Certified Tester

Foundation Level Syllabus

v4.0

International Software Testing Qualifications Board

1. Fundamentals of Testing – 180 minutes

Keywords

coverage, debugging, defect, error, failure, quality, quality assurance, root cause, test analysis, test basis, test case, test completion, test condition, test control, test data, test design, test execution, test implementation, test monitoring, test object, test objective, test planning, test procedure, test result, testing, testware, validation, verification

Learning Objectives for Chapter 1:

1.1 What is Testing?

FL-1.1.1 (K1) Identify typical test objectives

FL-1.1.2 (K2) Differentiate testing from debugging

1.2 Why is Testing Necessary?

FL-1.2.1 (K2) Exemplify why testing is necessary

FL-1.2.2 (K1) Recall the relation between testing and quality assurance

FL-1.2.3 (K2) Distinguish between root cause, error, defect, and failure

1.3 Testing Principles

FL-1.3.1 (K2) Explain the seven testing principles

1.4 Test Activities, Testware and Test Roles

FL-1.4.1 (K2) Summarize the different test activities and tasks

FL-1.4.2 (K2) Explain the impact of context on the test process

FL-1.4.3 (K2) Differentiate the testware that supports the test activities

FL-1.4.4 (K2) Explain the value of maintaining traceability

FL-1.4.5 (K2) Compare the different roles in testing

1.5 Essential Skills and Good Practices in Testing

FL-1.5.1 (K2) Give examples of the generic skills required for testing

FL-1.5.2 (K1) Recall the advantages of the whole team approach

FL-1.5.3 (K2) Distinguish the benefits and drawbacks of independence of testing

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