Certified Tester

Foundation Level Syllabus

Version 2005

International Software Testing Qualifications Board

Revision History

Version	Date	Remarks
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 Softwaretestens“
ISEB V2.0	25-Feb-1999	ISEB Software Testing Foundation Syllabus V2.0 25 February 1999

Table of Contents

Acknowledgements

This document was produced by the International Software Testing Qualifications Board Working Party Foundation Level: Thomas Müller (chair), Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhäjärvi, Geoff Thompson and Erik van Veendendal. The core team thanks the review team and all national boards for the suggestions to the current syllabus.

Particular thanks to (Austria) Anastasios Kyriakopoulos, (Denmark) Klaus Olsen, Christine Rosenbeck-Larsen, (Germany) Matthias Daigl, Uwe Hehn, Tilo Linz, Horst Pohlmann, Ina Schieferdecker, Sabine Uhde, Stephanie Ulrich, (India) Vipul Kocher, (Israel) Shmuel Knishinsky, Ester Zabar, (Sweden) Anders Claesson, Mattias Nordin, Ingvar Nordström, Stefan Ohlsson, Kennet Osbjer, Ingela Skytte, Klaus Zeuge, (Switzerland) Armin Born, Sandra Harries, Silvio Moser, Reto Müller, Joerg Pietzsch, (UK) Aran Ebbett, Isabel Evans, Julie Gardiner, Andrew Goslin, Brian Hambling, James Lyndsay, Helen Moore, Peter Morgan, Trevor Newton, Angelina Samaroo, Shane Saunders, Mike Smith, Richard Taylor, Neil Thompson, Pete Williams, (US) Dale Perry.

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 (hereinafter called ISTQB) provides it to the national examination bodies for them to accredit the training providers and to derive examination questions in their local language. Training providers will produce courseware and determine appropriate teaching methods for accreditation, and 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/level of knowledge

Cognitive levels are given for each section in this syllabus:

- K1: remember, recognize, recall;
- K2: understand, explain, give reasons, compare, classify, summarize;
- K3: apply.

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 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 centre).

Accreditation

Training providers whose course material follows this syllabus may be accredited by a national board recognized by ISTQB. Accreditation guidelines should be obtained 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 have understood.
- 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 shows the levels of learning objectives that are covered within the chapter, and specifies the time for the chapter. For example:

2. Testing throughout the software life cycle (K2)	135 minutes

shows that Chapter 2 has learning objectives of K1 (assumed when a higher level is shown) and K2 (but not K3), and is intended to take 135 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)

- 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)
- Distinguish between the root cause of a defect and its effects. (K2)
- Give reasons why testing is necessary by giving examples. (K2)
- Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality. (K2)
- Recall the terms mistake, defect, failure and corresponding terms error and bug. (K1)

1.2 What is testing? (K2)

- Recall the common objectives of testing. (K1)
- Describe the purpose of testing in software development, maintenance and operations as a means to find defects, provide confidence and information, and prevent defects. (K2)

1.3 General testing principles (K2)

- Explain the fundamental principles in testing. (K2)

1.4 Fundamental test process (K1)

- Recall the fundamental test activities from planning to test closure activities and the main tasks of each test activity. (K1)

1.5 The psychology of testing (K2)

- Recall that the success of testing is influenced by psychological factors (K1):
  - clear objectives;
  - a balance of self-testing and independent testing;
  - recognition of courteous communication and feedback on defects.
- 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, software, testing.

1.1.1 Software systems context (K1)

Software systems are an increasing 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 code, in software or a system, or in a document. If a defect in code is executed, the system will 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, changed technologies, and/or many system interactions.

Failures can be caused by environmental conditions as well: radiation, magnetism, electronic fields, and pollution can cause faults in firmware or influence the execution of software by changing 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 in an operational environment and contribute to the quality of the software system, if 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 and maintainability). 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 reoccurring and, as a consequence, improve the quality of future systems.

Testing should be integrated as one of the quality assurance activities (e.g. 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 and business product and project 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
Code, debugging, development (of software), requirement, review, test basis, test case, testing, test objectives.

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, activities such as planning and control, choosing test conditions, designing test cases and checking results, evaluating completion criteria, reporting on the testing process and system under test, and finalizing or closure (e.g. after a test phase has been completed). Testing also includes reviewing of documents (including source code) and static analysis.

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

There can be different test objectives:

- finding defects;
- gaining confidence about the level of quality and providing information;
- preventing defects.

The thought process of 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) 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 errors 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. Testing can show failures that are caused by defects. Debugging is the development activity that identifies the cause of a defect, repairs the code and checks that the defect has been fixed correctly. Subsequent confirmation testing by a tester ensures that the fix does indeed resolve the failure. The responsibility for each activity is very different, i.e. testers test and developers debug.

The process of testing and its activities is explained in Section 1.4.

1.3 *General 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, we use risk and priorities to focus testing efforts.

Principle 3 – Early testing
Testing activities should start as early as possible in the software or system development life cycle, and should be focused on defined objectives.

Principle 4 – Defect clustering
A small number of modules contain most of the defects discovered during pre-release testing, or show the most 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 bugs. To overcome this “pesticide paradox”, the 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 potentially find 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, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test strategy, test summary report, testware.

Background
The most visible part of testing is executing tests. 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 status.

The fundamental test process consists of the following main activities:

- planning and control;
- analysis and design;
- 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.

1.4.1 Test planning and control (K1)
Test planning is the activity of verifying the mission of testing, 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, it should be monitored throughout the project. Test planning takes into account the feedback from monitoring and control activities.

Test planning has the following major tasks:

- Determining the scope and risks, and identifying the objectives of testing.
- Determining the test approach (techniques, test items, coverage, identifying and interfacing the teams involved in testing, testware).
- Determining the required test resources (e.g. people, test environment, PCs).
- Implementing the test policy and/or the test strategy.
- Scheduling test analysis and design tasks.
- Scheduling test implementation, execution and evaluation.
- Determining the exit criteria.

Test control has the following major tasks:

- measuring and analyzing results;
- monitoring and documenting progress, test coverage and exit criteria;
- initiation of corrective actions;
- making decisions.

1.4.2 Test analysis and design (K1)
Test analysis and design is the activity where general testing objectives are transformed into tangible test conditions and test designs.

Test analysis and design has the following major tasks:

- Reviewing the test basis (such as requirements, architecture, design, interfaces).
- Identifying test conditions or test requirements and required test data based on analysis of test items, the specification, behavior and structure.
- Designing the tests.
- Evaluating testability of the requirements and system.
- Designing the test environment set-up and identifying any required infrastructure and tools.

1.4.3 Test implementation and execution (K1)

Test implementation and execution is the activity where test conditions are transformed into test cases and testware, and the environment is set up.

Test implementation and execution has the following major tasks:

- Developing and prioritizing test cases, creating test data, writing test procedures and, optionally, preparing test harnesses and writing automated test scripts.
- Creating test suites from the test cases for efficient test execution.
- Verifying that the test environment has been set up correctly.
- Executing test cases 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 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.

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. For example, 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, the closure of incident reports or raising of change records for any that remain open, and the documentation of the acceptance of the system.
- Finalizing and archiving testware, the test environment and the test infrastructure for later reuse.
- Handover of testware to the maintenance organization.
- Analyzing lessons learned for future releases and projects, and the improvement of test maturity.

1.5 *The psychology of testing (K2)* — *35 minutes*

Terms
Independent testing.

Background
The mindset to be used while testing and reviewing is different to that used while analyzing or developing. 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) is often 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:

- 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).
- 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 works. 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. Testing is, therefore, 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 reviewing 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.

References

1.1.5 Black, 2001, Kaner, 2002
1.2 Beizer, 1990, Black, 2001, Myers, 1979
1.3 Beizer, 1990, Hetzel, 1998, Myers, 1979
1.4 Hetzel, 1998
1.4.5 Black, 2001, Craig, 2002
1.5 Black, 2001, Hetzel, 1998

2. Testing throughout the software life cycle (K2) — 135 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)

- Understand the relationship between development, test activities and work products in the development life cycle, and give examples based on project and product characteristics and context (K2).
- Recognize the fact that software development models must be adapted to the context of project and product characteristics. (K1)
- Recall reasons for different levels of testing, and characteristics of good testing in any life cycle model. (K1)

2.2 Test levels (K2)

- 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: the targets of testing (K2)

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

2.4 Maintenance testing (K2)

- 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)
- Identify reasons for maintenance testing (modification, migration and retirement). (K1)
- 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), incremental development model, test level, 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 (K2)
Although variants of the V-model exist, a common type of V-model uses four test levels, corresponding to 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, requirement 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 development models (K2)
Iterative development is the process of establishing requirements, designing, building and testing a system, done as a series of smaller developments. Examples are: prototyping, rapid application development (RAD), Rational Unified Process (RUP) and agile development models. The increment produced by an iteration may be tested at several levels as part of its development. 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)* — *60 minutes*

Terms
Alpha testing, beta testing, component testing (also known as unit, module or program testing), contract acceptance testing, drivers, field testing, functional requirements, integration, integration testing, non-functional requirements, operational (acceptance) testing, regulation acceptance testing, robustness testing, stubs, system testing, test-driven development, test environment, user acceptance testing.

Background
For each of the test levels, the following can be identified: their 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.

2.2.1 Component testing (K2)
Component testing searches for defects in, and verifies the functioning of, software (e.g. 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. memory leaks) or robustness testing, as well as structural testing (e.g. branch 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 the development environment, such as a unit test framework or debugging tool, and, in practice, usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally recording incidents.

One approach in 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 until they pass.

2.2.2 Integration testing (K2)
Integration testing tests interfaces between components, interactions to different parts of a system, such as the operating system, file system, hardware or 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. For example:

- Component integration testing tests the interactions between software components and is done after component testing;
- System integration testing tests the interactions between different systems and may be done after system testing. In this case, the developing organization may control only one side of the interface, so changes may be destabilizing. 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 failures to a specific component or system, which may lead to increased risk.

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

component. In order to reduce the risk of late defect discovery, integration should normally be incremental rather than “big bang”.

Testing of specific non-functional characteristics (e.g. performance) may be included in integration 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 either module. 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, they can be built in the order required for most efficient testing.

2.2.3 System testing (K2)

System testing is concerned with the behavior of a whole system/product as defined by the scope of a development project or programme.

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 descriptions of system behavior, interactions with the operating system, and system resources.

System testing should investigate both functional and non-functional requirements of the system. Requirements may exist as text and/or models. 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)

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 as more than just a single test level, 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;
- 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 contract is agreed. Regulation acceptance testing is performed against any regulations which must be adhered to, such as governmental, 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. Beta testing, or field testing, is performed by people at their own locations. Both are performed by potential customers, not the developers of the product.

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: the targets of testing (K2)* — *40 minutes*

Terms
Automation, black-box testing, code coverage, confirmation testing, functional testing, interoperability testing, load testing, maintainability testing, performance testing, portability testing, regression testing, reliability testing, security testing, specification-based testing, stress testing, structural testing, test suite, 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 the testing of 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; or related to changes, 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 and functional testing. For example, in functional testing a process flow model, a state transition model or a plain language specification; and for structural testing a control flow model or menu structure model.

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 these functions and features (described in documents or understood by the testers), 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 behaviour 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.

2.3.2 Testing of software product characteristics (non-functional testing) (K2)
Non-functional testing includes, but is not limited to, performance testing, load testing, stress testing, usability testing, interoperability 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).

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 and, therefore, 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 (confirmation and regression testing) (K2)

When a defect is detected and fixed then the software should be retested to confirm that the original defect has been successfully removed. This is called confirmation testing. Debugging (defect fixing) 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 applies to 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, migration, modifications, retirement.

Background
Once deployed, a software system is often in service for years or decades. During this time the system and its environment is often corrected, changed or extended. 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, or patches to 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.

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 extensive 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.

Maintenance testing can be difficult if specifications are out of date or missing.

References
2.1.3 CMMI, Craig, 2002, Hetzel, 1998, IEEE 12207
2.2 Hetzel, 1998
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, 1998
2.3.4 Hetzel, 1998, IEEE 829
2.4 Black, 2001, Craig, 2002, Hetzel, 1998, IEEE 829

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 Reviews and the test process (K2)

- Recognize software work products that can be examined by the different static techniques. (K1)
- Describe the importance and value of considering static techniques for the assessment of software work products. (K2)
- Explain the difference between static and dynamic techniques. (K2)

3.2 Review process (K2)

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

3.3 Static analysis by tools (K2)

- Describe the objective of static analysis and compare it to dynamic testing. (K2)
- Recall typical defects and errors identified by static analysis and compare them to reviews and dynamic testing. (K1)
- List typical benefits of static analysis. (K1)
- List typical code and design defects that may be identified by static analysis tools. (K1)

3.1 *Reviews and the test process (K2)* — *15 minutes*

Terms
Dynamic testing, reviews, static analysis.

Background
Static testing techniques do not execute the software that is being tested; they are manual (reviews) or automated (static analysis).

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 are often much cheaper to remove than those detected while running tests (e.g. defects found in requirements).

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 requirement 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 defect effectively and efficiently. In contrast to dynamic testing, reviews find defects rather than failures.

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

3.2 *Review process (K2)* — *25 minutes*

Terms
Entry criteria, exit criteria, formal review, informal review, inspection, kick-off, metrics, moderator/ inspection leader, peer review, reviewer, review meeting, review process, scribe, technical review, walkthrough.

Background
Reviews vary from very informal to very formal (i.e. well structured and regulated). 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 objective of the review (e.g. find defects, gain understanding, or discussion and decision by consensus).

3.2.1 Phases of a formal review (K1)
A typical formal review has the following main phases:

- Planning: selecting the personnel, allocating roles; defining the entry and exit criteria for more formal review types (e.g. inspection); and selecting which parts of documents to look at.
- Kick-off: distributing documents; explaining the objectives, process and documents to the participants; and checking entry criteria (for more formal review types).
- Individual preparation: work done by each of the participants on their own before the review meeting, noting potential defects, questions and comments.
- Review meeting: discussion or logging, with documented results or minutes (for more formal review types). The meeting participants may simply note defects, make recommendations for handling the defects, or make decisions about the defects.
- Rework: fixing defects found, typically done by the author.
- Follow-up: checking that defects have been addressed, gathering metrics and 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 follow-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 they 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 documents from different perspectives, and using checklists, can make reviews more effective and efficient, for example, a checklist based on perspectives such as user, maintainer, tester or operations, or a checklist of typical requirements problems.

3.2.3 Types of review (K2)
A single document 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 requirement specification before a walkthrough with customers. The main characteristics, options and purposes of common review types are:

Informal review
Key characteristics:
- no formal process;
- there may be pair programming or a technical lead reviewing designs and code;
- optionally may be documented;
- may vary in usefulness depending on the reviewer;
- main purpose: inexpensive way to get some benefit.

Walkthrough
Key characteristics:
- meeting led by author;
- scenarios, dry runs, peer group;
- open-ended sessions;
- optionally a pre-meeting preparation of reviewers, review report, list of findings and scribe (who is not the author)
- may vary in practice from quite informal to very formal;
- main purposes: learning, gaining understanding, defect finding.

Technical review
Key characteristics:
- documented, defined defect-detection process that includes peers and technical experts;
- may be performed as a peer review without management participation;
- ideally led by trained moderator (not the author);
- pre-meeting preparation;
- optionally the use of checklists, review report, list of findings and management participation;
- may vary in practice from quite informal to very formal;
- main purposes: discuss, make decisions, evaluate alternatives, find defects, solve technical problems and check conformance to specifications and standards.

I**nspection**
Key characteristics:
- led by trained moderator (not the author);
- usually peer examination;
- defined roles;
- includes metrics;
- formal process based on rules and checklists with entry and exit criteria;
- pre-meeting preparation;
- inspection report, list of findings;
- formal follow-up process;
- optionally, process improvement and reader;
- main purpose: find defects.

3.2.4 Success factors for reviews (K2)

Success factors for reviews include:

- Each review has a clear predefined objective.
- The right people for the review objectives are involved.
- 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).
- Review techniques are applied that are suitable 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 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 interface between modules and components;
- variables that are never used;
- unreachable (dead) code;
- 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, 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 (K3) — 255 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 Identifying test conditions and designing test cases (K3)

- Differentiate between a test design specification, test case specification and test procedure specification. (K1)
- Compare the terms test condition, test case and test procedure. (K2)
- Write test cases: (K3)
  - showing a clear traceability to the requirements;
  - containing an expected result.
- Translate test cases into a well-structured test procedure specification at a level of detail relevant to the knowledge of the testers. (K3)
- Write a test execution schedule for a given set of test cases, considering prioritization, and technical and logical dependencies. (K3)

4.2 Categories of test design techniques (K2)

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

4.3 Specification-based or black-box techniques (K3)

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

4.4 Structure-based or white-box techniques (K3)

- Describe the concept and importance of code coverage. (K2)
- Explain the concepts of statement and decision coverage, and understand that these concepts can also be used at other test levels than component testing (e.g. on business procedures at system level). (K2)
- Write test cases from given control flows using the following test design techniques: (K3)
  - statement testing;
  - decision testing.
- Assess statement and decision coverage for completeness. (K3)

4.5 Experience-based techniques (K2)

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

4.6 Choosing test techniques (K2)

- List the factors that influence the selection of the appropriate test design technique for a particular kind of problem, such as the type of system, risk, customer requirements, models for use case modeling, requirements models or tester knowledge. (K2)

4.1 *Identifying test conditions and designing test cases (K3)* — *15 minutes*

Terms
Test cases, test case specification, test condition, test data, test procedure specification, test script, traceability.

Background
The process of identifying test conditions and designing tests consists of a number of steps:

- Designing tests by identifying test conditions.
- Specifying test cases.
- Specifying test procedures.

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

During test design, 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 impact analysis, when requirements change, and requirements coverage to be determined for a set of tests. During test design the detailed test approach is implemented based on, among other considerations, the risks identified (see Chapter 5 for more on risk analysis).

During test case specification the test cases and test data are developed and described in detail by using test design techniques. A test case consists of a set of input values, execution preconditions, expected results and execution post-conditions, developed to cover certain test condition(s). The ‘Standard for Software Test Documentation’ (IEEE 829) describes the content of test design specifications 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.

The test cases are put in an executable order; this is the test procedure specification. The test procedure (or manual test script) specifies the sequence of action 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, when they are to be carried out and by whom. 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 techniques, experience-based techniques, specification-based techniques, structure-based techniques, white-box techniques.

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

It is a classic distinction to denote test techniques as black box or white box. Black-box techniques (also called specification-based techniques) are a way to derive and select test conditions or test cases based on an analysis of the test basis documentation, whether functional or non-functional, for a component or system without reference to its internal structure. White-box techniques (also called structural or structure-based techniques) are based on an analysis of the internal structure of the component or system.

Some techniques fall clearly into a single category; others have elements of more than one category. This syllabus refers to specification-based or experience-based approaches as black-box techniques and structure-based as white-box techniques.

Common features of specification-based techniques:

- Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components.
- From these models test cases can be derived systematically.

Common features of structure-based techniques:

- Information about how the software is constructed is used to derive the test cases, for example, code and design.
- 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 features of experience-based techniques:

- The knowledge and experience of people are used to derive the test cases.
    - knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment;
    - knowledge about likely defects and their distribution.

4.3 *Specification-based or black-box techniques (K3)* — *120 minutes*

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

4.3.1 Equivalence partitioning (K3)

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 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. during integration testing). Tests can be designed to cover partitions. Equivalence partitioning (EP) is applicable at all levels of testing.

Equivalence partitioning as a technique can be used to achieve input and output coverage. 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, 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 value on each boundary 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.

This technique is often considered an extension of equivalence partitioning and can be used on input by humans as well as, for example, on timing or table boundaries. Boundary values may also be used for test data selection.

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. 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 can either 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 that 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, which typically involves covering all combinations of triggering conditions.

The strength of decision table testing is that it creates combinations of conditions that might not otherwise 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 as 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 specified from use cases or business scenarios. A use case describes interactions between actors, including users and the system, which produce a result of value to a system user. Each use case has preconditions, which need to be met for a use case to work successfully. Each use case terminates with post-conditions, 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 sometimes alternative branches.

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, often referred to as scenarios, 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.

4.4 *Structure-based or white-box techniques (K3)* — *60 minutes*

Terms
Code coverage, decision coverage, statement coverage, structural testing, structure-based testing, white-box testing.

Background
Structure-based testing/white-box testing is based on an identified structure of the software or system, as seen in the following examples:

- Component level: the structure is that of the code itself, i.e. statements, decisions or branches.
- 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, two code-related structural techniques for code coverage, based on statements 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 (K3)

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

4.4.2 Decision testing and coverage (K3)

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. Decision testing derives test cases to execute specific decision outcomes, normally to increase decision coverage.

Decision testing is a form of control flow testing as it generates 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 (e.g. at integration level) where 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
Error guessing, exploratory testing.

Background
Perhaps the most widely practiced technique is error guessing. Tests are derived from the tester's skill and intuition and their experience with similar applications and technologies. When used to augment systematic techniques, intuitive testing can be useful to identify 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 structured approach to the error guessing technique is to enumerate a list of possible errors and to design tests that attack these errors. 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 of types of defects found.

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels.

References
4.1 Craig, 2002, Hetzel, 1998, IEEE 829
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) — 180 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)

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

5.2 Test planning and estimation (K2)

- Recognize the different levels and objectives of test planning. (K1)
- 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 829). (K2)
- Recall typical factors that influence the effort related to testing. (K1)
- Differentiate between two conceptually different estimation approaches: the metrics-based approach and the expert-based approach. (K2)
- Differentiate between the subject of test planning for a project, for individual test levels (e.g. system test) or specific test targets (e.g. usability test), and for test execution. (K2)
- List test preparation and execution tasks that need planning. (K1)
- Recognize/justify adequate 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)

- Recall common metrics used for monitoring test preparation and execution. (K1)
- Understand and interpret test metrics for test reporting and test control (e.g. defects found and fixed, and tests passed and failed). (K2)
- Summarize the purpose and content of the test summary report document according to the 'Standard for Software Test Documentation' (IEEE 829). (K2)

5.4 Configuration management (K2)

- Summarize how configuration management supports testing. (K2)

5.5 Risk and testing (K2)

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

5.6 Incident Management (K3)

- Recognize the content of the 'Standard for Software Test Documentation' (IEEE 829) incident report. (K1)
- 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 are:

- 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, user community and IT.
- Independent test specialists for specific test targets 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).
- Independent testers may be the bottleneck as the last checkpoint.
- Developers lose a sense of responsibility for quality.

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 risks – including selecting test approaches, estimating the time, effort and cost of testing, acquiring resources, defining test levels, cycles, approach, and objectives, and planning incident management;
- Initiate the specification, preparation, implementation and execution of tests, and monitor and control the execution.
- 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.
- Schedule tests.
- 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 (K2)* — *50 minutes*

Terms
Entry criteria, exit criteria, exploratory testing, test approach, test level, test plan, test procedure, 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 project or master test plan, and in separate test plans for test levels, such as system testing and acceptance testing. Outlines of test planning documents are covered by the 'Standard for Software Test Documentation' (IEEE 829).

Planning is influenced by the test policy of the organization, the scope of testing, objectives, risks, constraints, criticality, testability and the availability of resources. The more the project and test planning progresses the more information is available and the more detail that 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 (K2)

Test planning activities may include:

- Defining the overall approach of testing (the test strategy), 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, when and how the test activities should be done, how the test results will be evaluated, and when to stop testing (exit criteria).
- Assigning resources for the different tasks 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 Exit criteria (K2)

The purpose of exit criteria is to define when to stop testing, such as at the end of a test level or when a set of tests has a specific goal.

Typically exit criteria may consist of:

- 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.

5.2.4 Test estimation (K2)

Two approaches for the estimation of test effort are covered in this syllabus:

- Estimating the testing effort based on metrics of former or similar projects or based on typical values.
- Estimating the tasks by the owner of these 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.

5.2.5 Test approaches (test strategies) (K2)

One way to classify test approaches or strategies is based on the point in time at which the bulk of the test design work is begun:

- Preventative approaches, where tests are designed as early as possible.
- Reactive approaches, where test design comes after the software or system has been produced.

Typical approaches or strategies 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), check-list 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 where 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.

The selection of a test approach should consider the context, including:

- Risk of failure of the project, hazards to the product and risks of product failure to humans, the environment and the company.
- Skills and experience of the people in the proposed techniques, tools and methods.
- The objective of the testing endeavour and the mission of the testing team.
- Regulatory aspects, such as external and internal regulations for the development process.
- The nature of the product and the business.

5.3 *Test progress monitoring and control (K2)* — *20 minutes*

Terms
Defect density, failure rate, test control, test coverage, test monitoring, test report.

5.3.1 Test progress monitoring (K1)

The purpose of test monitoring is to give 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 retest 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 endeavour, 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 tested software.

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

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 its 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 are:

- Re-prioritize tests when an identified risk occurs (e.g. software delivered late).
- Change the test schedule due to availability of a test environment.
- Set an entry criterion requiring fixes to have been retested 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 that:

- 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.

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 its undesirable consequences, 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 (K1, K2)
Project risks are the risks that surround the project's capability to deliver its objectives, such as:

- Supplier issues:
  - failure of a third party;
  - contractual issues.
- Organizational factors:
  - skill and staff shortages;
  - personal and training issues;
  - political issues, such as
    - problems with testers communicating their needs and test results;
    - failure 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 that requirements can be met given existing constraints;
  - the quality of the design, code and tests.

When analyzing, managing and mitigating these risks, the test manager is following well established project management principles. The 'Standard for Software Test Documentation' (IEEE 829) 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, such as:

- Error-prone software delivered.
- The potential that the software/hardware could cause harm to an individual or company.
- Poor software characteristics (e.g. functionality, security, reliability, usability and performance).
- 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 the test planning, 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.

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. Incidents 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 a 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 development documents, test documents or 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.

A tester or reviewer typically logs the following information, if known, regarding an incident:

- Date of issue, issuing organization, author, approvals and status.
- Scope, severity and priority of the incident.
- References, including the identity of the test case specification that revealed the problem.

Details of the incident report may include:

- Expected and actual results.
- Date the incident was discovered.
- Identification or configuration item of the software or system.
- Software or system life cycle process in which the incident was observed.
- Description of the anomaly to enable resolution.
- 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 confirmation test or closed).
- Conclusions and recommendations.
- 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.

The structure of an incident report is covered in the ‘Standard for Software Test Documentation’ (IEEE 829) and is called an anomaly report.

References
5.1.1 Black, 2001, Hetzel, 1998
5.1.2 Black, 2001, Hetzel, 1998
5.2.5 Black, 2001, Craig, 2002, IEEE 829, Kaner 2002
5.3.3 Black, 2001, Craig, 2002, Hetzel, 1998, IEEE 829

5.4 Craig, 2002
5.5.2 Black, 2001 , IEEE 829
5.6 Black, 2001, IEEE 829

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 tool (K2)

- Classify different types of test tools according to the test process activities. (K2)
- Recognize tools that may help developers in their testing. (K1)

6.2 Effective use of tools: potential benefits and risks (K2)

- Summarize the potential benefits and risks of test automation and tool support for testing. (K2)
- Recognize that test execution tools can have different scripting techniques, including data driven and keyword driven. (K1)

6.3 Introducing a tool into an organization (K1)

- State the main principles of introducing a tool into an organization. (K1)
- State the goals of a proof-of-concept/piloting phase for tool evaluation. (K1)
- Recognize that factors other than simply acquiring a tool are required for good tool support. (K1)

6.1 *Types of test tool (K2)* — *45 minutes*

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

6.1.1 Test tool classification (K2)

There are a number of tools that support different aspects of testing. 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. Some commercial tools offer support for only one type of activity; other commercial tool vendors offer suites or families of tools that provide support for many or all of these activities.

Testing tools can improve the efficiency of testing activities by automating repetitive tasks. Testing tools can also improve the reliability of testing by, for example, automating large data comparisons or simulating behavior.

Some types of test tool can be intrusive in that the tool itself can affect the actual outcome of the test. For example, the actual timing may be different depending on how you measure it with different performance tools, or you may get a different measure of code coverage depending on which coverage tool you use. The consequence of intrusive tools is called the probe effect.

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

6.1.2 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
Characteristics of test management tools include:

- Support for the management of tests and the testing activities carried out.
- Interfaces to test execution tools, defect tracking tools and requirement management tools.
- Independent version control or interface with an external configuration management tool.
- Support for traceability of tests, test results and incidents to source documents, such as requirement specifications.
- Logging of test results and generation of progress reports.
- Quantitative analysis (metrics) related to the tests (e.g. tests run and tests passed) and the test object (e.g. incidents raised), in order to give information about the test object, and to control and improve the test process.

Requirements management tools
Requirements management tools store requirement statements, check for consistency and undefined (missing) requirements, allow requirements to be prioritized and enable individual tests to be traceable to requirements, functions and/or features. Traceability may be reported in test management progress reports. The coverage of requirements, functions and/or features by a set of tests may also be reported.

Incident management tools
Incident management tools store and manage incident reports, i.e. defects, failures or perceived problems and anomalies, and support management of incident reports in ways that include:

- Facilitating their prioritization.
- Assignment of actions to people (e.g. fix or confirmation test).
- Attribution of status (e.g. rejected, ready to be tested or deferred to next release).

These tools enable the progress of incidents to be monitored over time, often provide support for statistical analysis and provide reports about incidents. They are also known as defect tracking tools.

Configuration management tools
Configuration management (CM) tools are not strictly testing tools, but are typically necessary to keep track of different versions and builds of the software and tests.

Configuration Management tools:

- Store information about versions and builds of software and testware.
- Enable traceability between testware and software work products and product variants.
- Are particularly useful when developing on more than one configuration of the hardware/software environment (e.g. for different operating system versions, different libraries or compilers, different browsers or different computers).

6.1.3 Tool support for static testing (K1)

Review process support tools
Review process support tools may store information about review processes, store and communicate review comments, report on defects and effort, manage references to review rules and/or checklists and keep track of traceability between documents and source code. They may also provide aid for online reviews, which is useful if the team is geographically dispersed.

Static analysis tools (D)
Static analysis tools support developers, testers and quality assurance personnel in finding defects before dynamic testing. Their major purposes include:

- The enforcement of coding standards.
- The analysis of structures and dependencies (e.g. linked web pages).
- Aiding in understanding the code.

Static analysis tools can calculate metrics from the code (e.g. complexity), which can give valuable information, for example, for planning or risk analysis.

Modeling tools (D)
Modeling tools are able to validate models of the software. For example, a database model checker may find defects and inconsistencies in the data model; other modeling tools may find defects in a state model or an object model. These tools can often aid in generating some test cases based on the model (see also Test design tools below).

The major benefit of static analysis tools and modeling tools is the cost effectiveness of finding more defects at an earlier time in the development process. As a result, the development process may accelerate and improve by having less rework.

6.1.4 Tool support for test specification (K1)

Test design tools
Test design tools generate test inputs or the actual tests from requirements, from a graphical user interface, from design models (state, data or object) or from code. This type of tool may generate expected outcomes as well (i.e. may use a test oracle). The generated tests from a state or object model are useful for verifying the implementation of the model in the software, but are seldom sufficient for verifying all aspects of the software or system. They can save valuable time and provide increased thoroughness of testing because of the completeness of the tests that the tool can generate.

Other tools in this category can aid in supporting the generation of tests by providing structured templates, sometimes called a test frame, that generate tests or test stubs, and thus speed up the test design process.

Test data preparation tools (D)
Test data preparation tools manipulate databases, files or data transmissions to set up test data to be used during the execution of tests. A benefit of these tools is to ensure that live data transferred to a test environment is made anonymous, for data protection.

6.1.5 Tool support for test execution and logging (K1)

Test execution tools
Test execution tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language. The scripting language makes it possible to manipulate the tests with limited effort, for example, to repeat the test with different data or to test a different part of the system with similar steps. Generally these tools include dynamic comparison features and provide a test log for each test run.

Test execution tools can also be used to record tests, when they may be referred to as capture playback tools. Capturing test inputs during exploratory testing or unscripted testing can be useful in order to reproduce and/or document a test, for example, if a failure occurs.

Test harness/unit test framework tools (D)
A test harness may facilitate the testing of components or part of a system by simulating the environment in which that test object will run. This may be done either because other components of that environment are not yet available and are replaced by stubs and/or drivers, or simply to provide a predictable and controllable environment in which any faults can be localized to the object under test.

A framework may be created where part of the code, object, method or function, unit or component can be executed, by calling the object to be tested and/or giving feedback to that object. It can do this by providing artificial means of supplying input to the test object, and/or by supplying stubs to take output from the object, in place of the real output targets.

Test harness tools can also be used to provide an execution framework in middleware, where languages, operating systems or hardware must be tested together.

They may be called unit test framework tools when they have a particular focus on the component test level. This type of tool aids in executing the component tests in parallel with building the code.

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)
Coverage measurement tools can be either intrusive or non-intrusive depending on the measurement techniques used, what is measured and the coding language. Code coverage tools measure the percentage of specific types of code structure that have been exercised (e.g. statements, branches or decisions, and module or function calls). These tools show how thoroughly the measured type of structure has been exercised by a set of tests.

Security tools
Security tools check for computer viruses and denial of service attacks. A firewall, for example, is not strictly a testing tool, but may be used in security testing. Other security tools stress the system by searching specific vulnerabilities of the system.

6.1.6 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. They simulate a load on an application, a database, or a system environment, such as a network or server. The tools are often named after the aspect of performance that it measures, such as load or stress, so are also known as load testing tools or stress testing tools. They are often based on automated repetitive execution of tests, controlled by parameters.

Monitoring tools
Monitoring tools are not strictly testing tools but provide information that can be used for testing purposes and which is not available by other means.

Monitoring tools continuously analyze, verify and report on usage of specific system resources, and give warnings of possible service problems. They store information about the version and build of the software and testware, and enable traceability.

6.1.7 Tool support for specific application areas (K1)

Individual examples of the types of tool classified above can be specialized for use in a particular type of application. For example, there are performance testing tools specifically for web-based applications, static analysis tools for specific development platforms, and dynamic analysis tools specifically for testing security aspects.

Commercial tool suites may target specific application areas (e.g. embedded systems).

6.1.8 Tool support using other tools (K1)

The test tools listed here are not the only types of tools used by testers – they may also use spreadsheets, SQL, resource or debugging tools (D), for example.

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 benefit 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, and tests derived from requirements).
- Objective assessment (e.g. static measures, coverage and system behavior).
- 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 where manual testing would be better).

6.2.2 Special considerations for some types of tool (K1)

Test execution tools
Test execution tools replay scripts designed to implement tests that are stored electronically. 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 tests. 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 approach separates out the test inputs (the data), usually into a spreadsheet, and uses a more generic script that can read the test data and perform the same test with different data. Testers who are not familiar with the scripting language can enter test data for these predefined scripts.
In a keyword-driven 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).

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

Performance testing tools
Performance testing tools need someone with expertise in performance testing to help design the tests and interpret the results.

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

Test management tools
Test management tools need to interface with other tools or spreadsheets in order to produce information in the best format for the current needs of the organization. The reports need to be designed and monitored so that they provide benefit.

6.3 *Introducing a tool into an organization (K1)* — *15 minutes*

Terms
No specific terms.

Background
The main principles of introducing a tool into 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 to test the required functionality and determine whether the product meets its objectives.
- Evaluation of the vendor (including training, support and commercial aspects).
- Identification of internal requirements for coaching and mentoring in the use of the tool.

The proof-of-concept could be done in a small-scale pilot project, making it possible to minimize impacts if major hurdles are found and the pilot is not successful.

A pilot project has the following objectives:

- Learn more detail about the tool.
- See how the tool would fit with existing processes and practices, and how they 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 learn lessons from tool use.
- Monitoring tool use and benefits.

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

7. References

Standards

ISTQB Glossary of terms used in Software Testing Version 1.0

[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 829] IEEE Std 829™ (1998/2005) IEEE Standard for Software Test Documentation (currently under revision)
See sections 2.3, 2.4, 4.1, 5.2, 5.3, 5.5, 5.6

[IEEE 1028] IEEE Std 1028™ (1997) IEEE Standard for Software Reviews
See section 3.2

[IEEE 12207] IEEE 12207/ISO/IEC 12207-1996, 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 (2nd 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:

See sections 1.1, 4.5, 5.2

[Myers 1979] Myers, Glenford J. (1979) The Art of Software Testing, John Wiley & Sons:
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

Appendix A – Syllabus background

History of this document

This document was prepared during 2004–2005 by a working party comprising 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). At the time of writing (2005), ISTQB membership includes Austria, Denmark, Finland, France, Germany, India, Israel, Japan, Korea, Norway, Poland, Portugal, Spain, Sweden, Switzerland, The Netherlands, UK and USA.

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 multi-national/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 some new 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.

Appendix B – Learning objectives/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.

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 and give examples for the testing concept.

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.

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.

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:

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.)

General rules

SG1. The syllabus should be understandable and absorbable by people with 0 to 6 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)

Current content

SC1. The syllabus should include recent testing concepts and should reflect current best practice 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).

Learning Objectives

LO1. Learning objectives should distinguish between items to be recognized/remembered (cognitive level K1), items the candidate should understand conceptually (K2) and those which the candidate should be able to practice/use (K3). (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)

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 ISTQB's 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.

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.

The specific areas of the syllabus requiring practical exercises are as follows:

4.3 Specification-based or black-box techniques

Practical work (short exercises) should be included covering the four techniques: equivalence partitioning, boundary value analysis, decision table testing and state transition testing. The lectures and exercises relating to these techniques should be based on the references provided for each technique.

4.4 Structure-based or white-box techniques

Practical work (short exercises) should be included to assess whether or not a set of tests achieve 100% statement and 100% decision coverage, as well as to design test cases for given control flows.

5.6 Incident management

Practical work (short exercise) should be included to cover the writing and/or assessment of an incident report.

Index