**Certified Tester**

**Foundation Level Syllabus**

Version 2017

International Software Testing Qualifications Board



Copyright Notice

This document may be copied in its entirety, or extracts made, if the source is acknowledged.

Copyright © International Software Testing Qualifications Board (hereinafter called ISTQB®).

# Revision History

Table of Contents

Revision History 3

Table of Contents 4

Acknowledgements 6

0. Introduction to this Syllabus 7

0.1 Purpose of this Document 7

0.2 Overview 7

0.3 Examinable Learning Objectives 7

1. XXXXX 9

1.1 CHAPTER 1 10

1.1.2 XXXX 10

1.1.3 XXXX 10

1.2 XXXXX 10

1.2.1 XXXXX 10

1.2.2 XXXX 10

1.2.3 XXXX 10

1.2.4 XXXXX 10

1.2.5 XXXXX 10

2. Testing Throughout the Software Life Cycle – 190 mins. 11

2.1 Software Development Models 12

2.1.1 Sequential, iterative and incremental development models 12

2.1.2 Test approaches based on the context of the project 14

2.2 Status of Testing in Agile Projects 14

2.2.1 Four test levels (K2) 14

2.3 Test Types 20

2.3.1 Software test types (K2) 20

2.3.2 Testing types at all any level of test 21

2.3.3 Confirmation and Regression Testing (K2) 22

2.4 Maintenance Testing (K2) 22

2.4.1 Testing an existing or a new developed system (K2) 22

2.4.2 Regression testing during maintenance (K2) 23

3. Static Techniques (K2) – 165 minutes. 24

3.1 Static Techniques (K2) 25

3.1.1 Examinable Work Products for Static Techniques (K1) 25

3.1.2 Benefits of Reviews (K2) 25

3.1.3 Differences between static and dynamic techniques (K2) 26

3.2 Review Process (K3) 26

3.2.1 Activities of a Formal Review ( K3) 26

3.2.2 Roles and Responsibilities (K1) 27

3.2.3 Types of Reviews (K2) 28

3.2.4 Success Factors for Reviews (K2) 29

3.3 Static Analysis by Tools (K2) 30

3.3.1 Benefits of static analysis (K2) 30

3.3.2 Typical defects found by static analysis tools (K1) 31

4. Test Design Techniques (K3) – XXX mins. 32

4.1 The Test Development Process 34

4.1.1 test suite, test condition, test case, test step 34

Need Material here 34

4.1.2 test suite, test condition, test case, test step 34

4.2 Categories of Test Design Techniques (K2) 35

4.2.1 Characteristics, commonalities, and differences between specification-based testing, structure-based testing and experience-based testing 35

4.3 Specification-based or Black-box Techniques 36

4.3.1 Equivalence Partitioning 36

4.3.2 Boundary Value Analysis 36

4.3.3 Decision Table Testing and other combinatorial test design techniques 36

4.3.4 State Transition Testing 37

4.3.5 Use Case Testing 37

4.4 Structure-based or White-box Techniques 38

4.4.1 Statement Testing and Coverage 38

4.4.2 Branch and Decision Testing and Coverage 38

4.4.3 The value of statement and decision coverage 39

4.5 Experience-based Testing 39

4.6 Choosing Test Techniques 39

5. Test Management (K3) - XXX minutes 41

5.1 Test Organization 42

5.1.1 Test Organization and Independence 42

5.1.2 Tasks of a Test Manager and Tester 42

5.2 Test Planning and Estimation 44

5.2.1 Purpose and content of a test plan 44

5.2.2 Test Execution Schedule 45

5.2.3 Factors influencing Testing Effort 45

5.2.4 Test Estimation 45

5.2.5 Entry Criteria and Exit criteria for testing 45

5.3Test Progress Monitoring and Control 47

5.3.1 Metrics used in testing 47

5.3.2 Purpose, content, and stakeholders of test Report and control 47

5.4 Configuration Management 48

5.4.1 Configuration management in support of testing 48

5.5 Risk and Testing 48

5.5.1 Definition of Risk 48

5.5.2 Project and Product Risks 48

5.5.3 Considering risk during test planning 49

5.6 Incident Management 50

5.6.1 Structure and content of an Incident report 50

6. Tool Support for Testing - 40 Min. 52

6.1 Test Tools 53

6.1.1 Test Tool Classifications 53

6.1.1.1 Tool Support for Management of Testing and Tests 53

6.1.1.2 Tool Support for Static Testing 54

6.1.1.3 Tool Support for Test Specification 54

6.1.1.4 Tool Support for Test Execution and Logging 55

6.1.1.5 Tool Support for Performance and Monitoring 55

6.1.1.6 Tool Support for Specific Testing Needs 56

6.1.2 Benefits and Risks of Automated Test Tools 56

6.1.3 Special Considerations for Some Types of Tools 57

6.2 Effective Use of Tools 59

6.2.1 Main considerations for selecting tools 59

6.2.2 Pilot Projects for tool evaluation 59

7. Index 60

Acknowledgements

This document was produced by a team from the International Software Testing Qualifications Board Foundation Level Working Group.

The Foundation team thanks the review team and the National Boards for their suggestions and input.

This document was formally approved for release by the General Assembly of the ISTQB® on DATE, YEAR.

# Introduction to this Syllabus

## 0.1 Purpose of this Document

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

To National Boards, to translate into their local language and to accredit training providers. National Boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.

To Examination Boards, to derive examination questions in their local language adapted to the learning objectives for each syllabus.

To training providers, to produce courseware and determine appropriate teaching methods.

To certification candidates, to prepare for the certification examination (either as part of a training course or independently).

To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and articles.

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

## 0.2 Overview

The Foundation Level Overview document [ISTQB\_FL\_OVIEW] includes the following information:

* Business Outcomes for the syllabus
* Summary for the syllabus
* Relationships among the syllabi
* Description of cognitive levels (K-levels)
* Appendices

## 0.3 Examinable Learning Objectives

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Certified Tester Foundation Level. In general, all parts of this syllabus are examinable at a K1 level. That is, the candidate will recognize, remember, and recall a term or concept. The specific learning objectives at K1, K2, and K3 levels are shown at the beginning of the pertinent chapter.

# Fundamentals of Testing

Keywords

Debugging, requirement, review, test case, testing, test objective, exhaustive testing, Confirmation testing, re-testing, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test procedure, test policy, test suite, test summary report, testware

Learning Objectives for Fundamentals of Testing

1.1 What is Testing?

FL-1.1.1 (K1) Identify the major objectives of testing

FL-1.1.2 (K2) Differentiate testing from debugging

FL-1.1.3 (K2) Identify the test activities and respective tasks within the fundamental test process.

1.2 Why is Testing Necessary?

FL-1.2.1 (K2) Give reasons why testing is necessary by giving examples

FL-1.2.2 (K2) Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality

FL-1.2.3 (K2) Distinguish between errors, defects and failures.

FL-1.2.4 (K2) Distinguish between the root cause of a defect and its effects

1.3 Seven Testing Principles

FL-1.3.1 (K2) Explain the seven testing principles.

1.4 The Psychology of Testing

FL-1.4.1 (K1) Identify the psychological factors that influence the success of testing.

FL-1.4.2 (K2) Clarify the difference in the mindset required for testing and development activities.

* 1. What is Testing?

Testing is an activity to ensure the right level of quality in software. This includes positive test to demonstrate functionality works as required, and negative tests to ensure robustness is at the required level.

ISTQB defines testing as:

The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

Testing starts day one in all projects, new development or maintenance work. It includes planning what we must test with traceability from requirements to test planned and later executed. Historical a common perception of testing was that test only consist of running tests, i.e., executing the software. Actually, this is a part of testing, but not all of the testing activities.

Deciding how much testing is enough should take account of the level of risk, and project constraints such as time and budget. Testing should provide metrics to stakeholders to make informed decisions about the release of the software. See more on metrics in chapter 5.

### 1.1.1 The major objectives of testing

Testing have the following objectives:

* Validate that the system under test works as required
* Verify that all specified requirements have been fulfilled
* Gaining confidence about the level of quality
* Finding defects
* Preventing defects.
* Providing information for decision-making

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

### 1.1.2 Differentiate testing from debugging

Debugging and testing are two different activities. Test is planned and executed to obtain the objectives of testing as listed in section 1.1.1. When test is executed defects can be found.

Debugging is the development activity that a developer uses to analyze what in the software is causing the defect and removing this cause of failures. Debugging is responsibility of a developer. Confirmation testing is then executed by first the developer at component level and subsequent by a tester at system test level to ensure that the corrective actions resolve the failure.

Ordinary defects are fixed and confirmation testing is executed. Additional a regression test is executed, to ensure all other parts of the software still works as expected. The magnitude of the regression test is based on the risk of the area where the defect was identified.

1.1.3 Test activities and respective tasks within the fundamental test process

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

* Test planning, monitoring and control
* Test analysis
* Test design
* Test implementation
* Test execution
* Evaluating exit criteria and reporting
* Test closure activities

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

Test Planning, monitoring and Control

Test planning is the activity of defining the objectives of testing and the specification of test activities in order to meet the objectives and mission.

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

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

Test Analysis

Test analysis and design is the activity during which general testing objectives are transformed into tangible test conditions and test cases.

The test analysis and design activity has the following major tasks:

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

Test Design

XXX

Test Implementation

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

Test implementation and execution has the following major tasks:

* Finalizing, implementing and prioritizing test cases (including the identification of test data) o Developing and prioritizing test procedures, creating test data and, optionally, preparing test harnesses and writing automated test scripts
* Creating test suites from the test procedures for efficient test execution
* Verifying that the test environment has been set up correctly
* Verifying and updating bi-directional traceability between the test basis and test cases
* Executing test procedures either manually or by using test execution tools, according to the planned sequence
* Logging the outcome of test execution and recording the identities and versions of the software under test, test tools and testware
* Comparing actual results with expected results
* Reporting discrepancies as incidents and analyzing them in order to establish their cause (e.g., a defect in the code, in specified test data, in the test document, or a mistake in the way the test was executed)
* Repeating test activities as a result of action taken for each discrepancy, for example, reexecution 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)

Test Execution

XXX

Evaluating Exit Criteria and Reporting

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

Evaluating exit criteria has the following major tasks:

* Checking test logs against the exit criteria specified in test planning
* Assessing if more tests are needed or if the exit criteria specified should be changed
* Writing a test summary report for stakeholders

Test Closure Activities

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

Test closure activities include the following major tasks:

* Checking which planned deliverables have been delivered
* Closing incident reports or raising change records for any that remain open
* Documenting the acceptance of the system
* Finalizing and archiving testware, the test environment and the test infrastructure for later reuse
* Handing over the testware to the maintenance organization
* Analyzing lessons learned to determine changes needed for future releases and projects
* Using the information gathered to improve test maturity
  1. Why is Testing Necessary?

We test because it is a good business case. Finding defects during testing of software is cheaper than during use of a system, and the consequence are a magnitude lower in testing. If the risk for failure in software is low, or the consequence when failure appears are low, we could consider to test less exhaustive.

### 1.2.1 Give reasons why testing is necessary

We test to reduce the risk of failure in software when we use it. When we have a risk of software not working at an expected quality level, test is used to reduce this risk or confirm we have a failure we must fix before we use the software. If software has defects causing failures when it is used, the consequence can be fatal for the user and company.

Rigorous testing of systems and documentation can help to reduce the risk of problems occurring during operation. When defects found are corrected before the system is released for operational use, and these corrected have passed confirmation and regression testing, then this will lead to an increased quality of the software system.

Software testing may also be required to meet contractual or legal requirements, or industry-specific standards. (should we include reference to nay standards here?)

### 1.2.2 Quality Assurance and Testing

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, performance, efficiency, maintainability and portability).

For more information on non-functional testing see Chapter 2; for more information on software characteristics see “Software Engineering – Software Product Quality” (ISO 25010).

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 and confirmation testing has documented they now work correct.

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

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

### 1.2.3 Errors, defects and failures

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

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

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

### 1.2.4 Root cause of a defect and its effects

Defects can be analyzed to identify the root cause as to why and where they were introduced, and how to avoid similar defects to be introduced in the future. e.g. lack of domain knowledge caused a misinterpretation when implementing a part of the software. The effect of this defect might be fatal, or it might only be cosmetic.

* 1. Seven Testing Principles

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

1.3.1 Explain the seven testing principles

Principle 1 – Testing shows presence of defects

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

Principle 2 – Exhaustive testing is impossible

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

Principle 3 – Early testing

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

Principle 4 – Defect clustering

Testing effort shall be focused proportionally to the expected and later observed defect density of modules. A small number of modules usually contains most of the defects discovered during prerelease testing, or is responsible for most of the operational failures.

Principle 5 – Pesticide paradox

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

Principle 6 – Testing is context dependent

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

Principle 7 – Absence-of-errors fallacy

Finding and fixing defects does not help if the system built is unusable and does not fulfill the users’ needs and expectations

* 1. The Psychology of Testing

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

### 1.4.1 Psychological factors that influence the success of testing

People are driven by goals. Goals can be explicitly or implicitly. It has been shown that testers find more defects with defined goals (e.g. Explore the user interface for finding inconsistencies). Therefore, test leaders shall assign goals when assigning testing tasks to testers.

When providing and accepting feedback, the developer or author of a document may interpret failures as criticism against himself instead as a means to improve the product quality and reduce product risks. When testers and reviewers communicate about failures in a constructive way, they can avoid negative feelings or misinterpretation by developers and authors.

Looking for failures in a system requires curiosity, professional pessimism, a critical eye, attention to detail, good communication with peers, and experience on which to base error guessing.

The tester and test managers need good interpersonal skills because communication problems may occur, e.g. testers might be seen only as messengers of bad news.

However, there are several ways to improve communication and relationship in teams:

* Language (considering culture, religion, and gender)
* Communicate findings on the product in an objective, factual way without criticizing the person who created it, e.g., write objective and factual incident reports and review findings
* Show understanding and share the feelings of another.
* Confirm that the other person has understood what you have said and vice versa.

In addition, test managers shall consider

* Set up teams with common goals
* Intercultural differences

### 1.4.2 Tester vs. Developer mindset

The mindset to be used while testing and reviewing is different from that used while developing software.

* Developers mindset is related to construction and creativity to create software
* Testers mindset is related to challenge the software to find deviations

Good developers can switch to the mindset of testers. Benefit of applying both mindsets is identifying defects early. Switching mindset is difficult. People are biased towards what they created. Therefore, a degree of independence often makes the tester more effective at finding defects and failures.

Levels of independence can be defined as shown here from low to high:

1. Tests designed by the person(s) who wrote the software under test
2. Tests designed by another person(s) (e.g., from the development team)
3. Tests designed by a person(s) from a different internal organizational group (e.g., an independent test team) or test specialists (e.g., usability or performance test specialists)
4. Tests designed by a person(s) from a different external organization or company (e.g., outsourcing or certification by an external body)

# Testing Throughout the Software Life Cycle – 190 mins.

Keywords

Alpha testing, beta testing, black-box testing, code coverage, Commercial Off-The-Shelf (COTS), component testing, driver, field testing, functional requirement, functional testing, Impact analysis, integration, integration testing,interoperability testing, iterative development model, incremental development model, load testing, maintainability testing, maintenance testing test approach, model non-functional requirement, performance testing, robustness testing, security testing, stress testing, structural testing, stub, system testing, test environment, test level, test-driven development, user portability testing, reliability testing, usability testing, acceptance testing, validation, verification, V-model., white-box testing

Learning Objectives for Testing Throughout the Software Life Cycle

2.1 Software Development Models

FL-2.1.1 (K2) Explain the relationship between software development activities and test activities in development life cycle.

FL-2.1.2 (K1) Identify reasons why software development models must be adapted to the context

of project and product characteristics

2.2 Test Levels

FL-2.2.1 (K2) Compare the different levels of testing from the perspectives of objectives, test objects, test targets related work products, responsibilities, and types of defects and failures to be identified.

2.3 Test Types

FL-2.3.1 (K2) Compare functional, non-functional, structural testing

FL-2.3.2 (K1) Recognize that functional and structural tests occur at any level

2.4 Change Related Testing (K2)

FL-2.4.1 (K2) Compare the purpose of confirmation testing and regression testing

2.5 Maintenance testing (K2)

FL-2.5.1 (K2) Identify triggers for change-related testing.

FL-2.5.2 (K2) Describe the role of impact analysis and regression testing.

## 2.1 Software Development Models

Over time different types of development models have been used. Waterfall is an example of sequential model. It is one of the first models developed to describe development activities aiming at transforming requirements into a final product. Another example of sequential model is the V–Model.

Although variants of the V-model exist, within ISTQB the V-model has four test levels. Each of the four test levels on the right of the V-model corresponds to development activities on the left in the V-model.

The four test levels used in this syllabus are:

* Component 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.

Incremental development is the process of establishing requirements, designing, building and testing a system, done as a series of shorter development cycles. An increment, added to others developed previously, forms a growing partial system. Those increments should be tested.

Iterative development breaks the project usually in a large number of iterations. An iteration is a complete development loop of defining, building and testing. The resulting executable product produced by an iteration may be tested at several test levels during each iteration.

Examples of incremental and iterative development models are Rapid Application Development (RAD) and Rational Unified Process (RUP) and agile development models. One can distinguish number of different methods and frameworks within Agile like eXtreme Programing, which is a development model, and Scrum, which is a management framework for doing agile projects.

### 2.1.1 Relationship between software and test activities

Testing does not exist in isolation and test activities relates to corresponding software development activities. Different development life cycle models need different approaches to testing.

In any life cycle model, there are number of characteristics of testing – development relations like:

* For every development activity there is a corresponding testing activity
* Each test level has test objectives specific to related development level.
* The analysis and design of tests for a given test level should begin during the corresponding development activity. Start as early as possible with testing activities.
* Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle.

### 2.1.2 Software development models in context (K1)

Software development models must be adapted to the context of project and product characteristics. Based on the project goals and the product risk assessment, a model is set up. It is possible to apply different models depending on the type of software. For example, a small internal administrative system will have a different approach to testing in comparison with a safety-critical system like an automobile’s brake control system. This will also influence a test approach.

It is possible to combine or reorganize development and test levels 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

Testing does not exist in isolation. The test activities relate to the corresponding software development activities. Different software development life cycle models (SDLC) have different approaches to testing. Identify for every SDLC the needed test levels and test types.

For each of the four test levels the following characteristics can be identified:

* Objectives for a specific test level;
* Work product(s) (test basis) needed for deriving test cases;
* Test object i.e., what is being tested;
* Typical defects and failures found during testing;
* Test targets to be met;

Besides the general test objectives, each test level has its own objectives.

Different viewpoints in testing take different objectives into account.

For example, in 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.

During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

Each test level needs its own specific test environment for an optimal result.

### 2.2.1 Different levels of testing (K2)

Component Testing

Component testing searches for defects in components. By testing components, one verifies the functioning of any component separately. It is possible to test components in isolation from the rest of the system, depending on the context of the development lifecycle and the system. As test basis for deriving test cases for component testing, use i.e. one of the following work products:

* Component requirements;
* Detailed design;
* Data model;

Typically, component testing occurs with access to the tested code. Typical test objects are:

Components, units, modules, program;

* Objects;
* Classes;
* Database modules;

Component testing may include testing of functionality and specific non-functional characteristics. Such as resource-behavior (e.g., searching for memory leaks) or robustness testing, as well as structural testing (e.g., decision coverage).

A common approach is that the developer who wrote the code usually does the component testing. The developer fixes the defects when found, without formally managing these defects. Another approach to component testing is test-driven development (TDD) or test-first approach. This highly iterative approach prepares and automate test cases before coding. The first step is developing test cases, then building and integrating small pieces of code, and executing the component tests correcting any defects found. This iteration cycle stops when all component tests pass.

Component testing usually uses a development environment such as a unit test framework. Tools like debuggers, stubs, drivers and simulators like mock objects are common in this kind of test environment.

When meeting the specified test target, the test process will move on to the next test level.

Integration testing

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

Integration testing searches for defects in interfaces and/or interactions:

* Between components
* With different parts of a system, such as:
* The operating system;
* File system and hardware;
* Interfaces between systems;

There may be more than one level of integration testing.

The size of the tested test objects may vary depending on the test phase:

* Component integration testing, done after component testing in a development environment, tests the interactions between software components.
* System integration testing, done after system testing, tests the interactions between different systems or between hardware and software. In this case, the developing organization may control only one side of the interface. This might be considered as a risk.
* Business processes implemented as workflows may involve multiple systems on different platforms. Cross-platform issues may be significant and time consuming to solve.

The greater the scope of integration, the more difficult it becomes to isolate defects in a specific component or system. This may lead to increased risk and additional time for solving integration problems.

As test basis for deriving test cases for integration testing, use i.e. one of the following work products:

* Software and system design;
* Architecture;
* Workflows;
* Use cases;

Typical test objects are:

* Subsystems;
* Database implementation;
* System infrastructure;
* Interfaces;
* System configuration and configuration data;

Systematic integration test strategies may be based on:

* the system architecture, such as top-down and bottom-up;
* functional tasks;
* transaction-processing sequences;
* some other aspect of the system or components;

In order to ease fault isolation and detect defects early, integration should normally be incremental rather than “big bang”. Testing of specific non-functional characteristics, e.g. performance may be included in integration testing.

When working in agile projects applying the mentioned strategies with continuous integration implemented is important.

At the component integration level, developers primarily do this testing. At the system integration level, testers primarily do this testing. Ideally, testers should understand the implemented architecture. If planning for integration tests before components are built, those components can be built in the order required for most efficient testing.

Doing integration testing at the component level, the development environment is the most obvious. When doing system integration testing, the environment must have interfaces to the other related systems. Organizing this type of testing can be challenging in large organizations. Those related systems have probably different owners and test teams. Maybe third parties are involved. Access authorization, security and use of consistent test data sets can be complex. Organizing this kind of integration test may take a lot of time in relation to the execution of the test.

System Testing

System testing searches for defects in the behavior of a whole system/product. System testing should investigate the functional and non-functional requirements of the system, and data quality characteristics. Testers also need to deal with incomplete, implied 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, by creating a decision table for combinations of causes and 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.

For system testing, use i.e. one of the following work products as test basis for deriving test cases:

* System and software requirement specification
* Use cases
* Functional specification
* Risk analysis reports
* Business processes
* Models of system behavior
* Epics / User stories

Typical test objects are:

* Systems
* User and operation manuals
* System configuration and configuration data

The responsibility of system testing is not with the developers anymore. An independent test team often carries out system testing. More independence helps to be more effective at finding defects and failures.

In system testing, the test environment must be suitable to test the system in order to minimize the risk of environment-specific failures. The test environment is often a scaled down environment of the production environment. For testing non-functional attributes like efficiency performance often a specific test environment is needed as close as possible to the production environment.

Acceptance Testing

The goal of acceptance testing is to establish confidence in the system, parts of the system or in specific non-functional characteristics of the system. Finding defects is not the main focus of 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.

For acceptance testing, use i.e. one of the following work products as test basis for deriving test cases:

* User requirements
* System requirements
* Use cases
* Business processes
* Risk analysis reports
* Epics / User stories

Typical test objects are:

* Business processes on fully integrated system
* Operational and maintenance processes
* User procedures
* Forms
* Reports
* Configuration data

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

Acceptance testing may occur at various times in the lifecycle, for example:

* A COTS software product may be acceptance tested when it is installed or integrated
* Acceptance testing of a new functional enhancement may come before system integration testing

Typical forms of acceptance testing include the following:

User acceptance testing: 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 includes:

* Testing of backup/restore
* Disaster recovery
* User management
* Maintenance tasks
* Data load and migration tasks
* Periodic checks of security vulnerabilities

For acceptance testing a test environment should be a copy based of the software and hardware in production or as close to it as possible.

Contract and regulation acceptance testing: Testing against a contract’s acceptance criteria for producing custom-developed software is called contract acceptance testing. Acceptance criteria should be defined when the parties agree to the contract. Regulation acceptance testing is testing the system under test against any regulations, such as government, legal or safety regulations. The system under test must adhere to those specified regulations.

A test environment with integration to third parties is suitable to perform this test.

Alpha and beta (or field) testing: Software development teams of market software (COTS) often want to get feedback from potential or existing customers before the product is released to the market.

A selected group of expert users of the product performs the Alpha test at the developing organization’s site. A selected group of users receives an invitation to perform the Alpha test. The developers are not the testers.

Customers or potential customers at their own locations perform beta testing or field-testing.

2.3 Test Types

Test types are a group of activities aimed at verifying the software system, or part of a system, based on a specific test objective, which could be one of the following:

* A function to be performed by the software
* A non-functional quality characteristic, such as reliability or usability
* The structure or architecture of the software or system

A model of the software may be developed and/or used in:

* Functional testing like a process flow model, a state transition model or a plain language specification;
* Non-functional testing like a performance model, usability model, security threat model;
* Structural testing like a control flow model or menu structure model;

### 2.3.1 Comparison of functional, non-functional and structural test types (K2)

Functional testing

Functional testing considers the external behavior of the software. The functions are “what” the system does. The functions that a system, subsystem or component shall perform are described in its work products. Examples of work products are:

* Requirement specifications
* Use cases
* Functional specifications
* Epics / user stories

The basis for functional tests are the functions and features described in documents. Requirement-based techniques may be used to derive test conditions and test cases from the described functionality of the software or system.

Functional testing may be considered as the external behavior of the software (black-box testing). Another type of functional testing, interoperability testing, evaluates the capability of the software product to interact with one or more specified components or systems.

Non-functional testing

The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified objectively, such as response times for performance testing or be judged to conform to requirements like, such a system is reliable over a given period of time (i.e. mean time between failure). Non-functional tests refer to a quality model, e.g. the one defined in ISO 25010. It is the testing of “how” the system works. Non-functional testing considers the external behavior of the software or component and in most cases uses black-box test design techniques to accomplish that. Non-functional testing includes, but is not limited to, performance efficiency testing, compatibility testing, security testing, usability testing, maintainability testing, reliability testing and portability testing

Structural testing

Use structural testing to analyze the internal structure of components, systems and architecture. Structural testing approaches can also be applied at system, system integration or acceptance testing levels (e.g., to business models or menu structures).

Structural techniques used after specification-based techniques is a good approach, in order to help measure the thoroughness of testing through assessment of coverage of a type of structure. Coverage is expressed as a percentage of the items being covered. If coverage is not 100%, then design more tests to test the missed items to increase coverage of course this must be based on the product risk.

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.

### 2.3.2 Occurrence of test types in test levels

At any test level, it is possible to perform any test type mentioned in section 2.3.1. It is obvious to perform functional or non-functional testing during one of the four test levels. Other test types to perform are e.g.,

* security testing on a component test or system test level
* usability testing on a system test level

Applying test types as early as possible helps to find defects before the last test phase when it might be more common to execute those tests.

## 2.4 Change related testing

To confirm the removal of a defect, the execution of confirmation testing is necessary. Confirmation testing checks change(s) if defects found are successfully fixed and have no possible side effects. Regression testing is the repeated testing of an already tested program after modification, to discover any defects introduced or uncovered because of the change(s). These defects may be either in the tested software, or in another related or unrelated software component. After the software or its environment is changed, execution of a regression test is necessary to verify if the quality of the unchanged software is still the same as before the change.

Changes in the software relate to two test types:

* Confirmation testing: After a defect is detected and fixed, the software should be tested again. A confirmation test executes a test case, which failed the last time. Confirmation testing is done to confirm that the original defect (functional, non-functional or structural) has been successfully removed. Tests should be repeatable if they are to be used for confirmation testing and to assist regression testing.

Debugging (locating and fixing a defect) is a development activity, not a testing activity.

* Regression testing: Regression testing covers the risk of not finding defects in software that was working previously (unchanged software). Regression testing may be performed at all test levels, and includes functional, non-functional and structural testing. Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation.

## 2.5 Maintenance Testing

Once deployed, a software system is often in service for years or decades. Often corrections, changes or extensions on features take place during the maintenance lifecycle of the system, its configuration data, or its environment. The test process between an existing or new system may not differ much. However, often the amount of testing during maintenance is more than testing a new system. Where testing a new system the focus is on testing the whole system as much as possible based on a product risk analysis. The focus for maintenance testing is testing the system initiated by changes as well as testing the non-changed parts. Depending on the changes, maintenance testing generally involves change related testing and may be done at any or all test levels and for any or all test types.

### 2.5.1 Triggers for maintenance testing (K2)

The planning of releases in advance is crucial for successful maintenance testing. For maintenance testing, make a distinction between planned releases and non-planned releases (hot fixes). Triggers for maintenance testing on an existing operational system are:

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

The different types of planned testing activities are:

Perfective – add/improve (non-) functional requirements: The system is modified e.g. for improving the performance. Some regression testing regarding time or resource behavior is needed to test if the system functions at it did before the change with a better performance

Preventive – preventing future problems:By monitoring the system future problems may be detected. Pro-actively make changes to the system will help to prevent a failure of the service

Adaptive – adjust the software to the environment: This could be changes in the software needed to communicate with another or updated environment. Testing if the communication still works is needed.

Corrective – fixing defects: After fixing the defects, confirmation testing is needed

The different types of non-planned testing activities are:

Corrective – emergency fixes of defects: When a service fails in production, emergency fixes are needed and also the related testing on different test levels under high time pressure.

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

Retirement of the system: Maintenance testing for the retirement of a system may include the testing of data migration or archiving if long data-retention periods are required. Testing restore/retrieve procedures after archiving for long retention periods.

Any of the test types can be applied during maintenance testing just like in testing a newly developed system.

### 2.5.2 The role of regression testing and impact analysis in maintenance

Besides testing the changed part of the system, maintenance testing includes regression testing to the unchanged parts of the system.

The scope of maintenance testing relates to:

* the risk of the change
* the size of the existing system
* the size of the change.

Impact analysis determines if the specified requirements affects the existing system. The impact analysis helps to decide how much regression testing is needed. The impact analysis may be used to determine the regression test suite. If the changes are in a high-risk area of the system under test, the regression set for this test will increase.

Doing an impact analysis can be difficult if:

* specifications are out of date or missing
* cases are not documented or out of date
* development documentation is missing or out of date
* people involved don’t have domain and/or system knowledge.

# Static Techniques – 165 minutes.

Keywords

Compiler, complexity, control flow, data flow, dynamic testing, entry criteria, formal review, informal review, inspection, metrics, moderator, peer review, reviewer, scribe, static code analysis, static testing, technical review, walkthrough, Compiler, complexity, control flow, data flow, static code analysis

Learning Objectives for Static Testing Techniques

3.1 Static Techniques

LO-3.1.1 (K1) Recognize software work products that can be examined by different static techniques

LO-3.1.2 (K2) Describe, using examples, the value of using static techniques

LO-3.1.3 (K2) Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software life cycle

3.2 Review Process

LO-3.2.1 (K2) Summarize the activities of a formal review

LO-3.2.2 (K1) Identify the roles and responsibilities in a formal review

LO-3.2.3 (K2) Explain the differences between different types of reviews: informal review, walkthrough, technical review and inspection

LO-3.2.4 (K3) Find defects in a specification by a review

LO-3.2.5 (K2) Explain the factors that may contribute to a successful review

## 3.1 Static Techniques

In comparison to dynamic testing, which requires the execution of software, static testing techniques use manual examination (reviews) or automated analysis (static analysis) of the code and other documentation, without the execution of code.

Static techniques, especially reviews, are the most widespread activity for early detection of defects. There are various work products that should be examined by static techniques to gain benefits, which can’t be obtained with dynamic testing.

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

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

### 3.1.1 Examinable Work Products for Static Techniques

Static techniques comprise reviews, which are a manual activity, and tool-supported static analysis. The main manual activity is to examine a work product and provide relevant comments. Any software work product can be reviewed, including but not restricted to:

* Requirements specifications
* Design specifications
* Code
* Test plans
* Test specifications
* Test cases
* Test scripts
* User guides
* Web pages

### 3.1.2 Benefits of Static Techniques

Benefits of static techniques include

* Early defect detection and correction (prior to test execution),
* It identifies defects not easily found by dynamic testing
* Development productivity inclusive maintainability of code and design improvements,
* Reduced development timescales,
* Reduced testing cost and time,
* Lifetime cost reduction,
* Fewer defects and
* Improved communication

### 3.1.3 Differences between static and dynamic techniques (K2)

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

Typical defects that are easier to find in reviews than in dynamic testing include e.g.:

* Deviations from standards
* Requirement defects
* Design defects
* Insufficient maintainability
* Incorrect interface specifications
* Security vulnerabilities

## 3.2 Review Process

The different types of reviews vary from informal to formal. Informal is characterized by no written instructions for reviewers. Formal is characterized by team participation, documented results of the review, and documented procedures for conducting the review. The formality of a review process is related to factors such as the maturity of the development process, any legal or regulatory requirements or the need for an audit trail.

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

### 3.2.1 Activities of a Formal Review (K2)

A typical formal review has the following main activities:

1. Planning

* Defining the review criteria
* Selecting the personnel
* Allocating roles
* Defining the entry and exit criteria for more formal review types (e.g., inspections)
* Selecting which parts of documents to review
* Checking entry criteria (for more formal review types)

1. Kick-off

* Distributing documents
* Explaining the objectives, process and documents to the participants

1. Individual preparation

* Preparing for the review meeting by reviewing the document(s)
* Noting potential defects, questions and comments

1. Examination/evaluation/recording of results (review meeting)

* Discussing or logging, with documented results or minutes (for more formal review types)
* Noting defects, making recommendations regarding handling the defects, making decisions about the defects
* Examining/evaluating and recording issues during any physical meetings or tracking any group electronic communications

1. Rework

* Fixing defects found (typically done by the author)
* Recording updated status of defects (in formal reviews)

1. Follow-up

* Checking that defects have been addressed
* Gathering metrics
* Checking, if exit criteria are met (for more formal review types)

### 3.2.2 Roles and Responsibilities in a formal review

A typical formal review will include the roles below:

###### Project manager

* Decides on the execution of reviews
* Allocates time in project schedules and
* Determines if the review objectives have been met

###### Moderator

* Plans the review
* Leads the meeting
* Performs follow-up after the meeting
* Is often the person upon whom the success of the review rests
* If necessary, the moderator may mediate between the (reviewers’) various points of view.

###### Author:

* the writer or person with chief responsibility for the document(s) to be reviewed.

###### Reviewers (also called checkers or inspectors)

* individuals with a specific technical or business background
* Identify and describe findings (e.g., defects) in the product under review after the necessary preparation
* Should be chosen to represent different perspectives and roles in the review process
* Should take part in any review meeting that requires their expertise.

###### Scribe (or recorder)

document all the issues, problems and open points that were identified during the meeting.

### 3.2.3 Types of Reviews

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

##### Informal Review

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

##### Walkthrough

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

##### Technical Review

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

##### Inspection

* Main purpose: finding defects and evaluating document quality
* Led by trained moderator (not the author)
* Usually peers are reviewer
* Defined roles
* Collecting metrics
* Formal process based on rules and checklists
* Specified entry and exit criteria for acceptance of the software product
* Pre-meeting preparation
* Inspection report including list of findings
* Formal follow-up process (with optional process improvement components)
* Optional reader

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

### 3.2.4 Finding defects in specifications using reviews (K3)

The main objective of reviews is to find major defects. This depends on how the reviewers do their work. Various factors influence this. Besides the appropriate review type, in case of a formal review the adherence of the process and using the defined roles there are further preconditions which influence the work of the reviewer.

* Reviewers must be qualified, i.e. they must be able to understand the document under review as well as its background.
* Reviewers need enough time to read the document itself, background documents and check possible checklists. Reading means reading in order to understand and checking against the background.
* There must be a nonthreatening atmosphere (i.e. a reviewer must not fear to be criticized for “idiotic” comments if something was misunderstood)

Defects to be found are

* “wrong”, i.e. the solution works differently from required or expected working,
* “extra”, i.e. the solution does unnecessary or not required things, or
* “missing”, i.e. some required features do not work or does not work in all necessary cases.

The meeting must concentrate on important (major) issues. A reviewer should sort out any minor points and report them outside of the review meeting, for example, as written comments.

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

### 3.2.5 Success Factors for Reviews

Success factors for reviews include:

* Each review has clear predefined objectives
* Involve the right people for the review objectives
* Testers are valued reviewers. They contribute to the review and also learn about the product which enables them to prepare tests earlier
* Defects found are welcomed and expressed objectively
* All involved persons deal with people issues and psychological aspects (e.g., making it a positive experience for all parties especially the author)
* Conduct the review in an atmosphere of trust; don’t use the outcome for evaluating the participants
* Apply review techniques that are suitable to achieve the objectives and to the type and level of software work products and reviewers
* Use Checklists or roles if appropriate to increase effectiveness of defect identification
* Arrange training 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

# Test Design Techniques – XXX mins.

Keywords

Test suite, test case, test condition, test step, test script, traceability, Black-box test design technique, specification-based test design technique, experience-based test design technique, test design technique, white-box test design technique, structure-based test design technique, Boundary value analysis, decision table testing, equivalence partitioning, state transition testing, use case testing, Code coverage, decision coverage, statement coverage, structure-based testing, white-box testing, exploratory testing, checklist-based testing, fault attack, error guessing

Learning Objectives Test Design Techniques

4.1 Test Design and Implementation

FL-4.1.1 (K2) Differentiate test suite, test condition, test case and test step

FL-4.1.2 (K2) Evaluate the quality of test cases in terms of clear traceability to the requirement and expected results

4.2 Categories of Test Design Techniques

FL-4.2.1 (K2) Explain the characteristics, commonalities, and differences between specification-based testing, structure-based testing and experience-based testing

4.3 Specification-based or Black-box Techniques

FL-4.3.1 (K3) Write test cases including coverage measurement from given requirements (text or model) using equivalence partitioning technique.

FL-4.3.2 (K3) Write test cases including coverage measurement from given requirements (text or model) using boundary value analysis technique.

FL-4.3.3 (K3) Write test cases including coverage measurement from given requirements (text or model) using decision table technique.

FL-4.3.4 (K3) Write test cases including coverage measurement from given requirements (text or model) using state transition diagrams/tables technique.

FL-4.3.5 (K3) Write test cases from given use cases to cover base path, alternate paths and exceptions.

FL-4.3.6 (K2) Understand Pairwise combination testing as a test technique.

4.4 Structure-based or White-box Techniques (K3)

LO-4.4.1 (K2) Explain the concepts of statement coverage

LO-4.4.2 (K2) Explain the concepts of decision coverage

LO-4.4.3 (K2) Exemplify the value of statement and decision coverage

4.5 Experience-based Testing (K3)

LO-4.5.1 (K2) Explain with example the concept of Error Guessing

LO-4.5.2 (K2) Explain with example the concept Exploratory Testing.

LO-4.5.3 (K2) Explain with example the concept of  Checklist-based testing

## 4.1 The Test Development Process

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

During test analysis testers implement the detailed test approach and select the test design techniques to use based on, among other considerations, the identified risks (see Chapter 5 for more on risk analysis).

During test design testers develop the test cases and test data.

During test implementation the tester develops, implements, prioritizes and organizes the test cases to make them ready for execution.

### 4.1.1 test suite, test condition, test case, test step

During test analysis, the tester analyzes the test basis 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).

During test design the tester specifies test cases and test data. A test case consists of a set of input values, execution preconditions, expected results and execution postconditions, defined to cover a certain test objective(s) or test condition(s).

The execution sequence in a test case can be further detailed by breaking down a test case into test steps. The test steps specify the sequence of actions for the execution of a test. If tests are run using a test execution tool, the sequence of actions is specified in a test script.

Test cases may be ordered into test suites. There is then a smaller number of test suites than the number of test cases, making overview easier.

### 4.1.2 test suite, test condition, test case, test step

Establishing traceability from test cases to test conditions and from test conditions back to the specifications and requirements enables both effective impact analysis when requirements change, and determining requirements coverage for a set of tests.

The tester should produce expected results 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 be defined prior to test execution.

## 4.2 Categories of Test Design Techniques (K2)

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

### 4.2.1 Characteristics, commonalities, and differences between specification-based testing, structure-based testing and experience-based testing

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

White-box test design techniques analyze the structure of the test object. In practice this is mostly the code. Black-box and white-box testing may also be combined with experience-based techniques. These leverage the experience of developers, testers and users to determine what should be tested.

Common characteristics of black-box / specification-based test design techniques include:

Common characteristics of specification-based test design techniques include:

* Use formal or informal models to specify
  + the problem to be solved,
  + the software design or
  + its components
* Test cases can be derived systematically from these models

Common characteristics of white box / structure-based test design techniques include:

* They use information about how the software is constructed to derive the test cases (e.g., information about the detailed design and code)

The extent of code coverage can be measured for existing test cases, and further test cases can be derived systematically to increase coverage

Common characteristics of experience-based test design techniques include:

* They use the knowledge and experience of people to derive the test cases
* Main sources of information:
* Knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment.
* Knowledge about likely defects and their distribution.

The international standard ISO/IEC/IEEE 29119-4 contains descriptions of test design techniques.

## 4.3 Specification-based or Black-box Techniques

### 4.3.1 Equivalence Partitioning

Equivalence partitioning divides data into groups that are expected to lead to similar behavior, so they are likely to be processed in the same way. There are equivalence partitions (or equivalence classes) for both valid and invalid values.

* Valid values are values that should be accepted. An equivalence class containing valid values is also called “valid equivalence class”.
* Invalid values are values that should be rejected. An equivalence class containing invalid values is also called “invalid equivalence class”.
* Partitions can be identified for any data, e.g., inputs, outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing).
* The equivalence classes for a data element taken together must cover all possible data values for this data element.
* When invalid equivalence classes are used in test cases, they should be used individually, i.e. not be combined with other invalid equivalence classes.

Test cases should cover all valid and invalid partitions. Coverage is measured as the percentage of defined equivalence classes used in designing test cases. Equivalence partitioning is applicable at all levels of testing.

### 4.3.2 Boundary Value Analysis

Behavior at the edge of each equivalence partition is more likely to be incorrect than behavior within the partition, so boundaries are an area where testing is likely to catch 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 should cover both valid and invalid boundary values.

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 help in determining the interesting boundaries.

This technique is an extension of equivalence partitioning. It can also be used to extend other black-box test design techniques. However, it can only be used for numerical data. It is useful for equivalence classes for user input as well as, for example, on time ranges (e.g., time out, transactional speed requirements) or table ranges (e.g., table size is 256\*256).

Coverage is measured as the percentage of defined boundary values used in designing test cases.

### 4.3.3 Decision Table Testing

Combinatorial test design techniques are useful for testing system requirements and system designs that contain logical conditions. Two main approaches are decision table testing and pair-wise testing (Section 4.3.6).

Decision tables are a good way to record complex business rules that a system must implement. When creating decision tables, the tester identifies conditions and actions of the system. The input conditions and actions are usually listed in such a way that they must be true or false (Boolean). Otherwise they may be discrete values (e.g., red, green, yellow). The columns of the decision table contain the triggering conditions, usually a combination of true and false for all input conditions, and the resulting actions. Each column corresponds to a business rule that defines a unique combination of conditions and which result in the execution of the actions associated with that rule.

In the worst case, the decision table may have one column for every possible combination of conditions. The number of columns can be reduced by deleting impossible combinations and combinations where one or more parameters can be ignored.

The common coverage standard with decision table testing is to have at least one test per column in the table. This typically involves covering all combinations of triggering conditions.

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

### 4.3.4 State Transition Testing

A system may exhibit a different response depending on current conditions or previous history (its state). In this case, a state transition diagram may show that aspect of the system. It allows viewing the software in terms of its states, transitions between states, the inputs or events that trigger state changes (transitions) and the actions resulting from those transitions.

The states of the test object are separate, identifiable and finite in number. A state transition 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. 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).

Coverage is usually measured as the percentage of identified states or state transitions used in designing test cases.

### 4.3.5 Use Case Testing

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

Test cases combine flows through the use cases. They start with the first flow and then use different flows. Use case test coverage is measured as the percentage of defined use case flows used in designing test cases. Use case testing does not require specific data values. The only requirement is that input values trigger the intended flows.

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

Coverage is usually measured as the percentage of identified use case flows used in designing test cases

### 4.3.6 Pair-Wise testing

Another combinatorial testing technique is using pairwise combinations. Test cases contain all combinations of each pair of inputs, but not all combinations of inputs. The number of test cases may be much smaller than with decision tables.

For example, if there are three parameters, all combinations of parameters 1 and 2, 1 and 3, and 2 and 3 are included, but not all combinations between all three parameters. The technique is used for cases where programs might implement wrong behavior for pairs of values but where possible value interaction is unknown. The technique is not strong enough if there must be found wrong interactions between more than two parameters.

This technique usually requires using tools. The technique is very useful for platform testing and testing parameter settings.

## 4.4 Structure-based or White-box Techniques

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

* Component level: the structure of a software component, i.e., statements, decisions, branches or even distinct paths
* Integration level: the structure may be a call tree (a diagram in which modules call other modules) or a description of the data flows
* System level: the structure may be a menu structure, business process or web page structure

This section discusses two code-related structural test design techniques for code coverage, based on statements, branches and decisions. Instead of code, a control flow diagram may be used to visualize the alternatives for each decision. In that case, covering the branches in the diagram covers decisions in the code. Code coverage intends to execute every executable part of the code during testing. In principle, the quality of code not executed during dynamic testing is unknown.

In practice, white-box testing is mostly used in lower test levels.

Branch and decision coverage denote the same technique. Coverage measures the percentage of decision outcomes (e.g., the True and False options of an IF statement) that have been exercised by a test.

### 4.4.1 Statement Testing and Coverage

Statement coverage aims to execute the executable statements in the code. In component testing, statement coverage for a test suite measures the percentage of executable statements that have been exercised. The statement testing technique derives test cases to execute specific statements, normally to increase statement coverage.

### 4.4.2 Decision Testing and Coverage

Decision coverage aims to execute the different decision outcomes in the code. Branch coverage tries to execute control flow branches in a control-flow diagram. Branches originate from decision points in the code and show the transfer of control to different locations in the code.

Decision testing is a form of control flow testing as it follows a specific flow of control through the decision points. Decision coverage is stronger than statement coverage; 100% decision coverage guarantees 100% statement coverage, but not vice versa. Decision coverage follows up even execution of implied branches / control flows. Statement coverage is only concerned with explicitly given branches / control flows.

### 4.4.3 The value of statement and decision coverage

Both statement and branch/decision techniques will execute all executable statements in the code. This makes sure there is no untested code in the system. Branch/decision coverage additionally executes control flow branches that lead to no executable statements. In case the programmer forgot to code something in an otherwise empty branch, this may be found.

## 4.5 Experience-based Testing

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

## 4.5.1 Error Guessing (K2)

## A commonly used experience-based technique is error guessing. Generally testers anticipate defects based on experience with

## how the application has worked in the past,

## experience with working with the developers and knowing the sorts of mistakes they make and

## experience about problems in other applications.

## 4.5.2 Exploratory Testing (K2)

## Another approach to do experience-based testing is exploratory testing. Exploratory testing is concurrent test design, test execution, test logging, evaluation and learning. Test cases are immediately executed after designing them. The tester then gets new test ideas from observing the results. This technique is based on a test charter containing test objectives, and carried out within predefined time-boxes, in order to control the time used. 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.

## In order to be manageable, experience based testing requires thorough documenting and reporting. Any applicable coverage criteria for other techniques may be followed up.

## 4.5.3 Checklist based-testing (K2)

## A structured approach is to use a checklist to enumerate a list of possible situations to test and to design tests that cover them. Such checklists can be built based on experience, available defect and failure data, and from common knowledge about what is important for the software user and why software fails. Other similar systems may also provide checklists for enhancement.

The choice of test techniques depends on factors like the type of system, the test level, regulatory standards, level and types of risk and other factors. When creating test cases, testers generally use a combination of test techniques to ensure adequate coverage of the test object. Black-box test design techniques should always be chosen. White-box techniques are most applicable for lower-level testing. Experience-based techniques can be used to augment the test. Equivalence partitioning is used to cover data variation and possible invalid data. Boundary value analysis is used for numerical input data. Decision table testing is used where logic is involved. Pairwise combination testing is used where decision tables get too complicated or where too many inputs exist with unknown interactions. State-transition testing is useful where states are important or where a system depends on input or execution history. No test technique precludes application of any other test technique.

# Test Management – XXX mins.

Keywords

Tester, test leader, test manager, test approach, test strategy, defect density, failure rate, test control, test monitoring, test summary report, configuration management, version control, risk, product risk, project risk, risk-based testing, incident logging, incident management, incident report

Learning Objectives for Test Management

5.1 Test Organization (K2)

FL-5.1.1 (K2) Explain the benefits and drawbacks of independent testing for various project situations and maintenance and support organizations

FL-5.1.2 (K1) Identify the tasks of a test manager and tester

5.2 Test Planning and Estimation (K3)

FL-5.2.1 (K2) Summarize the purpose and content of a test plan

FL-5.2.2 (K2) Differentiate between different test approaches

FL-5.2.3 (K2) Give examples of adequate entry and exit criteria for specific test levels and test types

FL-5.2.4 (K2) Explain how to schedule test execution for a given set of test cases, considering prioritization, and technical and logical dependencies.

FL-5.2.3 (K1) Identify the factors that influence the effort related to testing

FL-5.2.4 (K2) Explain the difference between the two estimation approaches: the metrics-based approach and the expert-based approach

FL-5.2.5 (K2) Give examples of adequate entry and exit criteria for specific test levels and test types

5.3 Test Progress Monitoring and Control (K2)

FL-5.3.1 (K1) Recall metrics used in testing

FL-5.3.2 (K2) Summarize the purpose, content, and stakeholders of a test report

5.4 Configuration Management (K2)

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

5.5 Risks and Testing (K2)

FL-5.5.1 (K1) Define risk using likelihood and impact

FL-5.5.2 (K2) Distinguish between the project and product risks

FL-5.5.3 (K2) Describe, using examples, how project and product risks may thoroughness and scope of testing

5.6 Incident Management (K3)

FL-5.6.1 (K3) Write an incident report covering events that require investigation during testing.

## 5.1 Test Organization

### 5.1.1 Test Organization and Independence

A certain degree of independence (avoiding the author bias) often makes the tester more effective at finding defects and failures. Independence is not, however, a replacement for familiarity, and developers can efficiently find many defects in their own code. The effectiveness of finding defects by testing and reviews can be improved by using independent testers. Several levels of independence can be defined from low level of independence to high level of independence as follows:

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

For large, complex or safety critical projects, it is usually best to have multiple levels of testing, with some of the levels tested 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 test 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 of test independence include:

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

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

### 5.1.2 Tasks of a Test Manager and Tester

In this syllabus two test positions are covered, test manager 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.

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.

Typical test manager tasks may include:

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

Typical tester’s 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
* Designing and implementing test cases at various test levels
* Execute 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

### 5.2.1 Purpose and content of a test plan

The test plan documents test planning activities for development and maintenance projects. Planning is influenced by the test policy and test strategy of the organization, the scope of testing, objectives, risks, constraints, criticality, testability and the availability of resources. As the project and test planning progress, more information becomes available and more detail can be included in the test 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. Planning may be documented in a master test plan and in separate test plans for test levels such as system testing and acceptance testing.

Test planning activities for an entire system or part of a system may include:

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

### 5.2.2 Test Approaches

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

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

Typical approaches include: o Analytical approaches, such as risk-based testing where testing is directed to areas of greatest risk

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

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

### 5.2.3 Adequate Entry and Exit Criteria for specific test levels and types

Entry Criteria

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

* Test environment availability and readiness
* Test tool readiness in the test environment
* Testable code availability
* Test data availability

Exit Criteria

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

* Thoroughness measures, such as coverage of code, functionality or risk
* Estimates of defect density or reliability measures o 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 Execution Schedule

XXX

### 5.2.5 Factors influencing Testing Effort

The testing effort may depend on a number of factors, including:

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

### 5.2.6 Test Estimation approaches

Two approaches for the estimation of test effort are:

* The metrics-based approach: estimating the testing effort based on metrics of former or similar projects or based on typical values
* The expert-based approach: estimating the tasks based on estimates made by the owner of the tasks or by experts

## 5.3Test Progress Monitoring and Control

### 5.3.1 Metrics used in testing

The purpose of test monitoring and reporting is to provide feedback and visibility about test activities. Information to be monitored may be collected manually or automatically and may be used to measure exit criteria, such as coverage.

Metrics should be collected during and at the end of a test level in order to assess:

* The adequacy of the test objectives for that test level
* The adequacy of the test approaches taken
* The effectiveness of the testing with respect to the objectives
* Assess progress against the planned schedule and budget

Common test metrics include:

* Percentage of work done in test case preparation (or percentage of planned test cases prepared)
* Percentage of work done in test environment preparation
* Test case execution (e.g., number of test cases run/not run, and test cases passed/failed)
* Defect information (e.g., defect density, defects found and fixed, failure rate, and re-test results)
* Test coverage of requirements, risks or code
* Subjective confidence of testers in the product
* Dates of test milestones
* Testing costs, including the cost compared to the benefit of finding the next defect or to run the next test

### 5.3.2 Purpose, content, and stakeholders of test report and control

Test reporting is concerned with summarizinginformation about the testing activities, including:

* What happened during a period of testing, such as dates when exit criteria were met
* Analyzed information and metrics to support recommendations and decisions about future actions, such as an assessment of defects remaining, the economic benefit of continued testing, outstanding risks, and the level of confidence in the tested software

Test controldescribes any guiding or corrective actions taken as a result of monitoring, gathering metrics and reporting. Actions may cover any test activity and may affect any other software life cycle activity or task.

Examples of test control actions include:

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

## 5.4 Configuration Management

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.

### 5.4.1 Configuration management in support of testing

For testing, configuration management may involve ensuring the following:

* All items of testware are identified, version controlled, tracked for changes, related to each other and related to development items (test objects) so that traceability can be maintained throughout the test process
* All identified documents and software items are referenced unambiguously in test documentation

For testers, configuration management helps identify (and to reproduce) the current tested item, test documents, the tests and the test harness(es). During test planning, configuration management procedures and infrastructure (tools) should be chosen, documented and implemented.

## 5.5 Risk and Testing

### 5.5.1 Definition of Risk

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

### 5.5.2 Project and Product Risks

##### Project Risks

Project risks are the risks that surround the project’s capability to deliver its objectives, such as:

* Organizational factors:
  + Skill, training and staff shortages
  + Personnel issues
  + Political issues, such as:
* Problems with testers communicating their needs and test results
* Failure by the team to follow up on information found in testing and reviews

(e.g., not improving development and testing practices)

* + Improper attitude toward or expectations of testing (e.g., not appreciating the value of finding defects during testing)
* Technical issues:
  + Problems in defining the right requirements
  + The extent to which requirements cannot be met given existing constraints
  + Test environment not ready on time
  + Late data conversion, migration planning and development and testing data conversion/migration tools
  + Low quality of the design, code, configuration data, test data and tests
* Supplier issues:
  + Failure of a third party
  + Contractual issues

##### Product Risks

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, they are also called quality risks. Examples of product risks include:

* Failure-prone software delivered
* The potential that the software/hardware could cause harm to an individual or company
* Poor software characteristics (e.g., functionality, reliability, usability and performance)
* Poor data integrity and quality (e.g., data migration issues, data conversion problems, data transport problems, violation of data standards)
* Software that does not perform its intended functions

### 5.5.3 Considering thoroughness and Scope of testing based on risk analysis

Risk is used to focus the effort required during testing. It is used to decide where and when to start testing and the areas that need more testing effort. Testing is used to reduce the risk of an adverse effect occurring, or to reduce the impact of an adverse effect.

Identifying project and product risks contributes to the success of a project. Testing is used as a risk-control activity to provide feedback about the identified risks as well as any 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 identification of product risks and their use in guiding test planning and control, specification, preparation and execution of tests. In a risk-based approach the risks identified may be used to:

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

Risk-based testing draws on the collective knowledge and insight of the project stakeholders to determine the risks and the levels of testing required to address those risks. To ensure that the chance of a product failure is minimized, risk management activities provide a disciplined approach to:

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

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

## 5.6 Incident Management

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

5.6.1 Structure and content of an Incident report Incidents may be reported during development, review, testing or use of a software product. They may be reported for issues in code or working system, or in any type of documentation including requirements, development documents, test documents, user manual, or installation guides.

Incident reports have the following objectives:

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

Details of the incident report may include:

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

# Tool Support for Testing - 40 Min.

Keywords

Application lifecycle management tool, configuration management tool, coverage tool, debugging tool, dynamic analysis tool, defect management tool, load testing tool, modeling tool, monitoring tool, performance testing tool, probe effect, requirements management tool, review tool, security tool, static analysis tool, stress testing tool, test comparator, test data preparation tool, test design tool, test harness, test execution tool, test management tool, unit test framework tool, data-driven testing, keyword-driven testing, scripting language

Learning Objectives for Test Tools

6.1 Test tool Classification

FL-6.1.1 (K2) Classify test tools according to their purpose and the test activities that they support.

FL-6.1.2 (K1) Identify benefits and risks of test automation ~~automated test tools and execution~~

FL-6.1.3 (K1) Remember special considerations for test execution tools, static analysis, and test management tools

6.2 Effective use of Tools

FL-6.2.1 (K1) Identify the main principles for selecting a tool

FL-6.2.2 (K1) Identify the objectives for a pilot project for tool evaluation

FL-6.2.3 (K1) Identify the success factors for evaluation, implementation, deployment and on-going

support of test tools in an organisation

## 6.1 Test Tools

Test tools can be used for one or more activities that support testing. These include:

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

Tool support for testing can have one or more of the following purposes depending on the context:

* Improve the efficiency of test activities by automating repetitive tasks or supporting manual test activities like test planning, test design, test reporting and monitoring
* Automate activities that require significant resources when done manually (e.g., static testing)
* Automate activities that cannot be executed manually (e.g., large scale performance testing of client-server applications)
* Increase reliability of testing (e.g., by automating large data comparisons or simulating behavior)

### 6.1.1 Test Tool Classifications

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

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

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

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

### 6.1.1.1 Tool Support for Management of Testing and Tests

Management tools apply to all test activities over the entire software life cycle. Examples of these tools include:

* Test Management and Application Lifecycle Management Tools
* Requirements Management Tools
* Incident Management Tools/Defect Tracking Tools
* Configuration Management Tools

6.1.1.2 Tool Support for Static Testing

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

* Review Tools
* Static Analysis Tools (D)
* Modeling Tools (D)

6.1.1.3 Tool Support for Test Specification

Test specification tools aid in the creation of robust test specifications, including test case and test procedure specifications. Examples of these tools include:

* Test Design Tools
* Test Data Preparation Tools

6.1.1.4 Tool Support for Test Execution and Logging

Many tools exist to support and enhance test execution and logging activities. Examples of these tools include:

* Test Execution Tools
* Test Harness/Unit Test Framework Tools (D)
* Test Comparators
* Coverage Measurement Tools (D)
* Security Testing Tools

6.1.1.5 Tool Support for Performance and Monitoring

Performance and Monitoring tools are essential in supporting these testing activities, as they cannot effectively be performed manually. Examples of these tools include:

* Dynamic Analysis Tools (D)
* Performance Testing/Load Testing/Stress Testing Tools
* Monitoring Tools

6.1.1.6 Tool Support for Specialized Testing Needs

In addition to tools that support the general testing process, there are many other tools that support more specific testing issues. Examples of these include tools that focus on data quality assessment, data conversion and migration, usability testing, accessibility testing, localization testing and other highly specialized testing activities.

6.1.2 Benefits and Risks of Automated Test Tools and Execution

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

Potential benefits of using automated tools and execution include:

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

Risks of using automated tools include:

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

6.1.3 Special Considerations for Some Types of Tools

Test Execution Tools

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

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

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

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

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

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

Static Analysis Tools

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

Test Management Tools

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

6.2 Effective Use of Tools

Have an introduction here

6.2.1 Main principles for selecting tools

The main considerations in selecting a tool for an organization include:

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

6.2.2 Pilot Projects for tool evaluation

Introducing the selected tool into an organization starts with a pilot project, which has the following objectives:

* Learn more detail about the tool
* Evaluate how the tool fits with existing processes and practices, and determine what would need to change
* Decide on standard ways of using, managing, storing and maintaining the tool and the test assets (e.g., deciding on naming conventions for files and tests, creating libraries and defining the modularity of test suites)
* Assess whether the benefits will be achieved at reasonable cost

6.2.3 Success factors for tools in organization

Success factors for evaluation, implementation, deployment, and on-going support of tools within an organization include:

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

# Index

acceptance tests, 10

black-box testing, 20

interoperability testing, 21

user story, 9