Certified Tester

Foundation Level Syllabus

Released
Version 2011

International Software Testing Qualifications Board

Revision History

Version	Date	Remarks
ISTQB 2011	Effective 1-Apr-2011	Certified Tester Foundation Level Syllabus Maintenance Release – see Appendix E – Release Notes
ISTQB 2010	Effective 30-Mar-2010	Certified Tester Foundation Level Syllabus Maintenance Release – see Appendix E – Release Notes
ISTQB 2007	01-May-2007	Certified Tester Foundation Level Syllabus Maintenance Release
ISTQB 2005	01-July-2005	Certified Tester Foundation Level Syllabus
ASQF V2.2	July-2003	ASQF Syllabus Foundation Level Version 2.2 “Lehrplan Grundlagen des Software-testens“
ISEB V2.0	25-Feb-1999	ISEB Software Testing Foundation Syllabus V2.0 25 February 1999

Table of Contents

Acknowledgements

International Software Testing Qualifications Board Working Group Foundation Level (Edition 2011): Thomas Müller (chair), Debra Friedenberg. The core team thanks the review team (Dan Almog, Armin Beer, Rex Black, Julie Gardiner, Judy McKay, Tuula Pääkkönen, Eric Riou du Cosquier Hans Schaefer, Stephanie Ulrich, Erik van Veenendaal) and all National Boards for the suggestions for the current version of the syllabus.

International Software Testing Qualifications Board Working Group Foundation Level (Edition 2010): Thomas Müller (chair), Rahul Verma, Martin Klonk and Armin Beer. The core team thanks the review team (Rex Black, Mette Bruhn-Pederson, Debra Friedenberg, Klaus Olsen, Judy McKay, Tuula Pääkkönen, Meile Posthuma, Hans Schaefer, Stephanie Ulrich, Pete Williams, Erik van Veenendaal) and all National Boards for their suggestions.

International Software Testing Qualifications Board Working Group Foundation Level (Edition 2007): Thomas Müller (chair), Dorothy Graham, Debra Friedenberg, and Erik van Veenendaal. The core team thanks the review team (Hans Schaefer, Stephanie Ulrich, Meile Posthuma, Anders Pettersson, and Wonil Kwon) and all the National Boards for their suggestions.

International Software Testing Qualifications Board Working Group Foundation Level (Edition 2005): Thomas Müller (chair), Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhäjärvi, Geoff Thompson and Erik van Veenendaal and the review team and all National Boards for their suggestions.

Introduction to this Syllabus

Purpose of this Document

This syllabus forms the basis for the International Software Testing Qualification at the Foundation Level. The International Software Testing Qualifications Board (ISTQB) provides it to the National Boards for them to accredit the training providers and to derive examination questions in their local language. Training providers will determine appropriate teaching methods and produce courseware for accreditation. The syllabus will help candidates in their preparation for the examination. Information on the history and background of the syllabus can be found in Appendix A.

The Certified Tester Foundation Level in Software Testing

The Foundation Level qualification is aimed at anyone involved in software testing. This includes people in roles such as testers, test analysts, test engineers, test consultants, test managers, user acceptance testers and software developers. This Foundation Level qualification is also appropriate for anyone who wants a basic understanding of software testing, such as project managers, quality managers, software development managers, business analysts, IT directors and management consultants. Holders of the Foundation Certificate will be able to go on to a higher-level software testing qualification.

Learning Objectives/Cognitive Level of Knowledge

Learning objectives are indicated for each section in this syllabus and classified as follows:

- K1: remember
- K2: understand
- K3: apply
- K4: analyze

Further details and examples of learning objectives are given in Appendix B.

All terms listed under “Terms” just below chapter headings shall be remembered (K1), even if not explicitly mentioned in the learning objectives.

The Examination

The Foundation Level Certificate examination will be based on this syllabus. Answers to examination questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable.

The format of the examination is multiple choice.

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

Accreditation

An ISTQB National Board may accredit training providers whose course material follows this syllabus. Training providers should obtain accreditation guidelines from the board or body that performs the accreditation. An accredited course is recognized as conforming to this syllabus, and is allowed to have an ISTQB examination as part of the course.

Further guidance for training providers is given in Appendix D.

Level of Detail

The level of detail in this syllabus allows internationally consistent teaching and examination. In order to achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the Foundation Level
- A list of information to teach, including a description, and references to additional sources if required
- Learning objectives for each knowledge area, describing the cognitive learning outcome and mindset to be achieved
- A list of terms that students must be able to recall and understand
- A description of the key concepts to teach, including sources such as accepted literature or standards

The syllabus content is not a description of the entire knowledge area of software testing; it reflects the level of detail to be covered in Foundation Level training courses.

How this Syllabus is Organized

There are six major chapters. The top-level heading for each chapter shows the highest level of learning objectives that is covered within the chapter and specifies the time for the chapter. For example:

2. Testing Throughout the Software Life Cycle (K2) 115 minutes

This heading shows that Chapter 2 has learning objectives of K1 (assumed when a higher level is shown) and K2 (but not K3), and it is intended to take 115 minutes to teach the material in the chapter. Within each chapter there are a number of sections. Each section also has the learning objectives and the amount of time required. Subsections that do not have a time given are included within the time for the section.

1. Fundamentals of Testing (K2) — 155 minutes

Learning Objectives for Fundamentals of Testing

The objectives identify what you will be able to do following the completion of each module.

1.1 Why is Testing Necessary? (K2)

LO-1.1.1 Describe, with examples, the way in which a defect in software can cause harm to a person, to the environment or to a company (K2)
LO-1.1.2 Distinguish between the root cause of a defect and its effects (K2)
LO-1.1.3 Give reasons why testing is necessary by giving examples (K2)
LO-1.1.4 Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality (K2)
LO-1.1.5 Explain and compare the terms error, defect, fault, failure, and the corresponding terms mistake and bug, using examples (K2)

1.2 What is Testing? (K2)

LO-1.2.1 Recall the common objectives of testing (K1)
LO-1.2.2 Provide examples for the objectives of testing in different phases of the software life cycle (K2)
LO-1.2.3 Differentiate testing from debugging (K2)

1.3 Seven Testing Principles (K2)

LO-1.3.1 Explain the seven principles in testing (K2)

1.4 Fundamental Test Process (K1)

LO-1.4.1 Recall the five fundamental test activities and respective tasks from planning to closure (K1)

1.5 The Psychology of Testing (K2)

LO-1.5.1 Recall the psychological factors that influence the success of testing (K1)
LO-1.5.2 Contrast the mindset of a tester and of a developer (K2)

1.1 Why is Testing Necessary (K2) — *20 minutes*

Terms
Bug, defect, error, failure, fault, mistake, quality, risk

1.1.1 Software Systems Context (K1)

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 could even cause injury or death.

1.1.2 Causes of Software Defects (K2)

A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so.

Defects occur because human beings are fallible and because there is time pressure, complex code, complexity of infrastructure, changing technologies, and/or many system interactions.

Failures can be caused by environmental conditions as well. For example, radiation, magnetism, electronic fields, and pollution can cause faults in firmware or influence the execution of software by changing the hardware conditions.

1.1.3 Role of Testing in Software Development, Maintenance and Operations (K2)

Rigorous testing of systems and documentation can help to reduce the risk of problems occurring during operation and contribute to the quality of the software system, if the defects found are corrected before the system is released for operational use.

Software testing may also be required to meet contractual or legal requirements, or industry-specific standards.

1.1.4 Testing and Quality (K2)

With the help of testing, it is possible to measure the quality of software in terms of defects found, for both functional and non-functional software requirements and characteristics (e.g., reliability, usability, efficiency, maintainability and portability). For more information on non-functional testing see Chapter 2; for more information on software characteristics see 'Software Engineering – Software Product Quality' (ISO 9126).

Testing can give confidence in the quality of the software if it finds few or no defects. A properly designed test that passes reduces the overall level of risk in a system. When testing does find defects, the quality of the software system increases when those defects are fixed.

Lessons should be learned from previous projects. By understanding the root causes of defects found in other projects, processes can be improved, which in turn should prevent those defects from reoccurring and, as a consequence, improve the quality of future systems. This is an aspect of quality assurance.

Testing should be integrated as one of the quality assurance activities (i.e., alongside development standards, training and defect analysis).

1.1.5 How Much Testing is Enough? (K2)

Deciding how much testing is enough should take account of the level of risk, including technical, safety, and business risks, and project constraints such as time and budget. Risk is discussed further in Chapter 5.

Testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system being tested, for the next development step or handover to customers.

1.2 What is Testing? (K2) — *30 minutes*

Terms
Debugging, requirement, review, test case, testing, test objective

Background
A common perception of testing is that it only consists of running tests, i.e., executing the software. This is part of testing, but not all of the testing activities.

Test activities exist before and after test execution. These activities include planning and control, choosing test conditions, designing and executing test cases, checking results, evaluating exit criteria, reporting on the testing process and system under test, and finalizing or completing closure activities after a test phase has been completed. Testing also includes reviewing documents (including source code) and conducting static analysis.

Both dynamic testing and static testing can be used as a means for achieving similar objectives, and will provide information that can be used to improve both the system being tested and the development and testing processes.

Testing can have the following objectives:
- Finding defects
- Gaining confidence about the level of quality
- Providing information for decision-making
- Preventing defects

The thought process and activities involved in designing tests early in the life cycle (verifying the test basis via test design) can help to prevent defects from being introduced into code. Reviews of documents (e.g., requirements) and the identification and resolution of issues also help to prevent defects appearing in the code.

Different viewpoints in testing take different objectives into account. For example, in development testing (e.g., component, integration and system testing), the main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed. In acceptance testing, the main objective may be to confirm that the system works as expected, to gain confidence that it has met the requirements. In some cases the main objective of testing may be to assess the quality of the software (with no intention of fixing defects), to give information to stakeholders of the risk of releasing the system at a given time. Maintenance testing often includes testing that no new defects have been introduced during development of the changes. During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

Debugging and testing are different. Dynamic testing can show failures that are caused by defects. Debugging is the development activity that finds, analyzes and removes the cause of the failure. Subsequent re-testing by a tester ensures that the fix does indeed resolve the failure. The responsibility for these activities is usually testers test and developers debug.

The process of testing and the testing activities are explained in Section 1.4.

1.3 Seven Testing Principles (K2) — *35 minutes*

Terms
Exhaustive testing

Principles
A number of testing principles have been suggested over the past 40 years and offer general guidelines common for all testing.

Principle 1 – Testing shows presence of defects
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, it is not a proof of correctness.

Principle 2 – Exhaustive testing is impossible
Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts.

Principle 3 – Early testing
To find defects early, testing activities shall be started as early as possible in the software or system development life cycle, and shall be focused on defined objectives.

Principle 4 – Defect clustering
Testing effort shall be focused proportionally to the expected and later observed defect density of modules. 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.

Principle 5 – Pesticide paradox
If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new defects. To overcome this “pesticide paradox”, test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to find potentially more defects.

Principle 6 – Testing is context dependent
Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.

Principle 7 – Absence-of-errors fallacy
Finding and fixing defects does not help if the system built is unusable and does not fulfill the users’ needs and expectations.

1.4 Fundamental Test Process (K1) — *35 minutes*

Terms
Confirmation testing, re-testing, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test procedure, test policy, test suite, test summary report, testware

Background
The most visible part of testing is test execution. But to be effective and efficient, test plans should also include time to be spent on planning the tests, designing test cases, preparing for execution and evaluating results.

The fundamental test process consists of the following main activities:
- Test planning and control
- Test analysis and design
- Test implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

Although logically sequential, the activities in the process may overlap or take place concurrently. Tailoring these main activities within the context of the system and the project is usually required.

1.4.1 Test Planning and Control (K1)
Test planning is the activity of defining the objectives of testing and the specification of test activities in order to meet the objectives and mission.

Test control is the ongoing activity of comparing actual progress against the plan, and reporting the status, including deviations from the plan. It involves taking actions necessary to meet the mission and objectives of the project. In order to control testing, the testing activities should be monitored throughout the project. Test planning takes into account the feedback from monitoring and control activities.

Test planning and control tasks are defined in Chapter 5 of this syllabus.

1.4.2 Test Analysis and Design (K1)
Test analysis and design is the activity during which general testing objectives are transformed into tangible test conditions and test cases.

The test analysis and design activity has the following major tasks:
- Reviewing the test basis (such as requirements, software integrity level[1] (risk level), risk analysis reports, architecture, design, interface specifications)
- Evaluating testability of the test basis and test objects
- Identifying and prioritizing test conditions based on analysis of test items, the specification, behavior and structure of the software
- Designing and prioritizing high level test cases
- Identifying necessary test data to support the test conditions and test cases
- Designing the test environment setup and identifying any required infrastructure and tools
- Creating bi-directional traceability between test basis and test cases

[1] The degree to which software complies or must comply with a set of stakeholder-selected software and/or software-based system characteristics (e.g., software complexity, risk assessment, safety level, security level, desired performance, reliability, or cost) which are defined to reflect the importance of the software to its stakeholders.

1.4.3 Test Implementation and Execution (K1)

Test implementation and execution is the activity where test procedures or scripts are specified by combining the test cases in a particular order and including any other information needed for test execution, the environment is set up and the tests are run.

Test implementation and execution has the following major tasks:
- Finalizing, implementing and prioritizing test cases (including the identification of test data)
- Developing and prioritizing test procedures, creating test data and, optionally, preparing test harnesses and writing automated test scripts
- Creating test suites from the test procedures for efficient test execution
- Verifying that the test environment has been set up correctly
- Verifying and updating bi-directional traceability between the test basis and test cases
- Executing test procedures either manually or by using test execution tools, according to the planned sequence
- Logging the outcome of test execution and recording the identities and versions of the software under test, test tools and testware
- Comparing actual results with expected results
- Reporting discrepancies as incidents and analyzing them in order to establish their cause (e.g., a defect in the code, in specified test data, in the test document, or a mistake in the way the test was executed)
- Repeating test activities as a result of action taken for each discrepancy, for example, re-execution of a test that previously failed in order to confirm a fix (confirmation testing), execution of a corrected test and/or execution of tests in order to ensure that defects have not been introduced in unchanged areas of the software or that defect fixing did not uncover other defects (regression testing)

1.4.4 Evaluating Exit Criteria and Reporting (K1)

Evaluating exit criteria is the activity where test execution is assessed against the defined objectives. This should be done for each test level (see Section 2.2).

Evaluating exit criteria has the following major tasks:
- Checking test logs against the exit criteria specified in test planning
- Assessing if more tests are needed or if the exit criteria specified should be changed
- Writing a test summary report for stakeholders

1.4.5 Test Closure Activities (K1)

Test closure activities collect data from completed test activities to consolidate experience, testware, facts and numbers. Test closure activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), a milestone has been achieved, or a maintenance release has been completed.

Test closure activities include the following major tasks:
- Checking which planned deliverables have been delivered
- Closing incident reports or raising change records for any that remain open
- Documenting the acceptance of the system
- Finalizing and archiving testware, the test environment and the test infrastructure for later reuse
- Handing over the testware to the maintenance organization
- Analyzing lessons learned to determine changes needed for future releases and projects
- Using the information gathered to improve test maturity

1.5 The Psychology of Testing (K2) — *25 minutes*

Terms
Error guessing, independence

Background
The mindset to be used while testing and reviewing is different from that used while developing software. With the right mindset developers are able to test their own code, but separation of this responsibility to a tester is typically done to help focus effort and provide additional benefits, such as an independent view by trained and professional testing resources. Independent testing may be carried out at any level of testing.

A certain degree of independence (avoiding the author bias) often makes the tester more effective at finding defects and failures. Independence is not, however, a replacement for familiarity, and developers can efficiently find many defects in their own code. Several levels of independence can be defined as shown here from low to high:

- Tests designed by the person(s) who wrote the software under test (low level of independence)
- Tests designed by another person(s) (e.g., from the development team)
- Tests designed by a person(s) from a different organizational group (e.g., an independent test team) or test specialists (e.g., usability or performance test specialists)
- Tests designed by a person(s) from a different organization or company (i.e., outsourcing or certification by an external body)

People and projects are driven by objectives. People tend to align their plans with the objectives set by management and other stakeholders, for example, to find defects or to confirm that software meets its objectives. Therefore, it is important to clearly state the objectives of testing.

Identifying failures during testing may be perceived as criticism against the product and against the author. As a result, testing is often seen as a destructive activity, even though it is very constructive in the management of product risks. Looking for failures in a system requires curiosity, professional pessimism, a critical eye, attention to detail, good communication with development peers, and experience on which to base error guessing.

If errors, defects or failures are communicated in a constructive way, bad feelings between the testers and the analysts, designers and developers can be avoided. This applies to defects found during reviews as well as in testing.

The tester and test leader need good interpersonal skills to communicate factual information about defects, progress and risks in a constructive way. For the author of the software or document, defect information can help them improve their skills. Defects found and fixed during testing will save time and money later, and reduce risks.

Communication problems may occur, particularly if testers are seen only as messengers of unwanted news about defects. However, there are several ways to improve communication and relationships between testers and others:

- Start with collaboration rather than battles – remind everyone of the common goal of better quality systems
- Communicate findings on the product in a neutral, fact-focused way without criticizing the person who created it, for example, write objective and factual incident reports and review findings
- Try to understand how the other person feels and why they react as they do
- Confirm that the other person has understood what you have said and vice versa

1.6 Code of Ethics — *10 minutes*

Involvement in software testing enables individuals to learn confidential and privileged information. A code of ethics is necessary, among other reasons to ensure that the information is not put to inappropriate use. Recognizing the ACM and IEEE code of ethics for engineers, the ISTQB states the following code of ethics:

PUBLIC - Certified software testers shall act consistently with the public interest

CLIENT AND EMPLOYER - Certified software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest

PRODUCT - Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible

JUDGMENT- Certified software testers shall maintain integrity and independence in their professional judgment

MANAGEMENT - Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing

PROFESSION - Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest

COLLEAGUES - Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers

SELF - Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession

References

1.1.5 Black, 2001, Kaner, 2002
1.2 Beizer, 1990, Black, 2001, Myers, 1979
1.3 Beizer, 1990, Hetzel, 1988, Myers, 1979
1.4 Hetzel, 1988
1.4.5 Black, 2001, Craig, 2002
1.5 Black, 2001, Hetzel, 1988

2. Testing Throughout the Software Life Cycle (K2) — 115 minutes

Learning Objectives for Testing Throughout the Software Life Cycle

The objectives identify what you will be able to do following the completion of each module.

2.1 Software Development Models (K2)

LO-2.1.1 Explain the relationship between development, test activities and work products in the development life cycle, by giving examples using project and product types (K2)
LO-2.1.2 Recognize the fact that software development models must be adapted to the context of project and product characteristics (K1)
LO-2.1.3 Recall characteristics of good testing that are applicable to any life cycle model (K1)

2.2 Test Levels (K2)

LO-2.2.1 Compare the different levels of testing: major objectives, typical objects of testing, typical targets of testing (e.g., functional or structural) and related work products, people who test, types of defects and failures to be identified (K2)

2.3 Test Types (K2)

LO-2.3.1 Compare four software test types (functional, non-functional, structural and change-related) by example (K2)
LO-2.3.2 Recognize that functional and structural tests occur at any test level (K1)
LO-2.3.3 Identify and describe non-functional test types based on non-functional requirements (K2)
LO-2.3.4 Identify and describe test types based on the analysis of a software system's structure or architecture (K2)
LO-2.3.5 Describe the purpose of confirmation testing and regression testing (K2)

2.4 Maintenance Testing (K2)

LO-2.4.1 Compare maintenance testing (testing an existing system) to testing a new application with respect to test types, triggers for testing and amount of testing (K2)
LO-2.4.2 Recognize indicators for maintenance testing (modification, migration and retirement) (K1)
LO-2.4.3. Describe the role of regression testing and impact analysis in maintenance (K2)

2.1 Software Development Models (K2) — *20 minutes*

Terms
Commercial Off-The-Shelf (COTS), iterative-incremental development model, validation, verification, V-model

Background
Testing does not exist in isolation; test activities are related to software development activities. Different development life cycle models need different approaches to testing.

2.1.1 V-model (Sequential Development Model) (K2)

Although variants of the V-model exist, a common type of V-model uses four test levels, corresponding to the four development levels.

The four levels used in this syllabus are:
- Component (unit) testing
- Integration testing
- System testing
- Acceptance testing

In practice, a V-model may have more, fewer or different levels of development and testing, depending on the project and the software product. For example, there may be component integration testing after component testing, and system integration testing after system testing.

Software work products (such as business scenarios or use cases, requirements specifications, design documents and code) produced during development are often the basis of testing in one or more test levels. References for generic work products include Capability Maturity Model Integration (CMMI) or 'Software life cycle processes' (IEEE/IEC 12207). Verification and validation (and early test design) can be carried out during the development of the software work products.

2.1.2 Iterative-incremental Development Models (K2)

Iterative-incremental development is the process of establishing requirements, designing, building and testing a system in a series of short development cycles. Examples are: prototyping, Rapid Application Development (RAD), Rational Unified Process (RUP) and agile development models. A system that is produced using these models may be tested at several test levels during each iteration. An increment, added to others developed previously, forms a growing partial system, which should also be tested. Regression testing is increasingly important on all iterations after the first one. Verification and validation can be carried out on each increment.

2.1.3 Testing within a Life Cycle Model (K2)

In any life cycle model, there are several characteristics of good testing:
- For every development activity there is a corresponding testing activity
- Each test level has test objectives specific to that level
- The analysis and design of tests for a given test level should begin during the corresponding development activity
- Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle

Test levels can be combined or reorganized depending on the nature of the project or the system architecture. For example, for the integration of a Commercial Off-The-Shelf (COTS) software product into a system, the purchaser may perform integration testing at the system level (e.g.,

integration to the infrastructure and other systems, or system deployment) and acceptance testing (functional and/or non-functional, and user and/or operational testing).

2.2 Test Levels (K2) — *40 minutes*

Terms
Alpha testing, beta testing, component testing, driver, field testing, functional requirement, integration, integration testing, non-functional requirement, robustness testing, stub, system testing, test environment, test level, test-driven development, user acceptance testing

Background
For each of the test levels, the following can be identified: the generic objectives, the work product(s) being referenced for deriving test cases (i.e., the test basis), the test object (i.e., what is being tested), typical defects and failures to be found, test harness requirements and tool support, and specific approaches and responsibilities.

Testing a system's configuration data shall be considered during test planning,

2.2.1 Component Testing (K2)
Test basis:
- Component requirements
- Detailed design
- Code

Typical test objects:
- Components
- Programs
- Data conversion / migration programs
- Database modules

Component testing (also known as unit, module or program testing) searches for defects in, and verifies the functioning of, software modules, programs, objects, classes, etc., that are separately testable. It may be done in isolation from the rest of the system, depending on the context of the development life cycle and the system. Stubs, drivers and simulators may be used.

Component testing may include testing of functionality and specific non-functional characteristics, such as resource-behavior (e.g., searching for memory leaks) or robustness testing, as well as structural testing (e.g., decision coverage). Test cases are derived from work products such as a specification of the component, the software design or the data model.

Typically, component testing occurs with access to the code being tested and with the support of a development environment, such as a unit test framework or debugging tool. In practice, component testing usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally managing these defects.

One approach to component testing is to prepare and automate test cases before coding. This is called a test-first approach or test-driven development. This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, and executing the component tests correcting any issues and iterating until they pass.

2.2.2 Integration Testing (K2)

Test basis:
- Software and system design
- Architecture
- Workflows
- Use cases

Typical test objects:
- Subsystems
- Database implementation
- Infrastructure
- Interfaces
- System configuration and configuration data

Integration testing tests interfaces between components, interactions with different parts of a system, such as the operating system, file system and hardware, and interfaces between systems.

There may be more than one level of integration testing and it may be carried out on test objects of varying size as follows:
1. Component integration testing tests the interactions between software components and is done after component testing
2. System integration testing tests the interactions between different systems or between hardware and software and may be done after system testing. In this case, the developing organization may control only one side of the interface. This might be considered as a risk. Business processes implemented as workflows may involve a series of systems. Cross-platform issues may be significant.

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.

Systematic integration strategies may be based on the system architecture (such as top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components. In order to ease fault isolation and detect defects early, integration should normally be incremental rather than “big bang”.

Testing of specific non-functional characteristics (e.g., performance) may be included in integration testing as well as functional testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating module A with module B they are interested in testing the communication between the modules, not the functionality of the individual module as that was done during component testing. Both functional and structural approaches may be used.

Ideally, testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, those components can be built in the order required for most efficient testing.

2.2.3 System Testing (K2)

Test basis:
- o System and software requirement specification
- o Use cases
- o Functional specification
- o Risk analysis reports

Typical test objects:
- o System, user and operation manuals
- o System configuration and configuration data

System testing is concerned with the behavior of a whole system/product. The testing scope shall be clearly addressed in the Master and/or Level Test Plan for that test level.

In system testing, the test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment-specific failures not being found in testing.

System testing may include tests based on risks and/or on requirements specifications, business processes, use cases, or other high level text descriptions or models of system behavior, interactions with the operating system, and system resources.

System testing should investigate functional and non-functional requirements of the system, and data quality characteristics. Testers also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. Structure-based techniques (white-box) may then be used to assess the thoroughness of the testing with respect to a structural element, such as menu structure or web page navigation (see Chapter 4).

An independent test team often carries out system testing.

2.2.4 Acceptance Testing (K2)

Test basis:
- o User requirements
- o System requirements
- o Use cases
- o Business processes
- o Risk analysis reports

Typical test objects:
- o Business processes on fully integrated system
- o Operational and maintenance processes
- o User procedures
- o Forms
- o Reports
- o Configuration data

Acceptance testing is often the responsibility of the customers or users of a system; other stakeholders may be involved as well.

The goal in acceptance testing is to establish confidence in the system, parts of the system or specific non-functional characteristics of the system. Finding defects is not the main focus in acceptance testing. Acceptance testing may assess the system's readiness for deployment and

use, although it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance test for a system.

Acceptance testing may occur at various times in the life cycle, for example:
- A COTS software product may be acceptance tested when it is installed or integrated
- Acceptance testing of the usability of a component may be done during component testing
- Acceptance testing of a new functional enhancement may come before system testing

Typical forms of acceptance testing include the following:

User acceptance testing
Typically verifies the fitness for use of the system by business users.

Operational (acceptance) testing
The acceptance of the system by the system administrators, including:
- Testing of backup/restore
- Disaster recovery
- User management
- Maintenance tasks
- Data load and migration tasks
- Periodic checks of security vulnerabilities

Contract and regulation acceptance testing
Contract 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. Regulation acceptance testing is performed against any regulations that must be adhered to, such as government, legal or safety regulations.

Alpha and beta (or field) testing
Developers of market, or COTS, software often want to get feedback from potential or existing customers in their market before the software product is put up for sale commercially. Alpha testing is performed at the developing organization's site but not by the developing team. Beta testing, or field-testing, is performed by customers or potential customers at their own locations.

Organizations may use other terms as well, such as factory acceptance testing and site acceptance testing for systems that are tested before and after being moved to a customer's site.

2.3 Test Types (K2) — *40 minutes*

Terms
Black-box testing, code coverage, functional testing, interoperability testing, load testing, maintainability testing, performance testing, portability testing, reliability testing, security testing, stress testing, structural testing, usability testing, white-box testing

Background
A group of test activities can be aimed at verifying the software system (or a part of a system) based on a specific reason or target for testing.

A test type is focused on a particular test objective, which could be any of the following:
- A function to be performed by the software
- A non-functional quality characteristic, such as reliability or usability
- The structure or architecture of the software or system
- Change related, i.e., confirming that defects have been fixed (confirmation testing) and looking for unintended changes (regression testing)

A model of the software may be developed and/or used in structural testing (e.g., a control flow model or menu structure model), non-functional testing (e.g., performance model, usability model security threat modeling), and functional testing (e.g., a process flow model, a state transition model or a plain language specification).

2.3.1 Testing of Function (Functional Testing) (K2)
The functions that a system, subsystem or component are to perform may be described in work products such as a requirements specification, use cases, or a functional specification, or they may be undocumented. The functions are “what” the system does.

Functional tests are based on functions and features (described in documents or understood by the testers) and their interoperability with specific systems, and may be performed at all test levels (e.g., tests for components may be based on a component specification).

Specification-based techniques may be used to derive test conditions and test cases from the functionality of the software or system (see Chapter 4). Functional testing considers the external behavior of the software (black-box testing).

A type of functional testing, security testing, investigates the functions (e.g., a firewall) relating to detection of threats, such as viruses, from malicious outsiders. Another type of functional testing, interoperability testing, evaluates the capability of the software product to interact with one or more specified components or systems.

2.3.2 Testing of Non-functional Software Characteristics (Non-functional Testing) (K2)
Non-functional testing includes, but is not limited to, performance testing, load testing, stress testing, usability testing, maintainability testing, reliability testing and portability testing. It is the testing of “how” the system works.

Non-functional testing may be performed at all test levels. The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified on a varying scale, such as response times for performance testing. These tests can be referenced to a quality model such as the one defined in ‘Software Engineering – Software Product Quality’ (ISO

9126). Non-functional testing considers the external behavior of the software and in most cases uses black-box test design techniques to accomplish that.

2.3.3 Testing of Software Structure/Architecture (Structural Testing) (K2)

Structural (white-box) testing may be performed at all test levels. Structural techniques are best used after specification-based techniques, in order to help measure the thoroughness of testing through assessment of coverage of a type of structure.

Coverage is the extent that a structure has been exercised by a test suite, expressed as a percentage of the items being covered. If coverage is not 100%, then more tests may be designed to test those items that were missed to increase coverage. Coverage techniques are covered in Chapter 4.

At all test levels, but especially in component testing and component integration testing, tools can be used to measure the code coverage of elements, such as statements or decisions. Structural testing may be based on the architecture of the system, such as a calling hierarchy.

Structural testing approaches can also be applied at system, system integration or acceptance testing levels (e.g., to business models or menu structures).

2.3.4 Testing Related to Changes: Re-testing and Regression Testing (K2)

After a defect is detected and fixed, the software should be re-tested to confirm that the original defect has been successfully removed. This is called confirmation. Debugging (locating and fixing a defect) is a development activity, not a testing activity.

Regression testing is the repeated testing of an already tested program, after modification, to discover any defects introduced or uncovered as a result of the change(s). These defects may be either in the software being tested, or in another related or unrelated software component. It is performed when the software, or its environment, is changed. The extent of regression testing is based on the risk of not finding defects in software that was working previously.

Tests should be repeatable if they are to be used for confirmation testing and to assist regression testing.

Regression testing may be performed at all test levels, and includes functional, non-functional and structural testing. Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation.

2.4 Maintenance Testing (K2) — *15 minutes*

Terms
Impact analysis, maintenance testing

Background
Once deployed, a software system is often in service for years or decades. During this time the system, its configuration data, or its environment are often corrected, changed or extended. The planning of releases in advance is crucial for successful maintenance testing. A distinction has to be made between planned releases and hot fixes. Maintenance testing is done on an existing operational system, and is triggered by modifications, migration, or retirement of the software or system.

Modifications include planned enhancement changes (e.g., release-based), corrective and emergency changes, and changes of environment, such as planned operating system or database upgrades, planned upgrade of Commercial-Off-The-Shelf software, or patches to correct newly exposed or discovered vulnerabilities of the operating system.

Maintenance testing for migration (e.g., from one platform to another) should include operational tests of the new environment as well as of the changed software. Migration testing (conversion testing) is also needed when data from another application will be migrated into the system being maintained.

Maintenance testing for the retirement of a system may include the testing of data migration or archiving if long data-retention periods are required.

In addition to testing what has been changed, maintenance testing includes regression testing to parts of the system that have not been changed. The scope of maintenance testing is related to the risk of the change, the size of the existing system and to the size of the change. Depending on the changes, maintenance testing may be done at any or all test levels and for any or all test types. Determining how the existing system may be affected by changes is called impact analysis, and is used to help decide how much regression testing to do. The impact analysis may be used to determine the regression test suite.

Maintenance testing can be difficult if specifications are out of date or missing, or testers with domain knowledge are not available.

References
2.1.3 CMMI, Craig, 2002, Hetzel, 1988, IEEE 12207
2.2 Hetzel, 1988
2.2.4 Copeland, 2004, Myers, 1979
2.3.1 Beizer, 1990, Black, 2001, Copeland, 2004
2.3.2 Black, 2001, ISO 9126
2.3.3 Beizer, 1990, Copeland, 2004, Hetzel, 1988
2.3.4 Hetzel, 1988, IEEE STD 829-1998
2.4 Black, 2001, Craig, 2002, Hetzel, 1988, IEEE STD 829-1998

3. Static Techniques (K2) — 60 minutes

Learning Objectives for Static Techniques

The objectives identify what you will be able to do following the completion of each module.

3.1 Static Techniques and the Test Process (K2)

LO-3.1.1 Recognize software work products that can be examined by the different static techniques (K1)
LO-3.1.2 Describe the importance and value of considering static techniques for the assessment of software work products (K2)
LO-3.1.3 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 life cycle (K2)

3.2 Review Process (K2)

LO-3.2.1 Recall the activities, roles and responsibilities of a typical formal review (K1)
LO-3.2.2 Explain the differences between different types of reviews: informal review, technical review, walkthrough and inspection (K2)
LO-3.2.3 Explain the factors for successful performance of reviews (K2)

3.3 Static Analysis by Tools (K2)

LO-3.3.1 Recall typical defects and errors identified by static analysis and compare them to reviews and dynamic testing (K1)
LO-3.3.2 Describe, using examples, the typical benefits of static analysis (K2)
LO-3.3.3 List typical code and design defects that may be identified by static analysis tools (K1)

3.1 Static Techniques and the Test Process (K2)	*15 minutes*

Terms
Dynamic testing, static testing

Background
Unlike dynamic testing, which requires the execution of software, static testing techniques rely on the manual examination (reviews) and automated analysis (static analysis) of the code or other project documentation without the execution of the code.

Reviews are a way of testing software work products (including code) and can be performed well before dynamic test execution. Defects detected during reviews early in the life cycle (e.g., defects found in requirements) are often much cheaper to remove than those detected by running tests on the executing code.

A review could be done entirely as a manual activity, but there is also tool support. The main manual activity is to examine a work product and make comments about it. Any software work product can be reviewed, including requirements specifications, design specifications, code, test plans, test specifications, test cases, test scripts, user guides or web pages.

Benefits of reviews include early defect detection and correction, development productivity improvements, reduced development timescales, reduced testing cost and time, lifetime cost reductions, fewer defects and improved communication. Reviews can find omissions, for example, in requirements, which are unlikely to be found in dynamic testing.

Reviews, static analysis and dynamic testing have the same objective – identifying defects. They are complementary; the different techniques can find different types of defects effectively and efficiently. Compared to dynamic testing, static techniques find causes of failures (defects) rather than the failures themselves.

Typical defects that are easier to find in reviews than in dynamic testing include: deviations from standards, requirement defects, design defects, insufficient maintainability and incorrect interface specifications.

3.2 Review Process (K2) — *25 minutes*

Terms
Entry criteria, formal review, informal review, inspection, metric, moderator, peer review, reviewer, scribe, technical review, walkthrough

Background
The different types of reviews vary from informal, characterized by no written instructions for reviewers, to systematic, 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 maturity of the development process, any legal or regulatory requirements or the need for an audit trail.

The way a review is carried out depends on the agreed objectives of the review (e.g., find defects, gain understanding, educate testers and new team members, or discussion and decision by consensus).

3.2.1 Activities of a Formal Review (K1)
A typical formal review has the following main activities:

1. Planning
   - Defining the review criteria
   - Selecting the personnel
   - Allocating roles
   - Defining the entry and exit criteria for more formal review types (e.g., inspections)
   - Selecting which parts of documents to review
   - Checking entry criteria (for more formal review types)
2. Kick-off
   - Distributing documents
   - Explaining the objectives, process and documents to the participants
3. Individual preparation
   - Preparing for the review meeting by reviewing the document(s)
   - Noting potential defects, questions and comments
4. Examination/evaluation/recording of results (review meeting)
   - Discussing or logging, with documented results or minutes (for more formal review types)
   - Noting defects, making recommendations regarding handling the defects, making decisions about the defects
   - Examining/evaluating and recording issues during any physical meetings or tracking any group electronic communications
5. Rework
   - Fixing defects found (typically done by the author)
   - Recording updated status of defects (in formal reviews)
6. Follow-up
   - Checking that defects have been addressed
   - Gathering metrics
   - Checking on exit criteria (for more formal review types)

3.2.2 Roles and Responsibilities (K1)
A typical formal review will include the roles below:
- Manager: decides on the execution of reviews, allocates time in project schedules and determines if the review objectives have been met.

- Moderator: the person who leads the review of the document or set of documents, including planning the review, running the meeting, and following-up after the meeting. If necessary, the moderator may mediate between the various points of view and is often the person upon whom the success of the review rests.
- Author: the writer or person with chief responsibility for the document(s) to be reviewed.
- Reviewers: individuals with a specific technical or business background (also called checkers or inspectors) who, after the necessary preparation, identify and describe findings (e.g., defects) in the product under review. Reviewers should be chosen to represent different perspectives and roles in the review process, and should take part in any review meetings.
- Scribe (or recorder): documents all the issues, problems and open points that were identified during the meeting.

Looking at software products or related work products from different perspectives and using checklists can make reviews more effective and efficient. For example, a checklist based on various perspectives such as user, maintainer, tester or operations, or a checklist of typical requirements problems may help to uncover previously undetected issues.

3.2.3 Types of Reviews (K2)

A single software product or related work product may be the subject of more than one review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, or an inspection may be carried out on a requirements specification before a walkthrough with customers. The main characteristics, options and purposes of common review types are:

Informal Review
- No formal process
- May take the form of pair programming or a technical lead reviewing designs and code
- Results may be documented
- Varies in usefulness depending on the reviewers
- Main purpose: inexpensive way to get some benefit

Walkthrough
- Meeting led by author
- May take the form of scenarios, dry runs, peer group participation
- Open-ended sessions
  - Optional pre-meeting preparation of reviewers
  - Optional preparation of a review report including list of findings
- Optional scribe (who is not the author)
- May vary in practice from quite informal to very formal
- Main purposes: learning, gaining understanding, finding defects

Technical Review
- Documented, defined defect-detection process that includes peers and technical experts with optional management participation
- May be performed as a peer review without management participation
- Ideally led by trained moderator (not the author)
- Pre-meeting preparation by reviewers
- Optional use of checklists
- Preparation of a review report which includes the list of findings, the verdict whether the software product meets its requirements and, where appropriate, recommendations related to findings
- May vary in practice from quite informal to very formal
- Main purposes: discussing, making decisions, evaluating alternatives, finding defects, solving technical problems and checking conformance to specifications, plans, regulations, and standards

I**nspection**
- Led by trained moderator (not the author)
- Usually conducted as a peer examination
- Defined roles
- Includes metrics gathering
- Formal process based on rules and checklists
- Specified entry and exit criteria for acceptance of the software product
- Pre-meeting preparation
- Inspection report including list of findings
- Formal follow-up process (with optional process improvement components)
- Optional reader
- Main purpose: finding defects

Walkthroughs, technical reviews and inspections can be performed within a peer group, i.e., colleagues at the same organizational level. This type of review is called a “peer review”.

3.2.4 Success Factors for Reviews (K2)

Success factors for reviews include:
- Each review has clear predefined objectives
- The right people for the review objectives are involved
- Testers are valued reviewers who contribute to the review and also learn about the product which enables them to prepare tests earlier
- Defects found are welcomed and expressed objectively
- People issues and psychological aspects are dealt with (e.g., making it a positive experience for the author)
- The review is conducted in an atmosphere of trust; the outcome will not be used for the evaluation of the participants
- Review techniques are applied that are suitable to achieve the objectives and to the type and level of software work products and reviewers
- Checklists or roles are used if appropriate to increase effectiveness of defect identification
- Training is given in review techniques, especially the more formal techniques such as inspection
- Management supports a good review process (e.g., by incorporating adequate time for review activities in project schedules)
- There is an emphasis on learning and process improvement

3.3 Static Analysis by Tools (K2) — *20 minutes*

Terms
Compiler, complexity, control flow, data flow, static analysis

Background
The objective of static analysis is to find defects in software source code and software models. Static analysis is performed without actually executing the software being examined by the tool; dynamic testing does execute the software code. Static analysis can locate defects that are hard to find in dynamic testing. As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g., control flow and data flow), as well as generated output such as HTML and XML.

The value of static analysis is:
- Early detection of defects prior to test execution
- Early warning about suspicious aspects of the code or design by the calculation of metrics, such as a high complexity measure
- Identification of defects not easily found by dynamic testing
- Detecting dependencies and inconsistencies in software models such as links
- Improved maintainability of code and design
- Prevention of defects, if lessons are learned in development

Typical defects discovered by static analysis tools include:
- Referencing a variable with an undefined value
- Inconsistent interfaces between modules and components
- Variables that are not used or are improperly declared
- Unreachable (dead) code
- Missing and erroneous logic (potentially infinite loops)
- Overly complicated constructs
- Programming standards violations
- Security vulnerabilities
- Syntax violations of code and software models

Static analysis tools are typically used by developers (checking against predefined rules or programming standards) before and during component and integration testing or when checking-in code to configuration management tools, and by designers during software modeling. Static analysis tools may produce a large number of warning messages, which need to be well-managed to allow the most effective use of the tool.

Compilers may offer some support for static analysis, including the calculation of metrics.

References
3.2 IEEE 1028
3.2.2 Gilb, 1993, van Veenendaal, 2004
3.2.4 Gilb, 1993, IEEE 1028
3.3 van Veenendaal, 2004

4. Test Design Techniques (K4) — 285 minutes

Learning Objectives for Test Design Techniques

The objectives identify what you will be able to do following the completion of each module.

4.1 The Test Development Process (K3)

LO-4.1.1 Differentiate between a test design specification, test case specification and test procedure specification (K2)
LO-4.1.2 Compare the terms test condition, test case and test procedure (K2)
LO-4.1.3 Evaluate the quality of test cases in terms of clear traceability to the requirements and expected results (K2)
LO-4.1.4 Translate test cases into a well-structured test procedure specification at a level of detail relevant to the knowledge of the testers (K3)

4.2 Categories of Test Design Techniques (K2)

LO-4.2.1 Recall reasons that both specification-based (black-box) and structure-based (white-box) test design techniques are useful and list the common techniques for each (K1)
LO-4.2.2 Explain the characteristics, commonalities, and differences between specification-based testing, structure-based testing and experience-based testing (K2)

4.3 Specification-based or Black-box Techniques (K3)

LO-4.3.1 Write test cases from given software models using equivalence partitioning, boundary value analysis, decision tables and state transition diagrams/tables (K3)
LO-4.3.2 Explain the main purpose of each of the four testing techniques, what level and type of testing could use the technique, and how coverage may be measured (K2)
LO-4.3.3 Explain the concept of use case testing and its benefits (K2)

4.4 Structure-based or White-box Techniques (K4)

LO-4.4.1 Describe the concept and value of code coverage (K2)
LO-4.4.2 Explain the concepts of statement and decision coverage, and give reasons why these concepts can also be used at test levels other than component testing (e.g., on business procedures at system level) (K2)
LO-4.4.3 Write test cases from given control flows using statement and decision test design techniques (K3)
LO-4.4.4 Assess statement and decision coverage for completeness with respect to defined exit criteria. (K4)

4.5 Experience-based Techniques (K2)

LO-4.5.1 Recall reasons for writing test cases based on intuition, experience and knowledge about common defects (K1)
LO-4.5.2 Compare experience-based techniques with specification-based testing techniques (K2)

4.6 Choosing Test Techniques (K2)

LO-4.6.1 Classify test design techniques according to their fitness to a given context, for the test basis, respective models and software characteristics (K2)

4.1 The Test Development Process (K3) — *15 minutes*

Terms
Test case specification, test design, test execution schedule, test procedure specification, test script, traceability

Background
The test development process described in this section can be done in different ways, from very informal with little or no documentation, to very formal (as it is described below). The level of formality depends on the context of the testing, including the maturity of testing and development processes, time constraints, safety or regulatory requirements, and the people involved.

During test analysis, the test basis documentation is analyzed in order to determine what to test, i.e., to identify the test conditions. A test condition is defined as an item or event that could be verified by one or more test cases (e.g., a function, transaction, quality characteristic or structural element).

Establishing traceability from test conditions back to the specifications and requirements enables both effective impact analysis when requirements change, and determining requirements coverage for a set of tests. During test analysis the detailed test approach is implemented to select the test design techniques to use based on, among other considerations, the identified risks (see Chapter 5 for more on risk analysis).

During test design the test cases and test data are created and specified. A test case consists of a set of input values, execution preconditions, expected results and execution postconditions, defined to cover a certain test objective(s) or test condition(s). The ‘Standard for Software Test Documentation’ (IEEE STD 829-1998) describes the content of test design specifications (containing test conditions) and test case specifications.

Expected results should be produced as part of the specification of a test case and include outputs, changes to data and states, and any other consequences of the test. If expected results have not been defined, then a plausible, but erroneous, result may be interpreted as the correct one. Expected results should ideally be defined prior to test execution.

During test implementation the test cases are developed, implemented, prioritized and organized in the test procedure specification (IEEE STD 829-1998). The test procedure specifies the sequence of actions for the execution of a test. If tests are run using a test execution tool, the sequence of actions is specified in a test script (which is an automated test procedure).

The various test procedures and automated test scripts are subsequently formed into a test execution schedule that defines the order in which the various test procedures, and possibly automated test scripts, are executed. The test execution schedule will take into account such factors as regression tests, prioritization, and technical and logical dependencies.

4.2 Categories of Test Design Techniques (K2) — *15 minutes*

Terms
Black-box test design technique, experience-based test design technique, test design technique, white-box test design technique

Background
The purpose of a test design technique is to identify test conditions, test cases, and test data.

It is a classic distinction to denote test techniques as black-box or white-box. Black-box test design techniques (also called specification-based techniques) are a way to derive and select test conditions, test cases, or test data based on an analysis of the test basis documentation. This includes both functional and non-functional testing. Black-box testing, by definition, does not use any information regarding the internal structure of the component or system to be tested. White-box test design techniques (also called structural or structure-based techniques) are based on an analysis of the structure of the component or system. Black-box and white-box testing may also be combined with experience-based techniques to leverage the experience of developers, testers and users to determine what should be tested.

Some techniques fall clearly into a single category; others have elements of more than one category.

This syllabus refers to specification-based test design techniques as black-box techniques and structure-based test design techniques as white-box techniques. In addition experience-based test design techniques are covered.

Common characteristics of specification-based test design techniques include:
- Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components
- Test cases can be derived systematically from these models

Common characteristics of structure-based test design techniques include:
- Information about how the software is constructed is used to derive the test cases (e.g., code and detailed design information)
- The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage

Common characteristics of experience-based test design techniques include:
- The knowledge and experience of people are used to derive the test cases
- The knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment is one source of information
- Knowledge about likely defects and their distribution is another source of information

4.3 Specification-based or Black-box Techniques (K3) — *150 minutes*

Terms
Boundary value analysis, decision table testing, equivalence partitioning, state transition testing, use case testing

4.3.1 Equivalence Partitioning (K3)

In equivalence partitioning, inputs to the software or system are divided into groups that are expected to exhibit similar behavior, so they are likely to be processed in the same way. Equivalence partitions (or classes) can be found for both valid data, i.e., values that should be accepted and invalid data, i.e., values that should be rejected. Partitions can also be identified for 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). Tests can be designed to cover all valid and invalid partitions. Equivalence partitioning is applicable at all levels of testing.

Equivalence partitioning can be used to achieve input and output coverage goals. It can be applied to human input, input via interfaces to a system, or interface parameters in integration testing.

4.3.2 Boundary Value Analysis (K3)

Behavior at the edge of each equivalence partition is more likely to be incorrect than behavior within the partition, so boundaries are an area where testing is likely to yield defects. The maximum and minimum values of a partition are its boundary values. A boundary value for a valid partition is a valid boundary value; the boundary of an invalid partition is an invalid boundary value. Tests can be designed to cover both valid and invalid boundary values. When designing test cases, a test for each boundary value is chosen.

Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect-finding capability is high. Detailed specifications are helpful in determining the interesting boundaries.

This technique is often considered as an extension of equivalence partitioning or other black-box test design techniques. It can be used on equivalence classes for user input on screen as well as, for example, on time ranges (e.g., time out, transactional speed requirements) or table ranges (e.g., table size is 256*256).

4.3.3 Decision Table Testing (K3)

Decision tables are a good way to capture system requirements that contain logical conditions, and to document internal system design. They may be used to record complex business rules that a system is to implement. When creating decision tables, the specification is analyzed, and conditions and actions of the system are identified. The input conditions and actions are most often stated in such a way that they must be true or false (Boolean). The decision table contains the triggering conditions, often combinations of true and false for all input conditions, and the resulting actions for each combination of conditions. Each column of the table corresponds to a business rule that defines a unique combination of conditions and which result in the execution of the actions associated with that rule. The coverage standard commonly used with decision table testing is to have at least one test per column in the table, which typically involves covering all combinations of triggering conditions.

The strength of decision table testing is that it creates combinations of conditions that otherwise might not have been exercised during testing. It may be applied to all situations when the action of the software depends on several logical decisions.

4.3.4 State Transition Testing (K3)

A system may exhibit a different response depending on current conditions or previous history (its state). In this case, that aspect of the system can be shown with a state transition diagram. It allows the tester to view the software in terms of its states, transitions between states, the inputs or events that trigger state changes (transitions) and the actions which may result from those transitions. The states of the system or object under test are separate, identifiable and finite in number.

A state table shows the relationship between the states and inputs, and can highlight possible transitions that are invalid.

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

State transition testing is much used within the embedded software industry and technical automation in general. However, the technique is also suitable for modeling a business object having specific states or testing screen-dialogue flows (e.g., for Internet applications or business scenarios).

4.3.5 Use Case Testing (K2)

Tests can be derived from use cases. A use case describes interactions between actors (users or systems), which produce a result of value to a system user or the customer. Use cases may be described at the abstract level (business use case, technology-free, business process level) or at the system level (system use case on the system functionality level). Each use case has preconditions which need to be met for the use case to work successfully. Each use case terminates with postconditions which are the observable results and final state of the system after the use case has been completed. A use case usually has a mainstream (i.e., most likely) scenario and alternative scenarios.

Use cases describe the “process flows” through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system. Use cases are very useful for designing acceptance tests with customer/user participation. They also help uncover integration defects caused by the interaction and interference of different components, which individual component testing would not see. Designing test cases from use cases may be combined with other specification-based test techniques.

4.4 Structure-based or White-box Techniques (K4) — *60 minutes*

Terms
Code coverage, decision coverage, statement coverage, structure-based testing

Background
Structure-based or white-box testing is based on an identified structure of the software or the system, as seen in the following examples:
- Component level: the structure of a software component, i.e., statements, decisions, branches or even distinct paths
- Integration level: the structure may be a call tree (a diagram in which modules call other modules)
- System level: the structure may be a menu structure, business process or web page structure

In this section, three code-related structural test design techniques for code coverage, based on statements, branches and decisions, are discussed. For decision testing, a control flow diagram may be used to visualize the alternatives for each decision.

4.4.1 Statement Testing and Coverage (K4)

In component testing, statement coverage is the assessment of the percentage of executable statements that have been exercised by a test case suite. The statement testing technique derives test cases to execute specific statements, normally to increase statement coverage.

Statement coverage is determined by the number of executable statements covered by (designed or executed) test cases divided by the number of all executable statements in the code under test.

4.4.2 Decision Testing and Coverage (K4)

Decision coverage, related to branch testing, is the assessment of the percentage of decision outcomes (e.g., the True and False options of an IF statement) that have been exercised by a test case suite. The decision testing technique derives test cases to execute specific decision outcomes. Branches originate from decision points in the code and show the transfer of control to different locations in the code.

Decision coverage is determined by the number of all decision outcomes covered by (designed or executed) test cases divided by the number of all possible decision outcomes in the code under test.

Decision testing is a form of control flow testing as it follows a specific flow of control through the decision points. Decision coverage is stronger than statement coverage; 100% decision coverage guarantees 100% statement coverage, but not vice versa.

4.4.3 Other Structure-based Techniques (K1)

There are stronger levels of structural coverage beyond decision coverage, for example, condition coverage and multiple condition coverage.

The concept of coverage can also be applied at other test levels For example, at the integration level the percentage of modules, components or classes that have been exercised by a test case suite could be expressed as module, component or class coverage.

Tool support is useful for the structural testing of code.

4.5 Experience-based Techniques (K2) — *30 minutes*

Terms
Exploratory testing, (fault) attack

Background
Experience-based testing is where tests are derived from the tester's skill and intuition and their experience with similar applications and technologies. When used to augment systematic techniques, these techniques can be useful in identifying special tests not easily captured by formal techniques, especially when applied after more formal approaches. However, this technique may yield widely varying degrees of effectiveness, depending on the testers' experience.

A commonly used experience-based technique is error guessing. Generally testers anticipate defects based on experience. A structured approach to the error guessing technique is to enumerate a list of possible defects and to design tests that attack these defects. This systematic approach is called fault attack. These defect and failure lists can be built based on experience, available defect and failure data, and from common knowledge about why software fails.

Exploratory testing is concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, and carried out within time-boxes. It is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing. It can serve as a check on the test process, to help ensure that the most serious defects are found.

4.6 Choosing Test Techniques (K2) — *15 minutes*

Terms
No specific terms.

Background
The choice of which test techniques to use depends on a number of factors, including the type of system, regulatory standards, customer or contractual requirements, level of risk, type of risk, test objective, documentation available, knowledge of the testers, time and budget, development life cycle, use case models and previous experience with types of defects found.

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 including process, rule and data-driven techniques to ensure adequate coverage of the object under test.

References
4.1 Craig, 2002, Hetzel, 1988, IEEE STD 829-1998
4.2 Beizer, 1990, Copeland, 2004
4.3.1 Copeland, 2004, Myers, 1979
4.3.2 Copeland, 2004, Myers, 1979
4.3.3 Beizer, 1990, Copeland, 2004
4.3.4 Beizer, 1990, Copeland, 2004
4.3.5 Copeland, 2004
4.4.3 Beizer, 1990, Copeland, 2004
4.5 Kaner, 2002
4.6 Beizer, 1990, Copeland, 2004

5. Test Management (K3) — 170 minutes

Learning Objectives for Test Management

The objectives identify what you will be able to do following the completion of each module.

5.1 Test Organization (K2)

LO-5.1.1 Recognize the importance of independent testing (K1)
LO-5.1.2 Explain the benefits and drawbacks of independent testing within an organization (K2)
LO-5.1.3 Recognize the different team members to be considered for the creation of a test team (K1)
LO-5.1.4 Recall the tasks of a typical test leader and tester (K1)

5.2 Test Planning and Estimation (K3)

LO-5.2.1 Recognize the different levels and objectives of test planning (K1)
LO-5.2.2 Summarize the purpose and content of the test plan, test design specification and test procedure documents according to the 'Standard for Software Test Documentation' (IEEE Std 829-1998) (K2)
LO-5.2.3 Differentiate between conceptually different test approaches, such as analytical, model-based, methodical, process/standard compliant, dynamic/heuristic, consultative and regression-averse (K2)
LO-5.2.4 Differentiate between the subject of test planning for a system and scheduling test execution (K2)
LO-5.2.5 Write a test execution schedule for a given set of test cases, considering prioritization, and technical and logical dependencies (K3)
LO-5.2.6 List test preparation and execution activities that should be considered during test planning (K1)
LO-5.2.7 Recall typical factors that influence the effort related to testing (K1)
LO-5.2.8 Differentiate between two conceptually different estimation approaches: the metrics-based approach and the expert-based approach (K2)
LO-5.2.9 Recognize/justify adequate entry and exit criteria for specific test levels and groups of test cases (e.g., for integration testing, acceptance testing or test cases for usability testing) (K2)

5.3 Test Progress Monitoring and Control (K2)

LO-5.3.1 Recall common metrics used for monitoring test preparation and execution (K1)
LO-5.3.2 Explain and compare test metrics for test reporting and test control (e.g., defects found and fixed, and tests passed and failed) related to purpose and use (K2)
LO-5.3.3 Summarize the purpose and content of the test summary report document according to the 'Standard for Software Test Documentation' (IEEE Std 829-1998) (K2)

5.4 Configuration Management (K2)

LO-5.4.1 Summarize how configuration management supports testing (K2)

5.5 Risk and Testing (K2)

LO-5.5.1 Describe a risk as a possible problem that would threaten the achievement of one or more stakeholders' project objectives (K2)
LO-5.5.2 Remember that the level of risk is determined by likelihood (of happening) and impact (harm resulting if it does happen) (K1)
LO-5.5.3 Distinguish between the project and product risks (K2)
LO-5.5.4 Recognize typical product and project risks (K1)
LO-5.5.5 Describe, using examples, how risk analysis and risk management may be used for test planning (K2)

5.6 Incident Management (K3)

LO-5.6.1 Recognize the content of an incident report according to the 'Standard for Software Test Documentation' (IEEE Std 829-1998) (K1)

LO-5.6.2 Write an incident report covering the observation of a failure during testing. (K3)

5.1 Test Organization (K2) — *30 minutes*

Terms
Tester, test leader, test manager

5.1.1 Test Organization and Independence (K2)

The effectiveness of finding defects by testing and reviews can be improved by using independent testers. Options for independence include the following:
- No independent testers; developers test their own code
- Independent testers within the development teams
- Independent test team or group within the organization, reporting to project management or executive management
- Independent testers from the business organization or user community
- Independent test specialists for specific test types such as usability testers, security testers or certification testers (who certify a software product against standards and regulations)
- Independent testers outsourced or external to the organization

For large, complex or safety critical projects, it is usually best to have multiple levels of testing, with some or all of the levels done by independent testers. Development staff may participate in testing, especially at the lower levels, but their lack of objectivity often limits their effectiveness. The independent testers may have the authority to require and define test processes and rules, but testers should take on such process-related roles only in the presence of a clear management mandate to do so.

The benefits of independence include:
- Independent testers see other and different defects, and are unbiased
- An independent tester can verify assumptions people made during specification and implementation of the system

Drawbacks include:
- Isolation from the development team (if treated as totally independent)
- Developers may lose a sense of responsibility for quality
- Independent testers may be seen as a bottleneck or blamed for delays in release

Testing tasks may be done by people in a specific testing role, or may be done by someone in another role, such as a project manager, quality manager, developer, business and domain expert, infrastructure or IT operations.

5.1.2 Tasks of the Test Leader and Tester (K1)

In this syllabus two test positions are covered, test leader and tester. The activities and tasks performed by people in these two roles depend on the project and product context, the people in the roles, and the organization.

Sometimes the test leader is called a test manager or test coordinator. The role of the test leader may be performed by a project manager, a development manager, a quality assurance manager or the manager of a test group. In larger projects two positions may exist: test leader and test manager. Typically the test leader plans, monitors and controls the testing activities and tasks as defined in Section 1.4.

Typical test leader tasks may include:
- Coordinate the test strategy and plan with project managers and others
- Write or review a test strategy for the project, and test policy for the organization

- Contribute the testing perspective to other project activities, such as integration planning
- Plan the tests – considering the context and understanding the test objectives and risks – including selecting test approaches, estimating the time, effort and cost of testing, acquiring resources, defining test levels, cycles, and planning incident management
- Initiate the specification, preparation, implementation and execution of tests, monitor the test results and check the exit criteria
- Adapt planning based on test results and progress (sometimes documented in status reports) and take any action necessary to compensate for problems
- Set up adequate configuration management of testware for traceability
- Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
- Decide what should be automated, to what degree, and how
- Select tools to support testing and organize any training in tool use for testers
- Decide about the implementation of the test environment
- Write test summary reports based on the information gathered during testing

Typical tester tasks may include:

- Review and contribute to test plans
- Analyze, review and assess user requirements, specifications and models for testability
- Create test specifications
- Set up the test environment (often coordinating with system administration and network management)
- Prepare and acquire test data
- Implement tests on all test levels, execute and log the tests, evaluate the results and document the deviations from expected results
- Use test administration or management tools and test monitoring tools as required
- Automate tests (may be supported by a developer or a test automation expert)
- Measure performance of components and systems (if applicable)
- Review tests developed by others

People who work on test analysis, test design, specific test types or test automation may be specialists in these roles. Depending on the test level and the risks related to the product and the project, different people may take over the role of tester, keeping some degree of independence. Typically testers at the component and integration level would be developers, testers at the acceptance test level would be business experts and users, and testers for operational acceptance testing would be operators.

5.2 Test Planning and Estimation (K3) — *40 minutes*

Terms
Test approach, test strategy

5.2.1 Test Planning (K2)

This section covers the purpose of test planning within development and implementation projects, and for maintenance activities. Planning may be documented in a master test plan and in separate test plans for test levels such as system testing and acceptance testing. The outline of a test-planning document is covered by the 'Standard for Software Test Documentation' (IEEE Std 829-1998).

Planning is influenced by the test policy of the organization, the scope of testing, objectives, risks, constraints, criticality, testability and the availability of resources. As the project and test planning progress, more information becomes available and more detail can be included in the plan.

Test planning is a continuous activity and is performed in all life cycle processes and activities. Feedback from test activities is used to recognize changing risks so that planning can be adjusted.

5.2.2 Test Planning Activities (K3)

Test planning activities for an entire system or part of a system may include:
- Determining the scope and risks and identifying the objectives of testing
- Defining the overall approach of testing, including the definition of the test levels and entry and exit criteria
- Integrating and coordinating the testing activities into the software life cycle activities (acquisition, supply, development, operation and maintenance)
- Making decisions about what to test, what roles will perform the test activities, how the test activities should be done, and how the test results will be evaluated
- Scheduling test analysis and design activities
- Scheduling test implementation, execution and evaluation
- Assigning resources for the different activities defined
- Defining the amount, level of detail, structure and templates for the test documentation
- Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues
- Setting the level of detail for test procedures in order to provide enough information to support reproducible test preparation and execution

5.2.3 Entry Criteria (K2)

Entry criteria define when to start testing such as at the beginning of a test level or when a set of tests is ready for execution.

Typically entry criteria may cover the following:
- Test environment availability and readiness
- Test tool readiness in the test environment
- Testable code availability
- Test data availability

5.2.4 Exit Criteria (K2)

Exit criteria define when to stop testing such as at the end of a test level or when a set of tests has achieved specific goal.

Typically exit criteria may cover the following:
- Thoroughness measures, such as coverage of code, functionality or risk
- Estimates of defect density or reliability measures
- Cost
- Residual risks, such as defects not fixed or lack of test coverage in certain areas
- Schedules such as those based on time to market

5.2.5 Test Estimation (K2)

Two approaches for the estimation of test effort are:
- The metrics-based approach: estimating the testing effort based on metrics of former or similar projects or based on typical values
- The expert-based approach: estimating the tasks based on estimates made by the owner of the tasks or by experts

Once the test effort is estimated, resources can be identified and a schedule can be drawn up.

The testing effort may depend on a number of factors, including:
- Characteristics of the product: the quality of the specification and other information used for test models (i.e., the test basis), the size of the product, the complexity of the problem domain, the requirements for reliability and security, and the requirements for documentation
- Characteristics of the development process: the stability of the organization, tools used, test process, skills of the people involved, and time pressure
- The outcome of testing: the number of defects and the amount of rework required

5.2.6 Test Strategy, Test Approach (K2)

The test approach is the implementation of the test strategy for a specific project. The test approach is defined and refined in the test plans and test designs. It typically includes the decisions made based on the (test) project's goal and risk assessment. It is the starting point for planning the test process, for selecting the test design techniques and test types to be applied, and for defining the entry and exit criteria.

The selected approach depends on the context and may consider risks, hazards and safety, available resources and skills, the technology, the nature of the system (e.g., custom built vs. COTS), test objectives, and regulations.

Typical approaches include:
- Analytical approaches, such as risk-based testing where testing is directed to areas of greatest risk
- Model-based approaches, such as stochastic testing using statistical information about failure rates (such as reliability growth models) or usage (such as operational profiles)
- Methodical approaches, such as failure-based (including error guessing and fault attacks), experience-based, checklist-based, and quality characteristic-based
- Process- or standard-compliant approaches, such as those specified by industry-specific standards or the various agile methodologies
- Dynamic and heuristic approaches, such as exploratory testing where testing is more reactive to events than pre-planned, and where execution and evaluation are concurrent tasks
- Consultative approaches, such as those in which test coverage is driven primarily by the advice and guidance of technology and/or business domain experts outside the test team
- Regression-averse approaches, such as those that include reuse of existing test material, extensive automation of functional regression tests, and standard test suites

Different approaches may be combined, for example, a risk-based dynamic approach.

5.3 Test Progress Monitoring and Control (K2)	*20 minutes*

Terms
Defect density, failure rate, test control, test monitoring, test summary report

5.3.1 Test Progress Monitoring (K1)

The purpose of test monitoring is to provide feedback and visibility about test activities. Information to be monitored may be collected manually or automatically and may be used to measure exit criteria, such as coverage. Metrics may also be used to assess progress against the planned schedule and budget. Common test metrics include:
- Percentage of work done in test case preparation (or percentage of planned test cases prepared)
- Percentage of work done in test environment preparation
- Test case execution (e.g., number of test cases run/not run, and test cases passed/failed)
- Defect information (e.g., defect density, defects found and fixed, failure rate, and re-test results)
- Test coverage of requirements, risks or code
- Subjective confidence of testers in the product
- Dates of test milestones
- Testing costs, including the cost compared to the benefit of finding the next defect or to run the next test

5.3.2 Test Reporting (K2)

Test reporting is concerned with summarizing information about the testing endeavor, including:
- What happened during a period of testing, such as dates when exit criteria were met
- Analyzed information and metrics to support recommendations and decisions about future actions, such as an assessment of defects remaining, the economic benefit of continued testing, outstanding risks, and the level of confidence in the tested software

The outline of a test summary report is given in ‘Standard for Software Test Documentation’ (IEEE Std 829-1998).

Metrics should be collected during and at the end of a test level in order to assess:
- The adequacy of the test objectives for that test level
- The adequacy of the test approaches taken
- The effectiveness of the testing with respect to the objectives

5.3.3 Test Control (K2)

Test control describes any guiding or corrective actions taken as a result of information and metrics gathered and reported. Actions may cover any test activity and may affect any other software life cycle activity or task.

Examples of test control actions include:
- Making decisions based on information from test monitoring
- Re-prioritizing tests when an identified risk occurs (e.g., software delivered late)
- Changing the test schedule due to availability or unavailability of a test environment
- Setting an entry criterion requiring fixes to have been re-tested (confirmation tested) by a developer before accepting them into a build

5.4 Configuration Management (K2) — *10 minutes*

Terms
Configuration management, version control

Background
The purpose of configuration management is to establish and maintain the integrity of the products (components, data and documentation) of the software or system through the project and product life cycle.

For testing, configuration management may involve ensuring the following:
- All items of testware are identified, version controlled, tracked for changes, related to each other and related to development items (test objects) so that traceability can be maintained throughout the test process
- All identified documents and software items are referenced unambiguously in test documentation

For the tester, configuration management helps to uniquely identify (and to reproduce) the tested item, test documents, the tests and the test harness(es).

During test planning, the configuration management procedures and infrastructure (tools) should be chosen, documented and implemented.

5.5 Risk and Testing (K2) — *30 minutes*

Terms
Product risk, project risk, risk, risk-based testing

Background
Risk can be defined as the chance of an event, hazard, threat or situation occurring and resulting in undesirable consequences or a potential problem. The level of risk will be determined by the likelihood of an adverse event happening and the impact (the harm resulting from that event).

5.5.1 Project Risks (K2)
Project risks are the risks that surround the project's capability to deliver its objectives, such as:
- Organizational factors:
  - Skill, training and staff shortages
  - Personnel issues
  - Political issues, such as:
    - Problems with testers communicating their needs and test results
    - Failure by the team to follow up on information found in testing and reviews (e.g., not improving development and testing practices)
  - Improper attitude toward or expectations of testing (e.g., not appreciating the value of finding defects during testing)
- Technical issues:
  - Problems in defining the right requirements
  - The extent to which requirements cannot be met given existing constraints
  - Test environment not ready on time
  - Late data conversion, migration planning and development and testing data conversion/migration tools
  - Low quality of the design, code, configuration data, test data and tests
- Supplier issues:
  - Failure of a third party
  - Contractual issues

When analyzing, managing and mitigating these risks, the test manager is following well-established project management principles. The 'Standard for Software Test Documentation' (IEEE Std 829-1998) outline for test plans requires risks and contingencies to be stated.

5.5.2 Product Risks (K2)
Potential failure areas (adverse future events or hazards) in the software or system are known as product risks, as they are a risk to the quality of the product. These include:
- Failure-prone software delivered
- The potential that the software/hardware could cause harm to an individual or company
- Poor software characteristics (e.g., functionality, reliability, usability and performance)
- Poor data integrity and quality (e.g., data migration issues, data conversion problems, data transport problems, violation of data standards)
- Software that does not perform its intended functions

Risks are used to decide where to start testing and where to test more; testing is used to reduce the risk of an adverse effect occurring, or to reduce the impact of an adverse effect.

Product risks are a special type of risk to the success of a project. Testing as a risk-control activity provides feedback about the residual risk by measuring the effectiveness of critical defect removal and of contingency plans.

A risk-based approach to testing provides proactive opportunities to reduce the levels of product risk, starting in the initial stages of a project. It involves the identification of product risks and their use in guiding test planning and control, specification, preparation and execution of tests. In a risk-based approach the risks identified may be used to:

- Determine the test techniques to be employed
- Determine the extent of testing to be carried out
- Prioritize testing in an attempt to find the critical defects as early as possible
- Determine whether any non-testing activities could be employed to reduce risk (e.g., providing training to inexperienced designers)

Risk-based testing draws on the collective knowledge and insight of the project stakeholders to determine the risks and the levels of testing required to address those risks.

To ensure that the chance of a product failure is minimized, risk management activities provide a disciplined approach to:

- Assess (and reassess on a regular basis) what can go wrong (risks)
- Determine what risks are important to deal with
- Implement actions to deal with those risks

In addition, testing may support the identification of new risks, may help to determine what risks should be reduced, and may lower uncertainty about risks.

5.6 Incident Management (K3) — *40 minutes*

Terms
Incident logging, incident management, incident report

Background
Since one of the objectives of testing is to find defects, the discrepancies between actual and expected outcomes need to be logged as incidents. An incident must be investigated and may turn out to be a defect. Appropriate actions to dispose incidents and defects should be defined. Incidents and defects should be tracked from discovery and classification to correction and confirmation of the solution. In order to manage all incidents to completion, an organization should establish an incident management process and rules for classification.

Incidents may be raised during development, review, testing or use of a software product. They may be raised for issues in code or the working system, or in any type of documentation including requirements, development documents, test documents, and user information such as “Help” or installation guides.

Incident reports have the following objectives:
- Provide developers and other parties with feedback about the problem to enable identification, isolation and correction as necessary
- Provide test leaders a means of tracking the quality of the system under test and the progress of the testing
- Provide ideas for test process improvement

Details of the incident report may include:
- Date of issue, issuing organization, and author
- Expected and actual results
- Identification of the test item (configuration item) and environment
- Software or system life cycle process in which the incident was observed
- Description of the incident to enable reproduction and resolution, including logs, database dumps or screenshots
- Scope or degree of impact on stakeholder(s) interests
- Severity of the impact on the system
- Urgency/priority to fix
- Status of the incident (e.g., open, deferred, duplicate, waiting to be fixed, fixed awaiting re-test, closed)
- Conclusions, recommendations and approvals
- Global issues, such as other areas that may be affected by a change resulting from the incident
- Change history, such as the sequence of actions taken by project team members with respect to the incident to isolate, repair, and confirm it as fixed
- References, including the identity of the test case specification that revealed the problem

The structure of an incident report is also covered in the ‘Standard for Software Test Documentation’ (IEEE Std 829-1998).

References

5.1.1 Black, 2001, Hetzel, 1988
5.1.2 Black, 2001, Hetzel, 1988
5.2.5 Black, 2001, Craig, 2002, IEEE Std 829-1998, Kaner 2002
5.3.3 Black, 2001, Craig, 2002, Hetzel, 1988, IEEE Std 829-1998
5.4 Craig, 2002
5.5.2 Black, 2001 , IEEE Std 829-1998
5.6 Black, 2001, IEEE Std 829-1998

6. Tool Support for Testing (K2) — 80 minutes

Learning Objectives for Tool Support for Testing

The objectives identify what you will be able to do following the completion of each module.

6.1 Types of Test Tools (K2)

LO-6.1.1 Classify different types of test tools according to their purpose and to the activities of the fundamental test process and the software life cycle (K2)

LO-6.1.3 Explain the term test tool and the purpose of tool support for testing (K2) [2]

6.2 Effective Use of Tools: Potential Benefits and Risks (K2)

LO-6.2.1 Summarize the potential benefits and risks of test automation and tool support for testing (K2)

LO-6.2.2 Remember special considerations for test execution tools, static analysis, and test management tools (K1)

6.3 Introducing a Tool into an Organization (K1)

LO-6.3.1 State the main principles of introducing a tool into an organization (K1)

LO-6.3.2 State the goals of a proof-of-concept for tool evaluation and a piloting phase for tool implementation (K1)

LO-6.3.3 Recognize that factors other than simply acquiring a tool are required for good tool support (K1)

[2] LO-6.1.2 Intentionally skipped

6.1 Types of Test Tools (K2) — *45 minutes*

Terms
Configuration management tool, coverage tool, debugging tool, dynamic analysis tool, incident management tool, load testing tool, modeling tool, monitoring tool, performance testing tool, probe effect, requirements management tool, review tool, security tool, static analysis tool, stress testing tool, test comparator, test data preparation tool, test design tool, test harness, test execution tool, test management tool, unit test framework tool

6.1.1 Tool Support for Testing (K2)

Test tools can be used for one or more activities that support testing. These include:
1. Tools that are directly used in testing such as test execution tools, test data generation tools and result comparison tools
2. Tools that help in managing the testing process such as those used to manage tests, test results, data, requirements, incidents, defects, etc., and for reporting and monitoring test execution
3. Tools that are used in reconnaissance, or, in simple terms: exploration (e.g., tools that monitor file activity for an application)
4. Any tool that aids in testing (a spreadsheet is also a test tool in this meaning)

Tool support for testing can have one or more of the following purposes depending on the context:
- Improve the efficiency of test activities by automating repetitive tasks or supporting manual test activities like test planning, test design, test reporting and monitoring
- Automate activities that require significant resources when done manually (e.g., static testing)
- Automate activities that cannot be executed manually (e.g., large scale performance testing of client-server applications)
- Increase reliability of testing (e.g., by automating large data comparisons or simulating behavior)

The term “test frameworks” is also frequently used in the industry, in at least three meanings:
- Reusable and extensible testing libraries that can be used to build testing tools (called test harnesses as well)
- A type of design of test automation (e.g., data-driven, keyword-driven)
- Overall process of execution of testing

For the purpose of this syllabus, the term “test frameworks” is used in its first two meanings as described in Section 6.1.6.

6.1.2 Test Tool Classification (K2)

There are a number of tools that support different aspects of testing. Tools can be classified based on several criteria such as purpose, commercial / free / open-source / shareware, technology used and so forth. Tools are classified in this syllabus according to the testing activities that they support.

Some tools clearly support one activity; others may support more than one activity, but are classified under the activity with which they are most closely associated. Tools from a single provider, especially those that have been designed to work together, may be bundled into one package.

Some types of test tools can be intrusive, which means that they can affect the actual outcome of the test. For example, the actual timing may be different due to the extra instructions that are executed by the tool, or you may get a different measure of code coverage. The consequence of intrusive tools is called the probe effect.

Some tools offer support more appropriate for developers (e.g., tools that are used during component and component integration testing). Such tools are marked with "(D)" in the list below.

6.1.3 Tool Support for Management of Testing and Tests (K1)

Management tools apply to all test activities over the entire software life cycle.

Test Management Tools
These tools provide interfaces for executing tests, tracking defects and managing requirements, along with support for quantitative analysis and reporting of the test objects. They also support tracing the test objects to requirement specifications and might have an independent version control capability or an interface to an external one.

Requirements Management Tools
These tools store requirement statements, store the attributes for the requirements (including priority), provide unique identifiers and support tracing the requirements to individual tests. These tools may also help with identifying inconsistent or missing requirements.

Incident Management Tools (Defect Tracking Tools)
These tools store and manage incident reports, i.e., defects, failures, change requests or perceived problems and anomalies, and help in managing the life cycle of incidents, optionally with support for statistical analysis.

Configuration Management Tools
Although not strictly test tools, these are necessary for storage and version management of testware and related software especially when configuring more than one hardware/software environment in terms of operating system versions, compilers, browsers, etc.

6.1.4 Tool Support for Static Testing (K1)

Static testing tools provide a cost effective way of finding more defects at an earlier stage in the development process.

Review Tools
These tools assist with review processes, checklists, review guidelines and are used to store and communicate review comments and report on defects and effort. They can be of further help by providing aid for online reviews for large or geographically dispersed teams.

Static Analysis Tools (D)
These tools help developers and testers find defects prior to dynamic testing by providing support for enforcing coding standards (including secure coding), analysis of structures and dependencies. They can also help in planning or risk analysis by providing metrics for the code (e.g., complexity).

Modeling Tools (D)
These tools are used to validate software models (e.g., physical data model (PDM) for a relational database), by enumerating inconsistencies and finding defects. These tools can often aid in generating some test cases based on the model.

6.1.5 Tool Support for Test Specification (K1)

Test Design Tools
These tools are used to generate test inputs or executable tests and/or test oracles from requirements, graphical user interfaces, design models (state, data or object) or code.

Test Data Preparation Tools
Test data preparation tools manipulate databases, files or data transmissions to set up test data to be used during the execution of tests to ensure security through data anonymity.

6.1.6 Tool Support for Test Execution and Logging (K1)

Test Execution Tools
These tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language and usually provide a test log for each test run. They can also be used to record tests, and usually support scripting languages or GUI-based configuration for parameterization of data and other customization in the tests.

Test Harness/Unit Test Framework Tools (D)
A unit test harness or framework facilitates the testing of components or parts of a system by simulating the environment in which that test object will run, through the provision of mock objects as stubs or drivers.

Test Comparators
Test comparators determine differences between files, databases or test results. Test execution tools typically include dynamic comparators, but post-execution comparison may be done by a separate comparison tool. A test comparator may use a test oracle, especially if it is automated.

Coverage Measurement Tools (D)
These tools, through intrusive or non-intrusive means, measure the percentage of specific types of code structures that have been exercised (e.g., statements, branches or decisions, and module or function calls) by a set of tests.

Security Testing Tools
These tools are used to evaluate the security characteristics of software. This includes evaluating the ability of the software to protect data confidentiality, integrity, authentication, authorization, availability, and non-repudiation. Security tools are mostly focused on a particular technology, platform, and purpose.

6.1.7 Tool Support for Performance and Monitoring (K1)

Dynamic Analysis Tools (D)
Dynamic analysis tools find defects that are evident only when software is executing, such as time dependencies or memory leaks. They are typically used in component and component integration testing, and when testing middleware.

Performance Testing/Load Testing/Stress Testing Tools
Performance testing tools monitor and report on how a system behaves under a variety of simulated usage conditions in terms of number of concurrent users, their ramp-up pattern, frequency and relative percentage of transactions. The simulation of load is achieved by means of creating virtual users carrying out a selected set of transactions, spread across various test machines commonly known as load generators.

Monitoring Tools
Monitoring tools continuously analyze, verify and report on usage of specific system resources, and give warnings of possible service problems.

6.1.8 Tool Support for Specific Testing Needs (K1)

Data Quality Assessment
Data is at the center of some projects such as data conversion/migration projects and applications like data warehouses and its attributes can vary in terms of criticality and volume. In such contexts, tools need to be employed for data quality assessment to review and verify the data conversion and

migration rules to ensure that the processed data is correct, complete and complies with a pre-defined context-specific standard.

Other testing tools exist for usability testing.

6.2 Effective Use of Tools: Potential Benefits and Risks (K2) — *20 minutes*

Terms
Data-driven testing, keyword-driven testing, scripting language

6.2.1 Potential Benefits and Risks of Tool Support for Testing (for all tools) (K2)

Simply purchasing or leasing a tool does not guarantee success with that tool. Each type of tool may require additional effort to achieve real and lasting benefits. There are potential benefits and opportunities with the use of tools in testing, but there are also risks.

Potential benefits of using tools include:
- Repetitive work is reduced (e.g., running regression tests, re-entering the same test data, and checking against coding standards)
- Greater consistency and repeatability (e.g., tests executed by a tool in the same order with the same frequency, and tests derived from requirements)
- Objective assessment (e.g., static measures, coverage)
- Ease of access to information about tests or testing (e.g., statistics and graphs about test progress, incident rates and performance)

Risks of using tools include:
- Unrealistic expectations for the tool (including functionality and ease of use)
- Underestimating the time, cost and effort for the initial introduction of a tool (including training and external expertise)
- Underestimating the time and effort needed to achieve significant and continuing benefits from the tool (including the need for changes in the testing process and continuous improvement of the way the tool is used)
- Underestimating the effort required to maintain the test assets generated by the tool
- Over-reliance on the tool (replacement for test design or use of automated testing where manual testing would be better)
- Neglecting version control of test assets within the tool
- Neglecting relationships and interoperability issues between critical tools, such as requirements management tools, version control tools, incident management tools, defect tracking tools and tools from multiple vendors
- Risk of tool vendor going out of business, retiring the tool, or selling the tool to a different vendor
- Poor response from vendor for support, upgrades, and defect fixes
- Risk of suspension of open-source / free tool project
- Unforeseen, such as the inability to support a new platform

6.2.2 Special Considerations for Some Types of Tools (K1)

Test Execution Tools
Test execution tools execute test objects using automated test scripts. This type of tool often requires significant effort in order to achieve significant benefits.

Capturing tests by recording the actions of a manual tester seems attractive, but this approach does not scale to large numbers of automated test scripts. A captured script is a linear representation with specific data and actions as part of each script. This type of script may be unstable when unexpected events occur.

A data-driven testing approach separates out the test inputs (the data), usually into a spreadsheet, and uses a more generic test script that can read the input data and execute the same test script with different data. Testers who are not familiar with the scripting language can then create the test data for these predefined scripts.

There are other techniques employed in data-driven techniques, where instead of hard-coded data combinations placed in a spreadsheet, data is generated using algorithms based on configurable parameters at run time and supplied to the application. For example, a tool may use an algorithm, which generates a random user ID, and for repeatability in pattern, a seed is employed for controlling randomness.

In a keyword-driven testing approach, the spreadsheet contains keywords describing the actions to be taken (also called action words), and test data. Testers (even if they are not familiar with the scripting language) can then define tests using the keywords, which can be tailored to the application being tested.

Technical expertise in the scripting language is needed for all approaches (either by testers or by specialists in test automation).

Regardless of the scripting technique used, the expected results for each test need to be stored for later comparison.

Static Analysis Tools
Static analysis tools applied to source code can enforce coding standards, but if applied to existing code may generate a large quantity of messages. Warning messages do not stop the code from being translated into an executable program, but ideally should be addressed so that maintenance of the code is easier in the future. A gradual implementation of the analysis tool with initial filters to exclude some messages is an effective approach.

Test Management Tools
Test management tools need to interface with other tools or spreadsheets in order to produce useful information in a format that fits the needs of the organization.

6.3 Introducing a Tool into an Organization (K1)	*15 minutes*

Terms
No specific terms.

Background
The main considerations in selecting a tool for an organization include:
- Assessment of organizational maturity, strengths and weaknesses and identification of opportunities for an improved test process supported by tools
- Evaluation against clear requirements and objective criteria
- A proof-of-concept, by using a test tool during the evaluation phase to establish whether it performs effectively with the software under test and within the current infrastructure or to identify changes needed to that infrastructure to effectively use the tool
- Evaluation of the vendor (including training, support and commercial aspects) or service support suppliers in case of non-commercial tools
- Identification of internal requirements for coaching and mentoring in the use of the tool
- Evaluation of training needs considering the current test team's test automation skills
- Estimation of a cost-benefit ratio based on a concrete business case

Introducing the selected tool into an organization starts with a pilot project, which has the following objectives:
- Learn more detail about the tool
- Evaluate how the tool fits with existing processes and practices, and determine what would need to change
- Decide on standard ways of using, managing, storing and maintaining the tool and the test assets (e.g., deciding on naming conventions for files and tests, creating libraries and defining the modularity of test suites)
- Assess whether the benefits will be achieved at reasonable cost

Success factors for the deployment of the tool within an organization include:
- Rolling out the tool to the rest of the organization incrementally
- Adapting and improving processes to fit with the use of the tool
- Providing training and coaching/mentoring for new users
- Defining usage guidelines
- Implementing a way to gather usage information from the actual use
- Monitoring tool use and benefits
- Providing support for the test team for a given tool
- Gathering lessons learned from all teams

References
6.2.2 Buwalda, 2001, Fewster, 1999
6.3 Fewster, 1999

7. References

Standards

ISTQB Glossary of Terms used in Software Testing Version 2.1

[CMMI] Chrissis, M.B., Konrad, M. and Shrum, S. (2004) CMMI, Guidelines for Process Integration *and Product Improvement*, Addison Wesley: Reading, MA
See Section 2.1

[IEEE Std 829-1998] IEEE Std 829™ (1998) IEEE Standard for Software Test Documentation,
See Sections 2.3, 2.4, 4.1, 5.2, 5.3, 5.5, 5.6

[IEEE 1028] IEEE Std 1028™ (2008) IEEE Standard for Software Reviews and Audits,
See Section 3.2

[IEEE 12207] IEEE 12207/ISO/IEC 12207-2008, Software life cycle processes,
See Section 2.1

[ISO 9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality,
See Section 2.3

Books

[Beizer, 1990] Beizer, B. (1990*) Software Testing Techniques* (2nd edition), Van Nostrand Reinhold: Boston
See Sections 1.2, 1.3, 2.3, 4.2, 4.3, 4.4, 4.6

[Black, 2001] Black, R. (2001) *Managing the Testing Process* (3rd edition), John Wiley & Sons: New York
See Sections 1.1, 1.2, 1.4, 1.5, 2.3, 2.4, 5.1, 5.2, 5.3, 5.5, 5.6

[Buwalda, 2001] Buwalda, H. et al. (2001) *Integrated Test Design and Automation*, Addison Wesley: Reading, MA
See Section 6.2

[Copeland, 2004] Copeland, L. (2004) *A Practitioner's Guide to Software Test Design*, Artech House: Norwood, MA
See Sections 2.2, 2.3, 4.2, 4.3, 4.4, 4.6

[Craig, 2002] Craig, Rick D. and Jaskiel, Stefan P. (2002) *Systematic Software Testing*, Artech House: Norwood, MA
See Sections 1.4.5, 2.1.3, 2.4, 4.1, 5.2.5, 5.3, 5.4

[Fewster, 1999] Fewster, M. and Graham, D. (1999) *Software Test Automation*, Addison Wesley: Reading, MA
See Sections 6.2, 6.3

[Gilb, 1993]: Gilb, Tom and Graham, Dorothy (1993) *Software Inspection*, Addison Wesley: Reading, MA
See Sections 3.2.2, 3.2.4

[Hetzel, 1988] Hetzel, W. (1988) *Complete Guide to Software Testing*, QED: Wellesley, MA
See Sections 1.3, 1.4, 1.5, 2.1, 2.2, 2.3, 2.4, 4.1, 5.1, 5.3

[Kaner, 2002] Kaner, C., Bach, J. and Pettticord, B. (2002) *Lessons Learned in Software Testing*, John Wiley & Sons: New York
See Sections 1.1, 4.5, 5.2

[Myers 1979] Myers, Glenford J. (1979) *The Art of Software Testing*, John Wiley & Sons: New York
See Sections 1.2, 1.3, 2.2, 4.3

[van Veenendaal, 2004] van Veenendaal, E. (ed.) (2004) *The Testing Practitioner* (Chapters 6, 8, 10), UTN Publishers: The Netherlands
See Sections 3.2, 3.3

8. Appendix A – Syllabus Background

History of this Document

This document was prepared between 2004 and 2011 by a Working Group comprised of members appointed by the International Software Testing Qualifications Board (ISTQB). It was initially reviewed by a selected review panel, and then by representatives drawn from the international software testing community. The rules used in the production of this document are shown in Appendix C.

This document is the syllabus for the International Foundation Certificate in Software Testing, the first level international qualification approved by the ISTQB (www.istqb.org).

Objectives of the Foundation Certificate Qualification

- To gain recognition for testing as an essential and professional software engineering specialization
- To provide a standard framework for the development of testers' careers
- To enable professionally qualified testers to be recognized by employers, customers and peers, and to raise the profile of testers
- To promote consistent and good testing practices within all software engineering disciplines
- To identify testing topics that are relevant and of value to industry
- To enable software suppliers to hire certified testers and thereby gain commercial advantage over their competitors by advertising their tester recruitment policy
- To provide an opportunity for testers and those with an interest in testing to acquire an internationally recognized qualification in the subject

Objectives of the International Qualification (adapted from ISTQB meeting at Sollentuna, November 2001)

- To be able to compare testing skills across different countries
- To enable testers to move across country borders more easily
- To enable multinational/international projects to have a common understanding of testing issues
- To increase the number of qualified testers worldwide
- To have more impact/value as an internationally-based initiative than from any country-specific approach
- To develop a common international body of understanding and knowledge about testing through the syllabus and terminology, and to increase the level of knowledge about testing for all participants
- To promote testing as a profession in more countries
- To enable testers to gain a recognized qualification in their native language
- To enable sharing of knowledge and resources across countries
- To provide international recognition of testers and this qualification due to participation from many countries

Entry Requirements for this Qualification

The entry criterion for taking the ISTQB Foundation Certificate in Software Testing examination is that candidates have an interest in software testing. However, it is strongly recommended that candidates also:

- Have at least a minimal background in either software development or software testing, such as six months experience as a system or user acceptance tester or as a software developer

- Take a course that has been accredited to ISTQB standards (by one of the ISTQB-recognized National Boards).

Background and History of the Foundation Certificate in Software Testing

The independent certification of software testers began in the UK with the British Computer Society's Information Systems Examination Board (ISEB), when a Software Testing Board was set up in 1998 (www.bcs.org.uk/iseb). In 2002, ASQF in Germany began to support a German tester qualification scheme (www.asqf.de). This syllabus is based on the ISEB and ASQF syllabi; it includes reorganized, updated and additional content, and the emphasis is directed at topics that will provide the most practical help to testers.

An existing Foundation Certificate in Software Testing (e.g., from ISEB, ASQF or an ISTQB-recognized National Board) awarded before this International Certificate was released, will be deemed to be equivalent to the International Certificate. The Foundation Certificate does not expire and does not need to be renewed. The date it was awarded is shown on the Certificate.

Within each participating country, local aspects are controlled by a national ISTQB-recognized Software Testing Board. Duties of National Boards are specified by the ISTQB, but are implemented within each country. The duties of the country boards are expected to include accreditation of training providers and the setting of exams.

9. Appendix B – Learning Objectives/Cognitive Level of Knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it.

Level 1: Remember (K1)

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

Keywords: Remember, retrieve, recall, recognize, know

Example

Can recognize the definition of "failure" as:

- "Non-delivery of service to an end user or any other stakeholder" or
- "Actual deviation of the component or system from its expected delivery, service or result"

Level 2: Understand (K2)

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

Keywords: Summarize, generalize, abstract, classify, compare, map, contrast, exemplify, interpret, translate, represent, infer, conclude, categorize, construct models

Examples

Can explain the reason why tests should be designed as early as possible:

- To find defects when they are cheaper to remove
- To find the most important defects first

Can explain the similarities and differences between integration and system testing:

- Similarities: testing more than one component, and can test non-functional aspects
- Differences: integration testing concentrates on interfaces and interactions, and system testing concentrates on whole-system aspects, such as end-to-end processing

Level 3: Apply (K3)

The candidate can select the correct application of a concept or technique and apply it to a given context.

Keywords: Implement, execute, use, follow a procedure, apply a procedure

Example

- Can identify boundary values for valid and invalid partitions
- Can select test cases from a given state transition diagram in order to cover all transitions

Level 4: Analyze (K4)

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

Keywords: Analyze, organize, find coherence, integrate, outline, parse, structure, attribute, deconstruct, differentiate, discriminate, distinguish, focus, select

Example
- Analyze product risks and propose preventive and corrective mitigation activities
- Describe which portions of an incident report are factual and which are inferred from results

Reference

(For the cognitive levels of learning objectives)
Anderson, L. W. and Krathwohl, D. R. (eds) (2001) *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*, Allyn & Bacon

10. Appendix C – Rules Applied to the ISTQB

Foundation Syllabus

The rules listed here were used in the development and review of this syllabus. (A "TAG" is shown after each rule as a shorthand abbreviation of the rule.)

10.1.1 General Rules

SG1. The syllabus should be understandable and absorbable by people with zero to six months (or more) experience in testing. (6-MONTH)
SG2. The syllabus should be practical rather than theoretical. (PRACTICAL)
SG3. The syllabus should be clear and unambiguous to its intended readers. (CLEAR)
SG4. The syllabus should be understandable to people from different countries, and easily translatable into different languages. (TRANSLATABLE)
SG5. The syllabus should use American English. (AMERICAN-ENGLISH)

10.1.2 Current Content

SC1. The syllabus should include recent testing concepts and should reflect current best practices in software testing where this is generally agreed. The syllabus is subject to review every three to five years. (RECENT)
SC2. The syllabus should minimize time-related issues, such as current market conditions, to enable it to have a shelf life of three to five years. (SHELF-LIFE).

10.1.3 Learning Objectives

LO1. Learning objectives should distinguish between items to be recognized/remembered (cognitive level K1), items the candidate should understand conceptually (K2), items the candidate should be able to practice/use (K3), and items the candidate should be able to use to analyze a document, software or project situation in context (K4). (KNOWLEDGE-LEVEL)
LO2. The description of the content should be consistent with the learning objectives. (LO-CONSISTENT)
LO3. To illustrate the learning objectives, sample exam questions for each major section should be issued along with the syllabus. (LO-EXAM)

10.1.4 Overall Structure

ST1. The structure of the syllabus should be clear and allow cross-referencing to and from other parts, from exam questions and from other relevant documents. (CROSS-REF)
ST2. Overlap between sections of the syllabus should be minimized. (OVERLAP)
ST3. Each section of the syllabus should have the same structure. (STRUCTURE-CONSISTENT)
ST4. The syllabus should contain version, date of issue and page number on every page. (VERSION)
ST5. The syllabus should include a guideline for the amount of time to be spent in each section (to reflect the relative importance of each topic). (TIME-SPENT)

References

SR1. Sources and references will be given for concepts in the syllabus to help training providers find out more information about the topic. (REFS)
SR2. Where there are not readily identified and clear sources, more detail should be provided in the syllabus. For example, definitions are in the Glossary, so only the terms are listed in the syllabus. (NON-REF DETAIL)

Sources of Information

Terms used in the syllabus are defined in the ISTQB Glossary of Terms used in Software Testing. A version of the Glossary is available from ISTQB.

A list of recommended books on software testing is also issued in parallel with this syllabus. The main book list is part of the References section.

11. Appendix D – Notice to Training Providers

Each major subject heading in the syllabus is assigned an allocated time in minutes. The purpose of this is both to give guidance on the relative proportion of time to be allocated to each section of an accredited course, and to give an approximate minimum time for the teaching of each section. Training providers may spend more time than is indicated and candidates may spend more time again in reading and research. A course curriculum does not have to follow the same order as the syllabus.

The syllabus contains references to established standards, which must be used in the preparation of training material. Each standard used must be the version quoted in the current version of this syllabus. Other publications, templates or standards not referenced in this syllabus may also be used and referenced, but will not be examined.

All K3 and K4 Learning Objectives require a practical exercise to be included in the training materials.

12. Appendix E – Release Notes

Release 2010

1. Changes to Learning Objectives (LO) include some clarification
    a. Wording changed for the following LOs (content and level of LO remains unchanged): LO-1.2.2, LO-1.3.1, LO-1.4.1, LO-1.5.1, LO-2.1.1, LO-2.1.3, LO-2.4.2, LO-4.1.3, LO-4.2.1, LO-4.2.2, LO-4.3.1, LO-4.3.2, LO-4.3.3, LO-4.4.1, LO-4.4.2, LO-4.4.3, LO-4.6.1, LO-5.1.2, LO-5.2.2, LO-5.3.2, LO-5.3.3, LO-5.5.2, LO-5.6.1, LO-6.1.1, LO-6.2.2, LO-6.3.2.
    b. LO-1.1.5 has been reworded and upgraded to K2. Because a comparison of terms of defect related terms can be expected.
    c. LO-1.2.3 (K2) has been added. The content was already covered in the 2007 syllabus.
    d. LO-3.1.3 (K2) now combines the content of LO-3.1.3 and LO-3.1.4.
    e. LO-3.1.4 has been removed from the 2010 syllabus, as it is partially redundant with LO-3.1.3.
    f. LO-3.2.1 has been reworded for consistency with the 2010 syllabus content.
    g. LO-3.3.2 has been modified, and its level has been changed from K1 to K2, for consistency with LO-3.1.2.
    h. LO 4.4.4 has been modified for clarity, and has been changed from a K3 to a K4. Reason: LO-4.4.4 had already been written in a K4 manner.
    i. LO-6.1.2 (K1) was dropped from the 2010 syllabus and was replaced with LO-6.1.3 (K2). There is no LO-6.1.2 in the 2010 syllabus.
2. Consistent use for test approach according to the definition in the glossary. The term test strategy will not be required as term to recall.
3. Chapter 1.4 now contains the concept of traceability between test basis and test cases.
4. Chapter 2.x now contains test objects and test basis.
5. Re-testing is now the main term in the glossary instead of confirmation testing.
6. The aspect data quality and testing has been added at several locations in the syllabus: data quality and risk in Chapter 2.2, 5.5, 6.1.8.
7. Chapter 5.2.3 Entry Criteria are added as a new subchapter. Reason: Consistency to Exit Criteria (-> entry criteria added to LO-5.2.9).
8. Consistent use of the terms test strategy and test approach with their definition in the glossary.
9. Chapter 6.1 shortened because the tool descriptions were too large for a 45 minute lesson.
10. IEEE Std 829:2008 has been released. This version of the syllabus does not yet consider this new edition. Section 5.2 refers to the document Master Test Plan. The content of the Master Test Plan is covered by the concept that the document “Test Plan” covers different levels of planning: Test plans for the test levels can be created as well as a test plan on the project level covering multiple test levels. Latter is named Master Test Plan in this syllabus and in the ISTQB Glossary.
11. Code of Ethics has been moved from the CTAL to CTFL.

Release 2011

Changes made with the “maintenance release” 2011

1. General: Working Party replaced by Working Group
2. Replaced post-conditions by postconditions in order to be consistent with the ISTQB Glossary 2.1.
3. First occurrence: ISTQB replaced by ISTQB®
4. Introduction to this Syllabus: Descriptions of Cognitive Levels of Knowledge removed, because this was redundant to Appendix B.

5. Section 1.6: Because the intent was not to define a Learning Objective for the “Code of Ethics”, the cognitive level for the section has been removed.
6. Section 2.2.1, 2.2.2, 2.2.3 and 2.2.4, 3.2.3: Fixed formatting issues in lists.
7. Section 2.2.2 The word failure was not correct for “…isolate failures to a specific component …”. Therefore replaced with “defect” in that sentence.
8. Section 2.3: Corrected formatting of bullet list of test objectives related to test terms in section Test Types (K2).
9. Section 2.3.4: Updated description of debugging to be consistent with Version 2.1 of the ISTQB Glossary.
10. Section 2.4 removed word “extensive” from “includes extensive regression testing”, because the “extensive” depends on the change (size, risks, value, etc.) as written in the next sentence.
11. Section 3.2: The word “including” has been removed to clarify the sentence.
12. Section 3.2.1: Because the activities of a formal review had been incorrectly formatted, the review process had 12 main activities instead of six, as intended. It has been changed back to six, which makes this section compliant with the Syllabus 2007 and the ISTQB Advanced Level Syllabus 2007.
13. Section 4: Word “developed” replaced by “defined” because test cases get defined and not developed.
14. Section 4.2: Text change to clarify how black-box and white-box testing could be used in conjunction with experience-based techniques.
15. Section 4.3.5 text change “..between actors, including users and the system..” to “ … between actors (users or systems), … “.
16. Section 4.3.5 alternative path replaced by alternative scenario.
17. Section 4.4.2: In order to clarify the term branch testing in the text of Section 4.4, a sentence to clarify the focus of branch testing has been changed.
18. Section 4.5, Section 5.2.6: The term “experienced-based” testing has been replaced by the correct term “experience-based”.
19. Section 6.1: Heading “6.1.1 Understanding the Meaning and Purpose of Tool Support for Testing (K2)” replaced by “6.1.1 Tool Support for Testing (K2)”.
20. Section 7 / Books: The 3rd edition of [Black,2001] listed, replacing 2nd edition.
21. Appendix D: Chapters requiring exercises have been replaced by the generic requirement that all Learning Objectives K3 and higher require exercises. This is a requirement specified in the ISTQB Accreditation Process (Version 1.26).
22. Appendix E: The changed learning objectives between Version 2007 and 2010 are now correctly listed.

13. Index