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.