

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 adaptations / 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.