4.1 The Test Development Process (K3) — *15 minutes*

Terms
Test case specification, test design, test execution schedule, test procedure specification, test script, traceability

Background
The test development process described in this section can be done in different ways, from very informal with little or no documentation, to very formal (as it is described below). The level of formality depends on the context of the testing, including the maturity of testing and development processes, time constraints, safety or regulatory requirements, and the people involved.

During test analysis, the test basis documentation is analyzed in order to determine what to test, i.e., to identify the test conditions. A test condition is defined as an item or event that could be verified by one or more test cases (e.g., a function, transaction, quality characteristic or structural element).

Establishing traceability from test conditions back to the specifications and requirements enables both effective impact analysis when requirements change, and determining requirements coverage for a set of tests. During test analysis the detailed test approach is implemented to select the test design techniques to use based on, among other considerations, the identified risks (see Chapter 5 for more on risk analysis).

During test design the test cases and test data are created and specified. A test case consists of a set of input values, execution preconditions, expected results and execution postconditions, defined to cover a certain test objective(s) or test condition(s). The ‘Standard for Software Test Documentation’ (IEEE STD 829-1998) describes the content of test design specifications (containing test conditions) and test case specifications.

Expected results should be produced as part of the specification of a test case and include outputs, changes to data and states, and any other consequences of the test. If expected results have not been defined, then a plausible, but erroneous, result may be interpreted as the correct one. Expected results should ideally be defined prior to test execution.

During test implementation the test cases are developed, implemented, prioritized and organized in the test procedure specification (IEEE STD 829-1998). The test procedure specifies the sequence of actions for the execution of a test. If tests are run using a test execution tool, the sequence of actions is specified in a test script (which is an automated test procedure).

The various test procedures and automated test scripts are subsequently formed into a test execution schedule that defines the order in which the various test procedures, and possibly automated test scripts, are executed. The test execution schedule will take into account such factors as regression tests, prioritization, and technical and logical dependencies.

4.2 Categories of Test Design Techniques (K2) — *15 minutes*

Terms
Black-box test design technique, experience-based test design technique, test design technique, white-box test design technique

Background
The purpose of a test design technique is to identify test conditions, test cases, and test data.

It is a classic distinction to denote test techniques as black-box or white-box. Black-box test design techniques (also called specification-based techniques) are a way to derive and select test conditions, test cases, or test data based on an analysis of the test basis documentation. This includes both functional and non-functional testing. Black-box testing, by definition, does not use any information regarding the internal structure of the component or system to be tested. White-box test design techniques (also called structural or structure-based techniques) are based on an analysis of the structure of the component or system. Black-box and white-box testing may also be combined with experience-based techniques to leverage the experience of developers, testers and users to determine what should be tested.

Some techniques fall clearly into a single category; others have elements of more than one category.

This syllabus refers to specification-based test design techniques as black-box techniques and structure-based test design techniques as white-box techniques. In addition experience-based test design techniques are covered.

Common characteristics of specification-based test design techniques include:
- Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components
- Test cases can be derived systematically from these models

Common characteristics of structure-based test design techniques include:
- Information about how the software is constructed is used to derive the test cases (e.g., code and detailed design information)
- The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage

Common characteristics of experience-based test design techniques include:
- The knowledge and experience of people are used to derive the test cases
- The knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment is one source of information
- Knowledge about likely defects and their distribution is another source of information

4.3 Specification-based or Black-box Techniques (K3) — *150 minutes*

Terms
Boundary value analysis, decision table testing, equivalence partitioning, state transition testing, use case testing

4.3.1 Equivalence Partitioning (K3)

In equivalence partitioning, inputs to the software or system are divided into groups that are expected to exhibit similar behavior, so they are likely to be processed in the same way. Equivalence partitions (or classes) can be found for both valid data, i.e., values that should be accepted and invalid data, i.e., values that should be rejected. Partitions can also be identified for outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing). Tests can be designed to cover all valid and invalid partitions. Equivalence partitioning is applicable at all levels of testing.

Equivalence partitioning can be used to achieve input and output coverage goals. It can be applied to human input, input via interfaces to a system, or interface parameters in integration testing.

4.3.2 Boundary Value Analysis (K3)

Behavior at the edge of each equivalence partition is more likely to be incorrect than behavior within the partition, so boundaries are an area where testing is likely to yield defects. The maximum and minimum values of a partition are its boundary values. A boundary value for a valid partition is a valid boundary value; the boundary of an invalid partition is an invalid boundary value. Tests can be designed to cover both valid and invalid boundary values. When designing test cases, a test for each boundary value is chosen.

Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect-finding capability is high. Detailed specifications are helpful in determining the interesting boundaries.

This technique is often considered as an extension of equivalence partitioning or other black-box test design techniques. It can be used on equivalence classes for user input on screen as well as, for example, on time ranges (e.g., time out, transactional speed requirements) or table ranges (e.g., table size is 256*256).

4.3.3 Decision Table Testing (K3)

Decision tables are a good way to capture system requirements that contain logical conditions, and to document internal system design. They may be used to record complex business rules that a system is to implement. When creating decision tables, the specification is analyzed, and conditions and actions of the system are identified. The input conditions and actions are most often stated in such a way that they must be true or false (Boolean). The decision table contains the triggering conditions, often combinations of true and false for all input conditions, and the resulting actions for each combination of conditions. Each column of the table corresponds to a business rule that defines a unique combination of conditions and which result in the execution of the actions associated with that rule. The coverage standard commonly used with decision table testing is to have at least one test per column in the table, which typically involves covering all combinations of triggering conditions.

The strength of decision table testing is that it creates combinations of conditions that otherwise might not have been exercised during testing. It may be applied to all situations when the action of the software depends on several logical decisions.

4.3.4 State Transition Testing (K3)

A system may exhibit a different response depending on current conditions or previous history (its state). In this case, that aspect of the system can be shown with a state transition diagram. It allows the tester to view the software in terms of its states, transitions between states, the inputs or events that trigger state changes (transitions) and the actions which may result from those transitions. The states of the system or object under test are separate, identifiable and finite in number.

A state table shows the relationship between the states and inputs, and can highlight possible transitions that are invalid.

Tests can be designed to cover a typical sequence of states, to cover every state, to exercise every transition, to exercise specific sequences of transitions or to test invalid transitions.

State transition testing is much used within the embedded software industry and technical automation in general. However, the technique is also suitable for modeling a business object having specific states or testing screen-dialogue flows (e.g., for Internet applications or business scenarios).

4.3.5 Use Case Testing (K2)

Tests can be derived from use cases. A use case describes interactions between actors (users or systems), which produce a result of value to a system user or the customer. Use cases may be described at the abstract level (business use case, technology-free, business process level) or at the system level (system use case on the system functionality level). Each use case has preconditions which need to be met for the use case to work successfully. Each use case terminates with postconditions which are the observable results and final state of the system after the use case has been completed. A use case usually has a mainstream (i.e., most likely) scenario and alternative scenarios.

Use cases describe the “process flows” through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system. Use cases are very useful for designing acceptance tests with customer/user participation. They also help uncover integration defects caused by the interaction and interference of different components, which individual component testing would not see. Designing test cases from use cases may be combined with other specification-based test techniques.

4.4 Structure-based or White-box Techniques (K4)	*60 minutes*

Terms
Code coverage, decision coverage, statement coverage, structure-based testing

Background
Structure-based or white-box testing is based on an identified structure of the software or the system, as seen in the following examples:
- Component level: the structure of a software component, i.e., statements, decisions, branches or even distinct paths
- Integration level: the structure may be a call tree (a diagram in which modules call other modules)
- System level: the structure may be a menu structure, business process or web page structure

In this section, three code-related structural test design techniques for code coverage, based on statements, branches and decisions, are discussed. For decision testing, a control flow diagram may be used to visualize the alternatives for each decision.

4.4.1 Statement Testing and Coverage (K4)
In component testing, statement coverage is the assessment of the percentage of executable statements that have been exercised by a test case suite. The statement testing technique derives test cases to execute specific statements, normally to increase statement coverage.

Statement coverage is determined by the number of executable statements covered by (designed or executed) test cases divided by the number of all executable statements in the code under test.

4.4.2 Decision Testing and Coverage (K4)
Decision coverage, related to branch testing, is the assessment of the percentage of decision outcomes (e.g., the True and False options of an IF statement) that have been exercised by a test case suite. The decision testing technique derives test cases to execute specific decision outcomes. Branches originate from decision points in the code and show the transfer of control to different locations in the code.

Decision coverage is determined by the number of all decision outcomes covered by (designed or executed) test cases divided by the number of all possible decision outcomes in the code under test.

Decision testing is a form of control flow testing as it follows a specific flow of control through the decision points. Decision coverage is stronger than statement coverage; 100% decision coverage guarantees 100% statement coverage, but not vice versa.

4.4.3 Other Structure-based Techniques (K1)
There are stronger levels of structural coverage beyond decision coverage, for example, condition coverage and multiple condition coverage.

The concept of coverage can also be applied at other test levels For example, at the integration level the percentage of modules, components or classes that have been exercised by a test case suite could be expressed as module, component or class coverage.

Tool support is useful for the structural testing of code.

4.5 Experience-based Techniques (K2) — *30 minutes*

Terms
Exploratory testing, (fault) attack

Background
Experience-based testing is where tests are derived from the tester's skill and intuition and their experience with similar applications and technologies. When used to augment systematic techniques, these techniques can be useful in identifying special tests not easily captured by formal techniques, especially when applied after more formal approaches. However, this technique may yield widely varying degrees of effectiveness, depending on the testers' experience.

A commonly used experience-based technique is error guessing. Generally testers anticipate defects based on experience. A structured approach to the error guessing technique is to enumerate a list of possible defects and to design tests that attack these defects. This systematic approach is called fault attack. These defect and failure lists can be built based on experience, available defect and failure data, and from common knowledge about why software fails.

Exploratory testing is concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, and carried out within time-boxes. It is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing. It can serve as a check on the test process, to help ensure that the most serious defects are found.

4.6 Choosing Test Techniques (K2) — *15 minutes*

Terms
No specific terms.

Background
The choice of which test techniques to use depends on a number of factors, including the type of system, regulatory standards, customer or contractual requirements, level of risk, type of risk, test objective, documentation available, knowledge of the testers, time and budget, development life cycle, use case models and previous experience with types of defects found.

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels.

When creating test cases, testers generally use a combination of test techniques including process, rule and data-driven techniques to ensure adequate coverage of the object under test.

References
4.1 Craig, 2002, Hetzel, 1988, IEEE STD 829-1998
4.2 Beizer, 1990, Copeland, 2004
4.3.1 Copeland, 2004, Myers, 1979
4.3.2 Copeland, 2004, Myers, 1979
4.3.3 Beizer, 1990, Copeland, 2004
4.3.4 Beizer, 1990, Copeland, 2004
4.3.5 Copeland, 2004
4.4.3 Beizer, 1990, Copeland, 2004
4.5 Kaner, 2002
4.6 Beizer, 1990, Copeland, 2004