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)