 4 boundary values (2 boundaries).

FL-4.2.3	(K3) Apply decision table testing to derive test cases from given requirements
HO-4.2.3	(H2) For a given specification item, design and implement a test suite, applying decision table testing. Execute the test suite with the corresponding software.

Guideline for hands-on objective:

Provide a specification item and the corresponding software as test item. The test item shall contain defects that can be detected by decision table testing.

The participants shall design, implement and execute the test cases and verify that all table entries are covered. If not, they shall add test cases until the test goal is reached and all defects are detected.

Optional: Fix the defects in the software and rerun the test cases to confirm that the defects are resolved and verify that no new issues have been introduced.

The table in the example shall contain at least 3 conditions.

FL-4.2.4	(K3) Apply state transition testing to derive test cases from given requirements
HO-4.2.4	(H2) For a given specification item, design and implement a test suite, applying state transition testing. Execute the test suite with the corresponding software.

Guideline for hands-on objective:

Provide a piece of code and the corresponding software component specification. A few executable statements should contain defects that can be detected using state transition testing (e.g., describing a finite state machine).

The participants shall design, implement and execute the test cases and verify that all state transitions are covered. If not, they shall add test cases until the test goal is reached and all defects are detected.

The example shall be non-trivial, leading to at least 5 test cases.

FL-4.2.5	(K2) Explain how to derive test cases from a use case

4.3 White-box Test Techniques

4.3.1 Application test

FL-4.3.1	(K2) Explain statement coverage

TTA-2.2.1	(K3) Design test cases for a given test object by applying statement testing to achieve a defined level of coverage
HO-2.2.1	(H2) For a given specification item and a corresponding piece of code, design and implement a test suite with the goal to reach 100% statement coverage, and verify after execution that the test goal has been reached.

Guideline for hands-on objective:

Provide a piece of code and the corresponding software component specification. A few executable statements should contain defects that can be detected by statement coverage.

The participants shall design, implement and execute the test cases and verify that 100% statement coverage is reached. If not, they shall add test cases until the test goal is reached and all defects are detected.

The example shall be non-trivial, leading to at least 3 test cases.

4.3.2 Decision Testing

FL-4.3.2	(K2) Explain decision coverage
TTA-2.3.1	(K3) Design test cases for a given test object by applying the Decision test technique to achieve a defined level of coverage
HO-2.3.1	(H2) For a given specification item and a corresponding piece of code, design and implement a test suite with the goal to reach 100% decision coverage, and verify after execution that the test goal has been reached.

Guideline for hands-on objective:

Provide a piece of code and the corresponding software component specification. A few decisions or execution paths should contain defects that can be detected by decision coverage but not necessarily by statement coverage.

The participants shall design, implement and execute the test cases and verify that 100% decision coverage is reached. If not, they shall add test cases until the test goal is reached and all defects are detected.

The example shall be non-trivial, leading to at least 3 test cases. It shall show the advantage compared to statement testing.

4.3.3 The Value of Statement and Decision Testing

FL-4.3.3	(K2) Explain the value of statement and decision coverage

4.3.4 Modified Condition/Decision Coverage (MC/DC) Testing

TTA-2.4.1	(K3) Design test cases for a given test object by applying the modified condition/decision test technique to achieve full modified condition/decision coverage (MC/DC)
HO-2.4.1	(H2) For a given specification item and a corresponding piece of code, that contains a decision with multiple atomic conditions, design, implement and execute a test suite with the goal to reach 100% modified condition / decision coverage.*Guideline for hands-on objective:*

Provide a piece of code that shows a decision with several independent atomic conditions and the corresponding software development specification. The decision should contain a defect and this defect should be identified by the test cases.

The participants shall design, implement and execute the test cases. The trainer shall verify that 100% MC/DC coverage is reached.

The decision example shall be non-trivial, with at least 3 atomic conditions. It shall show the advantage compared to Decision Testing. For MC/DC, manual design of the test cases is feasible, but the training may also use a tool to either generate the inputs or verify the coverage.

4.4 Experience-based Test Techniques (non-exam relevant)

FL-4.4.1	*(K2) Explain error guessing (non-exam relevant)*
FL-4.4.2	*(K2) Explain exploratory testing (non-exam relevant)*
FL-4.4.3	*(K2) Explain checklist-based testing (non-exam relevant)*

4.1 Test Techniques

The purpose of a test technique, including those discussed in this section, is to help in identifying test conditions, test cases, and test data.

The choice of which test techniques to use depends on a number of factors, including:

- Component or system complexity
- Regulatory standards
- Customer or contractual requirements
- Risk levels and types
- Available documentation
- Tester knowledge and skills
- Available tools
- Time and budget
- Software development lifecycle model
- The types of defects expected in the component or system

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels. When creating test cases, testers generally use a combination of test techniques to achieve the best results from the test effort.

The use of test techniques in the test analysis, test design, and test implementation activities can range from very informal (little to no documentation) to very formal. The appropriate level of formality depends on the context of testing, including the maturity of test and development processes, time constraints, safety or regulatory requirements, the knowledge and skills of the people involved, and the software development lifecycle model being followed.

4.1.1 Categories of Test Techniques and their Characteristics

In this syllabus, test techniques are classified as black-box, white-box, or experience-based.

Black-box test techniques (also called behavioral or behavior-based techniques) are based on an analysis of the appropriate test basis (e.g., formal requirements documents, specifications, use cases, user stories, or business processes). These techniques are applicable to both functional and non-functional testing. Black-box test techniques concentrate on the inputs and outputs of the test object without reference to its internal structure.

White-box test techniques (also called structural or structure-based techniques) are based on an analysis of the architecture, detailed design, internal structure, or the code of the test object. Unlike black-box test techniques, white-box test techniques concentrate on the structure and processing within the test object.

Experience-based test techniques leverage the experience of developers, testers and users to design, implement, and execute tests. These techniques are often combined with black-box and white-box test techniques.

Common characteristics of black-box test techniques include the following:

- Test conditions, test cases, and test data are derived from a test basis that may include software requirements, specifications, use cases, and user stories
- Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements
- Coverage is measured based on the items tested in the test basis and the technique applied to the test basis

Common characteristics of white-box test techniques include:

- Test conditions, test cases, and test data are derived from a test basis that may include code, software architecture, detailed design, or any other source of information regarding the structure of the software
- Coverage is measured based on the items tested within a selected structure (e.g., the code or interfaces) and the technique applied to the test basis

Common characteristics of experience-based test techniques include:

- Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders

This knowledge and experience includes expected use of the software, its environment, likely defects, and the distribution of those defects

The international standard (ISO/IEC/IEEE 29119-4) contains descriptions of test techniques and their corresponding coverage measures (see Craig 2002 and Copeland 2004 for more on techniques).

4.2 Black-box Test Techniques

4.2.1 Equivalence Partitioning

Equivalence partitioning divides data into partitions (also known as equivalence classes) in such a way that all the members of a given partition are expected to be processed in the same way (see Kaner 2013 and Jorgensen 2014). There are equivalence partitions for both valid and invalid values.

- Valid values are values that should be accepted by the component or system. An equivalence partition containing valid values is called a “valid equivalence partition.”
- Invalid values are values that should be rejected by the component or system. An equivalence partition containing invalid values is called an “invalid equivalence partition.”
- Partitions can be identified for any data element related to the test object, including inputs, outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing).
- Any partition may be divided into sub partitions if required.
- Each value must belong to one and only one equivalence partition.
- When invalid equivalence partitions are used in test cases, they should be tested individually, i.e., not combined with other invalid equivalence partitions, to ensure that failures are not masked. Failures can be masked when several failures occur at the same time but only one is visible, causing the other failures to be undetected.

To achieve 100% coverage with this technique, test cases must cover all identified partitions (including invalid partitions) by using a minimum of one value from each partition. Coverage is measured as the number of equivalence partitions tested by at least one value, divided by the total number of identified equivalence partitions, normally expressed as a percentage. Equivalence partitioning is applicable at all test levels.

4.2.2 Boundary Value Analysis

Boundary value analysis (BVA) is an extension of equivalence partitioning, but can only be used when the partition is ordered, consisting of numeric or sequential data. The minimum and maximum values (or first and last values) of a partition are its boundary values (see Beizer 1990).

For example, suppose an input field accepts a single integer value as an input, using a keypad to limit inputs so that non-integer inputs are impossible. The valid range is from 1 to 5, inclusive. So, there are three equivalence partitions: invalid (too low); valid; invalid (too high). For the valid equivalence partition, the boundary values are 1 and 5. For the invalid (too high) partition, the boundary value is 6. For the invalid (too low) partition, there is only one boundary value, 0, because this is a partition with only one member.

In the example above, we identify two boundary values per boundary. The boundary between invalid (too low) and valid gives the test values 0 and 1. The boundary between valid and invalid (too high) gives the test values 5 and 6. Some variations of this technique identify three boundary values per boundary: the values before, at, and just over the boundary. In the previous example, using three-point boundary values, the lower boundary test values are 0, 1, and 2, and the upper boundary test values are 4, 5, and 6 (see Jorgensen 2014).

Behavior at the boundaries of equivalence partitions is more likely to be incorrect than behavior within the partitions. It is important to remember that both specified and implemented boundaries may be displaced to positions above or below their intended positions, may be omitted altogether, or may be supplemented
with unwanted additional boundaries. Boundary value analysis and testing will reveal almost all such defects by forcing the software to show behaviors from a partition other than the one to which the boundary value should belong.

Boundary value analysis can be applied at all test levels. This technique is generally used to test requirements that call for a range of numbers (including dates and times). Boundary coverage for a partition is measured as the number of boundary values tested, divided by the total number of identified boundary test values, normally expressed as a percentage.

4.2.3 Decision Table Testing

Decision tables are a good way to record complex business rules that a system must implement. When creating decision tables, the tester identifies conditions (often inputs) and the resulting actions (often outputs) of the system. These form the rows of the table, usually with the conditions at the top and the actions at the bottom. Each column corresponds to a decision rule that defines a unique combination of conditions which results in the execution of the actions associated with that rule. The values of the conditions and actions are usually shown as Boolean values (true or false) or discrete values (e.g., red, green, blue), but can also be numbers or ranges of numbers. These different types of conditions and actions might be found together in the same table.

The common notation in decision tables is as follows:

For conditions:

- Y means the condition is true (may also be shown as T or 1)

- N means the condition is false (may also be shown as F or 0)
- — means the value of the condition doesn’t matter (may also be shown as N/A)

For actions:

- X means the action should occur (may also be shown as Y or T or 1)
- Blank means the action should not occur (may also be shown as – or N or F or 0)

A full decision table has enough columns (test cases) to cover every combination of conditions. By deleting columns that do not affect the outcome, the number of test cases can decrease considerably. For example by removing impossible combinations of conditions. For more information on how to collapse decision tables. (see ISTQB-CTAL-AT).

The common minimum coverage standard for decision table testing is to have at least one test case per decision rule in the table. This typically involves covering all combinations of conditions. Coverage is measured as the number of decision rules tested by at least one test case, divided by the total number of decision rules, normally expressed as a percentage.

The strength of decision table testing is that it helps to identify all the important combinations of conditions, some of which might otherwise be overlooked. It also helps in finding any gaps in the requirements. It may be applied to all situations in which the behavior of the software depends on a combination of conditions, at any test level.

4.2.4 State Transition Testing

Components or systems may respond differently to an event depending on current conditions or previous history (e.g., the events that have occurred since the system was initialized). The previous history can be summarized using the concept of states. A state transition diagram shows the possible software states, as well as how the software enters, exits, and transitions between states. A transition is initiated by an event (e.g., user input of a value into a field). The event results in a transition. The same event can result in two or more different transitions from the same state. The state change may result in the software taking an action (e.g., outputting a calculation or error message).

A state transition table shows all valid transitions and potentially invalid transitions between states, as well as the events, and resulting actions for valid transitions. State transition diagrams normally show only the valid transitions and exclude the invalid transitions.

Tests can be designed to cover a typical sequence of states, to exercise all states, to exercise every transition, to exercise specific sequences of transitions, or to test invalid transitions.

State transition testing is used for menu-based applications and is widely used within the embedded software industry. The technique is also suitable for modeling a business scenario having specific states or for testing screen navigation. The concept of a state is abstract – it may represent a few lines of code or an entire business process.

Coverage is commonly measured as the number of identified states or transitions tested, divided by the total number of identified states or transitions in the test object, normally expressed as a percentage. For more information on coverage criteria for state transition testing, (see ISTQB-CTAL-AT).

4.2.5 Use Case Testing

Tests can be derived from use cases, which are a specific way of designing interactions with software items. They incorporate requirements for the software functions. Use cases are associated with actors (human users, external hardware, or other components or systems) and subjects (the component or system to which the use case is applied).

Each use case specifies some behavior that a subject can perform in collaboration with one or more actors (UML 2.5.1 2017). A use case can be described by interactions and activities, as well as preconditions, postconditions and natural language where appropriate. Interactions between the actors and the subject may result in changes to the state of the subject. Interactions may be represented graphically by work flows, activity diagrams, or business process models.

A use case can include possible variations of its basic behavior, including exceptional behavior and error handling (system response and recovery from programming, application and communication errors, e.g., resulting in an error message). Tests are designed to exercise the defined behaviors (basic, exceptional or alternative, and error handling). Coverage can be measured by the number of use case behaviors tested divided by the total number of use case behaviors, normally expressed as a percentage.

For more information on coverage criteria for use case testing. (see the ISTQB-CTAL-AT).

4.3 White-box Test Techniques

White-box testing is based on the internal structure of the test object. White-box test techniques can be used at all test levels, but the two code-related techniques discussed in this section are most commonly used at the component test level. There are more advanced techniques that are used in some safety-critical, mission-critical, or high integrity environments to achieve more thorough coverage, but those are not discussed here. For more information on such techniques, see the ISTQB-CTAL-TTA.

4.3.1 Statement Testing and Coverage

Statement testing exercises the executable statements in the code. Coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage.

Applicability

Achieving full statement coverage should be considered as a minimum for all code being tested, although this is not always possible in practice.

Limitations/Difficulties

Achieving full statement coverage should be considered as a minimum for all code being tested, although this is not always possible in practice due to constraints on the available time and/or effort. Even high percentages of statement coverage may not detect certain defects in the code's logic. In many cases achieving 100% statement coverage is not possible due to unreachable code. Although unreachable code is generally not considered good programming practice, it may occur, for instance, if a switch statement must have a default case, but all possible cases are handled explicitly.

4.3.2 Decision Testing and Coverage

Decision testing exercises the decision outcomes in the code. To do this, the test cases follow the control flows from a decision point (e.g., for an IF statement, there is one control flow for the true outcome and one for the false outcome; for a CASE statement, there may be several possible outcomes; for a LOOP statement there is one control flow for the true outcome of the loop condition and one for the false outcome).

Coverage is measured as the number of decision outcomes exercised by the tests divided by the total number of decision outcomes in the test object, normally expressed as a percentage. Note that a single test case may exercise several decision outcomes.

Compared to the modified condition/decision and multiple condition techniques described below, decision testing considers the entire decision as a whole and evaluates only the TRUE and FALSE outcomes, regardless of the complexity of its internal structure.

Branch testing is often used interchangeably with decision testing, because covering all branches and covering all decision outcomes can be achieved with the same tests. Branch testing exercises the branches in the code, where a branch is normally considered to be an edge of the control flow graph. For programs with no decisions, the definition of decision coverage above results in a coverage of 0/0, which is undefined, no matter how many tests are run, while the single branch from entry to exit point (assuming one entry and exit point) will result in 100% branch coverage being achieved. To address this difference between the two measures, ISO 29119-4 requires at least one test to be run on code with no decisions to achieve 100% decision coverage, so making 100% decision coverage and 100% branch coverage equivalent for nearly all programs. Many test tools that provide coverage measures, including those used for testing safety-related systems, employ a similar approach.

Applicability

This level of coverage should be considered when the code being tested is important or even critical [see ISTQB_ATTA_SYL the tables in section 2.8.2 safety-related systems]. This technique can be used for code and for any model that involves decision points, like business process models.

Limitations/Difficulties

Decision testing does not consider the details of how a decision with multiple conditions is made and may fail to detect defects caused by combinations of the condition outcomes.

4.3.3 The Value of Statement and Decision Testing

When 100% statement coverage is achieved, it ensures that all executable statements in the code have been tested at least once, but it does not ensure that all decision logic has been tested. Of the two white-box techniques discussed in this syllabus, statement testing may provide less coverage than decision testing.

When 100% decision coverage is achieved, it executes all decision outcomes, which includes testing the true outcome and also the false outcome, even when there is no explicit false statement (e.g., in the case of an IF statement without an else in the code). Statement coverage helps to find defects in code that was not exercised by other tests. Decision coverage helps to find defects in code where other tests have not taken both true and false outcomes.

Achieving 100% decision coverage guarantees 100% statement coverage (but not vice versa).

4.3.4 Modified Condition/Decision Coverage (MC/DC) Testing

Compared to decision testing, which considers the entire decision as a whole and evaluates the TRUE and FALSE outcomes, modified condition/decision testing considers how a decision is structured when it includes multiple conditions (where a decision is composed of only one atomic condition, it is simply decision testing).

Each decision predicate is made up of one or more atomic conditions, each of which evaluates to a Boolean value. These are logically combined to determine the outcome of the decision. This technique checks that each of the atomic conditions independently and correctly affects the outcome of the overall decision.

This technique provides a stronger level of coverage than statement and decision coverage when there are decisions containing multiple conditions. Assuming N unique, mutually independent atomic conditions, MC/DC for a decision can usually be achieved by exercising the decision N+1 times. Modified condition/decision testing requires pairs of tests that show a change of a single atomic condition outcome can independently affect the result of a decision. Note that a single test case may exercise several condition combinations and therefore it is not always necessary to run N+1 separate test cases to achieve MC/DC.

Applicability

This technique is used in the aerospace and automotive industries, and other industry sectors for safety-critical systems. It is used when testing software where a failure may cause a catastrophe. Modified condition/decision testing can be a reasonable middle ground between decision testing and multiple condition testing (due to the large number of combinations to test). It is more rigorous than decision testing but requires far fewer test conditions to be exercised than multiple condition testing when there are several atomic conditions in the decision.

Limitations/Difficulties

Achieving MC/DC may be complicated when there are multiple occurrences of the same variable in a decision with multiple conditions; when this occurs, the conditions may be “coupled”. Depending on the decision, it may not be possible to vary the value of one condition such that it alone causes the decision outcome to change. One approach to addressing this issue is to specify that only uncoupled atomic conditions are tested using modified condition/decision testing . The other approach is to analyze each decision in which coupling occurs.

Some compilers and/or interpreters are designed such that they exhibit short-circuiting behavior when evaluating a complex decision statement in the code. That is, the executing code may not evaluate an entire expression if the final outcome of the evaluation can be determined after evaluating only a portion of the expression. For example, if evaluating the decision “A and B”, there is no reason to evaluate B if A has already been evaluated as FALSE. No value of B can change the final result, so the code may save execution time by not evaluating B. Short-circuiting may affect the ability to attain MC/DC since some required tests may not be achievable. Usually, it is possible to configure the compiler to switch off the short-circuiting optimization for the testing, but this may not be allowed for safety-critical applications, where the tested code and the delivered code must be identical.

4.4 Experience-based Test Techniques (non-exam relevant)

When applying experience-based test techniques, the test cases are derived from the tester’s skill and intuition, and their experience with similar applications and technologies. These techniques can be helpful in identifying tests that were not easily identified by other more systematic techniques. Depending on the tester’s approach and experience, these techniques may achieve widely varying degrees of coverage and effectiveness. Coverage can be difficult to assess and may not be measurable with these techniques.

Commonly used experience-based techniques are discussed in the following sections.

4.4.1 Error Guessing (non-exam relevant)

Error guessing is a technique used to anticipate the occurrence of errors, defects, and failures, based on the tester’s knowledge, including:

- *How the application has worked in the past*
- *What kind of errors tend to be made*
- *Failures that have occurred in other applications*

A methodical approach to the error guessing technique is to create a list of possible errors, defects, and failures, and design tests that will expose those failures and the defects that caused them. These error, defect, failure lists can be built based on experience, defect and failure data, or from common knowledge about why software fails.

4.4.2 Exploratory Testing (non-exam relevant)

In exploratory testing, informal (not pre-defined) tests are designed, executed, logged, and evaluated dynamically during test execution. The test results are used to learn more about the component or system, and to create tests for the areas that may need more testing.

Exploratory testing is sometimes conducted using session-based testing to structure the activity. In session-based testing, exploratory testing is conducted within a defined time-box, and the tester uses a test charter containing test objectives to guide the testing. The tester may use test session sheets to document the steps followed and the discoveries made.

Exploratory testing is most useful when there are few or inadequate specifications or significant time pressure on testing. Exploratory testing is also useful to complement other more formal testing techniques.

Exploratory testing is strongly associated with reactive test strategies (see section 5.2.2). Exploratory testing can incorporate the use of other black-box, white-box, and experience-based techniques.

4.4.3 Checklist-based Testing (non-exam relevant)

In checklist-based testing, testers design, implement, and execute tests to cover test conditions found in a checklist. As part of analysis, testers create a new checklist or expand an existing checklist, but testers may also use an existing checklist without modification. Such checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails.

Checklists can be created to support various test types, including functional and non-functional testing. In the absence of detailed test cases, checklist-based testing can provide guidelines and a degree of consistency. As these are high-level lists, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

5 References

5.1 Standards

ISO/IEC/IEEE 29119-1 (2013) Software and systems engineering - Software testing - Part 1: Concepts and definitions

ISO/IEC/IEEE 29119-2 (2013) Software and systems engineering - Software testing - Part 2: Test processes

ISO/IEC/IEEE 29119-3 (2013) Software and systems engineering - Software testing - Part 3: Test documentation

ISO/IEC/IEEE 29119-4 (2015) Software and systems engineering - Software testing - Part 4: Test techniques

ISO/IEC 25010, (2011) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models

ISO/IEC 20246: (2017) Software and systems engineering — Work product reviews

UML 2.5, Unified Modeling Language Reference Manual, http://www.omg.org/spec/UML/2.5.1/, 2017

The following standards are mentioned in these respective chapters.

[RTCA DO-178C/ED-12C]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12C. 2013. Chapter 2

[ISO9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality Chapter 4

[ISO25010] ISO/IEC 25010 (2014) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models Chapters 2 and 4

[ISO29119] ISO/IEC/IEEE 29119-4 International Standard for Software and Systems Engineering - Software Testing Part 4: Test techniques. 2015 Chapter 2

[ISO42010] ISO/IEC/IEEE 42010:2011 Systems and software engineering - Architecture description [ISTQB_FL_SYL - Chapter 5]

[IEC61508] IEC 61508-5 (2010) Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Part 5: Examples of methods for the determination of safety integrity levels Chapter 2

5.2 ISTQB® documents

[ISTQB_GLOSSARY] Glossary of Terms used in Software Testing, Version 3.2, 2019

[ISTQB_FL_SYL] Foundation Level Syllabus, Version 2018 V3.1

[ISTQB_FL_OVW] Foundation Level Overview 2018

[ISTQB_FLAT_SYL] Foundation Level Agile Tester Syllabus, Version 2014

[ISTQB_FLPT_SYL] Foundation Level Performance Testing Syllabus, Version 2018

[ISTQB_FLMBT_SYL] Foundation Level Model-Based Testing Syllabus, Version 2015

[ISTQB_FLMAT_SYL] Foundation Level Mobile Application Testing Syllabus, 2019

[ISTQB_AL_OVIEW] Advanced Level Overview, Version 2019

[ISTQB_ALSEC_SYL] Advanced Level Security Testing Syllabus, Version 2016

[ISTQB_ALTAE_SYL] Advanced Level Test Automation Engineer Syllabus, Version 2017

[ISTQB_AL_OVW] Advanced Level TA & TTA Overview 2019

[ISTQB_ALTA_SYL] Advanced Level Test Analyst Syllabus, Version 2019

[ISTQB_ATTA_SYL] Advanced Level Technical Test Analyst Syllabus, Version 2019

[ISTQB_ALTM_SYL] Advanced Level Test Manager Syllabus, Version 2012

[ISTQB_ELTM_SYL] Expert Level Test Management Syllabus, Version 2011

[ISTQB_EITP_SYL] Expert Level Improving the Test Process Syllabus, Version 2011

5.3 Books and Articles

Beizer, B. (1990*) Software Testing Techniques (2e)*, Van Nostrand Reinhold: Boston MA

Black, R. (2017) *Agile Testing Foundations*, BCS Learning & Development Ltd: Swindon UK

Black, R. (2009) *Managing the Testing Process (3e)*, John Wiley & Sons: New York NY

Buwalda, H. et al. (2001) *Integrated Test Design and Automation*, Addison Wesley: Reading MA

Copeland, L. (2004) *A Practitioner's Guide to Software Test Design*, Artech House: Norwood MA

Craig, R. and Jaskiel, S. (2002) *Systematic Software Testing*, Artech House: Norwood MA

Crispin, L. and Gregory, J. (2008) *Agile Testing*, Pearson Education: Boston MA

Fewster, M. and Graham, D. (1999) *Software Test Automation*, Addison Wesley: Harlow UK

Gilb, T. and Graham, D. (1993) *Software Inspection*, Addison Wesley: Reading MA

Graham, D. and Fewster, M. (2012) *Experiences of Test Automation*, Pearson Education: Boston MA

Gregory, J. and Crispin, L. (2015) *More Agile Testing*, Pearson Education: Boston MA

Jorgensen, P. (2014) *Software Testing, A Craftsman's Approach (4e)*, CRC Press: Boca Raton FL

Kaner, C., Bach, J. and Pettichord, B. (2002) *Lessons Learned in Software Testing*, John Wiley & Sons: New York NY

Kaner, C., Padmanabhan, S. and Hoffman, D. (2013) *The Domain Testing Workbook*, Context-Driven Press: New York NY

Kramer, A., Legeard, B. (2016) *Model-Based Testing Essentials: Guide to the ISTQB® Certified Model-Based Tester: Foundation Level*, John Wiley & Sons: New York NY

Myers, G. (2011) *The Art of Software Testing*, (3e), John Wiley & Sons: New York NY

Sauer, C. (2000) “The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research,” *IEEE Transactions on Software Engineering*, *Volume 26, Issue 1, pp 1-*

Shull, F., Rus, I., Basili, V. July 2000. “How Perspective-Based Reading can Improve Requirement Inspections.” *IEEE Computer, Volume 33, Issue 7, pp 73-79*

van Veenendaal, E. (ed.) (2004) *The Testing Practitioner* (Chapters 8 - 10), UTN Publishers: The Netherlands

Wiegers, K. (2002) *Peer Reviews in Software*, Pearson Education: Boston MA

Weinberg, G. (2008) *Perfect Software and Other Illusions about Testing*, Dorset House: New York NY

Other Resources (not directly referenced in this Syllabus)

Black, R., van Veenendaal, E. and Graham, D. (2019) *Foundations of Software Testing: ISTQB® Certification (4e)*, Cengage Learning: London UK

Hetzel, W. (1993) *Complete Guide to Software Testing (2e)*, QED Information Sciences: Wellesley MA

6 Appendix

6.1 Overview Learning Objectives

		SDET-BO1	SDET-BO2	SDET-BO3	SDET-BO4	SDET-BO5	SDET-BO6	SDET-BO7	Time SDET	Time SDET & Opt	Guidelines for Trainer / Provider / Accreditation / Exam For HO: Competency Guideline
									800	855	
Chapter 1	Fundamentals of Testing									110	
Keywords CTFL:	debugging; failure; defect; error; root cause; quality; quality assurance; traceability; test procedure; test analysis; test basis; test condition; test execution; test data; testing; test design; test case; test object; test oracle; test process; test implementation; test suite; test objective; coverage; validation; verification;										
Keywords TTA:											
1.1 What is Testing?											take over
FL-1.1.1	(K1) Identify typical objectives of testing		x						5		take over
FL-1.1.2	(K2) Differentiate testing from debugging	x							15		take over
1.2 Why is Testing Necessary?											
FL-1.2.1	(K2) Give examples of why testing is necessary	x							15		take over
FL-1.2.2	(K2) Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality		x						15		take over
FL-1.2.3	(K2) Distinguish between error, defect and failure	x					x		15		take over
FL-1.2.4	(K2) Distinguish between the root cause of a defect and its effects		x				x		15		take over
1.3 Seven Testing Principles											
FL-1.3.1	(K2) Explain the seven testing principles		x						15		take over Select examples matching the lower test levels (or create new examples)
1.4 Test Process										15	Handle chapter 1.4 in 15 - 20 minutes. No exam questions.
FL-1.4.1	(K2) Explain the impact of context on the test process										Short introduction: Limit factors to the relevant test levels
FL-1.4.2	(K2) Describe the test activities and respective tasks within the test process										The focus is on the activities analysis, design, implementation and execution, which are performed iteratively. The other activities are performed within the development activities.
FL-1.4.3	(K2) Differentiate the artifacts that support the test process										Limit to the artifacts that are not part of the development artifacts and are therefore created by other ways
FL-1.4.4	(K2) Explain the value of maintaining traceability between the test basis and test artifacts										Can be briefly explained as an extension of traceability between test basis and development artifacts.
Chapter 2	Testing Throughout the Software Development Lifecycle									70	
Keywords CTFL:	change-related testing; impact analysis; confirmation testing; functional testing; integration testing; component integration testing; component testing; non-functional testing; regression testing; test type; test basis; test case; test object; test level; test environment; test objective; maintenance testing; white-box testing;										
Keywords TTA:											
2.2 Test Levels										10	Shorten to 10 minutes to provide an overview. No exam questions.

FL-2.2.1	(K2) Compare the different levels of testing from the perspective of objectives, test objects, test targets (e.g. software characteristics), related work products, responsibilities, and types of defects and failures to be identified										E.g. table -> 1 slide/flipchart 1st axis Test levels 2nd axis typical (goals, test basis, test objects, failures, responsibilities)
2.3 Test Types											
FL-2.3.1	(K2) Compare functional, non-functional and white-box testing	x		x	x				15		take over
FL-2.3.3	(K2) Compare the purposes of confirmation testing and regression testing	x					x		15		take over
2.4 Maintenance Testing											
FL-2.4.1	(K2) Summarize triggers for maintenance testing		x						15		take over
FL-2.4.2	(K2) Describe the role of impact analysis in maintenance testing		x						15		take over
Chapter 3	Static Testing									225	
Keywords CTFL:	ad hoc review; checklist-based review; dynamic testing; perspective-based reading; review; role-based reviewing; static analysis; static testing; scenario-based review;										
Keywords TTA:	data flow analysis; definition-use pair; control flow analysis; static analysis; cyclomatic complexity;										
3.1 Static Testing Basics										15	Reduce core points to 15 minutes. No exam questions
FL-3.1.1	(K1) Recognize types of software work product that can be examined by the different static testing techniques										Reduce the examples. (Code, architectures and models)
FL-3.1.2	(K2) Use examples to describe the value of static testing										Chapter 3.1.2 without the section additional benefits
FL-3.1.3	(K2) Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software lifecycle										Replace by TTA Chapter 3.1
3.2 Applying review techniques											
FL-3.2.4	(K3) Apply a review technique to a work product to find defects					H2		x	60		take over
HO-3.2.4	(H2) Review a piece of code with a given checklist. Document the findings.							x			Provide a typical code review checklist with various anomalies. Provide a piece of code that contains several of the anomalies from the checklist. Provide a findings list template to participants to document findings. Review the findings list with the participants.
3.3 Static Analysis											
TTA-3.2.1	(K3) Use control flow analysis to detect if code has any control flow anomalies and to measure cyclomatic complexity					H1		x	60		take over

HO-3.2.1	(H1) For a piece of code, use a static analysis tool to find typical control flow anomalies. Understand the report of the tool and how the anomalies affect the product quality characteristics.										Provide one or more pieces of code that are syntactically correct, and contain different types of control flow anomalies mentioned in the Syllabus. Guide the participants in running the static analysis tool and displaying the reports on control flow anomalies. Participants shall discuss the defects found and indicate the quality characteristic affected (functional correctness, maintainability, security etc.).
TTA-3.2.2	(K3) Use data flow analysis to detect if code has any data flow anomalies					H0		x	30		take over; Changed with CTAL-TA v4.0 from K2 to K3
HO-3.2.2	(H1) For a piece of code, understand the report of a static analysis tool concerning data flow anomalies and how those anomalies affect functional correctness and maintainability.										Provide a piece of code that is syntactically correct and contains the main types of data flow anomalies for some variables. Run static code analysis, explain the data flow anomalies reported to the participants, and discuss their impact on functional correctness or on maintainability.
TTA-3.2.3	(K3) Propose ways to improve the maintainability of code by applying static analysis					H2		x	60		take over
HO-3.2.3	(H2) For a piece of code violating a given set of coding standards and guidelines, fix the maintainability defects reported by static code analysis. Subsequently confirm by re-testing that the defects are resolved and verify that no new issues have been introduced.										Provide a set of coding standards and guidelines out of the ones mentioned in the Syllabus. Provide a piece of code that is syntactically correct and contains violations against this set. Run a static analysis tool testing the code against this given set and provide the report on deviations to the participants. The participants shall fix the maintainability defects reported by the tool. They shall rerun the static analysis to confirm that the defects are resolved and verify that no new issues have been introduced.
Chapter 4	Test Techniques									450	
Keywords CTFL:	statement coverage; use case testing; equivalence partitioning; black-box test technique; decision table testing; decision coverage; experience-based test technique; boundary value analysis; test technique; coverage; white-box test technique; state transition testing;										
Keywords TTA:	statement testing; atomic condition; decision testing; multiple condition testing; modified condition/decision testing; white-box test technique;										
4.1 Test Techniques											
FL-4.1.1	(K2) Explain the characteristics, commonalities, and differences between black-box test techniques, white-box test techniques and experience-based test techniques	x		x	x				15		take over
4.2 Black-box Test Techniques											
FL-4.2.1	(K3) Apply equivalence partitioning to derive test cases from given requirements	x		x		H2			60		take over
HO-4.2.1	(H2) For a given specification item, design and implement a test suite, applying equivalence partitioning. Execute the test suite with the corresponding software.										Provide a specification item and the corresponding software as test item. The test item shall contain defects that can be detected by equivalence partitioning. The participants shall design, implement and execute the test cases and verify that all partitions are covered. If not, they shall add test cases until the test goal is reached and all defects are detected. Optional: Fix the defects in the software and rerun the test cases to confirm that the defects are resolved and verify that no new issues have been introduced. The example shall lead to test cases for at least 2 valid and at least 1 invalid equivalence classes.
FL-4.2.2	(K3) Apply boundary value analysis to derive test cases from given requirements	x		x		H2			60		take over

HO-4.2.2	(H2) For a given specification item, design and implement a test suite, applying boundary value analysis. Execute the test suite with the corresponding software.										Provide a specification item and the corresponding software as test item. The test item shall contain defects that can be detected by boundary value analysis. The participants shall design, implement and execute the test cases and verify that all boundary values are covered. If not, they shall add test cases until the test goal is reached and all defects are detected. Optional: Fix the defects in the software and rerun the test cases to confirm that the defects are resolved and verify that no new issues have been introduced. The example shall lead to test cases for at least 4 boundary values (2 boundaries).
FL-4.2.3	(K3) Apply decision table testing to derive test cases from given requirements	x		x		H2			60		take over
HO-4.2.3	(H2) For a given specification item, design and implement a test suite, applying decision table testing. Execute the test suite with the corresponding software.										Provide a specification item and the corresponding software as test item. The test item shall contain defects that can be detected by decision table testing. The participants shall design, implement and execute the test cases and verify that all table entries are covered. If not, they shall add test cases until the test goal is reached and all defects are detected. Optional: Fix the defects in the software and rerun the test cases to confirm that the defects are resolved and verify that no new issues have been introduced. The table in the example shall contain at least 3 conditions.
FL-4.2.4	(K3) Apply state transition testing to derive test cases from given requirements	x		x		H2			60		take over
HO-4.2.4	(H2) For a given specification item, design and implement a test suite, applying state transition testing. Execute the test suite with the corresponding software.										Provide a piece of code and the corresponding software component specification. A few executable statements should contain defects that can be detected using state transition testing (e.g., describing a finite state machine). The participants shall design, implement and execute the test cases and verify that all state transitions are covered. If not, they shall add test cases until the test goal is reached and all defects are detected. The example shall be non-trivial, leading to at least 5 test cases.
FL-4.2.5	(K2) Explain how to derive tests from a use case	x		x					15		take over
4.3 White-box Test Techniques											
4.3.1 Statement Testing											
FL-4.3.1	(K2) Explain statement coverage	x			x				15		Combine LOs, due to redundancy the FL LO text is not included in the syllabus.
TTA-2.2.1	(K3) Design test cases for a given test object by applying statement testing to achieve a defined level of coverage	x			x	H2			30		
HO-2.2.1	(H2) For a given specification item and a corresponding piece of code, design and implement a test suite with the goal to reach 100% statement coverage, and verify after execution that the test goal has been reached.										Provide a piece of code and the corresponding software component specification. A few executable statements should contain defects that can be detected by statement coverage. The participants shall design, implement and execute the test cases and verify that 100% statement coverage is reached. If not, they shall add test cases until the test goal is reached and all defects are detected. The example shall be non-trivial, leading to at least 3 test cases.

4.3.2 Decision Testing											
FL-4.3.2	(K2) Explain decision coverage	x			x				15		Combine LOs, due to redundancy the FL LO text is not included in the syllabus.
TTA-2.3.1	(K3) Design test cases for a given test object by applying the Decision test technique to achieve a defined level of coverage	x			x	H2			30		
HO-2.3.1	(H2) For a given specification item and a corresponding piece of code, design and implement a test suite with the goal to reach 100% decision coverage, and verify after execution that the test goal has been reached.										Provide a piece of code and the corresponding software component specification. A few decisions or execution paths should contain defects that can be detected by decision coverage but not necessarily by statement coverage. The participants shall design, implement and execute the test cases and verify that 100% decision coverage is reached. If not, they shall add test cases until the test goal is reached and all defects are detected. The example shall be non-trivial, leading to at least 3 test cases. It shall show the advantage compared to statement testing.
4.3.3 Value of statement and decision coverage											
FL-4.3.3	(K2) Explain the value of statement and decision coverage	x			x				15		take over
4.3.4 Modified Condition/Decision Coverage (MC/DC) Testing											
TTA-2.4.1	(K3) Design test cases for a given test object by applying the modified condition/decision test technique to achieve full modified condition/decision coverage (MC/DC)	x			x	H2			60		take over
HO-2.4.1	(H2) For a given specification item and a corresponding piece of code containing multiple atomic conditions in one decision, design, implement and execute a test suite that fullfills 100% MC/DC coverage of the decision.										Provide a piece of code that shows a decision with several independent atomic conditions and the corresponding software development specification. The decision should contain a defect and this defect should be identified by the test cases. The participants shall design, implement and execute the test cases. The trainer shall verify that 100% MC/DC coverage is reached. The decision example shall be non-trivial, with at least 3 atomic conditions. It shall show the advantage compared to Decision Testing. For MC/DC, manual design of the test cases is feasible, but the training may also use a tool to either generate the inputs or verify the coverage.
4.4 Experience-based Test Techniques										15	Reduce core points to 15 minutes. No exam questions
FL-4.4.1	(K2) Explain error guessing										Common characteristics of experience-based test techniques include: Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders. This knowledge and experience includes expected use of the software, its environment, likely defects, and the distribution of those defects.
FL-4.4.2	(K2) Explain exploratory testing										see remarks on FL-4.4.1
FL-4.4.3	(K2) Explain checklist-based testing										see remarks on FL-4.4.1