2.1 Software Development Models (K2) — *20 minutes*

Terms
Commercial Off-The-Shelf (COTS), iterative-incremental development model, validation, verification, V-model

Background
Testing does not exist in isolation; test activities are related to software development activities. Different development life cycle models need different approaches to testing.

2.1.1 V-model (Sequential Development Model) (K2)
Although variants of the V-model exist, a common type of V-model uses four test levels, corresponding to the four development levels.

The four levels used in this syllabus are:
- Component (unit) testing
- Integration testing
- System testing
- Acceptance testing

In practice, a V-model may have more, fewer or different levels of development and testing, depending on the project and the software product. For example, there may be component integration testing after component testing, and system integration testing after system testing.

Software work products (such as business scenarios or use cases, requirements specifications, design documents and code) produced during development are often the basis of testing in one or more test levels. References for generic work products include Capability Maturity Model Integration (CMMI) or 'Software life cycle processes' (IEEE/IEC 12207). Verification and validation (and early test design) can be carried out during the development of the software work products.

2.1.2 Iterative-incremental Development Models (K2)
Iterative-incremental development is the process of establishing requirements, designing, building and testing a system in a series of short development cycles. Examples are: prototyping, Rapid Application Development (RAD), Rational Unified Process (RUP) and agile development models. A system that is produced using these models may be tested at several test levels during each iteration. An increment, added to others developed previously, forms a growing partial system, which should also be tested. Regression testing is increasingly important on all iterations after the first one. Verification and validation can be carried out on each increment.

2.1.3 Testing within a Life Cycle Model (K2)
In any life cycle model, there are several characteristics of good testing:
- For every development activity there is a corresponding testing activity
- Each test level has test objectives specific to that level
- The analysis and design of tests for a given test level should begin during the corresponding development activity
- Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle

Test levels can be combined or reorganized depending on the nature of the project or the system architecture. For example, for the integration of a Commercial Off-The-Shelf (COTS) software product into a system, the purchaser may perform integration testing at the system level (e.g.,

integration to the infrastructure and other systems, or system deployment) and acceptance testing (functional and/or non-functional, and user and/or operational testing).

2.2 Test Levels (K2) — *40 minutes*

Terms
Alpha testing, beta testing, component testing, driver, field testing, functional requirement, integration, integration testing, non-functional requirement, robustness testing, stub, system testing, test environment, test level, test-driven development, user acceptance testing

Background
For each of the test levels, the following can be identified: the generic objectives, the work product(s) being referenced for deriving test cases (i.e., the test basis), the test object (i.e., what is being tested), typical defects and failures to be found, test harness requirements and tool support, and specific approaches and responsibilities.

Testing a system's configuration data shall be considered during test planning,

2.2.1 Component Testing (K2)
Test basis:
- Component requirements
- Detailed design
- Code

Typical test objects:
- Components
- Programs
- Data conversion / migration programs
- Database modules

Component testing (also known as unit, module or program testing) searches for defects in, and verifies the functioning of, software modules, programs, objects, classes, etc., that are separately testable. It may be done in isolation from the rest of the system, depending on the context of the development life cycle and the system. Stubs, drivers and simulators may be used.

Component testing may include testing of functionality and specific non-functional characteristics, such as resource-behavior (e.g., searching for memory leaks) or robustness testing, as well as structural testing (e.g., decision coverage). Test cases are derived from work products such as a specification of the component, the software design or the data model.

Typically, component testing occurs with access to the code being tested and with the support of a development environment, such as a unit test framework or debugging tool. In practice, component testing usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally managing these defects.

One approach to component testing is to prepare and automate test cases before coding. This is called a test-first approach or test-driven development. This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, and executing the component tests correcting any issues and iterating until they pass.

2.2.2 Integration Testing (K2)

Test basis:
- Software and system design
- Architecture
- Workflows
- Use cases

Typical test objects:
- Subsystems
- Database implementation
- Infrastructure
- Interfaces
- System configuration and configuration data

Integration testing tests interfaces between components, interactions with different parts of a system, such as the operating system, file system and hardware, and interfaces between systems.

There may be more than one level of integration testing and it may be carried out on test objects of varying size as follows:
1. Component integration testing tests the interactions between software components and is done after component testing
2. System integration testing tests the interactions between different systems or between hardware and software and may be done after system testing. In this case, the developing organization may control only one side of the interface. This might be considered as a risk. Business processes implemented as workflows may involve a series of systems. Cross-platform issues may be significant.

The greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting.

Systematic integration strategies may be based on the system architecture (such as top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components. In order to ease fault isolation and detect defects early, integration should normally be incremental rather than “big bang”.

Testing of specific non-functional characteristics (e.g., performance) may be included in integration testing as well as functional testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating module A with module B they are interested in testing the communication between the modules, not the functionality of the individual module as that was done during component testing. Both functional and structural approaches may be used.

Ideally, testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, those components can be built in the order required for most efficient testing.

2.2.3 System Testing (K2)

Test basis:
- o System and software requirement specification
- o Use cases
- o Functional specification
- o Risk analysis reports

Typical test objects:
- o System, user and operation manuals
- o System configuration and configuration data

System testing is concerned with the behavior of a whole system/product. The testing scope shall be clearly addressed in the Master and/or Level Test Plan for that test level.

In system testing, the test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment-specific failures not being found in testing.

System testing may include tests based on risks and/or on requirements specifications, business processes, use cases, or other high level text descriptions or models of system behavior, interactions with the operating system, and system resources.

System testing should investigate functional and non-functional requirements of the system, and data quality characteristics. Testers also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. Structure-based techniques (white-box) may then be used to assess the thoroughness of the testing with respect to a structural element, such as menu structure or web page navigation (see Chapter 4).

An independent test team often carries out system testing.

2.2.4 Acceptance Testing (K2)

Test basis:
- o User requirements
- o System requirements
- o Use cases
- o Business processes
- o Risk analysis reports

Typical test objects:
- o Business processes on fully integrated system
- o Operational and maintenance processes
- o User procedures
- o Forms
- o Reports
- o Configuration data

Acceptance testing is often the responsibility of the customers or users of a system; other stakeholders may be involved as well.

The goal in acceptance testing is to establish confidence in the system, parts of the system or specific non-functional characteristics of the system. Finding defects is not the main focus in acceptance testing. Acceptance testing may assess the system's readiness for deployment and

use, although it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance test for a system.

Acceptance testing may occur at various times in the life cycle, for example:
- A COTS software product may be acceptance tested when it is installed or integrated
- Acceptance testing of the usability of a component may be done during component testing
- Acceptance testing of a new functional enhancement may come before system testing

Typical forms of acceptance testing include the following:

User acceptance testing
Typically verifies the fitness for use of the system by business users.

Operational (acceptance) testing
The acceptance of the system by the system administrators, including:
- Testing of backup/restore
- Disaster recovery
- User management
- Maintenance tasks
- Data load and migration tasks
- Periodic checks of security vulnerabilities

Contract and regulation acceptance testing
Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance criteria should be defined when the parties agree to the contract. Regulation acceptance testing is performed against any regulations that must be adhered to, such as government, legal or safety regulations.

Alpha and beta (or field) testing
Developers of market, or COTS, software often want to get feedback from potential or existing customers in their market before the software product is put up for sale commercially. Alpha testing is performed at the developing organization's site but not by the developing team. Beta testing, or field-testing, is performed by customers or potential customers at their own locations.

Organizations may use other terms as well, such as factory acceptance testing and site acceptance testing for systems that are tested before and after being moved to a customer's site.

2.3 Test Types (K2) — *40 minutes*

Terms
Black-box testing, code coverage, functional testing, interoperability testing, load testing, maintainability testing, performance testing, portability testing, reliability testing, security testing, stress testing, structural testing, usability testing, white-box testing

Background
A group of test activities can be aimed at verifying the software system (or a part of a system) based on a specific reason or target for testing.

A test type is focused on a particular test objective, which could be any of the following:
- A function to be performed by the software
- A non-functional quality characteristic, such as reliability or usability
- The structure or architecture of the software or system
- Change related, i.e., confirming that defects have been fixed (confirmation testing) and looking for unintended changes (regression testing)

A model of the software may be developed and/or used in structural testing (e.g., a control flow model or menu structure model), non-functional testing (e.g., performance model, usability model security threat modeling), and functional testing (e.g., a process flow model, a state transition model or a plain language specification).

2.3.1 Testing of Function (Functional Testing) (K2)
The functions that a system, subsystem or component are to perform may be described in work products such as a requirements specification, use cases, or a functional specification, or they may be undocumented. The functions are “what” the system does.

Functional tests are based on functions and features (described in documents or understood by the testers) and their interoperability with specific systems, and may be performed at all test levels (e.g., tests for components may be based on a component specification).

Specification-based techniques may be used to derive test conditions and test cases from the functionality of the software or system (see Chapter 4). Functional testing considers the external behavior of the software (black-box testing).

A type of functional testing, security testing, investigates the functions (e.g., a firewall) relating to detection of threats, such as viruses, from malicious outsiders. Another type of functional testing, interoperability testing, evaluates the capability of the software product to interact with one or more specified components or systems.

2.3.2 Testing of Non-functional Software Characteristics (Non-functional Testing) (K2)
Non-functional testing includes, but is not limited to, performance testing, load testing, stress testing, usability testing, maintainability testing, reliability testing and portability testing. It is the testing of “how” the system works.

Non-functional testing may be performed at all test levels. The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified on a varying scale, such as response times for performance testing. These tests can be referenced to a quality model such as the one defined in ‘Software Engineering – Software Product Quality’ (ISO

9126). Non-functional testing considers the external behavior of the software and in most cases uses black-box test design techniques to accomplish that.

2.3.3 Testing of Software Structure/Architecture (Structural Testing) (K2)

Structural (white-box) testing may be performed at all test levels. Structural techniques are best used after specification-based techniques, in order to help measure the thoroughness of testing through assessment of coverage of a type of structure.

Coverage is the extent that a structure has been exercised by a test suite, expressed as a percentage of the items being covered. If coverage is not 100%, then more tests may be designed to test those items that were missed to increase coverage. Coverage techniques are covered in Chapter 4.

At all test levels, but especially in component testing and component integration testing, tools can be used to measure the code coverage of elements, such as statements or decisions. Structural testing may be based on the architecture of the system, such as a calling hierarchy.

Structural testing approaches can also be applied at system, system integration or acceptance testing levels (e.g., to business models or menu structures).

2.3.4 Testing Related to Changes: Re-testing and Regression Testing (K2)

After a defect is detected and fixed, the software should be re-tested to confirm that the original defect has been successfully removed. This is called confirmation. Debugging (locating and fixing a defect) is a development activity, not a testing activity.

Regression testing is the repeated testing of an already tested program, after modification, to discover any defects introduced or uncovered as a result of the change(s). These defects may be either in the software being tested, or in another related or unrelated software component. It is performed when the software, or its environment, is changed. The extent of regression testing is based on the risk of not finding defects in software that was working previously.

Tests should be repeatable if they are to be used for confirmation testing and to assist regression testing.

Regression testing may be performed at all test levels, and includes functional, non-functional and structural testing. Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation.