1.1. What is Testing?

Software systems are an integral part of our daily life. Most people have had experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time or business reputation, and, in extreme cases, even injury or death. Software testing assesses software quality and helps reducing the risk of software failure in operation.

Software testing is a set of activities to discover defects and evaluate the quality of software artifacts. These artifacts, when being tested, are known as test objects. A common misconception about testing is that it only consists of executing tests (i.e., running the software and checking the test results). However, software testing also includes other activities and must be aligned with the software development lifecycle (see chapter 2).

Another common misconception about testing is that testing focuses entirely on verifying the test object. Whilst testing involves verification, i.e., checking whether the system meets specified requirements, it also involves validation, which means checking whether the system meets users' and other stakeholders' needs in its operational environment.

Testing may be dynamic or static. Dynamic testing involves the execution of software, while static testing does not. Static testing includes reviews (see chapter 3) and static analysis. Dynamic testing uses different types of test techniques and test approaches to derive test cases (see chapter 4).

Testing is not only a technical activity. It also needs to be properly planned, managed, estimated, monitored and controlled (see chapter 5).

Testers use tools (see chapter 6), but it is important to remember that testing is largely an intellectual activity, requiring the testers to have specialized knowledge, use analytical skills and apply critical thinking and systems thinking (Myers 2011, Roman 2018).

The ISO/IEC/IEEE 29119-1 standard provides further information about software testing concepts.

1.1.1. Test Objectives

The typical test objectives are:

- Evaluating work products such as requirements, user stories, designs, and code
- Triggering failures and finding defects
- Ensuring required coverage of a test object
- Reducing the level of risk of inadequate software quality
- Verifying whether specified requirements have been fulfilled
- Verifying that a test object complies with contractual, legal, and regulatory requirements
- Providing information to stakeholders to allow them to make informed decisions
- Building confidence in the quality of the test object
- Validating whether the test object is complete and works as expected by the stakeholders

Objectives of testing can vary, depending upon the context, which includes the work product being tested, the test level, risks, the software development lifecycle (SDLC) being followed, and factors related to the business context, e.g., corporate structure, competitive considerations, or time to market.

1.1.2. Testing and Debugging

Testing and debugging are separate activities. Testing can trigger failures that are caused by defects in the software (dynamic testing) or can directly find defects in the test object (static testing).

When dynamic testing (see chapter 4) triggers a failure, debugging is concerned with finding causes of this failure (defects), analyzing these causes, and eliminating them. The typical debugging process in this case involves:

- Reproduction of a failure
- Diagnosis (finding the root cause)
- Fixing the cause

Subsequent confirmation testing checks whether the fixes resolved the problem. Preferably, confirmation testing is done by the same person who performed the initial test. Subsequent regression testing can also be performed, to check whether the fixes are causing failures in other parts of the test object (see section 2.2.3 for more information on confirmation testing and regression testing).

When static testing identifies a defect, debugging is concerned with removing it. There is no need for reproduction or diagnosis, since static testing directly finds defects, and cannot cause failures (see chapter 3).

1.2. Why is Testing Necessary?

Testing, as a form of quality control, helps in achieving the agreed upon goals within the set scope, time, quality, and budget constraints. Testing's contribution to success should not be restricted to the test team activities. Any stakeholder can use their testing skills to bring the project closer to success. Testing components, systems, and associated documentation helps to identify defects in software,

1.2.1. Testing's Contributions to Success

Testing provides a cost-effective means of detecting defects. These defects can then be removed (by debugging – a non-testing activity), so testing indirectly contributes to higher quality test objects.

Testing provides a means of directly evaluating the quality of a test object at various stages in the SDLC. These measures are used as part of a larger project management activity, contributing to decisions to move to the next stage of the SDLC, such as the release decision.

Testing provides users with indirect representation on the development project. Testers ensure that their understanding of users' needs are considered throughout the development lifecycle. The alternative is to involve a representative set of users as part of the development project, which is not usually possible due to the high costs and lack of availability of suitable users.

Testing may also be required to meet contractual or legal requirements, or to comply with regulatory standards.

1.2.2. Testing and Quality Assurance (QA)

While people often use the terms "testing" and "quality assurance" (QA) interchangeably, testing and QA are not the same. Testing is a form of quality control (QC).

QC is a product-oriented, corrective approach that focuses on those activities supporting the achievement of appropriate levels of quality. Testing is a major form of quality control, while others include formal methods (model checking and proof of correctness), simulation and prototyping.

QA is a process-oriented, preventive approach that focuses on the implementation and improvement of processes. It works on the basis that if a good process is followed correctly, then it will generate a good product. QA applies to both the development and testing processes, and is the responsibility of everyone on a project.

Test results are used by QA and QC. In QC they are used to fix defects, while in QA they provide feedback on how well the development and test processes are performing.

1.2.3. Errors, Defects, Failures, and Root Causes

Human beings make errors (mistakes), which produce defects (faults, bugs), which in turn may result in failures. Humans make errors for various reasons, such as time pressure, complexity of work products, processes, infrastructure or interactions, or simply because they are tired or lack adequate training.

Defects can be found in documentation, such as a requirements specification or a test script, in source code, or in a supporting artifact such as a build file. Defects in artifacts produced earlier in the SDLC, if undetected, often lead to defective artifacts later in the lifecycle. 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. Some defects will always result in a failure if executed, while others will only result in a failure in specific circumstances, and some may never result in a failure.

Errors and defects are not the only cause of failures. Failures can also be caused by environmental conditions, such as when radiation or electromagnetic field cause defects in firmware.

A root cause is a fundamental reason for the occurrence of a problem (e.g., a situation that leads to an error). Root causes are identified through root cause analysis, which is typically performed when a failure occurs or a defect is identified. It is believed that further similar failures or defects can be prevented or their frequency reduced by addressing the root cause, such as by removing it.

1.3. Testing Principles

A number of testing principles offering general guidelines applicable to all testing have been suggested over the years. This syllabus describes seven such principles.

1. Testing shows the presence, not the absence of defects. Testing can show that defects are present in the test object, but cannot prove that there are no defects (Buxton 1970). Testing reduces the probability of defects remaining undiscovered in the test object, but even if no defects are found, testing cannot prove test object correctness.

2. Exhaustive testing is impossible. Testing everything is not feasible except in trivial cases (Manna 1978). Rather than attempting to test exhaustively, test techniques (see chapter 4), test case prioritization (see section 5.1.5), and risk-based testing (see section 5.2), should be used to focus test efforts.

3. Early testing saves time and money. Defects that are removed early in the process will not cause subsequent defects in derived work products. The cost of quality will be reduced since fewer failures will occur later in the SDLC (Boehm 1981). To find defects early, both static testing (see chapter 3) and dynamic testing (see chapter 4) should be started as early as possible.

4. Defects cluster together. A small number of system components usually contain most of the defects discovered or are responsible for most of the operational failures (Enders 1975). This phenomenon is an

illustration of the Pareto principle. Predicted defect clusters, and actual defect clusters observed during testing or in operation, are an important input for risk-based testing (see section 5.2).

5. Tests wear out. If the same tests are repeated many times, they become increasingly ineffective in detecting new defects (Beizer 1990). To overcome this effect, existing tests and test data may need to be modified, and new tests may need to be written. However, in some cases, repeating the same tests can have a beneficial outcome, e.g., in automated regression testing (see section 2.2.3).

6. Testing is context dependent. There is no single universally applicable approach to testing. Testing is done differently in different contexts (Kaner 2011).

7. Absence-of-defects fallacy. It is a fallacy (i.e., a misconception) to expect that software verification will ensure the success of a system. Thoroughly testing all the specified requirements and fixing all the defects found could still produce a system that does not fulfill the users' needs and expectations, that does not help in achieving the customer's business goals, and that is inferior compared to other competing systems. In addition to verification, validation should also be carried out (Boehm 1981).

1.4. Test Activities, Testware and Test Roles

Testing is context dependent, but, at a high level, there are common sets of test activities without which testing is less likely to achieve test objectives. These sets of test activities form a test process. The test process can be tailored to a given situation based on various factors. Which test activities are included in this test process, how they are implemented, and when they occur is normally decided as part of the test planning for the specific situation (see section 5.1).

The following sections describe the general aspects of this test process in terms of test activities and tasks, the impact of context, testware, traceability between the test basis and testware, and testing roles.

The ISO/IEC/IEEE 29119-2 standard provides further information about test processes.

1.4.1. Test Activities and Tasks

A test process usually consists of the main groups of activities described below. Although many of these activities may appear to follow a logical sequence, they are often implemented iteratively or in parallel. These testing activities usually need to be tailored to the system and the project.

Test planning consists of defining the test objectives and then selecting an approach that best achieves the objectives within the constraints imposed by the overall context. Test planning is further explained in section 5.1.

Test monitoring and control. Test monitoring involves the ongoing checking of all test activities and the comparison of actual progress against the plan. Test control involves taking the actions necessary to meet the objectives of testing. Test monitoring and control are further explained in section 5.3.

Test analysis includes analyzing the test basis to identify testable features and to define and prioritize associated test conditions, together with the related risks and risk levels (see section 5.2). The test basis and the test objects are also evaluated to identify defects they may contain and to assess their testability. Test analysis is often supported by the use of test techniques (see chapter 4). Test analysis answers the question "what to test?" in terms of measurable coverage criteria.

Test design includes elaborating the test conditions into test cases and other testware (e.g., test charters). This activity often involves the identification of coverage items, which serve as a guide to specify test case inputs. Test techniques (see chapter 4) can be used to support this activity. Test design

also includes defining the test data requirements, designing the test environment and identifying any other required infrastructure and tools. Test design answers the question “how to test?”.

Test implementation includes creating or acquiring the testware necessary for test execution (e.g., test data). Test cases can be organized into test procedures and are often assembled into test suites. Manual and automated test scripts are created. Test procedures are prioritized and arranged within a test execution schedule for efficient test execution (see section 5.1.5). The test environment is built and verified to be set up correctly.

Test execution includes running the tests in accordance with the test execution schedule (test runs). Test execution may be manual or automated. Test execution can take many forms, including continuous testing or pair testing sessions. Actual test results are compared with the expected results. The test results are logged. Anomalies are analyzed to identify their likely causes. This analysis allows us to report the anomalies based on the failures observed (see section 5.5).

Test completion activities usually occur at project milestones (e.g., release, end of iteration, test level completion) for any unresolved defects, change requests or product backlog items created. Any testware that may be useful in the future is identified and archived or handed over to the appropriate teams. The test environment is shut down to an agreed state. The test activities are analyzed to identify lessons learned and improvements for future iterations, releases, or projects (see section 2.1.6). A test completion report is created and communicated to the stakeholders.

1.4.2. Test Process in Context

Testing is not performed in isolation. Test activities are an integral part of the development processes carried out within an organization. Testing is also funded by stakeholders and its final goal is to help fulfill the stakeholders’ business needs. Therefore, the way the testing is carried out will depend on a number of contextual factors including:

- Stakeholders (needs, expectations, requirements, willingness to cooperate, etc.)
- Team members (skills, knowledge, level of experience, availability, training needs, etc.)
- Business domain (criticality of the test object, identified risks, market needs, specific legal regulations, etc.)
- Technical factors (type of software, product architecture, technology used, etc.)
- Project constraints (scope, time, budget, resources, etc.)
- Organizational factors (organizational structure, existing policies, practices used, etc.)
- Software development lifecycle (engineering practices, development methods, etc.)
- Tools (availability, usability, compliance, etc.)

These factors will have an impact on many test-related issues, including: test strategy, test techniques used, degree of test automation, required level of coverage, level of detail of test documentation, reporting, etc.

1.4.3. Testware

Testware is created as output work products from the test activities described in section 1.4.1. There is a significant variation in how different organizations produce, shape, name, organize and manage their

work products. Proper configuration management (see section 5.4) ensures consistency and integrity of work products. The following list of work products is not exhaustive:

- **Test planning work products** include: test plan, test schedule, risk register, and entry and exit criteria (see section 5.1). Risk register is a list of risks together with risk likelihood, risk impact and information about risk mitigation (see section 5.2). Test schedule, risk register and entry and exit criteria are often a part of the test plan.
- **Test monitoring and control work products** include: test progress reports (see section 5.3.2), documentation of control directives (see section 5.3) and risk information (see section 5.2).
- **Test analysis work products** include: (prioritized) test conditions (e.g., acceptance criteria, see section 4.5.2), and defect reports regarding defects in the test basis (if not fixed directly).
- **Test design work products** include: (prioritized) test cases, test charters, coverage items, test data requirements and test environment requirements.
- **Test implementation work products** include: test procedures, automated test scripts, test suites, test data, test execution schedule, and test environment elements. Examples of test environment elements include: stubs, drivers, simulators, and service virtualizations.
- **Test execution work products** include: test logs, and defect reports (see section 5.5).
- **Test completion work products** include: test completion report (see section 5.3.2), action items for improvement of subsequent projects or iterations, documented lessons learned, and change requests (e.g., as product backlog items).

1.4.4. Traceability between the Test Basis and Testware

In order to implement effective test monitoring and control, it is important to establish and maintain traceability throughout the test process between the test basis elements, testware associated with these elements (e.g., test conditions, risks, test cases), test results, and detected defects.

Accurate traceability supports coverage evaluation, so it is very useful if measurable coverage criteria are defined in the test basis. The coverage criteria can function as key performance indicators to drive the activities that show to what extent the test objectives have been achieved (see section 1.1.1). For example:

- Traceability of test cases to requirements can verify that the requirements are covered by test cases.
- Traceability of test results to risks can be used to evaluate the level of residual risk in a test object.

In addition to evaluating coverage, good traceability makes it possible to determine the impact of changes, facilitates test audits, and helps meet IT governance criteria. Good traceability also makes test progress and completion reports more easily understandable by including the status of test basis elements. This can also assist in communicating the technical aspects of testing to stakeholders in an understandable manner. Traceability provides information to assess product quality, process capability, and project progress against business goals.

1.4.5. Roles in Testing

In this syllabus, two principal roles in testing are covered: a test management role and a testing role. The activities and tasks assigned to these two roles depend on factors such as the project and product context, the skills of the people in the roles, and the organization.

The test management role takes overall responsibility for the test process, test team and leadership of the test activities. The test management role is mainly focused on the activities of test planning, test monitoring and control and test completion. The way in which the test management role is carried out varies depending on the context. For example, in Agile software development, some of the test management tasks may be handled by the Agile team. Tasks that span multiple teams or the entire organization may be performed by test managers outside of the development team.

The testing role takes overall responsibility for the engineering (technical) aspect of testing. The testing role is mainly focused on the activities of test analysis, test design, test implementation and test execution.

Different people may take on these roles at different times. For example, the test management role can be performed by a team leader, by a test manager, by a development manager, etc. It is also possible for one person to take on the roles of testing and test management at the same time.

1.5. Essential Skills and Good Practices in Testing

Skill is the ability to do something well that comes from one's knowledge, practice and aptitude. Good testers should possess some essential skills to do their job well. Good testers should be effective team players and should be able to perform testing on different levels of test independence.

1.5.1. Generic Skills Required for Testing

While being generic, the following skills are particularly relevant for testers:

- Testing knowledge (to increase effectiveness of testing, e.g., by using test techniques)
- Thoroughness, carefulness, curiosity, attention to details, being methodical (to identify defects, especially the ones that are difficult to find)
- Good communication skills, active listening, being a team player (to interact effectively with all stakeholders, to convey information to others, to be understood, and to report and discuss defects)
- Analytical thinking, critical thinking, creativity (to increase effectiveness of testing)
- Technical knowledge (to increase efficiency of testing, e.g., by using appropriate test tools)
- Domain knowledge (to be able to understand and to communicate with end users/business representatives)

Testers are often the bearers of bad news. It is a common human trait to blame the bearer of bad news. This makes communication skills crucial for testers. Communicating test results may be perceived as criticism of the product and of its author. Confirmation bias can make it difficult to accept information that disagrees with currently held beliefs. Some people may perceive testing as a destructive activity, even though it contributes greatly to project success and product quality. To try to improve this view, information about defects and failures should be communicated in a constructive way.

1.5.2. Whole Team Approach

One of the important skills for a tester is the ability to work effectively in a team context and to contribute positively to the team goals. The whole team approach – a practice coming from Extreme Programming (see section 2.1) – builds upon this skill.

In the whole-team approach any team member with the necessary knowledge and skills can perform any task, and everyone is responsible for quality. The team members share the same workspace (physical or virtual), as co-location facilitates communication and interaction. The whole team approach improves team dynamics, enhances communication and collaboration within the team, and creates synergy by allowing the various skill sets within the team to be leveraged for the benefit of the project.

Testers work closely with other team members to ensure that the desired quality levels are achieved. This includes collaborating with business representatives to help them create suitable acceptance tests and working with developers to agree on the test strategy and decide on test automation approaches. Testers can thus transfer testing knowledge to other team members and influence the development of the product.

Depending on the context, the whole team approach may not always be appropriate. For instance, in some situations, such as safety-critical, a high level of test independence may be needed.

1.5.3. Independence of Testing

A certain degree of independence makes the tester more effective at finding defects due to differences between the author's and the tester's cognitive biases (cf. Salman 1995). Independence is not, however, a replacement for familiarity, e.g., developers can efficiently find many defects in their own code.

Work products can be tested by their author (no independence), by the author's peers from the same team (some independence), by testers from outside the author's team but within the organization (high independence), or by testers from outside the organization (very high independence). For most projects, it is usually best to carry out testing with multiple levels of independence (e.g., developers performing component and component integration testing, test team performing system and system integration testing, and business representatives performing acceptance testing).

The main benefit of independence of testing is that independent testers are likely to recognize different kinds of failures and defects compared to developers because of their different backgrounds, technical perspectives, and biases. Moreover, an independent tester can verify, challenge, or disprove assumptions made by stakeholders during specification and implementation of the system.

However, there are also some drawbacks. Independent testers may be isolated from the development team, which may lead to a lack of collaboration, communication problems, or an adversarial relationship with the development team. Developers may lose a sense of responsibility for quality. Independent testers may be seen as a bottleneck or be blamed for delays in release.

2. Testing Throughout the Software Development Lifecycle – 130 minutes

Keywords

acceptance testing, black-box testing, component integration testing, component testing, confirmation testing, functional testing, integration testing, maintenance testing, non-functional testing, regression testing, shift-left, system integration testing, system testing, test level, test object, test type, white-box testing

Learning Objectives for Chapter 2:

2.1 Testing in the Context of a Software Development Lifecycle

FL-2.1.1 (K2) Explain the impact of the chosen software development lifecycle on testing

FL-2.1.2 (K1) Recall good testing practices that apply to all software development lifecycles

FL-2.1.3 (K1) Recall the examples of test-first approaches to development

FL-2.1.4 (K2) Summarize how DevOps might have an impact on testing

FL-2.1.5 (K2) Explain the shift-left approach

FL-2.1.6 (K2) Explain how retrospectives can be used as a mechanism for process improvement

2.2 Test Levels and Test Types

FL-2.2.1 (K2) Distinguish the different test levels

FL-2.2.2 (K2) Distinguish the different test types

FL-2.2.3 (K2) Distinguish confirmation testing from regression testing

2.3 Maintenance Testing

FL-2.3.1 (K2) Summarize maintenance testing and its triggers

2.1. Testing in the Context of a Software Development Lifecycle

A software development lifecycle (SDLC) model is an abstract, high-level representation of the software development process. A SDLC model defines how different development phases and types of activities performed within this process relate to each other, both logically and chronologically. Examples of SDLC models include: sequential development models (e.g., waterfall model, V-model), iterative development models (e.g., spiral model, prototyping), and incremental development models (e.g., Unified Process).

Some activities within software development processes can also be described by more detailed software development methods and Agile practices. Examples include: acceptance test-driven development (ATDD), behavior-driven development (BDD), domain-driven design (DDD), extreme programming (XP), feature-driven development (FDD), Kanban, Lean IT, Scrum, and test-driven development (TDD).

2.1.1. Impact of the Software Development Lifecycle on Testing

Testing must be adapted to the SDLC to succeed. The choice of the SDLC impacts on the:

- Scope and timing of test activities (e.g., test levels and test types)
- Level of detail of test documentation
- Choice of test techniques and test approach
- Extent of test automation
- Role and responsibilities of a tester

In sequential development models, in the initial phases testers typically participate in requirement reviews, test analysis, and test design. The executable code is usually created in the later phases, so typically dynamic testing cannot be performed early in the SDLC.

In some iterative and incremental development models, it is assumed that each iteration delivers a working prototype or product increment. This implies that in each iteration both static and dynamic testing may be performed at all test levels. Frequent delivery of increments requires fast feedback and extensive regression testing.

Agile software development assumes that change may occur throughout the project. Therefore, lightweight work product documentation and extensive test automation to make regression testing easier are favored in agile projects. Also, most of the manual testing tends to be done using experience-based test techniques (see Section 4.4) that do not require extensive prior test analysis and design.

2.1.2. Software Development Lifecycle and Good Testing Practices

Good testing practices, independent of the chosen SDLC model, include the following:

- For every software development activity, there is a corresponding test activity, so that all development activities are subject to quality control
- Different test levels (see chapter 2.2.1) have specific and different test objectives, which allows for testing to be appropriately comprehensive while avoiding redundancy
- Test analysis and design for a given test level begins during the corresponding development phase of the SDLC, so that testing can adhere to the principle of early testing (see section 1.3)

- Testers are involved in reviewing work products as soon as drafts of this documentation are available, so that this earlier testing and defect detection can support the shift-left strategy (see section 2.1.5)

2.1.3. Testing as a Driver for Software Development

TDD, ATDD and BDD are similar development approaches, where tests are defined as a means of directing development. Each of these approaches implements the principle of early testing (see section 1.3) and follows a shift-left approach (see section 2.1.5), since the tests are defined before the code is written. They support an iterative development model. These approaches are characterized as follows:

Test-Driven Development (TDD):

- Directs the coding through test cases (instead of extensive software design) (Beck 2003)
- Tests are written first, then the code is written to satisfy the tests, and then the tests and code are refactored

Acceptance Test-Driven Development (ATDD) (see section 4.5.3):

- Derives tests from acceptance criteria as part of the system design process (Gärtner 2011)
- Tests are written before the part of the application is developed to satisfy the tests

Behavior-Driven Development (BDD):

- Expresses the desired behavior of an application with test cases written in a simple form of natural language, which is easy to understand by stakeholders – usually using the Given/When/Then format. (Chelimsky 2010)
- Test cases are then automatically translated into executable tests

For all the above approaches, tests may persist as automated tests to ensure the code quality in future adaptions / refactoring.

2.1.4. DevOps and Testing

DevOps is an organizational approach aiming to create synergy by getting development (including testing) and operations to work together to achieve a set of common goals. DevOps requires a cultural shift within an organization to bridge the gaps between development (including testing) and operations while treating their functions with equal value. DevOps promotes team autonomy, fast feedback, integrated toolchains, and technical practices like continuous integration (CI) and continuous delivery (CD). This enables the teams to build, test and release high-quality code faster through a DevOps delivery pipeline (Kim 2016).

From the testing perspective, some of the benefits of DevOps are:

- Fast feedback on the code quality, and whether changes adversely affect existing code
- CI promotes a shift-left approach in testing (see section 2.1.5) by encouraging developers to submit high quality code accompanied by component tests and static analysis
- Promotes automated processes like CI/CD that facilitate establishing stable test environments
- Increases the view on non-functional quality characteristics (e.g., performance, reliability)

- Automation through a delivery pipeline reduces the need for repetitive manual testing
- The risk in regression is minimized due to the scale and range of automated regression tests

DevOps is not without its risks and challenges, which include:

- The DevOps delivery pipeline must be defined and established
- CI / CD tools must be introduced and maintained
- Test automation requires additional resources and may be difficult to establish and maintain

Although DevOps comes with a high level of automated testing, manual testing – especially from the user's perspective – will still be needed.

2.1.5. Shift-Left Approach

The principle of early testing (see section 1.3) is sometimes referred to as shift-left because it is an approach where testing is performed earlier in the SDLC. Shift-left normally suggests that testing should be done earlier (e.g., not waiting for code to be implemented or for components to be integrated), but it does not mean that testing later in the SDLC should be neglected.

There are some good practices that illustrate how to achieve a “shift-left” in testing, which include:

- Reviewing the specification from the perspective of testing. These review activities on specifications often find potential defects, such as ambiguities, incompleteness, and inconsistencies
- Writing test cases before the code is written and have the code run in a test harness during code implementation
- Using CI and even better CD as it comes with fast feedback and automated component tests to accompany source code when it is submitted to the code repository
- Completing static analysis of source code prior to dynamic testing, or as part of an automated process
- Performing non-functional testing starting at the component test level, where possible. This is a form of shift-left as these non-functional test types tend to be performed later in the SDLC when a complete system and a representative test environment are available

A shift-left approach might result in extra training, effort and/or costs earlier in the process but is expected to save efforts and/or costs later in the process.

For the shift-left approach it is important that stakeholders are convinced and bought into this concept.

2.1.6. Retrospectives and Process Improvement

Retrospectives (also known as “post-project meetings” and project retrospectives) are often held at the end of a project or an iteration, at a release milestone, or can be held when needed. The timing and organization of the retrospectives depend on the particular SDLC model being followed. In these meetings the participants (not only testers, but also e.g., developers, architects, product owner, business analysts) discuss:

- What was successful, and should be retained?

- What was not successful and could be improved?
- How to incorporate the improvements and retain the successes in the future?

The results should be recorded and are normally part of the test completion report (see section 5.3.2). Retrospectives are critical for the successful implementation of continuous improvement and it is important that any recommended improvements are followed up.

Typical benefits for testing include:

- Increased test effectiveness / efficiency (e.g., by implementing suggestions for process improvement)
- Increased quality of testware (e.g., by jointly reviewing the test processes)
- Team bonding and learning (e.g., as a result of the opportunity to raise issues and propose improvement points)
- Improved quality of the test basis (e.g., as deficiencies in the extent and quality of the requirements could be addressed and solved)
- Better cooperation between development and testing (e.g., as collaboration is reviewed and optimized regularly)

2.2. Test Levels and Test Types

Test levels are groups of test activities that are organized and managed together. Each test level is an instance of the test process, performed in relation to software at a given stage of development, from individual components to complete systems or, where applicable, systems of systems.

Test levels are related to other activities within the SDLC. In sequential SDLC models, the test levels are often defined such that the exit criteria of one level are part of the entry criteria for the next level. In some iterative models, this may not apply. Development activities may span through multiple test levels. Test levels may overlap in time.

Test types are groups of test activities related to specific quality characteristics and most of those test activities can be performed at every test level.

2.2.1. Test Levels

In this syllabus, the following five test levels are described:

- **Component testing** (also known as unit testing) focuses on testing components in isolation. It often requires specific support, such as test harnesses or unit test frameworks. Component testing is normally performed by developers in their development environments.
- **Component integration testing** (also known as unit integration testing) focuses on testing the interfaces and interactions between components. Component integration testing is heavily dependent on the integration strategy approaches like bottom-up, top-down or big-bang.
- **System testing** focuses on the overall behavior and capabilities of an entire system or product, often including functional testing of end-to-end tasks and the non-functional testing of quality characteristics. For some non-functional quality characteristics, it is preferable to test them on a complete system in a representative test environment (e.g., usability). Using simulations of sub-

systems is also possible. System testing may be performed by an independent test team, and is related to specifications for the system.

- **System integration testing** focuses on testing the interfaces of the system under test and other systems and external services . System integration testing requires suitable test environments preferably similar to the operational environment.
- **Acceptance testing** focuses on validation and on demonstrating readiness for deployment, which means that the system fulfills the user's business needs. Ideally, acceptance testing should be performed by the intended users. The main forms of acceptance testing are: user acceptance testing (UAT), operational acceptance testing, contractual and regulatory acceptance testing, alpha testing and beta testing.

Test levels are distinguished by the following non-exhaustive list of attributes, to avoid overlapping of test activities:

- Test object
- Test objectives
- Test basis
- Defects and failures
- Approach and responsibilities

2.2.2. Test Types

A lot of test types exist and can be applied in projects. In this syllabus, the following four test types are addressed:

Functional testing evaluates the functions that a component or system should perform. The functions are "what" the test object should do. The main objective of functional testing is checking the functional completeness, functional correctness and functional appropriateness.

Non-functional testing evaluates attributes other than functional characteristics of a component or system. Non-functional testing is the testing of "how well the system behaves". The main objective of non-functional testing is checking the non-functional software quality characteristics. The ISO/IEC 25010 standard provides the following classification of the non-functional software quality characteristics:

- Performance efficiency
- Compatibility
- Usability
- Reliability
- Security
- Maintainability
- Portability

It is sometimes appropriate for non-functional testing to start early in the life cycle (e.g., as part of reviews and component testing or system testing). Many non-functional tests are derived from functional tests as

they use the same functional tests, but check that while performing the function, a non-functional constraint is satisfied (e.g., checking that a function performs within a specified time, or a function can be ported to a new platform). The late discovery of non-functional defects can pose a serious threat to the success of a project. Non-functional testing sometimes needs a very specific test environment, such as a usability lab for usability testing.

Black-box testing (see section 4.2) is specification-based and derives tests from documentation external to the test object. The main objective of black-box testing is checking the system's behavior against its specifications.

White-box testing (see section 4.3) is structure-based and derives tests from the system's implementation or internal structure (e.g., code, architecture, work-flows, and data flows). The main objective of white-box testing is to cover the underlying structure by the tests to the acceptable level.

All the four above mentioned test types can be applied to all test levels, although the focus will be different at each level. Different test techniques can be used to derive test conditions and test cases for all the mentioned test types.

2.2.3. Confirmation Testing and Regression Testing

Changes are typically made to a component or system to either enhance it by adding a new feature or to fix it by removing a defect. Testing should then also include confirmation testing and regression testing.

Confirmation testing confirms that an original defect has been successfully fixed. Depending on the risk, one can test the fixed version of the software in several ways, including:

- executing all test cases that previously have failed due to the defect, or, also by
- adding new tests to cover any changes that were needed to fix the defect

However, when time or money is short when fixing defects, confirmation testing might be restricted to simply exercising the steps that should reproduce the failure caused by the defect and checking that the failure does not occur.

Regression testing confirms that no adverse consequences have been caused by a change, including a fix that has already been confirmation tested. These adverse consequences could affect the same component where the change was made, other components in the same system, or even other connected systems. Regression testing may not be restricted to the test object itself but can also be related to the environment. It is advisable first to perform an impact analysis to optimize the extent of the regression testing. Impact analysis shows which parts of the software could be affected.

Regression test suites are run many times and generally the number of regression test cases will increase with each iteration or release, so regression testing is a strong candidate for automation. Automation of these tests should start early in the project. Where CI is used, such as in DevOps (see section 2.1.4), it is good practice to also include automated regression tests. Depending on the situation, this may include regression tests on different levels.

Confirmation testing and/or regression testing for the test object are needed on all test levels if defects are fixed and/or changes are made on these test levels.

2.3. Maintenance Testing

There are different categories of maintenance, it can be corrective, adaptive to changes in the environment or improve performance or maintainability (see ISO/IEC 14764 for details), so maintenance can involve planned releases/deployments and unplanned releases/deployments (hot fixes). Impact analysis may be done before a change is made, to help decide if the change should be made, based on the potential consequences in other areas of the system. Testing the changes to a system in production includes both evaluating the success of the implementation of the change and the checking for possible regressions in parts of the system that remain unchanged (which is usually most of the system).

The scope of maintenance testing typically depends on:

- The degree of risk of the change
- The size of the existing system
- The size of the change

The triggers for maintenance and maintenance testing can be classified as follows:

- Modifications, such as planned enhancements (i.e., release-based), corrective changes or hot fixes.
- Upgrades or migrations of the operational environment, such as from one platform to another, which can require tests associated with the new environment as well as of the changed software, or tests of data conversion when data from another application is migrated into the system being maintained.
- Retirement, such as when an application reaches the end of its life. When a system is retired, this can require testing of data archiving if long data-retention periods are required. Testing of restore and retrieval procedures after archiving may also be needed in the event that certain data is required during the archiving period.

3. Static Testing – 80 minutes

Keywords

anomaly, dynamic testing, formal review, informal review, inspection, review, static analysis, static testing, technical review, walkthrough

Learning Objectives for Chapter 3:

3.1 Static Testing Basics

FL-3.1.1 (K1) Recognize types of products that can be examined by the different static test techniques

FL-3.1.2 (K2) Explain the value of static testing

FL-3.1.3 (K2) Compare and contrast static and dynamic testing

3.2 Feedback and Review Process

FL-3.2.1 (K1) Identify the benefits of early and frequent stakeholder feedback

FL-3.2.2 (K2) Summarize the activities of the review process

FL-3.2.3 (K1) Recall which responsibilities are assigned to the principal roles when performing reviews

FL-3.2.4 (K2) Compare and contrast the different review types

FL-3.2.5 (K1) Recall the factors that contribute to a successful review

3.1. Static Testing Basics

In contrast to dynamic testing, in static testing the software under test does not need to be executed. Code, process specification, system architecture specification or other work products are evaluated through manual examination (e.g., reviews) or with the help of a tool (e.g., static analysis). Test objectives include improving quality, detecting defects and assessing characteristics like readability, completeness, correctness, testability and consistency. Static testing can be applied for both verification and validation.

Testers, business representatives and developers work together during example mappings, collaborative user story writing and backlog refinement sessions to ensure that user stories and related work products meet defined criteria, e.g., the Definition of Ready (see section 5.1.3). Review techniques can be applied to ensure user stories are complete and understandable and include testable acceptance criteria. By asking the right questions, testers explore, challenge and help improve the proposed user stories.

Static analysis can identify problems prior to dynamic testing while often requiring less effort, since no test cases are required, and tools (see chapter 6) are typically used. Static analysis is often incorporated into CI frameworks (see section 2.1.4). While largely used to detect specific code defects, static analysis is also used to evaluate maintainability and security. Spelling checkers and readability tools are other examples of static analysis tools.

3.1.1. Work Products Examinable by Static Testing

Almost any work product can be examined using static testing. Examples include requirement specification documents, source code, test plans, test cases, product backlog items, test charters, project documentation, contracts and models.

Any work product that can be read and understood can be the subject of a review. However, for static analysis, work products need a structure against which they can be checked (e.g., models, code or text with a formal syntax).

Work products that are not appropriate for static testing include those that are difficult to interpret by human beings and that should not be analyzed by tools (e.g., 3rd party executable code due to legal reasons).

3.1.2. Value of Static Testing

Static testing can detect defects in the earliest phases of the SDLC, fulfilling the principle of early testing (see section 1.3). It can also identify defects which cannot be detected by dynamic testing (e.g., unreachable code, design patterns not implemented as desired, defects in non-executable work products).

Static testing provides the ability to evaluate the quality of, and to build confidence in work products. By verifying the documented requirements, the stakeholders can also make sure that these requirements describe their actual needs. Since static testing can be performed early in the SDLC, a shared understanding can be created among the involved stakeholders. Communication will also be improved between the involved stakeholders. For this reason, it is recommended to involve a wide variety of stakeholders in static testing.

Even though reviews can be costly to implement, the overall project costs are usually much lower than when no reviews are performed because less time and effort needs to be spent on fixing defects later in the project.

Code defects can be detected using static analysis more efficiently than in dynamic testing, usually resulting in both fewer code defects and a lower overall development effort.

3.1.3. Differences between Static Testing and Dynamic Testing

Static testing and dynamic testing practices complement each other. They have similar objectives, such as supporting the detection of defects in work products (see section 1.1.1), but there are also some differences, such as:

- Static and dynamic testing (with analysis of failures) can both lead to the detection of defects, however there are some defect types that can only be found by either static or dynamic testing.
- Static testing finds defects directly, while dynamic testing causes failures from which the associated defects are determined through subsequent analysis
- Static testing may more easily detect defects that lay on paths through the code that are rarely executed or hard to reach using dynamic testing
- Static testing can be applied to non-executable work products, while dynamic testing can only be applied to executable work products
- Static testing can be used to measure quality characteristics that are not dependent on executing code (e.g., maintainability), while dynamic testing can be used to measure quality characteristics that are dependent on executing code (e.g., performance efficiency)

Typical defects that are easier and/or cheaper to find through static testing include:

- Defects in requirements (e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, duplications)
- Design defects (e.g., inefficient database structures, poor modularization)
- Certain types of coding defects (e.g., variables with undefined values, undeclared variables, unreachable or duplicated code, excessive code complexity)
- Deviations from standards (e.g., lack of adherence to naming conventions in coding standards)
- Incorrect interface specifications (e.g., mismatched number, type or order of parameters)
- Specific types of security vulnerabilities (e.g., buffer overflows)
- Gaps or inaccuracies in test basis coverage (e.g., missing tests for an acceptance criterion)

3.2. Feedback and Review Process

3.2.1. Benefits of Early and Frequent Stakeholder Feedback

Early and frequent feedback allows for the early communication of potential quality problems. If there is little stakeholder involvement during the SDLC, the product being developed might not meet the stakeholder's original or current vision. A failure to deliver what the stakeholder wants can result in costly rework, missed deadlines, blame games, and might even lead to complete project failure.

Frequent stakeholder feedback throughout the SDLC can prevent misunderstandings about requirements and ensure that changes to requirements are understood and implemented earlier. This helps the development team to improve their understanding of what they are building. It allows them to focus on those features that deliver the most value to the stakeholders and that have the most positive impact on identified risks.

3.2.2. Review Process Activities

The ISO/IEC 20246 standard defines a generic review process that provides a structured but flexible framework from which a specific review process may be tailored to a particular situation. If the required review is more formal, then more of the tasks described for the different activities will be needed.

The size of many work products makes them too large to be covered by a single review. The review process may be invoked a couple of times to complete the review for the entire work product.

The activities in the review process are:

- **Planning.** During the planning phase, the scope of the review, which comprises the purpose, the work product to be reviewed, quality characteristics to be evaluated, areas to focus on, exit criteria, supporting information such as standards, effort and the timeframes for the review, shall be defined.
- **Review initiation.** During review initiation, the goal is to make sure that everyone and everything involved is prepared to start the review. This includes making sure that every participant has access to the work product under review, understands their role and responsibilities and receives everything needed to perform the review.
- **Individual review.** Every reviewer performs an individual review to assess the quality of the work product under review, and to identify anomalies, recommendations, and questions by applying one or more review techniques (e.g., checklist-based reviewing, scenario-based reviewing). The ISO/IEC 20246 standard provides more depth on different review techniques. The reviewers log all their identified anomalies, recommendations, and questions.
- **Communication and analysis.** Since the anomalies identified during a review are not necessarily defects, all these anomalies need to be analyzed and discussed. For every anomaly, the decision should be made on its status, ownership and required actions. This is typically done in a review meeting, during which the participants also decide what the quality level of reviewed work product is and what follow-up actions are required. A follow-up review may be required to complete actions.
- **Fixing and reporting.** For every defect, a defect report should be created so that corrective actions can be followed-up. Once the exit criteria are reached, the work product can be accepted. The review results are reported.

3.2.3. Roles and Responsibilities in Reviews

Reviews involve various stakeholders, who may take on several roles. The principal roles and their responsibilities are:

- Manager – decides what is to be reviewed and provides resources, such as staff and time for the review
- Author – creates and fixes the work product under review

- Moderator (also known as the facilitator) – ensures the effective running of review meetings, including mediation, time management, and a safe review environment in which everyone can speak freely
- Scribe (also known as recorder) – collates anomalies from reviewers and records review information, such as decisions and new anomalies found during the review meeting
- Reviewer – performs reviews. A reviewer may be someone working on the project, a subject matter expert, or any other stakeholder
- Review leader – takes overall responsibility for the review such as deciding who will be involved, and organizing when and where the review will take place

Other, more detailed roles are possible, as described in the ISO/IEC 20246 standard.

3.2.4. Review Types

There exist many review types ranging from informal reviews to formal reviews. The required level of formality depends on factors such as the SDLC being followed, the maturity of the development process, the criticality and complexity of the work product being reviewed, legal or regulatory requirements, and the need for an audit trail. The same work product can be reviewed with different review types, e.g., first an informal one and later a more formal one.

Selecting the right review type is key to achieving the required review objectives (see section 3.2.5). The selection is not only based on the objectives, but also on factors such as the project needs, available resources, work product type and risks, business domain, and company culture.

Some commonly used review types are:

- **Informal review.** Informal reviews do not follow a defined process and do not require a formal documented output. The main objective is detecting anomalies.
- **Walkthrough.** A walkthrough, which is led by the author, can serve many objectives, such as evaluating quality and building confidence in the work product, educating reviewers, gaining consensus, generating new ideas, motivating and enabling authors to improve and detecting anomalies. Reviewers might perform an individual review before the walkthrough, but this is not required.
- **Technical Review.** A technical review is performed by technically qualified reviewers and led by a moderator. The objectives of a technical review are to gain consensus and make decisions regarding a technical problem, but also to detect anomalies, evaluate quality and build confidence in the work product, generate new ideas, and to motivate and enable authors to improve.
- **Inspection.** As inspections are the most formal type of review, they follow the complete generic process (see section 3.2.2). The main objective is to find the maximum number of anomalies. Other objectives are to evaluate quality, build confidence in the work product, and to motivate and enable authors to improve. Metrics are collected and used to improve the SDLC, including the inspection process. In inspections, the author cannot act as the review leader or scribe.

3.2.5. Success Factors for Reviews

There are several factors that determine the success of reviews, which include:

- Defining clear objectives and measurable exit criteria. Evaluation of participants should never be an objective
- Choosing the appropriate review type to achieve the given objectives, and to suit the type of work product, the review participants, the project needs and context
- Conducting reviews on small chunks, so that reviewers do not lose concentration during an individual review and/or the review meeting (when held)
- Providing feedback from reviews to stakeholders and authors so they can improve the product and their activities (see section 3.2.1)
- Providing adequate time to participants to prepare for the review
- Support from management for the review process
- Making reviews part of the organization's culture, to promote learning and process improvement
- Providing adequate training for all participants so they know how to fulfil their role
- Facilitating meetings

4. Test Analysis and Design – 390 minutes

Keywords

acceptance criteria, acceptance test-driven development, black-box test technique, boundary value analysis, branch coverage, checklist-based testing, collaboration-based test approach, coverage, coverage item, decision table testing, equivalence partitioning, error guessing, experience-based test technique, exploratory testing, state transition testing, statement coverage, test technique, white-box test technique

Learning Objectives for Chapter 4:

4.1 Test Techniques Overview

FL-4.1.1 (K2) Distinguish black-box, white-box and experience-based test techniques

4.2 Black-box Test Techniques

FL-4.2.1 (K3) Use equivalence partitioning to derive test cases

FL-4.2.2 (K3) Use boundary value analysis to derive test cases

FL-4.2.3 (K3) Use decision table testing to derive test cases

FL-4.2.4 (K3) Use state transition testing to derive test cases

4.3 White-box Test Techniques

FL-4.3.1 (K2) Explain statement testing

FL-4.3.2 (K2) Explain branch testing

FL-4.3.3 (K2) Explain the value of white-box testing

4.4 Experience-based Test Techniques

FL-4.4.1 (K2) Explain error guessing

FL-4.4.2 (K2) Explain exploratory testing

FL-4.4.3 (K2) Explain checklist-based testing

4.5. Collaboration-based Test Approaches

FL-4.5.1 (K2) Explain how to write user stories in collaboration with developers and business representatives

FL-4.5.2 (K2) Classify the different options for writing acceptance criteria

FL-4.5.3 (K3) Use acceptance test-driven development (ATDD) to derive test cases

4.1. Test Techniques Overview

Test techniques support the tester in test analysis (what to test) and in test design (how to test). Test techniques help to develop a relatively small, but sufficient, set of test cases in a systematic way. Test techniques also help the tester to define test conditions, identify coverage items, and identify test data during the test analysis and design. Further information on test techniques and their corresponding measures can be found in the ISO/IEC/IEEE 29119-4 standard, and in (Beizer 1990, Craig 2002, Copeland 2004, Koomen 2006, Jorgensen 2014, Ammann 2016, Forgács 2019).

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

Black-box test techniques (also known as specification-based techniques) are based on an analysis of the specified behavior of the test object without reference to its internal structure. Therefore, the test cases are independent of how the software is implemented. Consequently, if the implementation changes, but the required behavior stays the same, then the test cases are still useful.

White-box test techniques (also known as structure-based techniques) are based on an analysis of the test object's internal structure and processing. As the test cases are dependent on how the software is designed, they can only be created after the design or implementation of the test object.

Experience-based test techniques effectively use the knowledge and experience of testers for the design and implementation of test cases. The effectiveness of these techniques depends heavily on the tester's skills. Experience-based test techniques can detect defects that may be missed using the black-box and white-box test techniques. Hence, experience-based test techniques are complementary to the black-box and white-box test techniques.

4.2. Black-Box Test Techniques

Commonly used black-box test techniques discussed in the following sections are:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State Transition Testing

4.2.1. Equivalence Partitioning

Equivalence Partitioning (EP) divides data into partitions (known as equivalence partitions) based on the expectation that all the elements of a given partition are to be processed in the same way by the test object. The theory behind this technique is that if a test case, that tests one value from an equivalence partition, detects a defect, this defect should also be detected by test cases that test any other value from the same partition. Therefore, one test for each partition is sufficient.

Equivalence partitions can be identified for any data element related to the test object, including inputs, outputs, configuration items, internal values, time-related values, and interface parameters. The partitions may be continuous or discrete, ordered or unordered, finite or infinite. The partitions must not overlap and must be non-empty sets.

For simple test objects EP can be easy, but in practice, understanding how the test object will treat different values is often complicated. Therefore, partitioning should be done with care.

A partition containing valid values is called a valid partition. A partition containing invalid values is called an invalid partition. The definitions of valid and invalid values may vary among teams and organizations. For example, valid values may be interpreted as those that should be processed by the test object or as those for which the specification defines their processing. Invalid values may be interpreted as those that should be ignored or rejected by the test object or as those for which no processing is defined in the test object specification.

In EP, the coverage items are the equivalence partitions. To achieve 100% coverage with this technique, test cases must exercise all identified partitions (including invalid partitions) by covering each partition at least once. Coverage is measured as the number of partitions exercised by at least one test case, divided by the total number of identified partitions, and is expressed as a percentage.

Many test objects include multiple sets of partitions (e.g., test objects with more than one input parameter), which means that a test case will cover partitions from different sets of partitions. The simplest coverage criterion in the case of multiple sets of partitions is called Each Choice coverage (Ammann 2016). Each Choice coverage requires test cases to exercise each partition from each set of partitions at least once. Each Choice coverage does not take into account combinations of partitions.

4.2.2. Boundary Value Analysis

Boundary Value Analysis (BVA) is a technique based on exercising the boundaries of equivalence partitions. Therefore, BVA can only be used for ordered partitions. The minimum and maximum values of a partition are its boundary values. In the case of BVA, if two elements belong to the same partition, all elements between them must also belong to that partition.

BVA focuses on the boundary values of the partitions because developers are more likely to make errors with these boundary values. Typical defects found by BVA are located where implemented boundaries are misplaced to positions above or below their intended positions or are omitted altogether.

This syllabus covers two versions of the BVA: 2-value and 3-value BVA. They differ in terms of coverage items per boundary that need to be exercised to achieve 100% coverage.

In 2-value BVA (Craig 2002, Myers 2011), for each boundary value there are two coverage items: this boundary value and its closest neighbor belonging to the adjacent partition. To achieve 100% coverage with 2-value BVA, test cases must exercise all coverage items, i.e., all identified boundary values. Coverage is measured as the number of boundary values that were exercised, divided by the total number of identified boundary values, and is expressed as a percentage.

In 3-value BVA (Koomen 2006, O'Regan 2019), for each boundary value there are three coverage items: this boundary value and both its neighbors. Therefore, in 3-value BVA some of the coverage items may not be boundary values. To achieve 100% coverage with 3-value BVA, test cases must exercise all coverage items, i.e., identified boundary values and their neighbors. Coverage is measured as the number of boundary values and their neighbors exercised, divided by the total number of identified boundary values and their neighbors, and is expressed as a percentage.

3-value BVA is more rigorous than 2-value BVA as it may detect defects overlooked by 2-value BVA. For example, if the decision "if ($x \leq 10$) …" is incorrectly implemented as "if ($x = 10$) …", no test data derived from the 2-value BVA ($x = 10$, $x = 11$) can detect the defect. However, $x = 9$, derived from the 3-value BVA, is likely to detect it.

4.2.3. Decision Table Testing

Decision tables are used for testing the implementation of system requirements that specify how different combinations of conditions result in different outcomes. Decision tables are an effective way of recording complex logic, such as business rules.

When creating decision tables, the conditions and the resulting actions of the system are defined. These form the rows of the table. Each column corresponds to a decision rule that defines a unique combination of conditions, along with the associated actions. In limited-entry decision tables all the values of the conditions and actions (except for irrelevant or infeasible ones; see below) are shown as Boolean values (true or false). Alternatively, in extended-entry decision tables some or all the conditions and actions may also take on multiple values (e.g., ranges of numbers, equivalence partitions, discrete values).

The notation for conditions is as follows: “T” (true) means that the condition is satisfied. “F” (false) means that the condition is not satisfied. “–” means that the value of the condition is irrelevant for the action outcome. “N/A” means that the condition is infeasible for a given rule. For actions: “X” means that the action should occur. Blank means that the action should not occur. Other notations may also be used.

A full decision table has enough columns to cover every combination of conditions. The table can be simplified by deleting columns containing infeasible combinations of conditions. The table can also be minimized by merging columns, in which some conditions do not affect the outcome, into a single column. Decision table minimization algorithms are out of scope of this syllabus.

In decision table testing, the coverage items are the columns containing feasible combinations of conditions. To achieve 100% coverage with this technique, test cases must exercise all these columns. Coverage is measured as the number of exercised columns, divided by the total number of feasible columns, and is expressed as a percentage.

The strength of decision table testing is that it provides a systematic approach to identify all the combinations of conditions, some of which might otherwise be overlooked. It also helps to find any gaps or contradictions in the requirements. If there are many conditions, exercising all the decision rules may be time consuming, since the number of rules grows exponentially with the number of conditions. In such a case, to reduce the number of rules that need to be exercised, a minimized decision table or a risk-based approach may be used.

4.2.4. State Transition Testing

A state transition diagram models the behavior of a system by showing its possible states and valid state transitions. A transition is initiated by an event, which may be additionally qualified by a guard condition. The transitions are assumed to be instantaneous and may sometimes result in the software taking action. The common transition labeling syntax is as follows: “event [guard condition] / action”. Guard conditions and actions can be omitted if they do not exist or are irrelevant for the tester.

A state table is a model equivalent to a state transition diagram. Its rows represent states, and its columns represent events (together with guard conditions if they exist). Table entries (cells) represent transitions, and contain the target state, as well as the resulting actions, if defined. In contrast to the state transition diagram, the state table explicitly shows invalid transitions, which are represented by empty cells.

A test case based on a state transition diagram or state table is usually represented as a sequence of events, which results in a sequence of state changes (and actions, if needed). One test case may, and usually will, cover several transitions between states.

There exist many coverage criteria for state transition testing. This syllabus discusses three of them.

In **all states coverage**, the coverage items are the states. To achieve 100% all states coverage, test cases must ensure that all the states are visited. Coverage is measured as the number of visited states divided by the total number of states, and is expressed as a percentage.

In **valid transitions coverage** (also called 0-switch coverage), the coverage items are single valid transitions. To achieve 100% valid transitions coverage, test cases must exercise all the valid transitions. Coverage is measured as the number of exercised valid transitions divided by the total number of valid transitions, and is expressed as a percentage.

In **all transitions coverage**, the coverage items are all the transitions shown in a state table. To achieve 100% all transitions coverage, test cases must exercise all the valid transitions and attempt to execute invalid transitions. Testing only one invalid transition in a single test case helps to avoid fault masking, i.e., a situation in which one defect prevents the detection of another. Coverage is measured as the number of valid and invalid transitions exercised or attempted to be covered by executed test cases, divided by the total number of valid and invalid transitions, and is expressed as a percentage.

All states coverage is weaker than valid transitions coverage, because it can typically be achieved without exercising all the transitions. Valid transitions coverage is the most widely used coverage criterion. Achieving full valid transitions coverage guarantees full all states coverage. Achieving full all transitions coverage guarantees both full all states coverage and full valid transitions coverage and should be a minimum requirement for mission and safety-critical software.

4.3. White-Box Test Techniques

Because of their popularity and simplicity, this section focuses on two code-related white-box test techniques:

- Statement testing
- Branch testing

There are more rigorous techniques that are used in some safety-critical, mission-critical, or high-integrity environments to achieve more thorough code coverage. There are also white-box test techniques used in higher test levels (e.g., API testing), or using coverage not related to code (e.g., neuron coverage in neural network testing). These techniques are not discussed in this syllabus.

4.3.1. Statement Testing and Statement Coverage

In statement testing, the coverage items are executable statements. The aim is to design test cases that exercise statements in the code until an acceptable level of coverage is achieved. Coverage is measured as the number of statements exercised by the test cases divided by the total number of executable statements in the code, and is expressed as a percentage.

When 100% statement coverage is achieved, it ensures that all executable statements in the code have been exercised at least once. In particular, this means that each statement with a defect will be executed, which may cause a failure demonstrating the presence of the defect. However, exercising a statement with a test case will not detect defects in all cases. For example, it may not detect defects that are data dependent (e.g., a division by zero that only fails when a denominator is set to zero). Also, 100% statement coverage does not ensure that all the decision logic has been tested as, for instance, it may not exercise all the branches (see chapter 4.3.2) in the code.

4.3.2. Branch Testing and Branch Coverage

A branch is a transfer of control between two nodes in the control flow graph, which shows the possible sequences in which source code statements are executed in the test object. Each transfer of control can be either unconditional (i.e., straight-line code) or conditional (i.e., a decision outcome).

In branch testing the coverage items are branches and the aim is to design test cases to exercise branches in the code until an acceptable level of coverage is achieved. Coverage is measured as the number of branches exercised by the test cases divided by the total number of branches, and is expressed as a percentage.

When 100% branch coverage is achieved, all branches in the code, unconditional and conditional, are exercised by test cases. Conditional branches typically correspond to a true or false outcome from an "if...then" decision, an outcome from a switch/case statement, or a decision to exit or continue in a loop. However, exercising a branch with a test case will not detect defects in all cases. For example, it may not detect defects requiring the execution of a specific path in a code.

Branch coverage subsumes statement coverage. This means that any set of test cases achieving 100% branch coverage also achieves 100% statement coverage (but not vice versa).

4.3.3. The Value of White-box Testing

A fundamental strength that all white-box techniques share is that the entire software implementation is taken into account during testing, which facilitates defect detection even when the software specification is vague, outdated or incomplete. A corresponding weakness is that if the software does not implement one or more requirements, white box testing may not detect the resulting defects of omission (Watson 1996).

White-box techniques can be used in static testing (e.g., during dry runs of code). They are well suited to reviewing code that is not yet ready for execution (Hetzel 1988), as well as pseudocode and other high-level or top-down logic which can be modeled with a control flow graph.

Performing only black-box testing does not provide a measure of actual code coverage. White-box coverage measures provide an objective measurement of coverage and provide the necessary information to allow additional tests to be generated to increase this coverage, and subsequently increase confidence in the code.

4.4. Experience-based Test Techniques

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

- Error guessing
- Exploratory testing
- Checklist-based testing

4.4.1. Error Guessing

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

- How the application has worked in the past

- The types of errors the developers tend to make and the types of defects that result from these errors
- The types of failures that have occurred in other, similar applications

In general, errors, defects and failures may be related to: input (e.g., correct input not accepted, parameters wrong or missing), output (e.g., wrong format, wrong result), logic (e.g., missing cases, wrong operator), computation (e.g., incorrect operand, wrong computation), interfaces (e.g., parameter mismatch, incompatible types), or data (e.g., incorrect initialization, wrong type).

Fault attacks are a methodical approach to the implementation of error guessing. This technique requires the tester to create or acquire a list of possible errors, defects and failures, and to design tests that will identify defects associated with the errors, expose the defects, or cause the failures. These lists can be built based on experience, defect and failure data, or from common knowledge about why software fails.

See (Whittaker 2002, Whittaker 2003, Andrews 2006) for more information on error guessing and fault attacks.

4.4.2. Exploratory Testing

In exploratory testing, tests are simultaneously designed, executed, and evaluated while the tester learns about the test object. The testing is used to learn more about the test object, to explore it more deeply with focused tests, and to create tests for untested areas.

Exploratory testing is sometimes conducted using session-based testing to structure the testing. In a session-based approach, exploratory testing is conducted within a defined time-box. The tester uses a test charter containing test objectives to guide the testing. The test session is usually followed by a debriefing that involves a discussion between the tester and stakeholders interested in the test results of the test session. In this approach test objectives may be treated as high-level test conditions. Coverage items are identified and exercised during the test session. The tester may use test session sheets to document the steps followed and the discoveries made.

Exploratory testing is useful when there are few or inadequate specifications or there is significant time pressure on the testing. Exploratory testing is also useful to complement other more formal test techniques. Exploratory testing will be more effective if the tester is experienced, has domain knowledge and has a high degree of essential skills, like analytical skills, curiosity and creativeness (see section 1.5.1).

Exploratory testing can incorporate the use of other test techniques (e.g., equivalence partitioning). More information about exploratory testing can be found in (Kaner 1999, Whittaker 2009, Hendrickson 2013).

4.4.3. Checklist-Based Testing

In checklist-based testing, a tester designs, implements, and executes tests to cover test conditions from a checklist. Checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails. Checklists should not contain items that can be checked automatically, items better suited as entry/exit criteria, or items that are too general (Brykczynski 1999).

Checklist items are often phrased in the form of a question. It should be possible to check each item separately and directly. These items may refer to requirements, graphical interface properties, quality characteristics or other forms of test conditions. Checklists can be created to support various test types, including functional and non-functional testing (e.g., 10 heuristics for usability testing (Nielsen 1994)).

Some checklist entries may gradually become less effective over time because the developers will learn to avoid making the same errors. New entries may also need to be added to reflect newly found high severity defects. Therefore, checklists should be regularly updated based on defect analysis. However, care should be taken to avoid letting the checklist become too long (Gawande 2009).

In the absence of detailed test cases, checklist-based testing can provide guidelines and some degree of consistency for the testing. If the checklists are high-level, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

4.5. Collaboration-based Test Approaches

Each of the above-mentioned techniques (see sections 4.2, 4.3, 4.4) has a particular objective with respect to defect detection. Collaboration-based approaches, on the other hand, focus also on defect avoidance by collaboration and communication.

4.5.1. Collaborative User Story Writing

A user story represents a feature that will be valuable to either a user or purchaser of a system or software. User stories have three critical aspects (Jeffries 2000), called together the “3 C’s”:

- Card – the medium describing a user story (e.g., an index card, an entry in an electronic board)
- Conversation – explains how the software will be used (can be documented or verbal)
- Confirmation – the acceptance criteria (see section 4.5.2)

The most common format for a user story is “As a [role], I want [goal to be accomplished], so that I can [resulting business value for the role]”, followed by the acceptance criteria.

Collaborative authorship of the user story can use techniques such as brainstorming and mind mapping. The collaboration allows the team to obtain a shared vision of what should be delivered, by taking into account three perspectives: business, development and testing.

Good user stories should be: Independent, Negotiable, Valuable, Estimable, Small and Testable (INVEST). If a stakeholder does not know how to test a user story, this may indicate that the user story is not clear enough, or that it does not reflect something valuable to them, or that the stakeholder just needs help in testing (Wake 2003).

4.5.2. Acceptance Criteria

Acceptance criteria for a user story are the conditions that an implementation of the user story must meet to be accepted by stakeholders. From this perspective, acceptance criteria may be viewed as the test conditions that should be exercised by the tests. Acceptance criteria are usually a result of the Conversation (see section 4.5.1).

Acceptance criteria are used to:

- Define the scope of the user story
- Reach consensus among the stakeholders
- Describe both positive and negative scenarios
- Serve as a basis for the user story acceptance testing (see section 4.5.3)

- Allow accurate planning and estimation

There are several ways to write acceptance criteria for a user story. The two most common formats are:

- Scenario-oriented (e.g., Given/When/Then format used in BDD, see section 2.1.3)
- Rule-oriented (e.g., bullet point verification list, or tabulated form of input-output mapping)

Most acceptance criteria can be documented in one of these two formats. However, the team may use another, custom format, as long as the acceptance criteria are well-defined and unambiguous.

4.5.3. Acceptance Test-driven Development (ATDD)

ATDD is a test-first approach (see section 2.1.3). Test cases are created prior to implementing the user story. The test cases are created by team members with different perspectives, e.g., customers, developers, and testers (Adzic 2009). Test cases may be executed manually or automated.

The first step is a specification workshop where the user story and (if not yet defined) its acceptance criteria are analyzed, discussed, and written by the team members. Incompleteness, ambiguities, or defects in the user story are resolved during this process. The next step is to create the test cases. This can be done by the team as a whole or by the tester individually. The test cases are based on the acceptance criteria and can be seen as examples of how the software works. This will help the team implement the user story correctly.

Since examples and tests are the same, these terms are often used interchangeably. During the test design the test techniques described in sections 4.2, 4.3 and 4.4 may be applied.

Typically, the first test cases are positive, confirming the correct behavior without exceptions or error conditions, and comprising the sequence of activities executed if everything goes as expected. After the positive test cases are done, the team should perform negative testing. Finally, the team should cover non-functional quality characteristics as well (e.g., performance efficiency, usability). Test cases should be expressed in a way that is understandable for the stakeholders. Typically, test cases contain sentences in natural language involving the necessary preconditions (if any), the inputs, and the postconditions.

The test cases must cover all the characteristics of the user story and should not go beyond the story. However, the acceptance criteria may detail some of the issues described in the user story. In addition, no two test cases should describe the same characteristics of the user story.

When captured in a format supported by a test automation framework, the developers can automate the test cases by writing the supporting code as they implement the feature described by a user story. The acceptance tests then become executable requirements.

5. Managing the Test Activities – 335 minutes

Keywords

defect management, defect report, entry criteria, exit criteria, product risk, project risk, risk, risk analysis, risk assessment, risk control, risk identification, risk level, risk management, risk mitigation, risk monitoring, risk-based testing, test approach, test completion report, test control, test monitoring, test plan, test planning, test progress report, test pyramid, testing quadrants

Learning Objectives for Chapter 5:

5.1 Test Planning

FL-5.1.1 (K2) Exemplify the purpose and content of a test plan

FL-5.1.2 (K1) Recognize how a tester adds value to iteration and release planning

FL-5.1.3 (K2) Compare and contrast entry criteria and exit criteria

FL-5.1.4 (K3) Use estimation techniques to calculate the required test effort

FL-5.1.5 (K3) Apply test case prioritization

FL-5.1.6 (K1) Recall the concepts of the test pyramid

FL-5.1.7 (K2) Summarize the testing quadrants and their relationships with test levels and test types

5.2 Risk Management

FL-5.2.1 (K1) Identify risk level by using risk likelihood and risk impact

FL-5.2.2 (K2) Distinguish between project risks and product risks

FL-5.2.3 (K2) Explain how product risk analysis may influence thoroughness and scope of testing

FL-5.2.4 (K2) Explain what measures can be taken in response to analyzed product risks

5.3 Test Monitoring, Test Control and Test Completion

FL-5.3.1 (K1) Recall metrics used for testing

FL-5.3.2 (K2) Summarize the purposes, content, and audiences for test reports

FL-5.3.3 (K2) Exemplify how to communicate the status of testing

5.4 Configuration Management

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

5.5 Defect Management

FL-5.5.1 (K3) Prepare a defect report

5.1. Test Planning

5.1.1. Purpose and Content of a Test Plan

A test plan describes the objectives, resources and processes for a test project. A test plan:

- Documents the means and schedule for achieving test objectives
- Helps to ensure that the performed test activities will meet the established criteria
- Serves as a means of communication with team members and other stakeholders
- Demonstrates that testing will adhere to the existing test policy and test strategy (or explains why the testing will deviate from them)

Test planning guides the testers' thinking and forces the testers to confront the future challenges related to risks, schedules, people, tools, costs, effort, etc. The process of preparing a test plan is a useful way to think through the efforts needed to achieve the test project objectives.

The typical content of a test plan includes:

- Context of testing (e.g., scope, test objectives, constraints, test basis)
- Assumptions and constraints of the test project
- Stakeholders (e.g., roles, responsibilities, relevance to testing, hiring and training needs)
- Communication (e.g., forms and frequency of communication, documentation templates)
- Risk register (e.g., product risks, project risks)
- Test approach (e.g., test levels, test types, test techniques, test deliverables, entry criteria and exit criteria, independence of testing, metrics to be collected, test data requirements, test environment requirements, deviations from the organizational test policy and test strategy)
- Budget and schedule

More details about the test plan and its content can be found in the ISO/IEC/IEEE 29119-3 standard.

5.1.2. Tester's Contribution to Iteration and Release Planning

In iterative SDLCs, typically two kinds of planning occur: release planning and iteration planning.

Release planning looks ahead to the release of a product, defines and re-defines the product backlog, and may involve refining larger user stories into a set of smaller user stories. It also serves as the basis for the test approach and test plan across all iterations. Testers involved in release planning participate in writing testable user stories and acceptance criteria (see section 4.5), participate in project and quality risk analyses (see section 5.2), estimate test effort associated with user stories (see section 5.1.4), determine the test approach, and plan the testing for the release.

Iteration planning looks ahead to the end of a single iteration and is concerned with the iteration backlog. Testers involved in iteration planning participate in the detailed risk analysis of user stories, determine the testability of user stories, break down user stories into tasks (particularly testing tasks), estimate test effort for all testing tasks, and identify and refine functional and non-functional aspects of the test object.

5.1.3. Entry Criteria and Exit Criteria

Entry criteria define the preconditions for undertaking a given activity. If entry criteria are not met, it is likely that the activity will prove to be more difficult, time-consuming, costly, and riskier. Exit criteria define what must be achieved in order to declare an activity completed. Entry criteria and exit criteria should be defined for each test level, and will differ based on the test objectives.

Typical entry criteria include: availability of resources (e.g., people, tools, environments, test data, budget, time), availability of testware (e.g., test basis, testable requirements, user stories, test cases), and initial quality level of a test object (e.g., all smoke tests have passed).

Typical exit criteria include: measures of thoroughness (e.g., achieved level of coverage, number of unresolved defects, defect density, number of failed test cases), and completion criteria (e.g., planned tests have been executed, static testing has been performed, all defects found are reported, all regression tests are automated).

Running out of time or budget can also be viewed as valid exit criteria. Even without other exit criteria being satisfied, it can be acceptable to end testing under such circumstances, if the stakeholders have reviewed and accepted the risk to go live without further testing.

In Agile software development, exit criteria are often called Definition of Done, defining the team's objective metrics for a releasable item. Entry criteria that a user story must fulfill to start the development and/or testing activities are called Definition of Ready.

5.1.4. Estimation Techniques

Test effort estimation involves predicting the amount of test-related work needed to meet the objectives of a test project. It is important to make it clear to the stakeholders that the estimate is based on a number of assumptions and is always subject to estimation error. Estimation for small tasks is usually more accurate than for the large ones. Therefore, when estimating a large task, it can be decomposed into a set of smaller tasks which then in turn can be estimated.

In this syllabus, the following four estimation techniques are described.

Estimation based on ratios. In this metrics-based technique, figures are collected from previous projects within the organization, which makes it possible to derive "standard" ratios for similar projects. The ratios of an organization's own projects (e.g., taken from historical data) are generally the best source to use in the estimation process. These standard ratios can then be used to estimate the test effort for the new project. For example, if in the previous project the development-to-test effort ratio was 3:2, and in the current project the development effort is expected to be 600 person-days, the test effort can be estimated to be 400 person-days.

Extrapolation. In this metrics-based technique, measurements are made as early as possible in the current project to gather the data. Having enough observations, the effort required for the remaining work can be approximated by extrapolating this data (usually by applying a mathematical model). This method is very suitable in iterative SDLCs. For example, the team may extrapolate the test effort in the forthcoming iteration as the averaged effort from the last three iterations.

Wideband Delphi. In this iterative, expert-based technique, experts make experience-based estimations. Each expert, in isolation, estimates the effort. The results are collected and if there are deviations that are out of range of the agreed upon boundaries, the experts discuss their current estimates. Each expert is then asked to make a new estimation based on that feedback, again in isolation. This process is repeated until a consensus is reached. Planning Poker is a variant of Wideband Delphi, commonly used in Agile

software development. In Planning Poker, estimates are usually made using cards with numbers that represent the effort size.

Three-point estimation. In this expert-based technique, three estimations are made by the experts: the most optimistic estimation (a), the most likely estimation (m) and the most pessimistic estimation (b). The final estimate (E) is their weighted arithmetic mean. In the most popular version of this technique, the estimate is calculated as E = (a + 4*m + b) / 6. The advantage of this technique is that it allows the experts to calculate the measurement error: SD = (b – a) / 6. For example, if the estimates (in person-hours) are: a=6, m=9 and b=18, then the final estimation is 10±2 person-hours (i.e., between 8 and 12 person-hours), because E = (6 + 4*9 + 18) / 6 = 10 and SD = (18 – 6) / 6 = 2.

See (Kan 2003, Koomen 2006, Westfall 2009) for these and many other test estimation techniques.

5.1.5. Test Case Prioritization

Once the test cases and test procedures are specified and assembled into test suites, these test suites can be arranged in a test execution schedule that defines the order in which they are to be run. When prioritizing test cases, different factors can be taken into account. The most commonly used test case prioritization strategies are as follows:

- Risk-based prioritization, where the order of test execution is based on the results of risk analysis (see section 5.2.3). Test cases covering the most important risks are executed first.
- Coverage-based prioritization, where the order of test execution is based on coverage (e.g., statement coverage). Test cases achieving the highest coverage are executed first. In another variant, called additional coverage prioritization, the test case achieving the highest coverage is executed first; each subsequent test case is the one that achieves the highest additional coverage.
- Requirements-based prioritization, where the order of test execution is based on the priorities of the requirements traced back to the corresponding test cases. Requirement priorities are defined by stakeholders. Test cases related to the most important requirements are executed first.

Ideally, test cases would be ordered to run based on their priority levels, using, for example, one of the above-mentioned prioritization strategies. However, this practice may not work if the test cases or the features being tested have dependencies. If a test case with a higher priority is dependent on a test case with a lower priority, the lower priority test case must be executed first.

The order of test execution must also take into account the availability of resources. For example, the required test tools, test environments or people that may only be available for a specific time window.

5.1.6. Test Pyramid

The test pyramid is a model showing that different tests may have different granularity. The test pyramid model supports the team in test automation and in test effort allocation by showing that different goals are supported by different levels of test automation. The pyramid layers represent groups of tests. The higher the layer, the lower the test granularity, test isolation and test execution time. Tests in the bottom layer are small, isolated, fast, and check a small piece of functionality, so usually a lot of them are needed to achieve a reasonable coverage. The top layer represents complex, high-level, end-to-end tests. These high-level tests are generally slower than the tests from the lower layers, and they typically check a large piece of functionality, so usually just a few of them are needed to achieve a reasonable coverage. The number and naming of the layers may differ. For example, the original test pyramid model (Cohn 2009) defines three layers: “unit tests”, “service tests” and “UI tests”. Another popular model defines unit

(component) tests, integration (component integration) tests, and end-to-end tests. Other test levels (see section 2.2.1) can also be used.

5.1.7. Testing Quadrants

The testing quadrants, defined by Brian Marick (Marick 2003, Crispin 2008), group the test levels with the appropriate test types, activities, test techniques and work products in the Agile software development. The model supports test management in visualizing these to ensure that all appropriate test types and test levels are included in the SDLC and in understanding that some test types are more relevant to certain test levels than others. This model also provides a way to differentiate and describe the types of tests to all stakeholders, including developers, testers, and business representatives.

In this model, tests can be business facing or technology facing. Tests can also support the team (i.e., guide the development) or critique the product (i.e., measure its behavior against the expectations). The combination of these two viewpoints determines the four quadrants:

- Quadrant Q1 (technology facing, support the team). This quadrant contains component and component integration tests. These tests should be automated and included in the CI process.
- Quadrant Q2 (business facing, support the team). This quadrant contains functional tests, examples, user story tests, user experience prototypes, API testing, and simulations. These tests check the acceptance criteria and can be manual or automated.
- Quadrant Q3 (business facing, critique the product). This quadrant contains exploratory testing, usability testing, user acceptance testing. These tests are user-oriented and often manual.
- Quadrant Q4 (technology facing, critique the product). This quadrant contains smoke tests and non-functional tests (except usability tests). These tests are often automated.

5.2. Risk Management

Organizations face many internal and external factors that make it uncertain whether and when they will achieve their objectives (ISO 31000). Risk management allows the organizations to increase the likelihood of achieving objectives, improve the quality of their products and increase the stakeholders' confidence and trust.

The main risk management activities are:

- Risk analysis (consisting of risk identification and risk assessment; see section 5.2.3)
- Risk control (consisting of risk mitigation and risk monitoring; see section 5.2.4)

The test approach, in which test activities are selected, prioritized, and managed based on risk analysis and risk control, is called risk-based testing.

5.2.1. Risk Definition and Risk Attributes

Risk is a potential event, hazard, threat, or situation whose occurrence causes an adverse effect. A risk can be characterized by two factors:

- Risk likelihood – the probability of the risk occurrence (greater than zero and less than one)
- Risk impact (harm) – the consequences of this occurrence

These two factors express the risk level, which is a measure for the risk. The higher the risk level, the more important is its treatment.

5.2.2. Project Risks and Product Risks

In software testing one is generally concerned with two types of risks: project risks and product risks.

Project risks are related to the management and control of the project. Project risks include:

- Organizational issues (e.g., delays in work products deliveries, inaccurate estimates, cost-cutting)
- People issues (e.g., insufficient skills, conflicts, communication problems, shortage of staff)
- Technical issues (e.g., scope creep, poor tool support)
- Supplier issues (e.g., third-party delivery failure, bankruptcy of the supporting company)

Project risks, when they occur, may have an impact on the project schedule, budget or scope, which affects the project's ability to achieve its objectives.

Product risks are related to the product quality characteristics (e.g., described in the ISO 25010 quality model). Examples of product risks include: missing or wrong functionality, incorrect calculations, runtime errors, poor architecture, inefficient algorithms, inadequate response time, poor user experience, security vulnerabilities. Product risks, when they occur, may result in various negative consequences, including:

- User dissatisfaction
- Loss of revenue, trust, reputation
- Damage to third parties
- High maintenance costs, overload of the helpdesk
- Criminal penalties
- In extreme cases, physical damage, injuries or even death

5.2.3. Product Risk Analysis

From a testing perspective, the goal of product risk analysis is to provide an awareness of product risk in order to focus the testing effort in a way that minimizes the residual level of product risk. Ideally, product risk analysis begins early in the SDLC.

Product risk analysis consists of risk identification and risk assessment. Risk identification is about generating a comprehensive list of risks. Stakeholders can identify risks by using various techniques and tools, e.g., brainstorming, workshops, interviews, or cause-effect diagrams. Risk assessment involves: categorization of identified risks, determining their risk likelihood, risk impact and level, prioritizing, and proposing ways to handle them. Categorization helps in assigning mitigation actions, because usually risks falling into the same category can be mitigated using a similar approach.

Risk assessment can use a quantitative or qualitative approach, or a mix of them. In the quantitative approach the risk level is calculated as the multiplication of risk likelihood and risk impact. In the qualitative approach the risk level can be determined using a risk matrix.

Product risk analysis may influence the thoroughness and scope of testing. Its results are used to:

- Determine the scope of testing to be carried out
- Determine the particular test levels and propose test types to be performed
- Determine the test techniques to be employed and the coverage to be achieved
- Estimate the test effort required for each task
- Prioritize testing in an attempt to find the critical defects as early as possible
- Determine whether any activities in addition to testing could be employed to reduce risk

5.2.4. Product Risk Control

Product risk control comprises all measures that are taken in response to identified and assessed product risks. Product risk control consists of risk mitigation and risk monitoring. Risk mitigation involves implementing the actions proposed in risk assessment to reduce the risk level. The aim of risk monitoring is to ensure that the mitigation actions are effective, to obtain further information to improve risk assessment, and to identify emerging risks.

With respect to product risk control, once a risk has been analyzed, several response options to risk are possible, e.g., risk mitigation by testing, risk acceptance, risk transfer, or contingency plan (Veenendaal 2012). Actions that can be taken to mitigate the product risks by testing are as follows:

- Select the testers with the right level of experience and skills, suitable for a given risk type
- Apply an appropriate level of independence of testing
- Conduct reviews and perform static analysis
- Apply the appropriate test techniques and coverage levels
- Apply the appropriate test types addressing the affected quality characteristics
- Perform dynamic testing, including regression testing

5.3. Test Monitoring, Test Control and Test Completion

Test monitoring is concerned with gathering information about testing. This information is used to assess test progress and to measure whether the test exit criteria or the test tasks associated with the exit criteria are satisfied, such as meeting the targets for coverage of product risks, requirements, or acceptance criteria.

Test control uses the information from test monitoring to provide, in a form of the control directives, guidance and the necessary corrective actions to achieve the most effective and efficient testing. Examples of control directives include:

- Reprioritizing tests when an identified risk becomes an issue
- Re-evaluating whether a test item meets entry criteria or exit criteria due to rework
- Adjusting the test schedule to address a delay in the delivery of the test environment
- Adding new resources when and where needed

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

5.3.1. Metrics used in Testing

Test metrics are gathered to show progress against the planned schedule and budget, the current quality of the test object, and the effectiveness of the test activities with respect to the objectives or an iteration goal. Test monitoring gathers a variety of metrics to support the test control and test completion.

Common test metrics include:

- Project progress metrics (e.g., task completion, resource usage, test effort)
- Test progress metrics (e.g., test case implementation progress, test environment preparation progress, number of test cases run/not run, passed/failed, test execution time)
- Product quality metrics (e.g., availability, response time, mean time to failure)
- Defect metrics (e.g., number and priorities of defects found/fixed, defect density, defect detection percentage)
- Risk metrics (e.g., residual risk level)
- Coverage metrics (e.g., requirements coverage, code coverage)
- Cost metrics (e.g., cost of testing, organizational cost of quality)

5.3.2. Purpose, Content and Audience for Test Reports

Test reporting summarizes and communicates test information during and after testing. Test progress reports support the ongoing control of the testing and must provide enough information to make modifications to the test schedule, resources, or test plan, when such changes are needed due to deviation from the plan or changed circumstances. Test completion reports summarize a specific stage of testing (e.g., test level, test cycle, iteration) and can give information for subsequent testing.

During test monitoring and control, the test team generates test progress reports for stakeholders to keep them informed. Test progress reports are usually generated on a regular basis (e.g., daily, weekly, etc.) and include:

- Test period
- Test progress (e.g., ahead or behind schedule), including any notable deviations
- Impediments for testing, and their workarounds
- Test metrics (see section 5.3.1 for examples)
- New and changed risks within testing period
- Testing planned for the next period

A test completion report is prepared during test completion, when a project, test level, or test type is complete and when, ideally, its exit criteria have been met. This report uses test progress reports and other data. Typical test completion reports include:

- Test summary
- Testing and product quality evaluation based on the original test plan (i.e., test objectives and exit criteria)
- Deviations from the test plan (e.g., differences from the planned schedule, duration, and effort).
- Testing impediments and workarounds
- Test metrics based on test progress reports
- Unmitigated risks, defects not fixed
- Lessons learned that are relevant to the testing

Different audiences require different information in the reports, and influence the degree of formality and the frequency of reporting. Reporting on test progress to others in the same team is often frequent and informal, while reporting on testing for a completed project follows a set template and occurs only once.

The ISO/IEC/IEEE 29119-3 standard includes templates and examples for test progress reports (called test status reports) and test completion reports.

5.3.3. Communicating the Status of Testing

The best means of communicating test status varies, depending on test management concerns, organizational test strategies, regulatory standards, or, in the case of self-organizing teams (see section 1.5.2), on the team itself. The options include:

- Verbal communication with team members and other stakeholders
- Dashboards (e.g., CI/CD dashboards, task boards, and burn-down charts)
- Electronic communication channels (e.g., email, chat)
- Online documentation
- Formal test reports (see section 5.3.2)

One or more of these options can be used. More formal communication may be more appropriate for distributed teams where direct face-to-face communication is not always possible due to geographical distance or time differences. Typically, different stakeholders are interested in different types of information, so communication should be tailored accordingly.

5.4. Configuration Management

In testing, configuration management (CM) provides a discipline for identifying, controlling, and tracking work products such as test plans, test strategies, test conditions, test cases, test scripts, test results, test logs, and test reports as configuration items.

For a complex configuration item (e.g., a test environment), CM records the items it consists of, their relationships, and versions. If the configuration item is approved for testing, it becomes a baseline and can only be changed through a formal change control process.

Configuration management keeps a record of changed configuration items when a new baseline is created. It is possible to revert to a previous baseline to reproduce previous test results.

To properly support testing, CM ensures the following:

- All configuration items, including test items (individual parts of the test object), are uniquely identified, version controlled, tracked for changes, and related to other configuration items so that traceability can be maintained throughout the test process
- All identified documentation and software items are referenced unambiguously in test documentation

Continuous integration, continuous delivery, continuous deployment and the associated testing are typically implemented as part of an automated DevOps pipeline (see section 2.1.4), in which automated CM is normally included.

5.5. Defect Management

Since one of the major test objectives is to find defects, an established defect management process is essential. Although we refer to "defects" here, the reported anomalies may turn out to be real defects or something else (e.g., false positive, change request) - this is resolved during the process of dealing with the defect reports. Anomalies may be reported during any phase of the SDLC and the form depends on the SDLC. At a minimum, the defect management process includes a workflow for handling individual anomalies from their discovery to their closure and rules for their classification. The workflow typically comprises activities to log the reported anomalies, analyze and classify them, decide on a suitable response such as to fix or keep it as it is and finally to close the defect report. The process must be followed by all involved stakeholders. It is advisable to handle defects from static testing (especially static analysis) in a similar way.

Typical defect reports have the following objectives:

- Provide those responsible for handling and resolving reported defects with sufficient information to resolve the issue
- Provide a means of tracking the quality of the work product
- Provide ideas for improvement of the development and test process

A defect report logged during dynamic testing typically includes:

- Unique identifier
- Title with a short summary of the anomaly being reported
- Date when the anomaly was observed, issuing organization, and author, including their role
- Identification of the test object and test environment
- Context of the defect (e.g., test case being run, test activity being performed, SDLC phase, and other relevant information such as the test technique, checklist or test data being used)

- Description of the failure to enable reproduction and resolution including the steps that detected the anomaly, and any relevant test logs, database dumps, screenshots, or recordings
- Expected results and actual results
- Severity of the defect (degree of impact) on the interests of stakeholders or requirements
- Priority to fix
- Status of the defect (e.g., open, deferred, duplicate, waiting to be fixed, awaiting confirmation testing, re-opened, closed, rejected)
- References (e.g., to the test case)

Some of this data may be automatically included when using defect management tools (e.g., identifier, date, author and initial status). Document templates for a defect report and example defect reports can be found in the ISO/IEC/IEEE 29119-3 standard, which refers to defect reports as incident reports.

6. Test Tools – 20 minutes

Keywords

test automation

Learning Objectives for Chapter 6:

6.1 Tool Support for Testing

FL-6.1.1 (K2) Explain how different types of test tools support testing

6.2 Benefits and Risks of Test Automation

FL-6.2.1 (K1) Recall the benefits and risks of test automation

6.1. Tool Support for Testing

Test tools support and facilitate many test activities. Examples include, but are not limited to:

- Management tools – increase the test process efficiency by facilitating management of the SDLC, requirements, tests, defects, configuration
- Static testing tools – support the tester in performing reviews and static analysis
- Test design and implementation tools – facilitate generation of test cases, test data and test procedures
- Test execution and coverage tools – facilitate automated test execution and coverage measurement
- Non-functional testing tools – allow the tester to perform non-functional testing that is difficult or impossible to perform manually
- DevOps tools – support the DevOps delivery pipeline, workflow tracking, automated build process(es), CI/CD
- Collaboration tools – facilitate communication
- Tools supporting scalability and deployment standardization (e.g., virtual machines, containerization tools)
- Any other tool that assists in testing (e.g., a spreadsheet is a test tool in the context of testing)

6.2. Benefits and Risks of Test Automation

Simply acquiring a tool does not guarantee success. Each new tool will require effort to achieve real and lasting benefits (e.g., for tool introduction, maintenance and training). There are also some risks, which need analysis and mitigation.

Potential benefits of using test automation include:

- Time saved by reducing repetitive manual work (e.g., execute regression tests, re-enter the same test data, compare expected results vs actual results, and check against coding standards)
- Prevention of simple human errors through greater consistency and repeatability (e.g., tests are consistently derived from requirements, test data is created in a systematic manner, and tests are executed by a tool in the same order with the same frequency)
- More objective assessment (e.g., coverage) and providing measures that are too complicated for humans to derive
- Easier access to information about testing to support test management and test reporting (e.g., statistics, graphs, and aggregated data about test progress, defect rates, and test execution duration)
- Reduced test execution times to provide earlier defect detection, faster feedback and faster time to market
- More time for testers to design new, deeper and more effective tests

Potential risks of using test automation include:

- Unrealistic expectations about the benefits of a tool (including functionality and ease of use).
- Inaccurate estimations of time, costs, effort required to introduce a tool, maintain test scripts and change the existing manual test process.
- Using a test tool when manual testing is more appropriate.
- Relying on a tool too much, e.g., ignoring the need of human critical thinking.
- The dependency on the tool vendor which may go out of business, retire the tool, sell the tool to a different vendor or provide poor support (e.g., responses to queries, upgrades, and defect fixes).
- Using an open-source software which may be abandoned, meaning that no further updates are available, or its internal components may require quite frequent updates as a further development.
- The automation tool is not compatible with the development platform.
- Choosing an unsuitable tool that did not comply with the regulatory requirements and/or safety standards.

7. References

Standards

ISO/IEC/IEEE 29119-1 (2022) Software and systems engineering – Software testing – Part 1: General Concepts

ISO/IEC/IEEE 29119-2 (2021) Software and systems engineering – Software testing – Part 2: Test processes

ISO/IEC/IEEE 29119-3 (2021) Software and systems engineering – Software testing – Part 3: Test documentation

ISO/IEC/IEEE 29119-4 (2021) Software and systems engineering – Software testing – Part 4: Test techniques

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

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

ISO/IEC/IEEE 14764:2022 – Software engineering – Software life cycle processes – Maintenance

ISO 31000 (2018) Risk management – Principles and guidelines

Books

Adzic, G. (2009) Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, Neuri Limited

Ammann, P. and Offutt, J. (2016) Introduction to Software Testing (2e), Cambridge University Press

Andrews, M. and Whittaker, J. (2006) How to Break Web Software: Functional and Security Testing of Web Applications and Web Services, Addison-Wesley Professional

Beck, K. (2003) Test Driven Development: By Example, Addison-Wesley

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

Boehm, B. (1981) Software Engineering Economics, Prentice Hall, Englewood Cliffs, NJ

Buxton, J.N. and Randell B., eds (1970), Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969, p. 16

Chelimsky, D. et al. (2010) The Rspec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends, The Pragmatic Bookshelf: Raleigh, NC

Cohn, M. (2009) Succeeding with Agile: Software Development Using Scrum, Addison-Wesley

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

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

Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams, Pearson Education: Boston MA

Forgács, I., and Kovács, A. (2019) Practical Test Design: Selection of traditional and automated test design techniques, BCS, The Chartered Institute for IT

Gawande A. (2009) The Checklist Manifesto: How to Get Things Right, New York, NY: Metropolitan Books

Gärtner, M. (2011), ATDD by Example: A Practical Guide to Acceptance Test-Driven Development, Pearson Education: Boston MA

Gilb, T., Graham, D. (1993) Software Inspection, Addison Wesley

Hendrickson, E. (2013) Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing, The Pragmatic Programmers

Hetzel, B. (1988) The Complete Guide to Software Testing, 2nd ed., John Wiley and Sons

Jeffries, R., Anderson, A., Hendrickson, C. (2000) Extreme Programming Installed, Addison-Wesley Professional

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

Kan, S. (2003) Metrics and Models in Software Quality Engineering, 2nd ed., Addison-Wesley

Kaner, C., Falk, J., and Nguyen, H.Q. (1999) Testing Computer Software, 2nd ed., Wiley

Kaner, C., Bach, J., and Pettichord, B. (2011) Lessons Learned in Software Testing: A Context-Driven Approach, 1st ed., Wiley

Kim, G., Humble, J., Debois, P. and Willis, J. (2016) The DevOps Handbook, Portland, OR

Koomen, T., van der Aalst, L., Broekman, B. and Vroon, M. (2006) TMap Next for result-driven testing, UTN Publishers, The Netherlands

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

O'Regan, G. (2019) Concise Guide to Software Testing, Springer Nature Switzerland

Pressman, R.S. (2019) Software Engineering. A Practitioner's Approach, 9th ed., McGraw Hill

Roman, A. (2018) Thinking-Driven Testing. The Most Reasonable Approach to Quality Control, Springer Nature Switzerland

Van Veenendaal, E (ed.) (2012) Practical Risk-Based Testing, The PRISMA Approach, UTN Publishers: The Netherlands

Watson, A.H., Wallace, D.R. and McCabe, T.J. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, U.S. Dept. of Commerce, Technology Administration, NIST

Westfall, L. (2009) The Certified Software Quality Engineer Handbook, ASQ Quality Press

Whittaker, J. (2002) How to Break Software: A Practical Guide to Testing, Pearson

Whittaker, J. (2009) Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design, Addison Wesley

Whittaker, J. and Thompson, H. (2003) How to Break Software Security, Addison Wesley

Wiegers, K. (2001) Peer Reviews in Software: A Practical Guide, Addison-Wesley Professional

Articles and Web Pages

Brykczynski, B. (1999) "A survey of software inspection checklists," *ACM SIGSOFT Software Engineering Notes, 24(1), pp. 82-89*

Enders, A. (1975) “An Analysis of Errors and Their Causes in System Programs,” *IEEE Transactions on Software Engineering 1(2), pp. 140-149*

Manna, Z., Waldinger, R. (1978) “The logic of computer programming,” *IEEE Transactions on Software Engineering 4(3), pp. 199-229*

Marick, B. (2003) Exploration through Example, http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1

Nielsen, J. (1994) “Enhancing the explanatory power of usability heuristics,” *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence, ACM Press, pp. 152–158*

Salman. I. (2016) “Cognitive biases in software quality and testing,” *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16), ACM, pp. 823-826.*

Wake, B. (2003) “INVEST in Good Stories, and SMART Tasks*,”* https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

8. Appendix A – Learning Objectives/Cognitive Level of Knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it. The learning objectives begin with an action verb corresponding to its cognitive level of knowledge as listed below.

Level 1: Remember (K1) – the candidate will remember, recognize and recall a term or concept.

Action verbs: identify, recall, remember, recognize.

Examples:

- "Identify typical test objectives ."
- "Recall the concepts of the test pyramid."
- "Recognize how a tester adds value to iteration and release planning"

Level 2: Understand (K2) – the candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify and give examples for the testing concept.

Action verbs: classify, compare, contrast, differentiate, distinguish, exemplify, explain, give examples, interpret, summarize.

Examples:

- "Classify the different options for writing acceptance criteria."
- "Compare the different roles in testing" (look for similarities, differences or both).
- "Distinguish between project risks and product risks" (allows concepts to be differentiated).
- "Exemplify the purpose and content of a test plan."
- "Explain the impact of context on the test process."
- "Summarize the activities of the review process."

Level 3: Apply (K3) – the candidate can carry out a procedure when confronted with a familiar task, or select the correct procedure and apply it to a given context.

Action verbs: apply, implement, prepare, use.

Examples:

- "Apply test case prioritization" (should refer to a procedure, technique, process, algorithm etc.).
- "Prepare a defect report."
- "Use boundary value analysis to derive test cases."

References for the cognitive levels of learning objectives:

Anderson, L. W. and Krathwohl, D. R. (eds) (2001) A Taxonomy for Learning, Teaching

Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Allyn & Bacon

9. Appendix B – Business Outcomes traceability matrix with Learning Objectives

This section lists the number of Foundation Level Learning Objectives related to the Business Outcomes and the traceability between Foundation Level Business Outcomes and Foundation Level Learning Objectives.

Business Outcomes: Foundation Level		FL-BO1	FL-BO2	FL-BO3	FL-BO4	FL-BO5	FL-BO6	FL-BO7	FL-BO8	FL-BO9	FL-BO10	FL-BO11	FL-BO12	FL-BO13	FL-BO14
BO1	Understand what testing is and why it is beneficial	6													
BO2	Understand fundamental concepts of software testing		22												
BO3	Identify the test approach and activities to be implemented depending on the context of testing			6											
BO4	Assess and improve the quality of documentation				9										
BO5	Increase the effectiveness and efficiency of testing					20									
BO6	Align the test process with the software development lifecycle						6								
BO7	Understand test management principles							6							
BO8	Write and communicate clear and understandable defect reports								1						
BO9	Understand the factors that influence the priorities and efforts related to testing									7					
BO10	Work as part of a cross-functional team										8				
BO11	Know risks and benefits related to test automation.											1			
BO12	Identify essential skills required for testing												5		
BO13	Understand the impact of risk on testing													4	
BO14	Effectively report on test progress and quality														4

Chapter/ section/ subsection	Learning objective	K-level	BUSINESS OUTCOMES														
			FL-BO1	FL-BO2	FL-BO3	FL-BO4	FL-BO5	FL-BO6	FL-BO7	FL-BO8	FL-BO9	FL-BO10	FL-BO11	FL-BO12	FL-BO13	FL-BO14
Chapter 1	**Fundamentals of Testing**																
1.1	**What is Testing?**																
1.1.1	Identify typical test objectives	K1	X														
1.1.2	Differentiate testing from debugging	K2		X													
1.2	**Why is Testing Necessary?**																
1.2.1	Exemplify why testing is necessary	K2	X														
1.2.2	Recall the relation between testing and quality assurance	K1		X													
1.2.3	Distinguish between root cause, error, defect, and failure	K2		X													
1.3	**Testing Principles**																
1.3.1	Explain the seven testing principles	K2		X													
1.4	**Test Activities, Testware and Test Roles**																
1.4.1	Summarize the different test activities and tasks	K2			X												
1.4.2	Explain the impact of context on the test process	K2			X			X									
1.4.3	Differentiate the testware that support the test activities	K2			X												
1.4.4	Explain the value of maintaining traceability	K2				X	X										
1.4.5	Compare the different roles in testing	K2										X					

1.5	**Essential Skills and Good Practices in Testing**															
1.5.1	Give examples of the generic skills required for testing	K2												X		
1.5.2	Recall the advantages of the whole team approach	K1										X				
1.5.3	Distinguish the benefits and drawbacks of independence of testing	K2			X											
Chapter 2	**Testing Throughout the Software Development Lifecycle**															
2.1	**Testing in the Context of a Software Development Lifecycle**															
2.1.1	Explain the impact of the chosen software development lifecycle on testing	K2						X								
2.1.2	Recall good testing practices that apply to all software development lifecycles	K1						X								
2.1.3	Recall the examples of test-first approaches to development	K1					X									
2.1.4	Summarize how DevOps might have an impact on testing	K2					X	X			X	X				
2.1.5	Explain the shift-left approach	K2					X	X								
2.1.6	Explain how retrospectives can be used as a mechanism for process improvement	K2					X					X				
2.2	**Test Levels and Test Types**															
2.2.1	Distinguish the different test levels	K2		X	X											
2.2.2	Distinguish the different test types	K2		X												
2.2.3	Distinguish confirmation testing from regression testing	K2		X												
2.3	**Maintenance Testing**															
2.3.1	Summarize maintenance testing and its triggers	K2		X					X							
Chapter 3	**Static Testing**															
3.1	**Static Testing Basics**															

3.1.1	Recognize types of products that can be examined by the different static test techniques	K1				X	X									
3.1.2	Explain the value of static testing	K2	X			X	X									
3.1.3	Compare and contrast static and dynamic testing	K2				X	X									
3.2	**Feedback and Review Process**															
3.2.1	Identify the benefits of early and frequent stakeholder feedback	K1	X			X						X				
3.2.2	Summarize the activities of the review process	K2			X	X										
3.2.3	Recall which responsibilities are assigned to the principal roles when performing reviews	K1				X								X		
3.2.4	Compare and contrast the different review types	K2		X												
3.2.5	Recall the factors that contribute to a successful review	K1					X							X		
Chapter 4	**Test Analysis and Design**															
4.1	**Test Techniques Overview**															
4.1.1	Distinguish black-box, white-box and experience-based test techniques	K2		X												
4.2	**Black-box Test Techniques**															
4.2.1	Use equivalence partitioning to derive test cases	K3					X									
4.2.2	Use boundary value analysis to derive test cases	K3					X									
4.2.3	Use decision table testing to derive test cases	K3					X									
4.2.4	Use state transition testing to derive test cases	K3					X									
4.3	**White-box Test Techniques**															
4.3.1	Explain statement testing	K2		X												

4.3.2	Explain branch testing	K2		X												
4.3.3	Explain the value of white box testing	K2	X	X												
4.4	**Experience-based Test Techniques**															
4.4.1	Explain error guessing	K2		X												
4.4.2	Explain exploratory testing	K2		X												
4.4.3	Explain checklist-based testing	K2		X												
4.5	**Collaboration-based Test Approaches**															
4.5.1	Explain how to write user stories in collaboration with developers and business representatives	K2				X						X				
4.5.2	Classify the different options for writing acceptance criteria	K2										X				
4.5.3	Use acceptance test-driven development (ATDD) to derive test cases	K3					X									
Chapter 5	**Managing the Test Activities**															
5.1	**Test Planning**															
5.1.1	Exemplify the purpose and content of a test plan	K2		X					X							
5.1.2	Recognize how a tester adds value to iteration and release planning	K1	X									X		X		
5.1.3	Compare and contrast entry criteria and exit criteria	K2				X		X								X
5.1.4	Use estimation techniques to calculate the required test effort	K3							X		X					
5.1.5	Apply test case prioritization	K3							X		X					
5.1.6	Recall the concepts of the test pyramid	K1		X												
5.1.7	Summarize the testing quadrants and their relationships with test levels and test types	K2		X							X					
5.2	**Risk Management**															

5.2.1	Identify risk level by using risk likelihood and risk impact	K1							X					X	
5.2.2	Distinguish between project risks and product risks	K2		X										X	
5.2.3	Explain how product risk analysis may influence thoroughness and scope of testing	K2					X				X			X	
5.2.4	Explain what measures can be taken in response to analyzed product risks	K2		X			X							X	
5.3	**Test Monitoring, Test Control and Test Completion**														
5.3.1	Recall metrics used for testing	K1									X				X
5.3.2	Summarize the purposes, content, and audiences for test reports	K2					X				X				X
5.3.3	Exemplify how to communicate the status of testing	K2											X		X
5.4	**Configuration Management**														
5.4.1	Summarize how configuration management supports testing	K2					X		X						
5.5	**Defect Management**														
5.5.1	Prepare a defect report	K3		X						X					
Chapter 6	**Test Tools**														
6.1	**Tool Support for Testing**														
6.1.1	Explain how different types of test tools support testing	K2					X								
6.2	**Benefits and Risks of Test Automation**														
6.2.1	Recall the benefits and risks of test automation	K1					X					X			

10. Appendix C – Release Notes

ISTQB® Foundation Syllabus v4.0 is a major update based on the Foundation Level syllabus (v3.1.1) and the Agile Tester 2014 syllabus. For this reason, there are no detailed release notes per chapter and section. However, a summary of principal changes is provided below. Additionally, in a separate Release Notes document, ISTQB® provides traceability between the learning objectives (LO) in the version 3.1.1 of the Foundation Level Syllabus, 2014 version of the Agile Tester Syllabus, and the learning objectives in the new Foundation Level v4.0 Syllabus, showing which LOs have been added, updated, or removed.

At the time the syllabus was written (2022-2023) more than one million people in more than 100 countries have taken the Foundation Level exam, and more than 800,000 are certified testers worldwide. With the expectation that all of them have read the Foundation Syllabus to be able to pass the exam, this makes the Foundation Syllabus likely to be the most read software testing document ever! This major update is made in respect of this heritage and to improve the views of hundreds of thousands more people on the level of quality that ISTQB® delivers to the global testing community.

In this version all LOs have been edited to make them atomic, and to create one-to-one traceability between LOs and syllabus sections, thus not having content without also having a LO. The goal is to make this version easier to read, understand, learn, and translate, focusing on increasing practical usefulness and the balance between knowledge and skills.

This major release has made the following changes:

- Size reduction of the overall syllabus. Syllabus is not a textbook, but a document that serves to outline the basic elements of an introductory course on software testing, including what topics should be covered and on what level. Therefore, in particular:
    - In most cases examples are excluded from the text. It is a task of a training provider to provide the examples, as well as the exercises, during the training
    - The “Syllabus writing checklist” was followed, which suggests the maximum text size for LOs at each K-level (K1 = max. 10 lines, K2 = max. 15 lines, K3 = max. 25 lines)
- Reduction of the number of LOs compared to the Foundation v3.1.1 and Agile v2014 syllabi
    - 14 K1 LOs compared with 21 LOs in FL v3.1.1 (15) and AT 2014 (6)
    - 42 K2 LOs compared with 53 LOs in FL v3.1.1 (40) and AT 2014 (13)
    - 8 K3 LOs compared with 15 LOs in FL v3.1.1 (7) and AT 2014 (8)
- More extensive references to classic and/or respected books and articles on software testing and related topics are provided
- Major changes in chapter 1 (Fundamentals of Testing)
    - Section on testing skills expanded and improved
    - Section on the whole team approach (K1) added
    - Section on the independence of testing moved to Chapter 1 from Chapter 5
- Major changes in chapter 2 (Testing Throughout the SDLCs)
    - Sections 2.1.1 and 2.1.2 rewritten and improved, the corresponding LOs are modified

- More focus on practices like: test-first approach (K1), shift-left (K2), retrospectives (K2)
- New section on testing in the context of DevOps (K2)
- Integration testing level split into two separate test levels: component integration testing and system integration testing

- Major changes in chapter 3 (Static Testing)
  - Section on review techniques, together with the K3 LO (apply a review technique) removed
- Major changes in chapter 4 (Test Analysis and Design)
  - Use case testing removed (but still present in the Advanced Test Analyst syllabus)
  - More focus on collaboration-based approach to testing: new K3 LO about using ATDD to derive test cases and two new K2 LOs about user stories and acceptance criteria
  - Decision testing and coverage replaced with branch testing and coverage (first, branch coverage is more commonly used in practice; second, different standards define the decision differently, as opposed to “branch”; third, this solves a subtle, but serious flaw from the old FL2018 which claims that „100% decision coverage implies 100% statement coverage” – this sentence is not true in case of programs with no decisions)
  - Section on the value of white-box testing improved
- Major changes in chapter 5 (Managing the Test Activities)
  - Section on test strategies/approaches removed
  - New K3 LO on estimation techniques for estimating the test effort
  - More focus on the well-known Agile-related concepts and tools in test management: iteration and release planning (K1), test pyramid (K1), and testing quadrants (K2)
  - Section on risk management better structured by describing four main activities: risk identification, risk assessment, risk mitigation and risk monitoring
- Major changes in chapter 6 (Test Tools)
  - Content on some test automation issues reduced as being too advanced for the foundation level – section on tools selection, performing pilot projects and introducing tools into organization removed

11. Index