ISTQB® Foundation

Course Reader

TSG Training Ltd
Tel: 0800 0199 337

www.tsg-training.co.uk
enquiries@tsg-training.co.uk

Introductions and Housekeeping

Table of Contents

1. Introduction to the Course

Introduction to the course

- Introductions
  - You are……?
  - I am…
- Administration

TOILETS

SMOKING

MESSAGES
(no mobiles
please)

SECURITY

BREAKS

FIRE

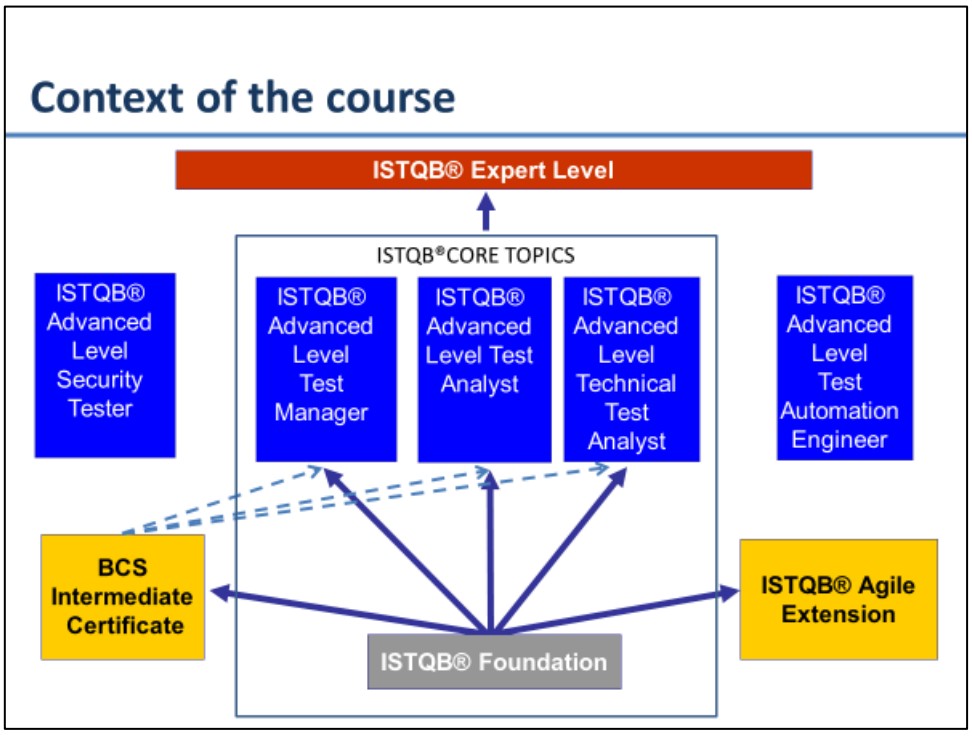
LAPTOPS
(closed please)

2. About TSG Training

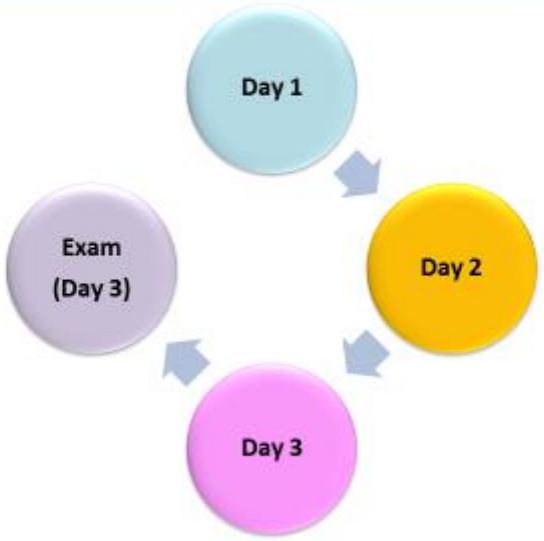
About TSG Training …

- Our certificated and non-certificated training courses provide the range of technical and managerial skills to increase organizational capability and support continual career growth and progression
- Investigate other courses at www.tsg-training.co.uk

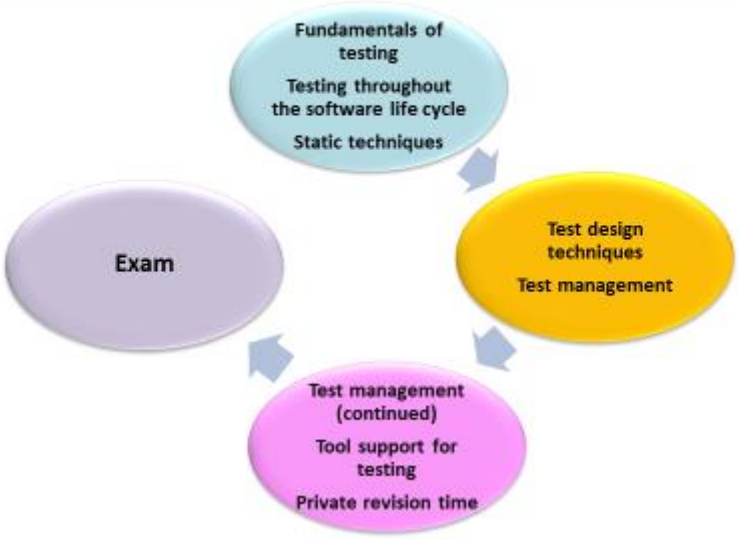
3. Context of the Course



Foundation Course Structure



Foundation Course Structure



4. Learning Objectives and Levels of Knowledge

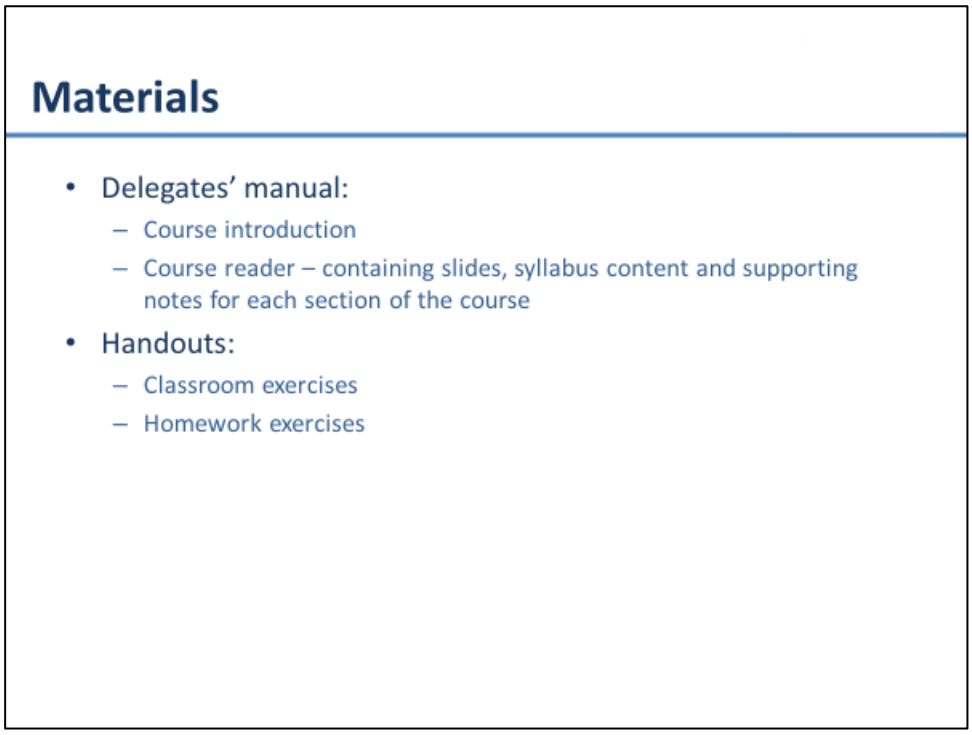
Learning objectives

- Learning objectives are defined for each topic in the syllabus
- A 'level of knowledge' is defined for each learning objective
- Each topic will be examined according to its learning objectives and the level of knowledge defined for it
- .....

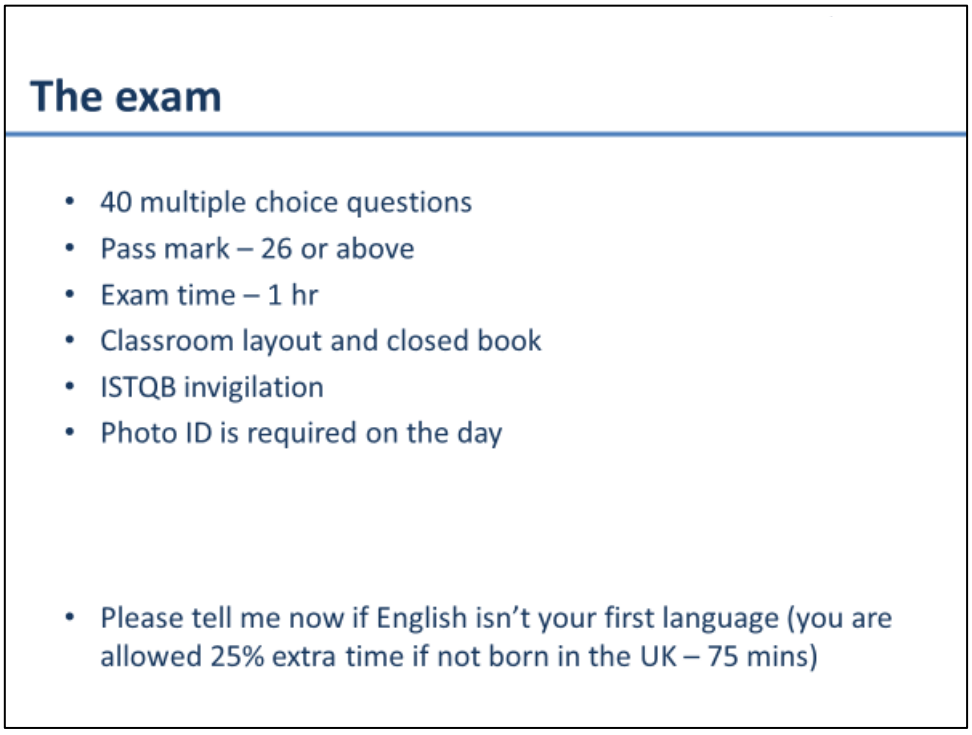
Levels of knowledge

- Level 1: Remember (K1)
  - The candidate will recognize, remember and recall a term or concept
- Level 2: Understand (K2)
  - The candidate can select the reasons or explanations for statements related to the topic and can summarize, compare, classify, categorize and give examples for the testing concept
- Level 3: Apply (K3)
  - The candidate can select the correct application of a concept or technique and apply it to a given context.

5. Course Materials



6. The Exam



The exam

- Questions weighted by topic according to time allocation on the syllabus
- Questions allocated by "K" level
  - 8 K1 (Remember) questions (20%)
  - 24 K2 (Understand) questions (60%)
  - 8 K3 (Apply) questions (20%)
- In general, all of the syllabus is examinable at a K1 level
- LO's and Keywords are on the final page of each section – there will be a keyword question (K1) for chapters 1 and 4

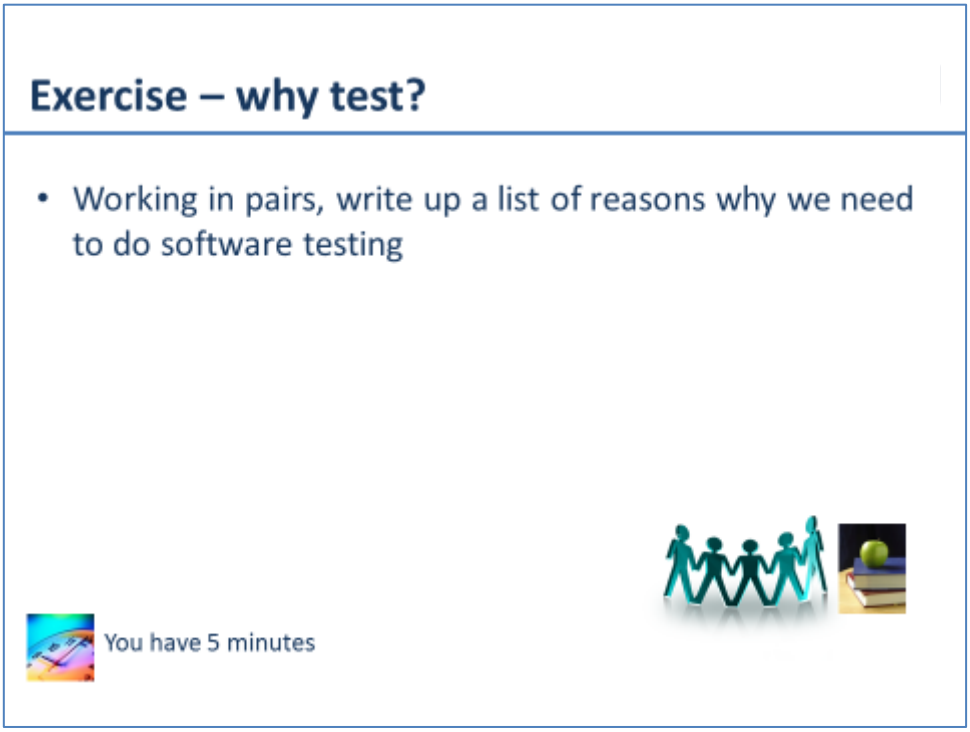
7. Revision Work

Revision work

- Exercises at the end of each section
- Mock exam at the end of the course (prior to the exam!)
- Read and review the syllabus and the course material
- Revision exercises provided at the end of day 1 and day 2
- Use the evenings to go over notes – write out key points from the day, and attempt the revision questions
- This course is HARD WORK!

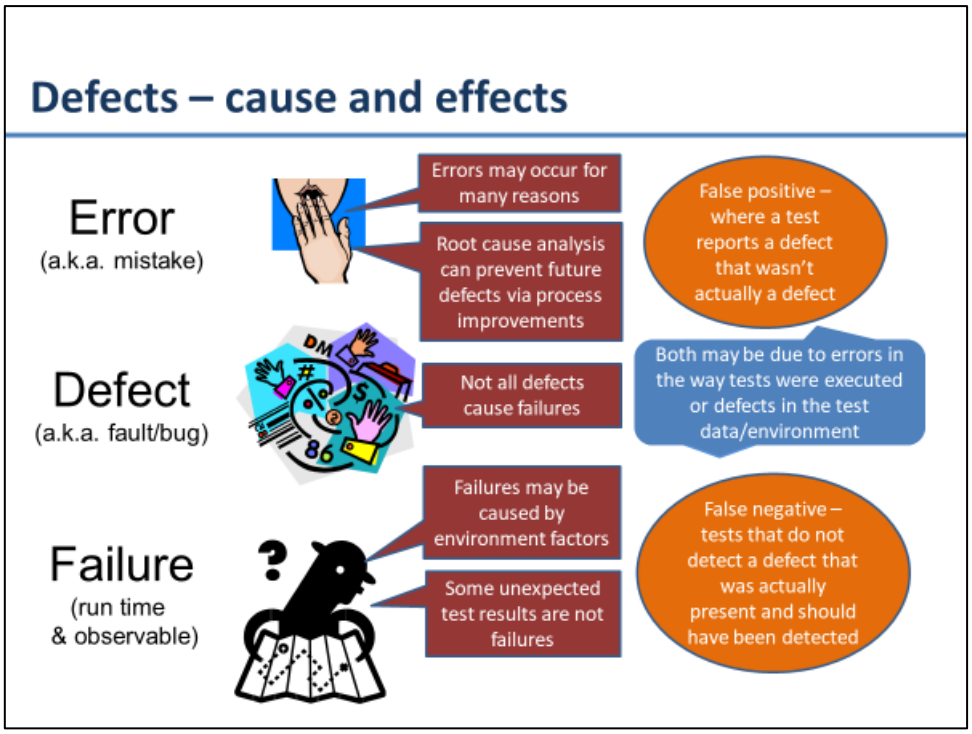
Fundamentals of Testing

Table of Contents



Space for you to note down your answers

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death. Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.



A person can make an error (mistake), which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product. An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product. For example, a requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code.

If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances.

For example, some defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.

Errors may occur for many reasons, such as:

- Time pressure
- Human fallibility
- Inexperienced or insufficiently skilled project participants
- Miscommunication between project participants, including miscommunication about requirements and design
- Complexity of the code, design, architecture, the underlying problem to be solved, and/or the technologies used
- Misunderstandings about intra-system and inter-system interfaces, especially when such intra-system and inter-system interactions are large in number
- New, unfamiliar technologies

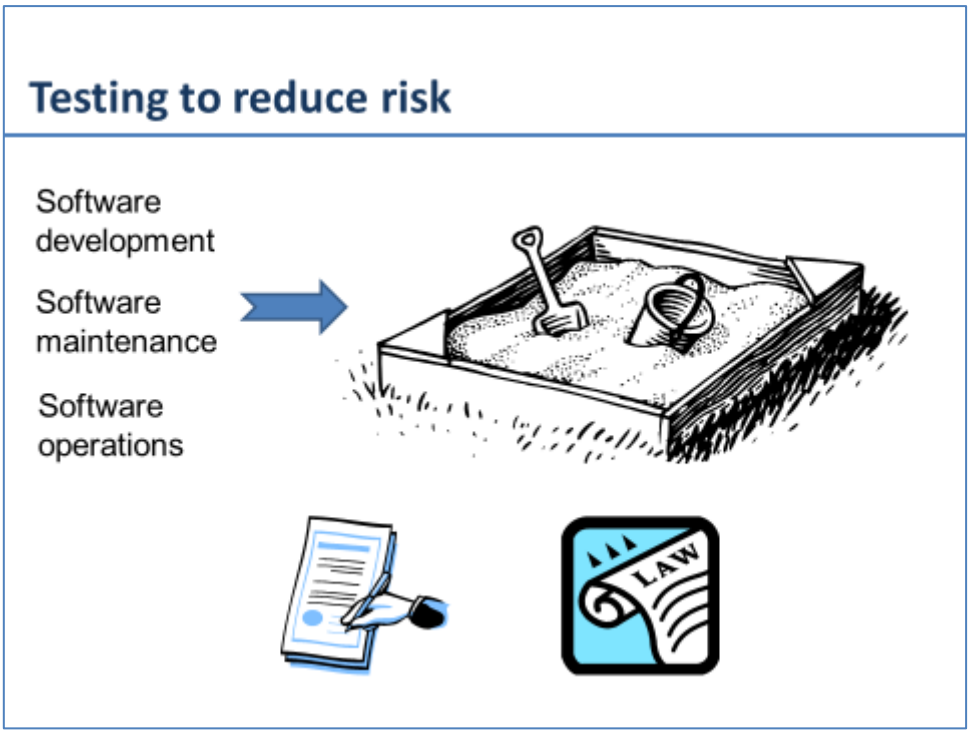
In addition to failures caused due to defects in the code, failures can also be caused by environmental conditions. For example, radiation, electromagnetic fields, and pollution can cause defects in firmware or influence the execution of software by changing hardware conditions.

Not all unexpected test results are failures. False positives may occur due to errors in the way tests were executed, or due to defects in the test data, the test environment, or other testware, or for other reasons. The inverse situation can also occur, where similar errors or defects lead to false negatives. False negatives are tests that do not detect defects that they should have detected; false positives are reported as defects, but aren't actually defects.

The root causes of defects are the earliest actions or conditions that contributed to creating the defects. Defects can be analyzed to identify their root causes, so as to reduce the occurrence of similar defects in the future. By focusing on the most significant root causes, root cause analysis can lead to process improvements that prevent a significant number of future defects from being introduced.

For example, suppose incorrect interest payments, due to a single line of incorrect code, result in customer complaints. The defective code was written for a user story which was ambiguous, due to the product owner's misunderstanding of how to calculate interest. If a large percentage of defects exist in interest calculations, and these defects have their root cause in similar misunderstandings, the product owners could be trained in the topic of interest calculations to reduce such defects in the future.

In this example, the customer complaints are effects. The incorrect interest payments are failures. The improper calculation in the code is a defect, and it resulted from the original defect, the ambiguity in the user story. The root cause of the original defect was a lack of knowledge on the part of the product owner, which resulted in the product owner making an error while writing the user story. The process of root cause analysis is discussed in ISTQB-CTEL-TM and ISTQB-CTEL-ITP.



Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation. When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems. In addition, software testing may also be required to meet contractual or legal requirements or industry-specific standards.

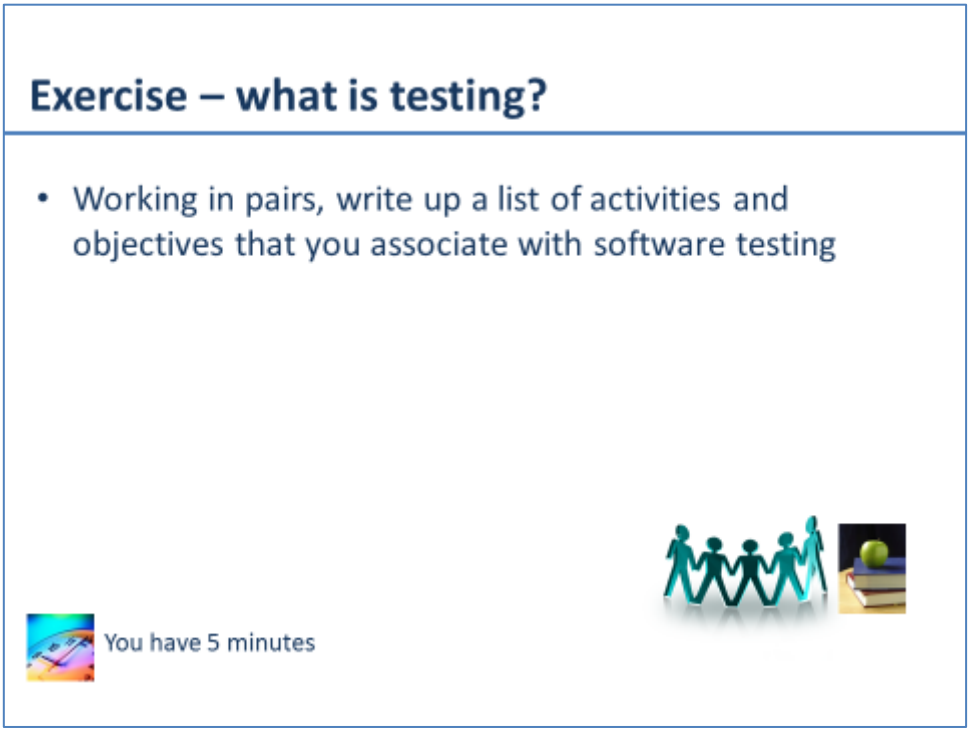
While people often use the phrase *quality assurance* (or just *QA*) to refer to testing, quality assurance and testing are not the same, but they are related. A larger concept, quality management, ties them together. Quality management includes all activities that direct and control an organization with regard to quality. Among other activities, **quality management** includes both quality assurance and quality control.

Quality assurance is typically focused on adherence to proper processes, in order to provide confidence that the appropriate levels of quality will be achieved. When processes are carried out properly, the work products created by those processes are generally of higher quality, which contributes to defect prevention. In addition, the use of root cause analysis to detect and remove the causes of defects, along with the proper application of the findings of retrospective meetings to improve processes, are important for effective quality assurance.

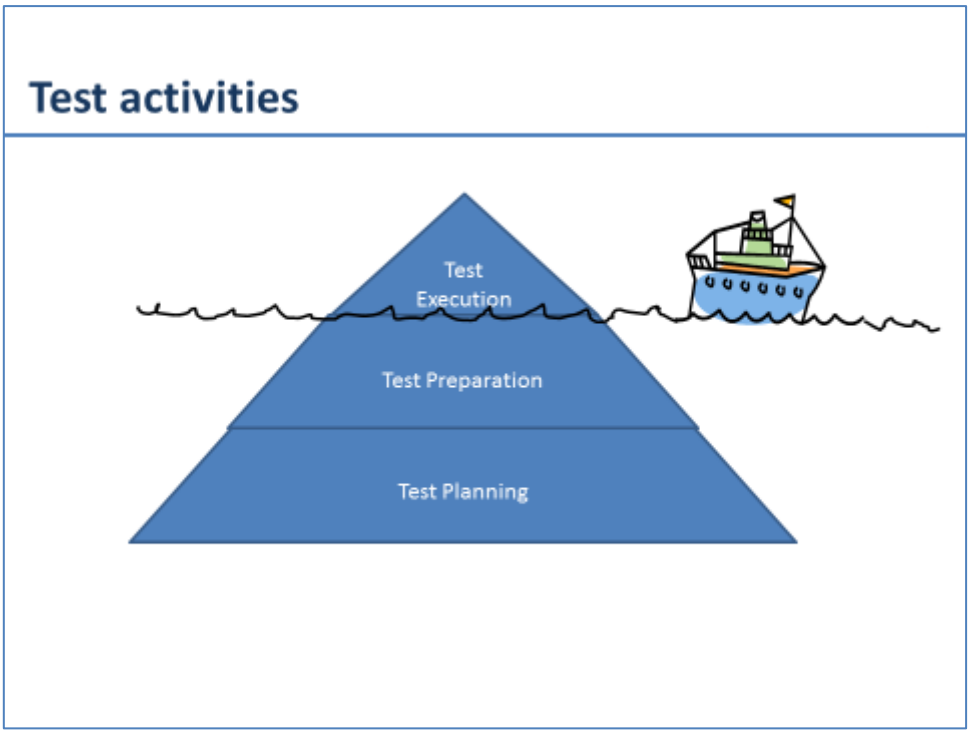
Quality control involves various activities, including test activities, that support the achievement of appropriate levels of quality. Test activities are part of the overall software development or maintenance process. Since quality assurance is concerned with the proper execution of the entire process, quality assurance supports proper testing. As described in sections 1.1.1 and 1.2.1, testing contributes to the achievement of quality in a variety of ways.

Summary

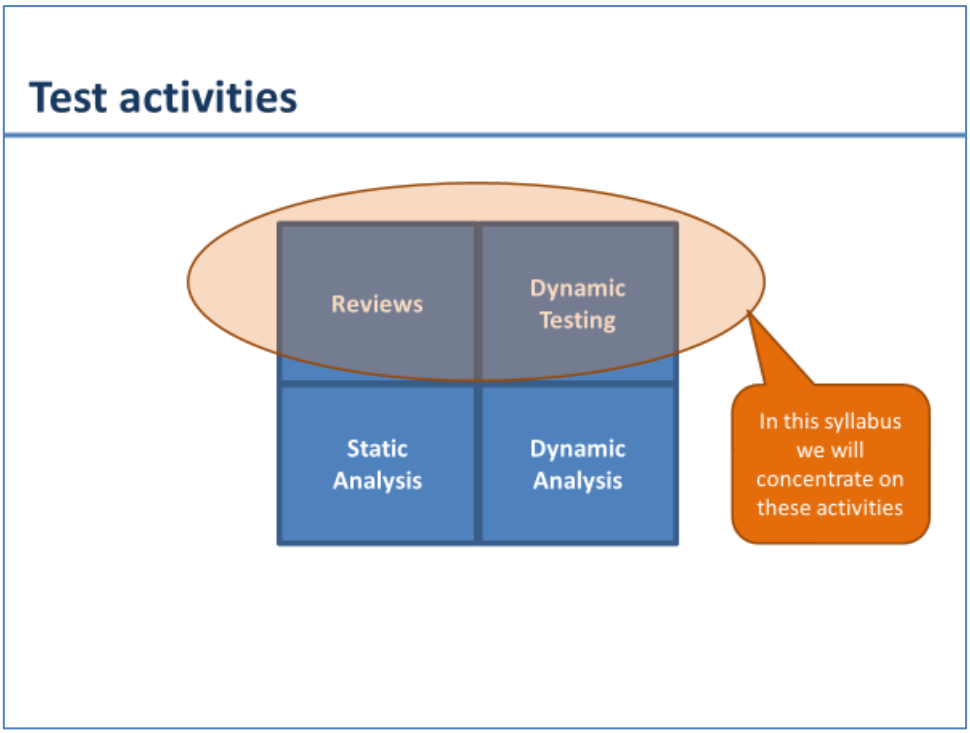
- Software defects can potentially harm individuals or companies
- There is a difference between the root cause of a defect (the error) and its consequences (the failure)
- There can be many reasons why testing is necessary
- Testing can contribute to improving quality by finding failures and removing the underlying defects
- Testing is part of quality assurance, by understanding the root causes of the defects the development and testing processes can be improved



Space for you to note down your answers



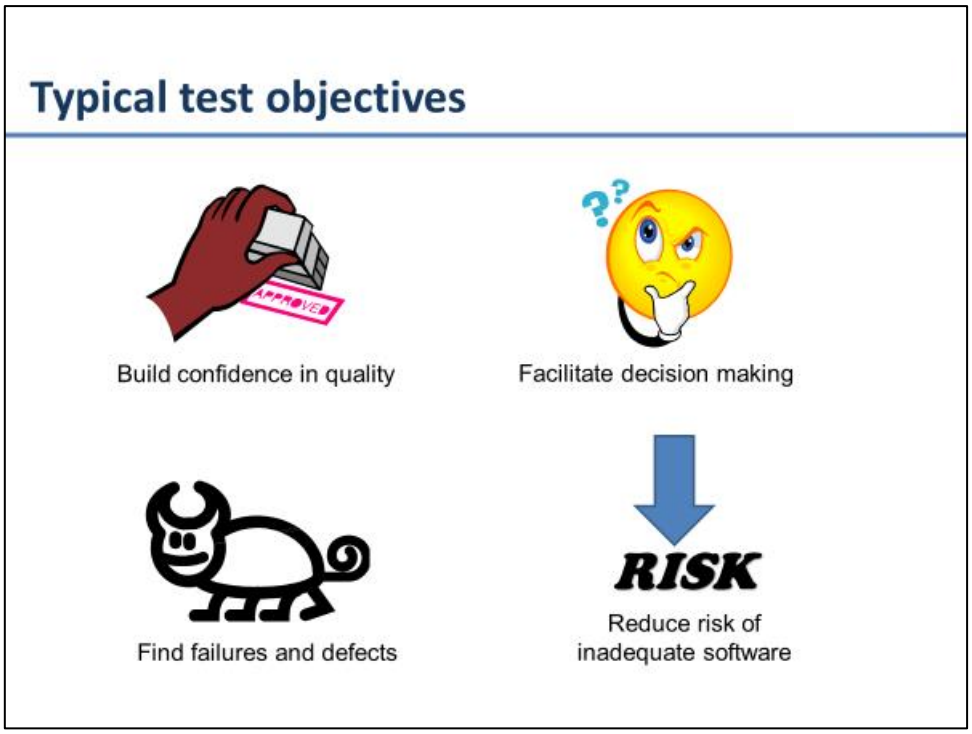
A common misperception of testing is that it only consists of running tests, i.e., executing the software and checking the results. As described in section 1.4, software testing is a process which includes many different activities; test execution (including checking of results) is only one of these activities. The test process also includes activities such as test planning, analyzing, designing, and implementing tests, reporting test progress and results, and evaluating the quality of a test object.



Some testing does involve the execution of the component or system being tested; such testing is called dynamic testing. Other testing does not involve the execution of the component or system being tested; such testing is called static testing. So, testing also includes reviewing work products such as requirements, user stories, and source code.

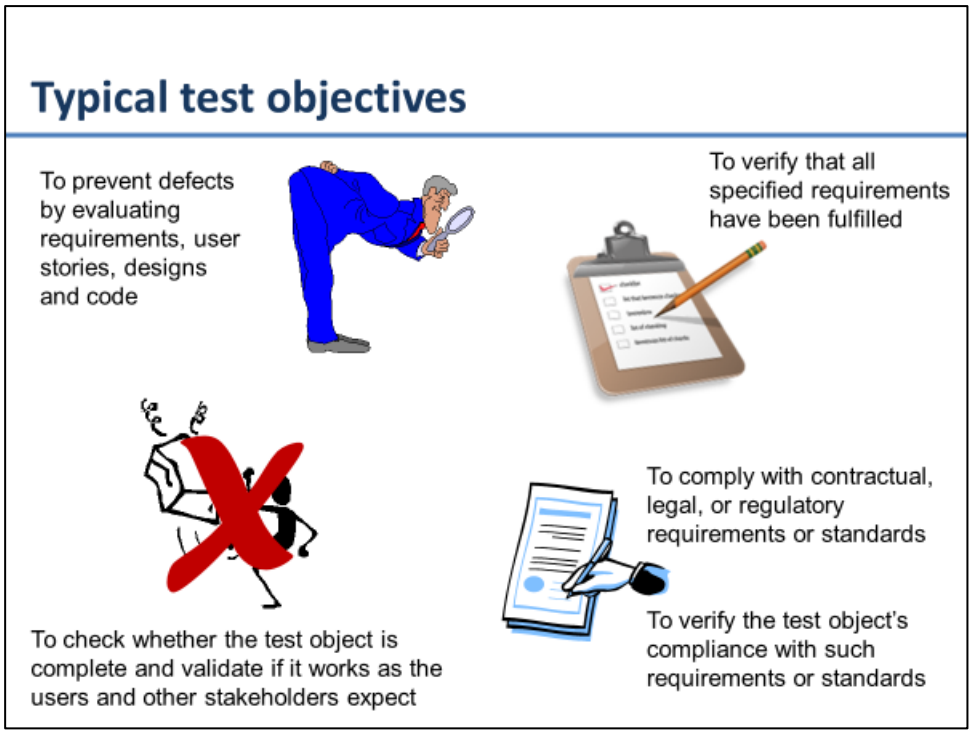
Another common misperception of testing is that it focuses entirely on verification of requirements, user stories, or other specifications. While testing does involve checking whether the system meets specified requirements, it also involves validation, which is checking whether the system will meet user and other stakeholder needs in its operational environment(s).

Test activities are organized and carried out differently in different lifecycles (see section 2.1).



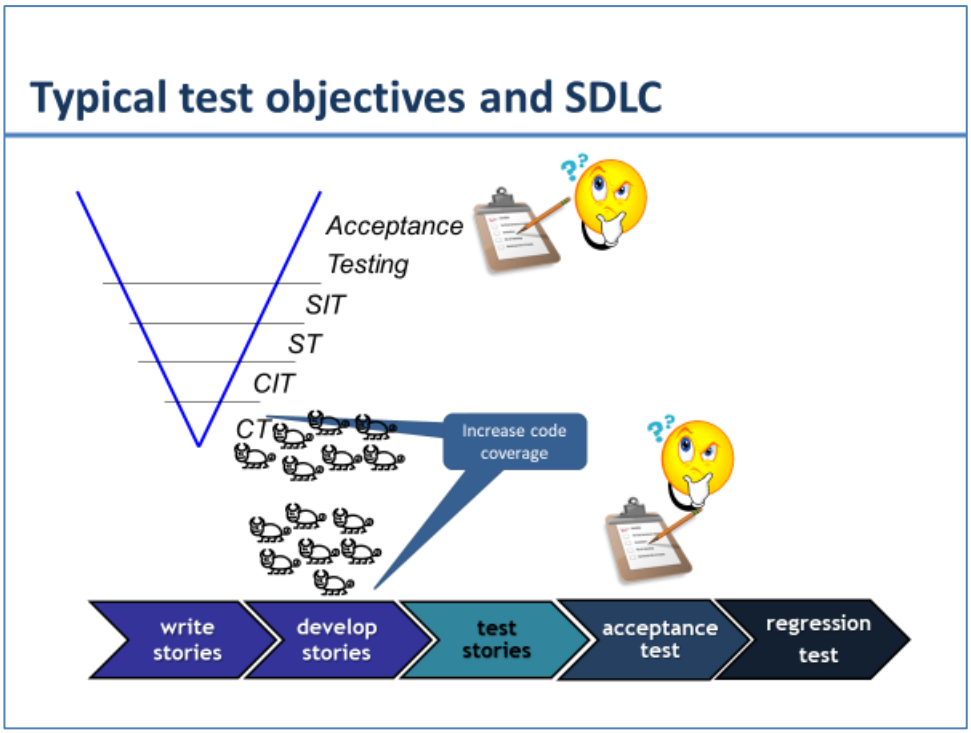
For any given project, the objectives of testing may include:

- To build confidence in the level of quality of the test object
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object
- To find defects and failures thus reducing the level of risk of inadequate software quality



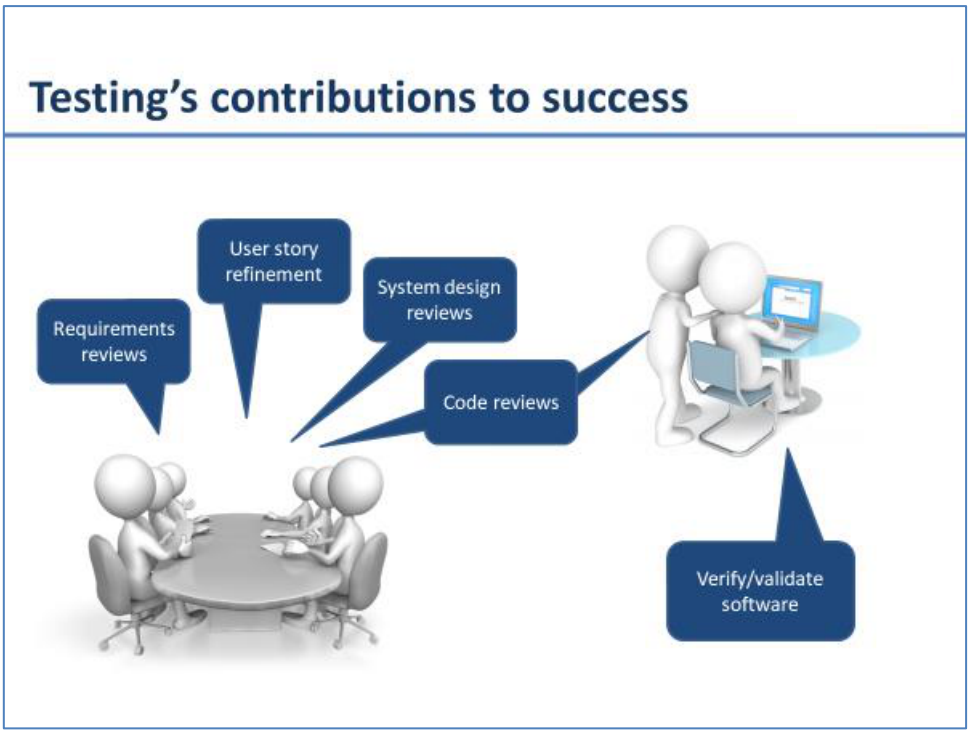
For any given project, the objectives of testing may include:

- To prevent defects by evaluating work products such as requirements, user stories, design, and code
- To verify whether all specified requirements have been fulfilled
- To check whether the test object is complete and validate if it works as the users and other stakeholders expect
- To comply with contractual, legal, or regulatory requirements or standards
- To verify the test object's compliance with such requirements or standards



The objectives of testing can vary, depending upon the context of the component or system being tested, the test level, and the software development lifecycle model. These differences may include, for example:

- During component testing, one objective may be to find as many failures as possible so that the underlying defects are identified and fixed early. Another objective may be to increase code coverage of the component tests.
- During acceptance testing, one objective may be to confirm that the system works as expected and satisfies requirements. Another objective of this testing may be to give information to stakeholders about the risk of releasing the system at a given time.



Throughout the history of computing, it is quite common for software and systems to be delivered into operation and, due to the presence of defects, to subsequently cause failures or otherwise not meet the stakeholders' needs. However, using appropriate test techniques can reduce the frequency of such problematic deliveries, when those techniques are applied with the appropriate level of test expertise, in the appropriate test levels, and at the appropriate points in the software development lifecycle. Examples include:

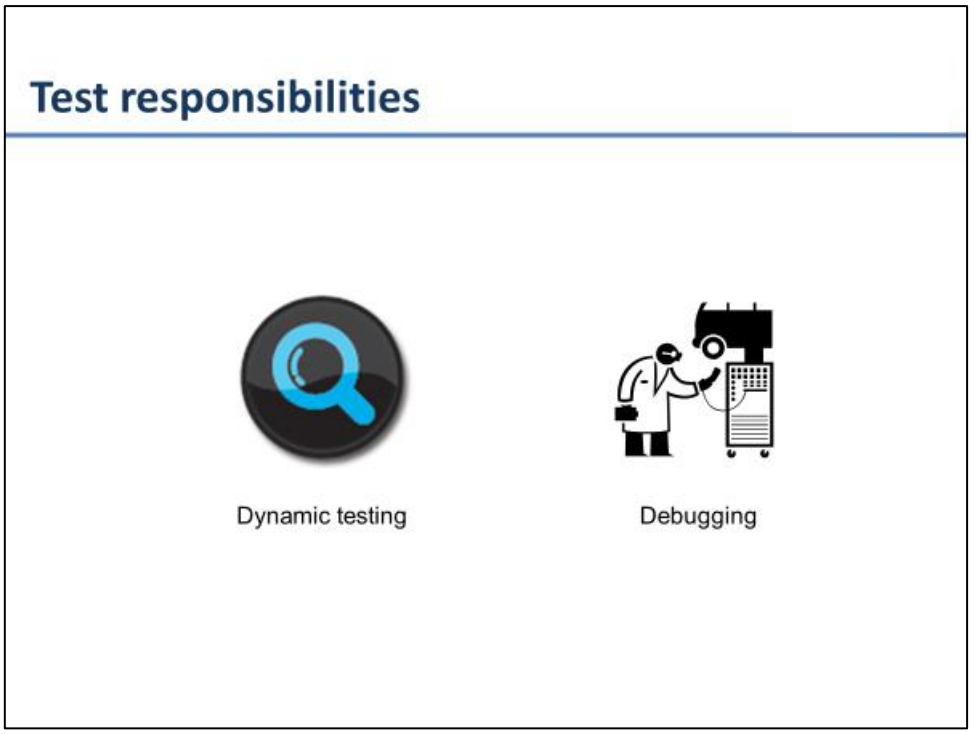
- Having testers involved in requirements reviews or user story refinement could detect defects in these work products. The identification and removal of requirements defects reduces the risk of incorrect or untestable features being developed.

- Having testers work closely with system designers while the system is being designed can increase each party's understanding of the design and how to test it. This increased understanding can reduce the risk of fundamental design defects and enable tests to be identified at an early stage.

- Having testers work closely with developers while the code is under development can increase each party's understanding of the code and how to test it. This increased understanding can reduce the risk of defects within the code and the tests.

- Having testers verify and validate the software prior to release can detect failures that might otherwise have been missed, and support the process of removing the defects that caused the failures (i.e., debugging). This increases the likelihood that the software meets stakeholder needs and satisfies requirements.

In addition to these examples, the achievement of defined test objectives (see section 1.1.1) contributes to overall software development and maintenance success.



Testing and debugging are different. Executing tests can show failures that are caused by defects in the software. Debugging is the development activity that finds, analyzes, and fixes such defects. Subsequent confirmation testing checks whether the fixes resolved the defects. In some cases, testers are responsible for the initial test and the final confirmation test, while developers do the debugging, associated component and component integration testing (continues integration). However, in Agile development and in some other software development lifecycles, testers may be involved in debugging and component testing.

ISO standard (ISO/IEC/IEEE 29119-1) has further information about software testing concepts.

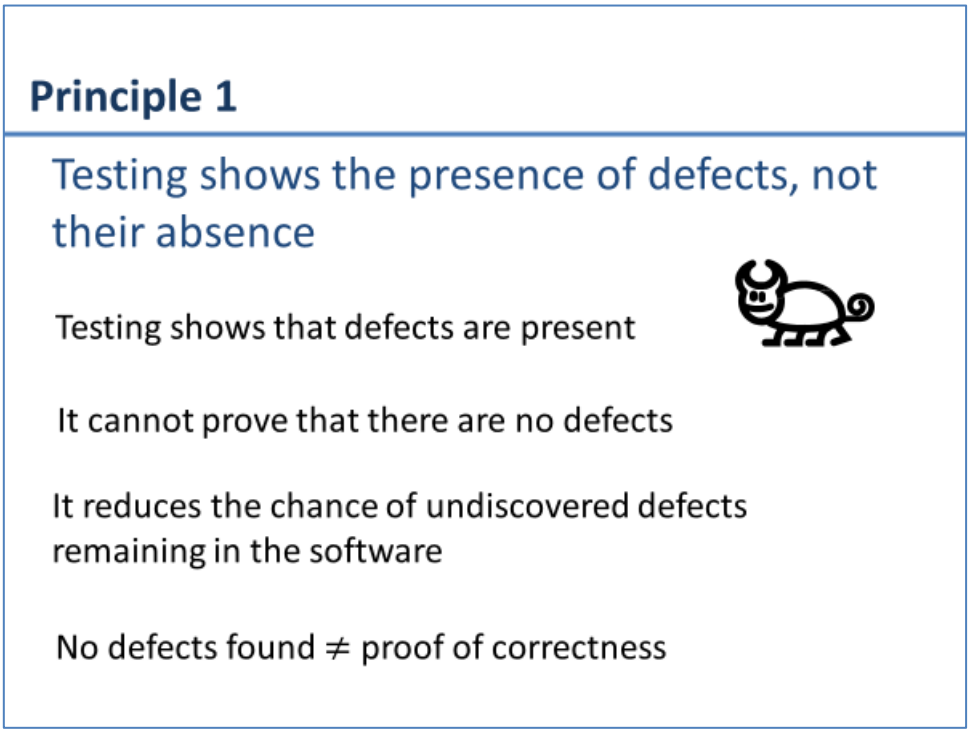
*The fundamental test process and its underlying activities are explained in detail in Section 1.4.

Summary

- Test activities take place before, during and after test execution
- Testing involves both static and dynamic activities
- Testing can have different objectives dependent on the phase of the development lifecycle
- Testing activities can both find and prevent defects
- Debugging and testing are different

Section 1.3 - Seven Testing Principles

Principles
A number of testing principles have been suggested over the past 50 years and offer general guidelines common for all testing.



Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, testing is not a proof of correctness.

Principle 2

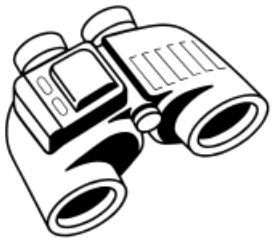
Exhaustive testing is impossible

Testing everything is infeasible

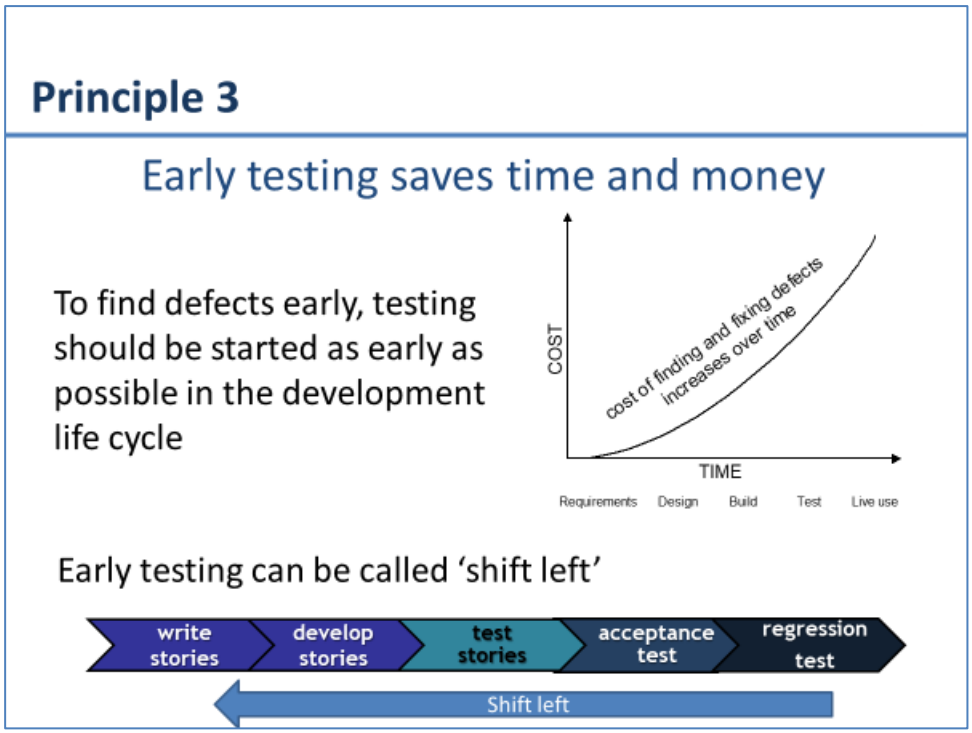
It is only possible in trivial cases

Focus test effort using:

- test techniques
- risk analysis
- test priorities



Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Rather than attempting to test exhaustively, risk analysis, test techniques, and priorities should be used to focus test efforts.



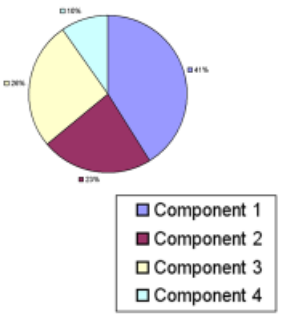
To find defects early, both static and dynamic test activities should be started as early as possible in the software development lifecycle. Early testing is sometimes referred to as *shift left*. Testing early in the software development lifecycle helps reduce or eliminate costly changes (see section 3.1).

Principle 4

Defects cluster together

A small number of modules usually contain most of the defects found during testing or operational use

Predicted or observed defect clusters support risk analysis to focus testing



A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort (as mentioned in principle 2).

Principle 5

Beware of the pesticide paradox

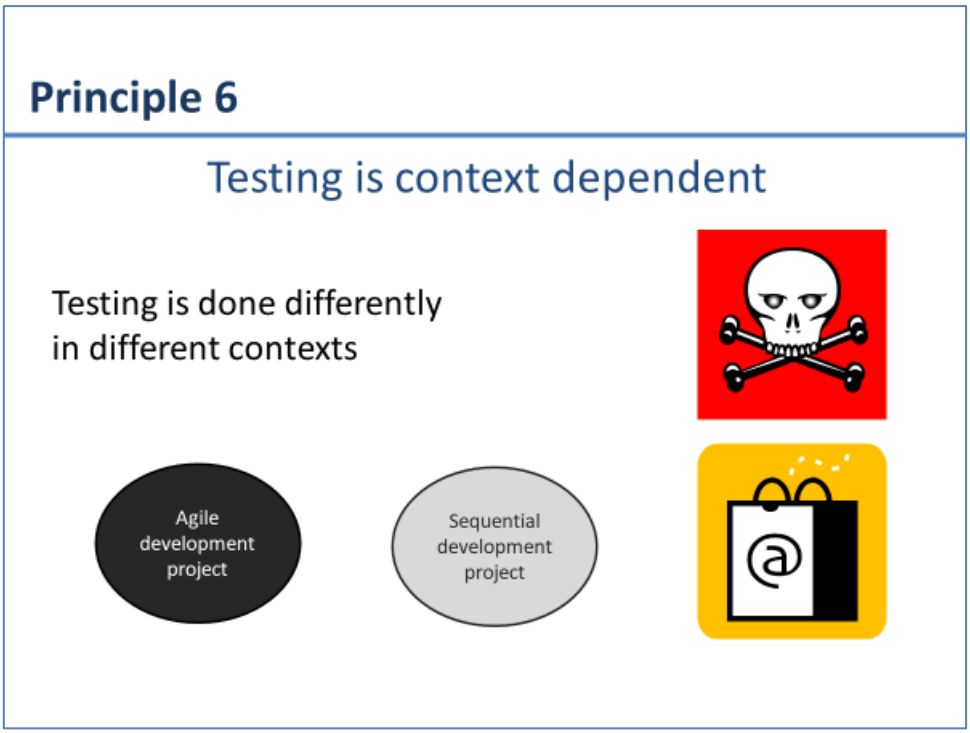
The same tests, repeated continually over time, will eventually stop finding new defects

This can be beneficial in some cases (e.g. automated regression testing when it can confirm a low number of regression defects)

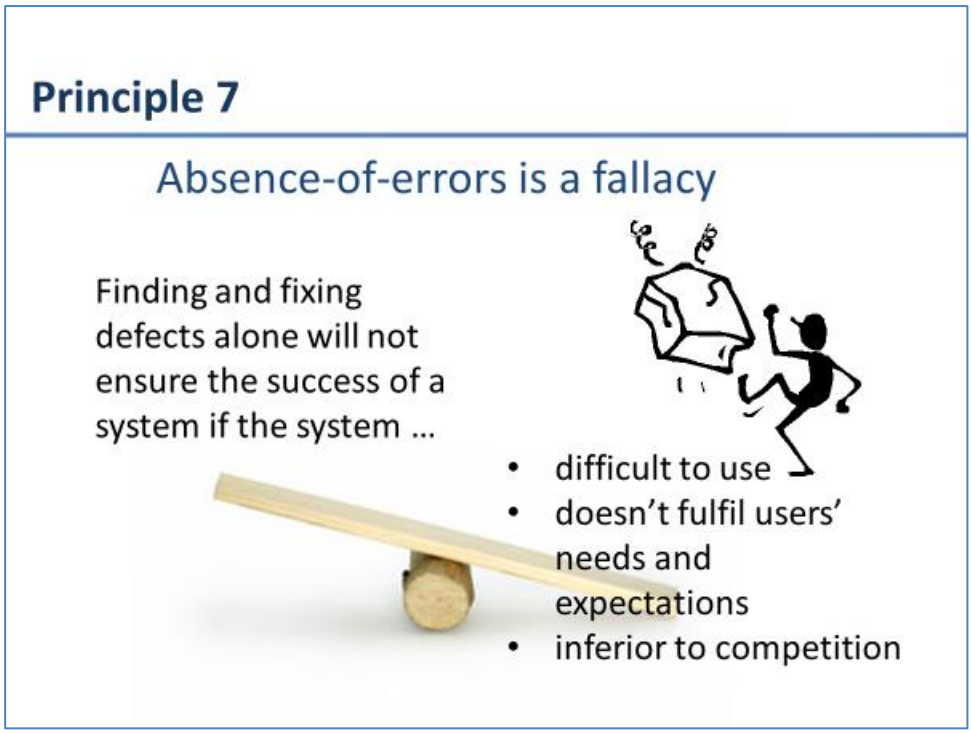
But to find new defects, tests / data must be regularly revised and new tests written

If the same tests are repeated over and over again, eventually these tests no longer find any new defects. To detect new defects, existing tests and test data may need changing, and new tests may need to be written. (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while.)

In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.



Testing is done differently in different contexts. For example, safety-critical industrial control software is tested differently from an e-commerce mobile app. As another example, testing in an Agile project is done differently than testing in a sequential software development lifecycle project (see section 2.1).



Some organizations expect that testers can run all possible tests and find all possible defects, but principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy (i.e., a mistaken belief) to expect that just finding and fixing a large number of defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users' needs and expectations, or that is inferior compared to other competing systems.

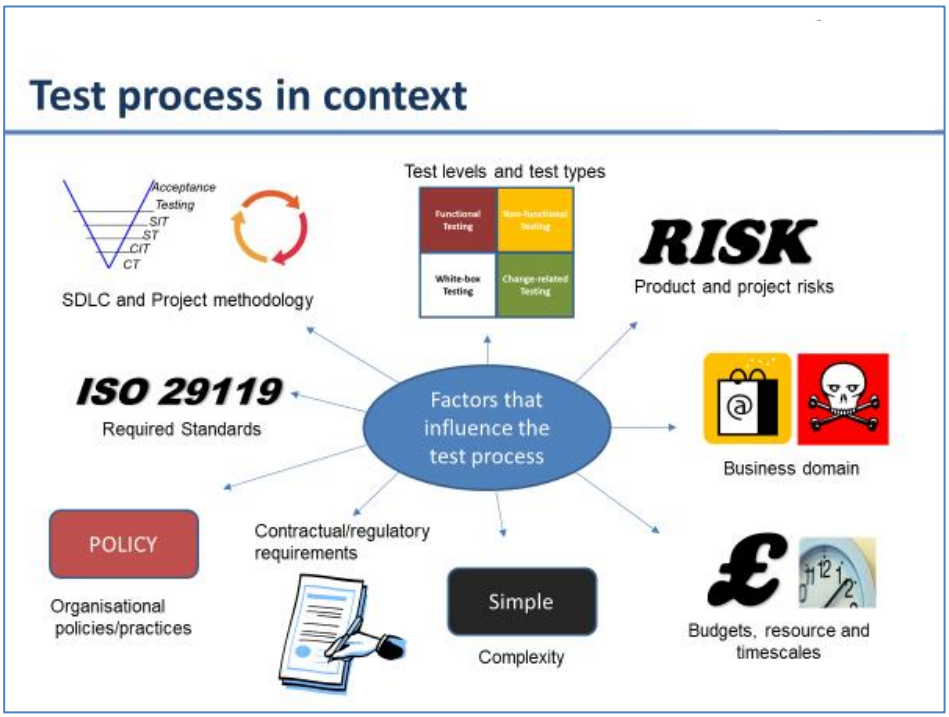
See Myers 2011, Kaner 2002, Weinberg 2008, and Beizer 1990 for examples of these and other testing principles.

Summary

There are seven testing principles:

1. Testing shows the presence of defects, not their absence
2. Exhaustive testing is impossible
3. Early testing saves time and money
4. Defects cluster together
5. Beware of the pesticide paradox
6. Testing is context dependent
7. Absence-of-errors is a fallacy

Section 1.4 - Test Process



Contextual factors that influence the test process for an organization, include, but are not limited to:

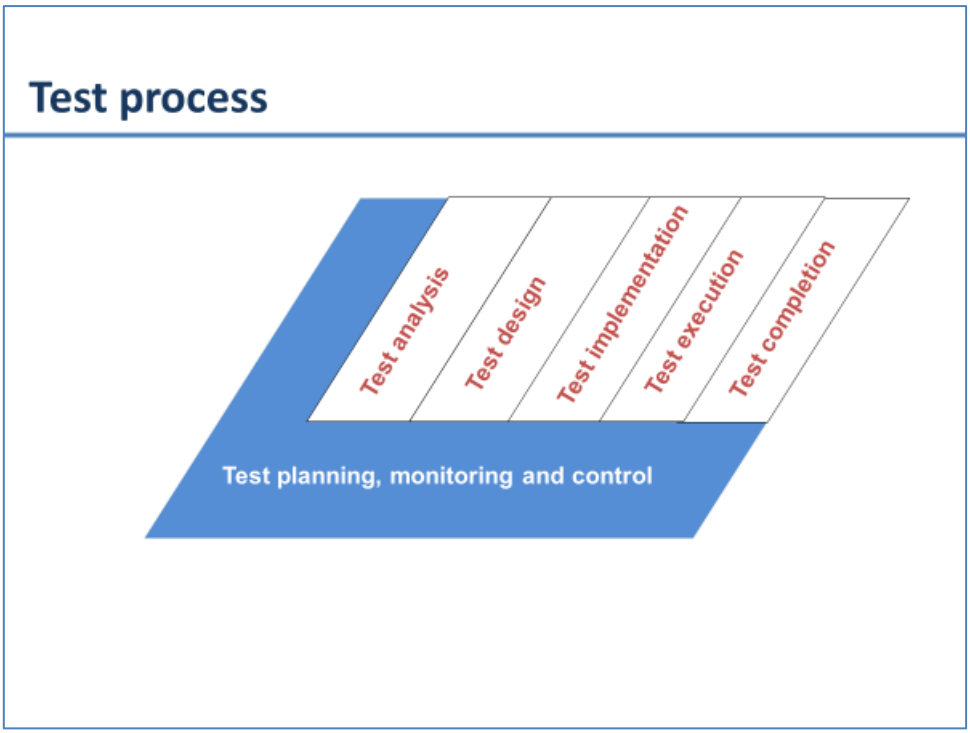
- Software development lifecycle model and project methodologies being used
- Test levels and test types being considered
- Product and project risks
- Business domain
- Operational constraints, including but not limited to:
  - Budgets and resources
  - Timescales
  - Complexity
  - Contractual and regulatory requirements
- Organizational policies and practices
- Required internal and external standards

The following sections describe general aspects of organizational test processes in terms of the following:

- Test activities and tasks
- Test work products
- Traceability between the test basis and test work products

It is very useful if the test basis (for any level or type of testing that is being considered) has measurable coverage criteria defined. The coverage criteria can act effectively as key performance indicators (KPIs) to drive the activities that demonstrate achievement of software test objectives (see section 1.1.1).

For example, for a mobile application, the test basis may include a list of requirements and a list of supported mobile devices. Each requirement is an element of the test basis. Each supported device is also an element of the test basis. The coverage criteria may require at least one test case for each element of the test basis. Once executed, the results of these tests tell stakeholders whether specified requirements are fulfilled and whether failures were observed on supported devices.
ISO standard (ISO/IEC/IEEE 29119-2) has further information about test processes.



A test process consists of the following main groups of activities:

- Test planning
- Test monitoring and control
- Test analysis
- Test design
- Test implementation
- Test execution
- Test completion

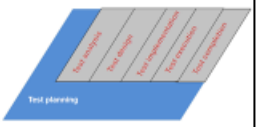
Each main group of activities is composed of constituent activities, which will be described in the subsections below. Each constituent activity consists of multiple individual tasks, which would vary from one project or release to another.

Further, although many of these main activity groups may appear logically sequential, they are often implemented iteratively. For example, Agile development involves small iterations of software design, build, and test that happen on a continuous basis, supported by on-going planning. So test activities are also happening on an iterative, continuous basis within this software development approach. Even in sequential software development, the stepped logical sequence of main groups of activities will involve overlap, combination, concurrency, or omission, so tailoring these main groups of activities within the context of the system and the project is usually required.

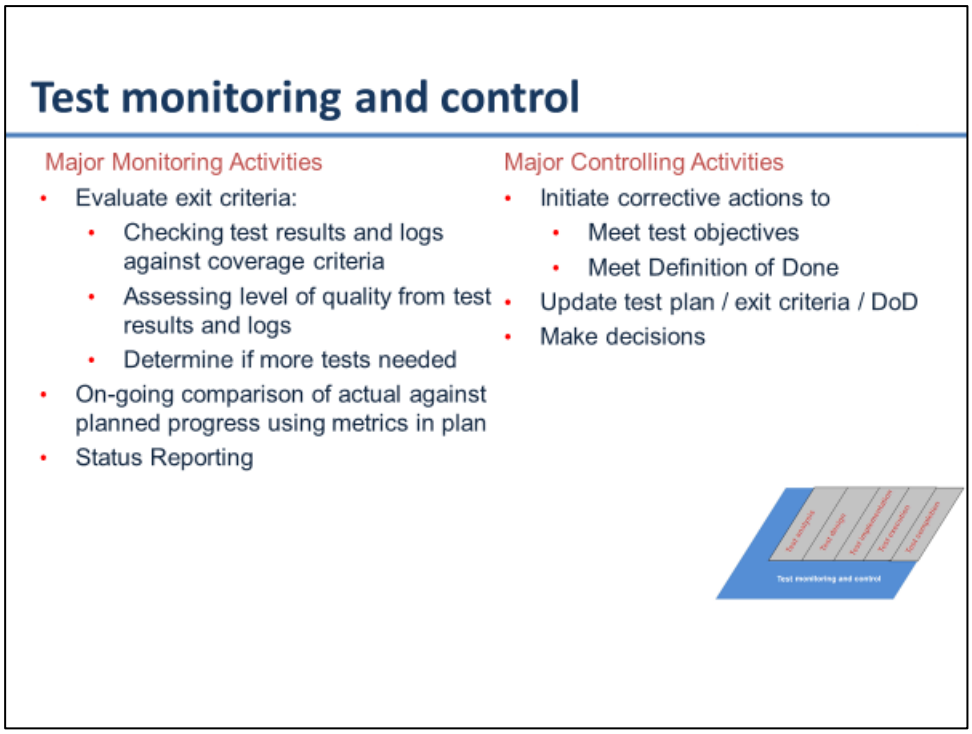
Test planning

Major Planning Activities

- Determine Scope and Risks
- Determine Test Objectives
- Determine Test Approach
- Implement Test Policy/Strategy
- Integrating & co-ordinating test activities into SDLC
- Determine Test Resources
- Schedule Test Analysis, Design, Implementation, Execution and Evaluation activities
- Selecting metrics for monitoring and control
- Budgeting for metrics' tasks
- Set level of detail for test documentation coverage



Test planning involves activities that define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context (e.g., specifying suitable test techniques and tasks, and formulating a test schedule for meeting a deadline). Test plans may be revisited based on feedback from monitoring and control activities. Test planning is further explained in section 5.2.

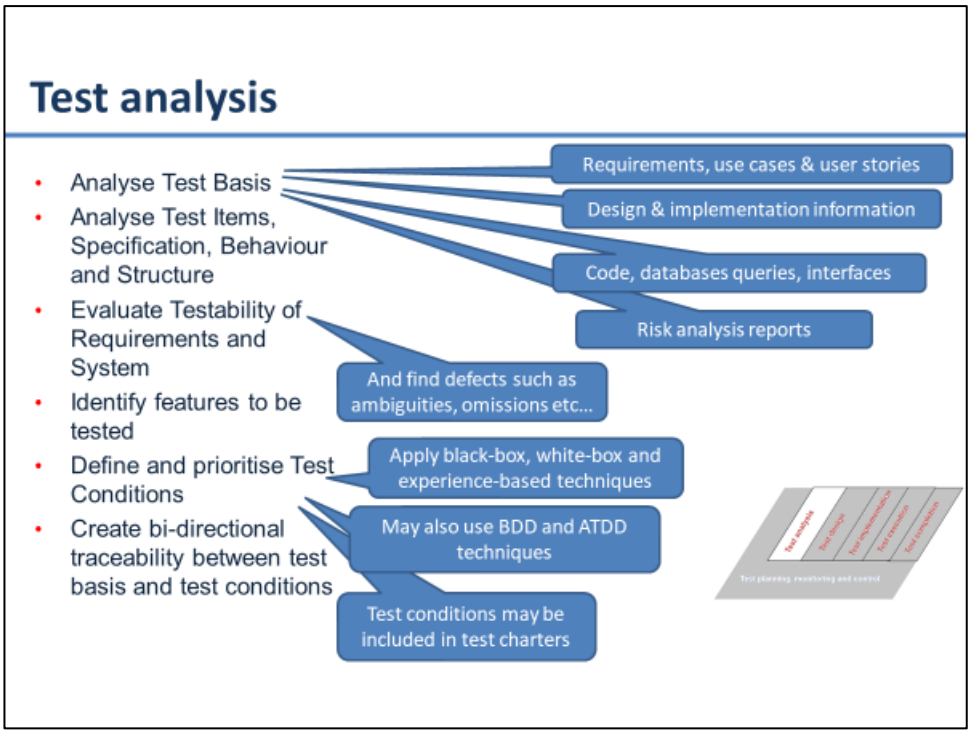


Test monitoring involves the on-going comparison of actual progress against planned progress using any test monitoring metrics defined in the test plan. Test control involves taking actions necessary to meet the objectives of the test plan (which may be updated over time). Test monitoring and control are supported by the evaluation of exit criteria, which are referred to as the definition of done in some software development lifecycle models (see ISTQB-CTFL-AT). For example, the evaluation of exit criteria for test execution as part of a given test level may include:

- Checking test results and logs against specified coverage criteria
- Assessing the level of component or system quality based on test results and logs
- Determining if more tests are needed (e.g., if tests originally intended to achieve a certain level of product risk coverage failed to do so, requiring additional tests to be written and executed)

Test progress against the plan is communicated to stakeholders in test progress reports, including deviations from the plan and information to support any decision to stop testing.

Test monitoring and control are further explained in section 5.3.



ISTQB Glossary Definition of Test Condition: “An aspect of the test basis that is relevant in order to achieve specific test objectives.”.

During test analysis, the test basis is analyzed to identify testable features and define associated test conditions. In other words, test analysis determines “what to test” in terms of measurable coverage criteria.

Test analysis includes the following major activities:

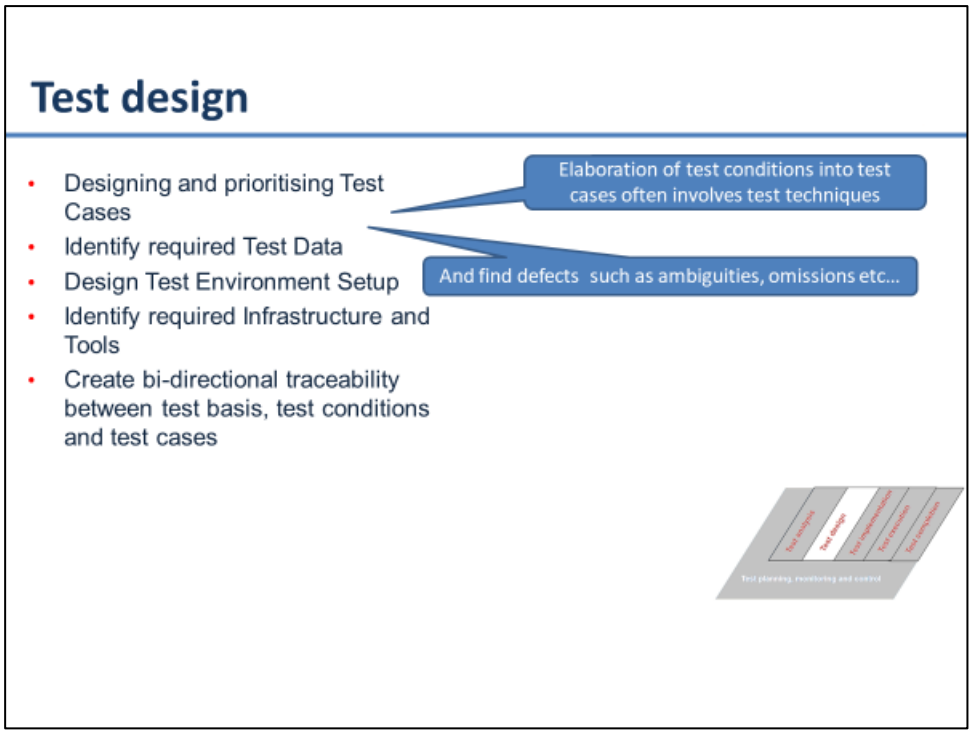
- Analyzing the test basis appropriate to the test level being considered, for example:
    - Requirement specifications, such as business requirements, functional requirements, system requirements, user stories, epics, use cases, or similar work products that specify desired functional and non-functional component or system behavior
    - Design and implementation information, such as system or software architecture diagrams or documents, design specifications, call flow graphs, modelling diagrams (e.g., UML or entity-relationship diagrams), interface specifications, or similar work products that specify component or system structure
    - The implementation of the component or system itself, including code, database metadata and queries, and interfaces
    - Risk analysis reports, which may consider functional, non-functional, and structural aspects of the component or system

- Evaluating the test basis and test items to identify defects of various types, such as:
    - Ambiguities
    - Omissions
    - Inconsistencies
    - Inaccuracies
    - Contradictions
    - Superfluous statements
- Identifying features and sets of features to be tested
- Defining and prioritizing test conditions for each feature based on analysis of the test basis, and considering functional, non-functional, and structural characteristics, other business and technical factors, and levels of risks
- Capturing bi-directional traceability between each element of the test basis and the associated test conditions (see sections 1.4.3 and 1.4.4)

The application of black-box, white-box, and experience-based test techniques can be useful in the process of test analysis (see chapter 4) to reduce the likelihood of omitting important test conditions and to define more precise and accurate test conditions.

In some cases, test analysis produces test conditions which are to be used as test objectives in test charters. Test charters are typical work products in some types of experience-based testing (see section 4.4.2). When these test objectives are traceable to the test basis, coverage achieved during such experience-based testing can be measured.

The identification of defects during test analysis is an important potential benefit, especially where no other review process is being used and/or the test process is closely connected with the review process. Such test analysis activities not only verify whether the requirements are consistent, properly expressed, and complete, but also validate whether the requirements properly capture customer, user, and other stakeholder needs. For example, techniques such as behavior driven development (BDD) and acceptance test driven development (ATDD), which involve generating test conditions and test cases from user stories and acceptance criteria prior to coding. These techniques also verify, validate, and detect defects in the user stories and acceptance criteria (see ISTQB-CTFL-AT).



ISTQB Glossary Definition of Test Case: “A set of preconditions, inputs, actions (where applicable), expected results and post conditions, developed based on test conditions.”.

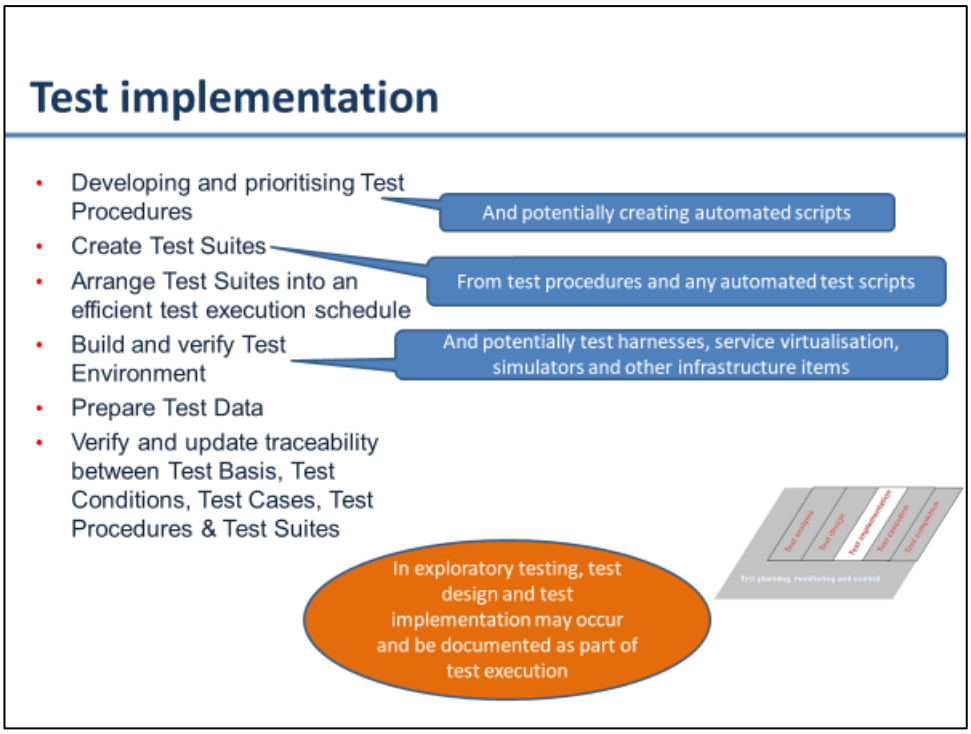
During test design, the test conditions are elaborated into high-level test cases, sets of high-level test cases, and other testware. So, test analysis answers the question “what to test?” while test design answers the question “how to test?”

Test design includes the following major activities:

- Designing and prioritizing test cases and sets of test cases
- Identifying necessary test data to support test conditions and test cases
- Designing the test environment and identifying any required infrastructure and tool
- Capturing bi-directional traceability between the test basis, test conditions, and test cases (see section 1.4.4)

The elaboration of test conditions into test cases and sets of test cases during test design often involves using test techniques (see chapter 4).

As with test analysis, test design may also result in the identification of similar types of defects in the test basis. Also, as with test analysis, the identification of defects during test design is an important potential benefit.



ISTQB Glossary Definition of Test Procedure: “A sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution.”.

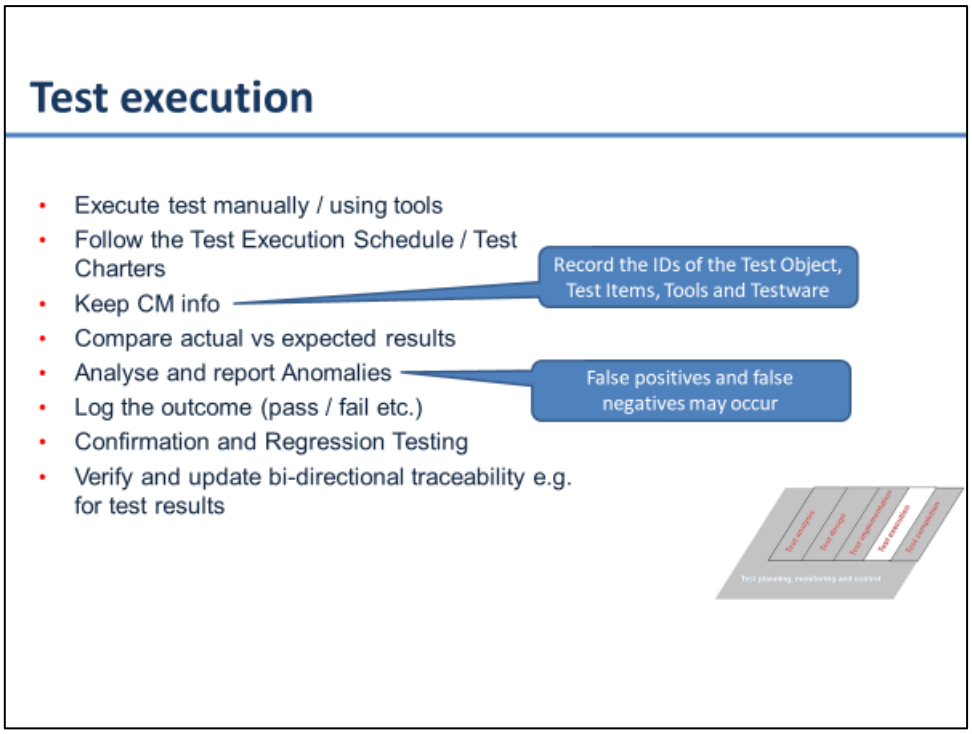
During test implementation, the testware necessary for test execution is created and/or completed, including sequencing the test cases into test procedures. So, test design answers the question “how to test?” while test implementation answers the question “do we now have everything in place to run the tests?”

Test implementation includes the following major activities:

- Developing and prioritizing test procedures, and, potentially, creating automated test scripts
- Creating test suites from the test procedures and (if any) automated test scripts
- Arranging the test suites within a test execution schedule in a way that results in efficient test execution (see section 5.2.4)
- Building the test environment (including, potentially, test harnesses, service virtualization, simulators, and other infrastructure items) and verifying that everything needed has been set up correctly
- Preparing test data and ensuring it is properly loaded in the test environment
- Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test suites (see section 1.4.4)

Test design and test implementation tasks are often combined.

In exploratory testing and other types of experience-based testing, test design and implementation may occur, and may be documented, as part of test execution. Exploratory testing may be based on test charters (produced as part of test analysis), and exploratory tests are executed immediately as they are designed and implemented (see section 4.4.2).



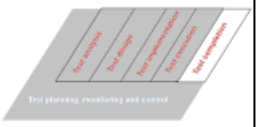
During test execution, test suites are run in accordance with the test execution schedule.

Test execution includes the following major activities:

- Recording the IDs and versions of the test item(s) or test object, test tool(s), and testware
- Executing tests either manually or by using test execution tools
- Comparing actual results with expected results
- Analyzing anomalies to establish their likely causes (e.g., failures may occur due to defects in the code, but false positives also may occur (see section 1.2.3)
- Reporting defects based on the failures observed (see section 5.6)
- Logging the outcome of test execution (e.g., pass, fail, blocked)
- Repeating test activities either as a result of action taken for an anomaly, or as part of the planned testing (e.g., execution of a corrected test, confirmation testing, and/or regression testing)
- Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test results.

Test completion

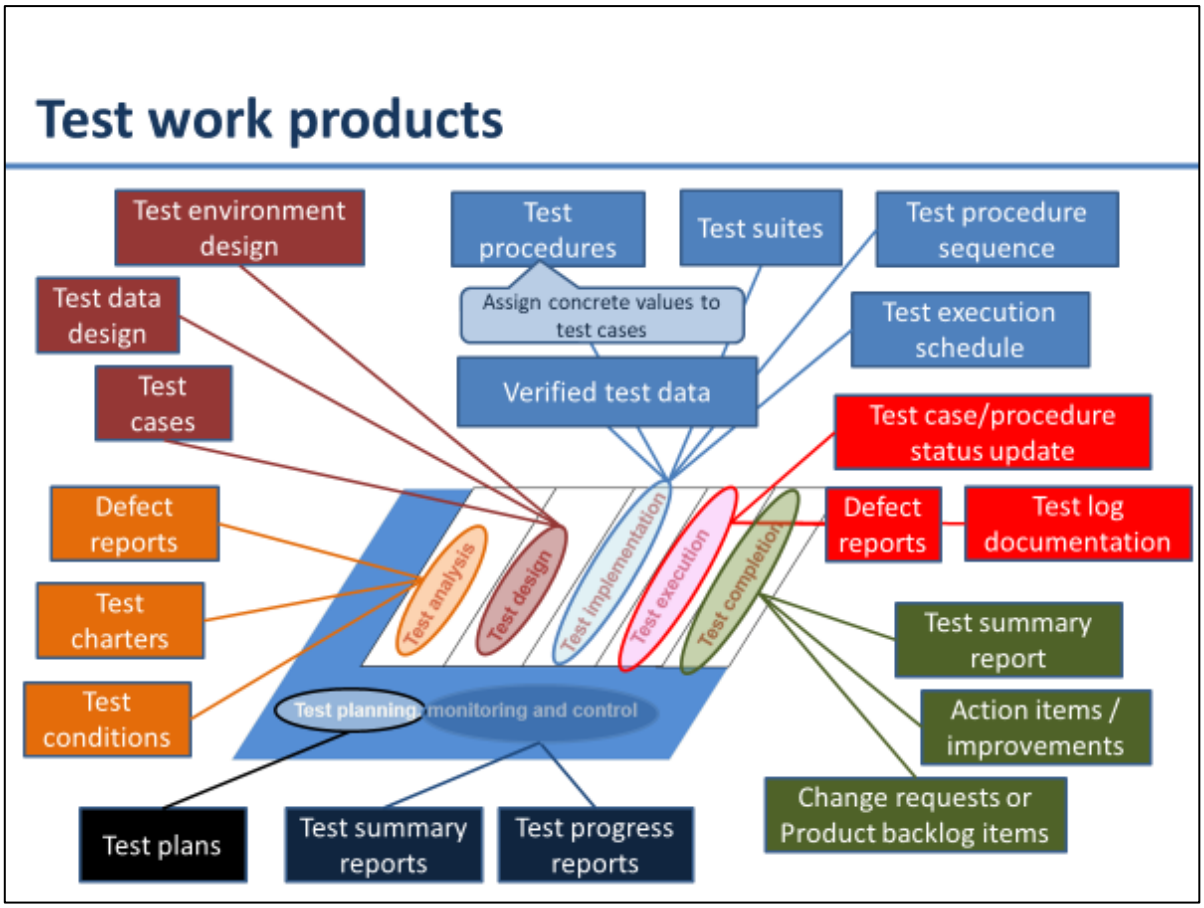
- Check all defects are closed and raise change requests / product backlog items for those that remain open
- Create Test Summary Report
- Finalise and archive Testware, Test Environment and Infrastructure for future use
- Handover Testware to maintenance teams
- Analyse Lessons Learned
- Use the information gathered to improve test process maturity



Test completion activities collect 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 software system is released, a test project is completed (or cancelled), an Agile project iteration is finished, a test level is completed, or a maintenance release has been completed.

Test completion includes the following major activities:

- Checking whether all defect reports are closed, entering change requests or product backlog items for any defects that remain unresolved at the end of test execution
- Creating a test summary report to be communicated to stakeholders
- Finalizing and archiving the test environment, the test data, the test infrastructure, and other testware for later reuse
- Handing over the testware to the maintenance teams, other project teams, and/or other stakeholders who could benefit from its use
- Analyzing lessons learned from the completed test activities to determine changes needed for future iterations, releases, and projects
- Using the information gathered to improve test process maturity



Test work products are created as part of the test process. Just as there is significant variation in the way that organizations implement the test process, there is also significant variation in the types of work products created during that process, in the ways those work products are organized and managed, and in the names used for those work products. This syllabus adheres to the test process outlined above, and the work products described in this syllabus and in the ISTQB® Glossary. ISO standard (ISO/IEC/IEEE 29119-3) may also serve as a guideline for test work products. Many of the test work products described in this section can be captured and managed using test management tools and defect management tools (see chapter 6).

Test planning work products
Test planning work products typically include one or more test plans. The test plan includes information about the test basis, to which the other test work products will be related via traceability information (see below and section 1.4.4), as well as exit criteria (or definition of done) which will be used during test monitoring and control. Test plans are described in section 5.2.

Test monitoring and control work products
Test monitoring and control work products typically include various types of test reports, including test progress reports produced on an ongoing and/or a regular basis, and test summary reports produced at various completion milestones. All test reports should provide audience-relevant details about the test progress as of the date of the report, including summarizing the test execution results once those become available.

Test monitoring and control work products should also address project management concerns, such as task completion, resource allocation and usage, and effort.

Test monitoring and control, and the work products created during these activities, are further explained in section 5.3 of this syllabus.

Test analysis work products
Test analysis work products include defined and prioritized test conditions, each of which is ideally bidirectionally traceable to the specific element(s) of the test basis it covers. For exploratory testing, test analysis may involve the creation of test charters. Test analysis may also result in the discovery and reporting of defects in the test basis.

Test design work products
Test design results in test cases and sets of test cases to exercise the test conditions defined in test analysis. It is often a good practice to design high-level test cases, without concrete values for input data and expected results. Such high-level test cases are reusable across multiple test cycles with different concrete data, while still adequately documenting the scope of the test case. Ideally, each test case is bidirectionally traceable to the test condition(s) it covers.

Test design also results in:

- The design and/or identification of the necessary test data
- The design of the test environment
- The identification of infrastructure and tools

though the extent to which these results are documented varies significantly.

Test implementation work products
Test implementation work products include:

- Test procedures and the sequencing of those test procedures
- Test suites
- A test execution schedule

Ideally, once test implementation is complete, achievement of coverage criteria established in the test plan can be demonstrated via bi-directional traceability between test procedures and specific elements of the test basis, through the test cases and test conditions.

In some cases, test implementation involves creating work products using or used by tools, such as service virtualization and automated test scripts.

Test implementation also may result in the creation and verification of test data and the test environment. The completeness of the documentation of the data and/or environment verification results may vary significantly.

The test data serve to assign concrete values to the inputs and expected results of test cases. Such concrete values, together with explicit directions about the use of the concrete values, turn high-level test cases into executable low-level test cases. The same high-level test case may use different test data when executed on different releases of the test object. The concrete expected results which are associated with concrete test data are identified by using a test oracle.

In exploratory testing, some test design and implementation work products may be created during test execution, though the extent to which exploratory tests (and their traceability to specific elements of the test basis) are documented may vary significantly.

Test conditions defined in test analysis may be further refined in test implementation.

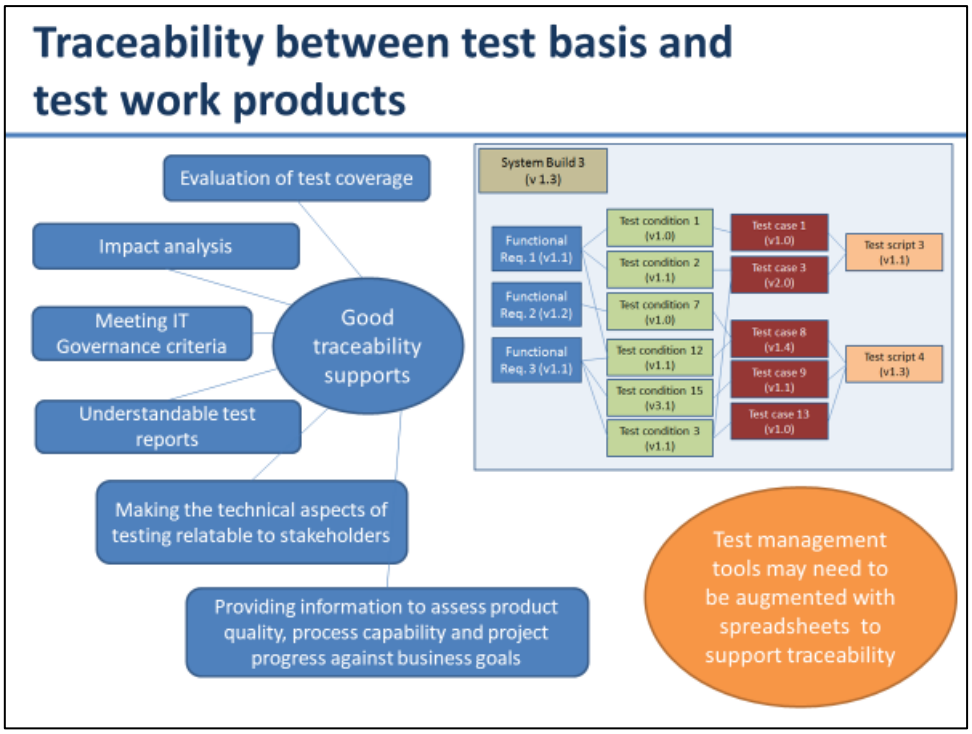
Test execution work products
Test execution work products include:

- Documentation of the status of individual test cases or test procedures (e.g., ready to run, pass, fail, blocked, deliberately skipped, etc.)
- Defect reports (see section 5.6)
- Documentation about which test item(s), test object(s), test tools, and testware were involved in the testing

Ideally, once test execution is complete, the status of each element of the test basis can be determined and reported via bi-directional traceability with the associated the test procedure(s). For example, we can say which requirements have passed all planned tests, which requirements have failed tests and/or have defects associated with them, and which requirements have planned tests still waiting to be run. This enables verification that the coverage criteria have been met and enables the reporting of test results in terms that are understandable to stakeholders.

Test completion work products
Test completion work products include:

- Test summary reports
- Action items for improvement of subsequent projects or iterations
- Change requests or product backlog items
- Finalized testware.



As mentioned in section 1.4.3, test work products and the names of those work products vary significantly. Regardless of these variations, in order to implement effective test monitoring and control, it is important to establish and maintain traceability throughout the test process between each element of the test basis and the various test work products associated with that element, as described above.

In addition to the evaluation of test coverage, good traceability supports:

- Analyzing the impact of changes
- Making testing auditable
- Meeting IT governance criteria
- Improving the understandability of test progress reports and test summary reports to include the status of elements of the test basis (e.g., requirements that passed their tests, requirements that failed their tests, and requirements that have pending tests)
- Relating the technical aspects of testing to stakeholders in terms that they can understand
- Providing information to assess product quality, process capability, and project progress against business goals

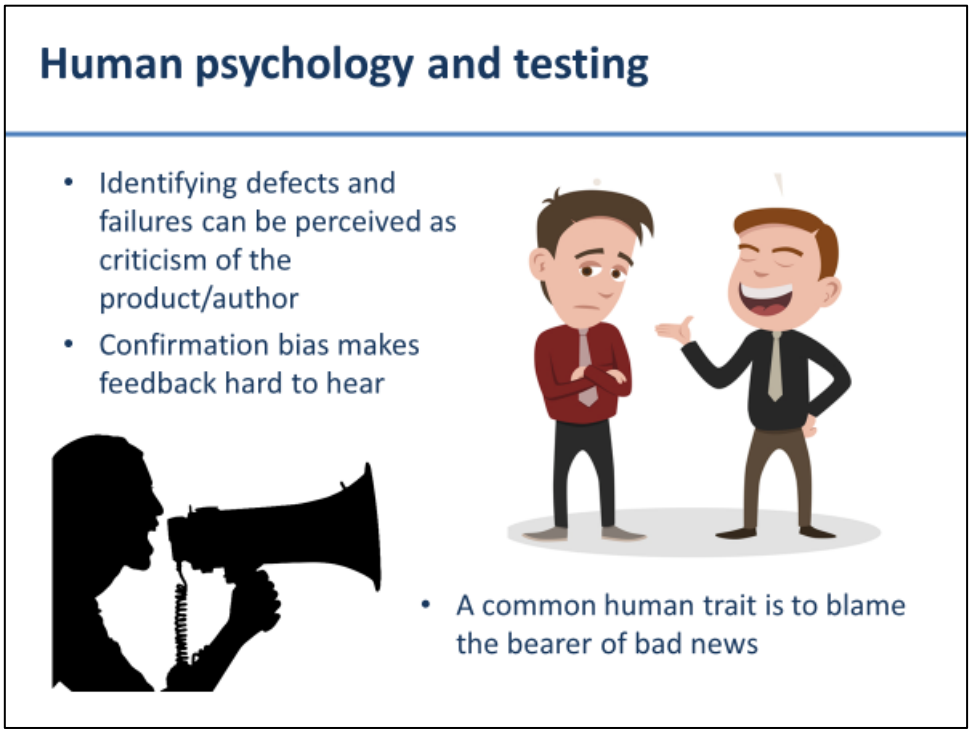
Some test management tools provide test work product models that match part or all of the test work products outlined in this section. Some organizations build their own management systems to organize the work products and provide the information traceability they require.

Summary

The stages of the test process are:

- Test Planning
- Test Monitoring and Control
- Test Analysis
- Test Design
- Test Implementation
- Test Execution
- Test Completion

Section 1.5 - The Psychology of Testing



Software development, including software testing, involves human beings. Therefore, human psychology has important effects on software testing.

Identifying defects during a static test such as a requirements review or user story refinement session, or identifying failures during dynamic test execution, may be perceived as criticism of the product and of its author. An element of human psychology called confirmation bias can make it difficult to accept information that disagrees with currently held beliefs. For example, since developers expect their code to be correct, they have a confirmation bias that makes it difficult to accept that the code is incorrect. In addition to confirmation bias, other cognitive biases may make it difficult for people to understand or accept information produced by testing. Further, it is a common human trait to blame the bearer of bad news, and information produced by testing often contains bad news.

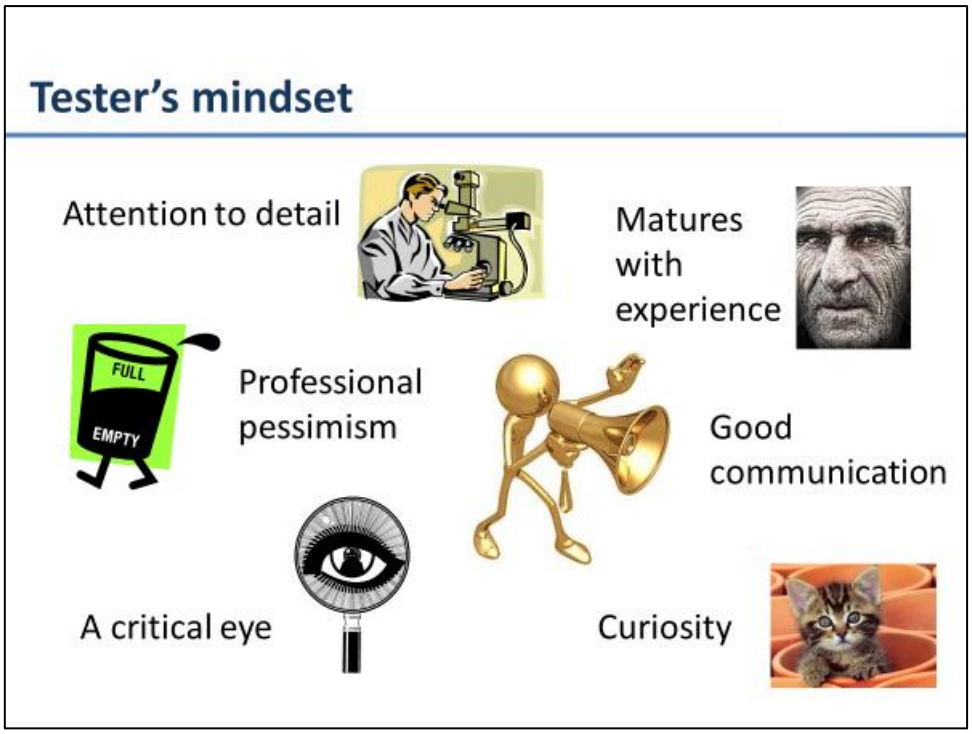
Tester's and developer's mindsets

- Testers think about verifying, validating and finding defects prior to release
- Developers often focus more on creating rather than contemplating what might be wrong
- Confirmation bias makes it difficult for developers to become aware of their errors
- Bringing the two mindsets together can help to achieve a higher level of quality

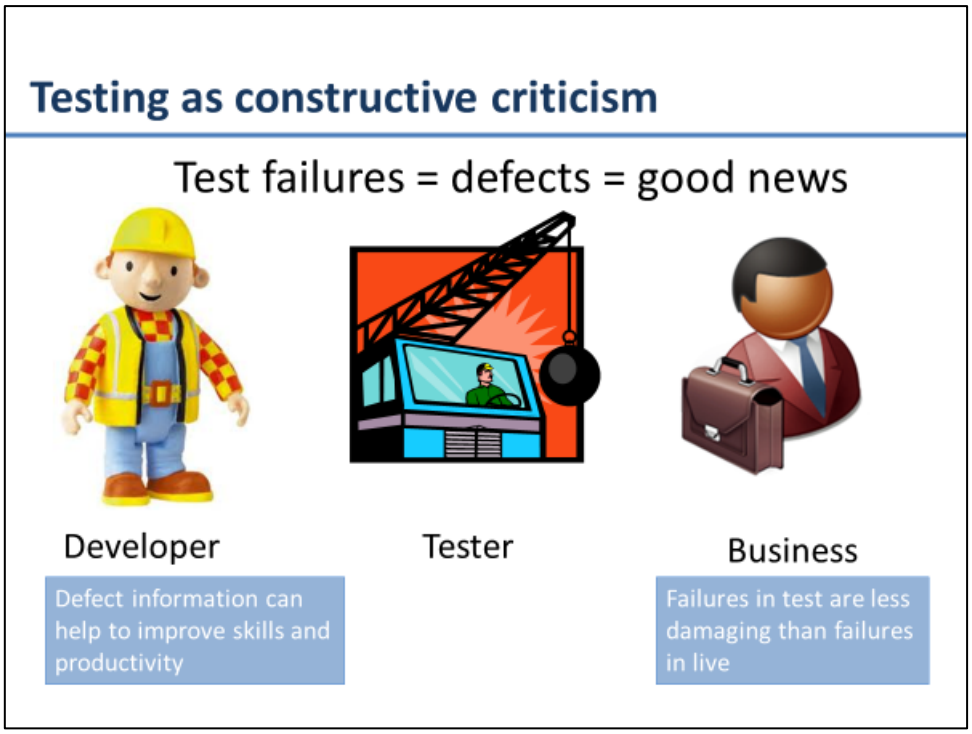
Developers and testers often think differently. The primary objective of development is to design and build a product. As discussed earlier, the objectives of testing include verifying and validating the product, finding defects prior to release, and so forth. These are different sets of objectives which require different mindsets. Bringing these mindsets together helps to achieve a higher level of product quality.

A developer's mindset may include some of the elements of a tester's mindset, but successful developers are often more interested in designing and building solutions than in contemplating what might be wrong with those solutions. In addition, confirmation bias makes it difficult to become aware of errors committed by themselves.

With the right mindset, developers are able to test their own code. Different software development lifecycle models often have different ways of organizing the testers and test activities.

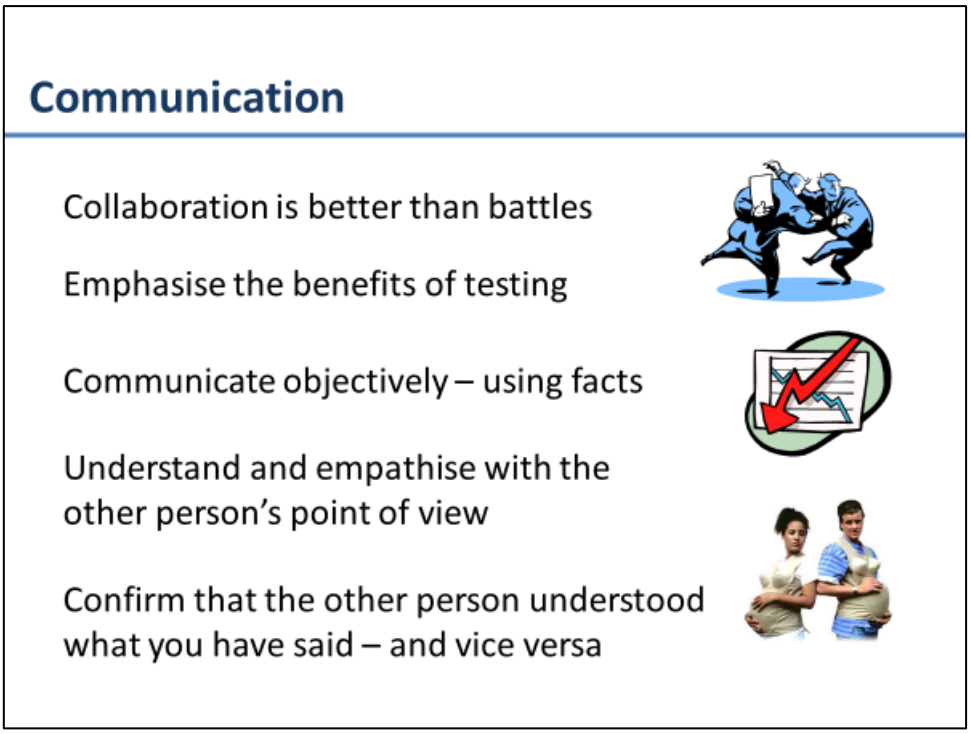


A mindset reflects an individual's assumptions and preferred methods for decision making and problem-solving. A tester's mindset should include curiosity, professional pessimism, a critical eye, attention to detail, and a motivation for good and positive communications and relationships. A tester's mindset tends to grow and mature as the tester gains experience.



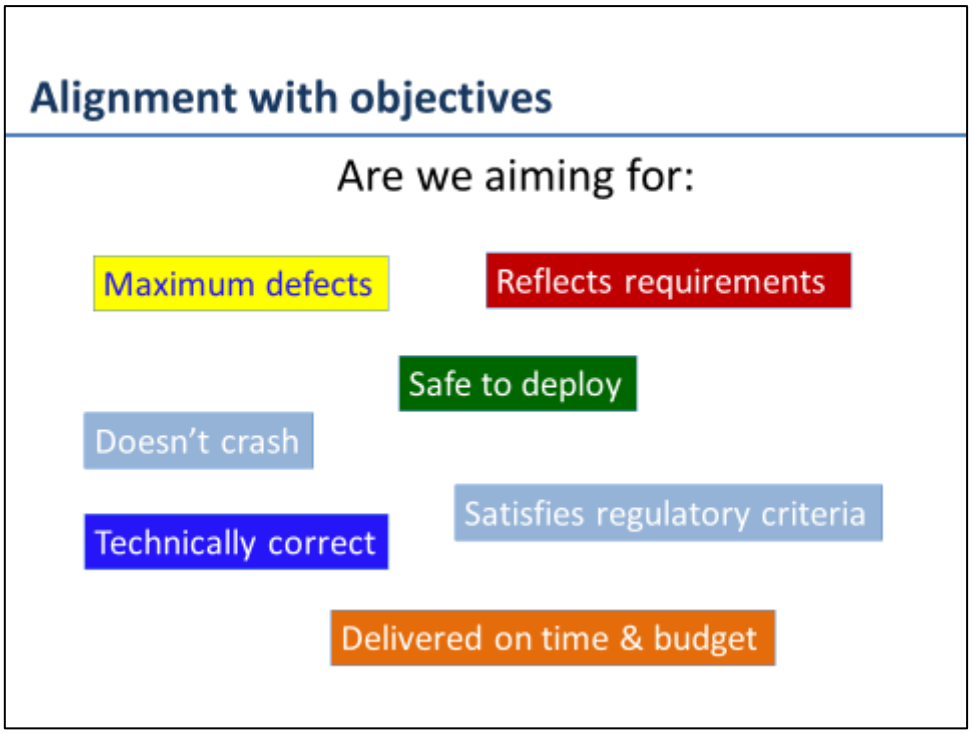
As a result of these psychological factors, some people may perceive testing as a destructive activity, even though it contributes greatly to project progress and product quality (see sections 1.1 and 1.2). To try to reduce these perceptions, information about defects and failures should be communicated in a constructive way. This way, tensions between the testers and the analysts, product owners, designers, and developers can be reduced. This applies during both static and dynamic testing.

Testers and test managers need to have good interpersonal skills to be able to communicate effectively about defects, failures, test results, test progress, and risks, and to build positive relationships with colleagues.



Ways to communicate well include the following examples:

- Start with collaboration rather than battles. Remind everyone of the common goal of better quality systems.
- Emphasize the benefits of testing. For example, for the authors, defect information can help them improve their work products and their skills. For the organization, defects found and fixed during testing will save time and money and reduce overall risk to product quality.
- Communicate test results and other findings in a neutral, fact-focused way without criticizing the person who created the defective item. Write objective and factual defect reports and review findings.
- Try to understand how the other person feels and the reasons they may react negatively to the information.
- Confirm that the other person has understood what has been said and vice versa.



Typical test objectives were discussed earlier (see section 1.1). Clearly defining the right set of test objectives has important psychological implications. Most people tend to align their plans and behaviors with the objectives set by the team, management, and other stakeholders. It is also important that testers adhere to these objectives with minimal personal bias.

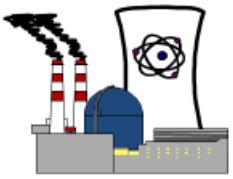
Independent testing

- Different SDLCs often have different ways of organising the testers and some make independent testing mandatory

- Some independent testing increases defect detection effectiveness

- Brings additional perspective

- Invaluable in large, complex or safety critical projects



With the right mindset, developers are able to test their own code. Different software development lifecycle models often have different ways of organizing the testers and test activities. Having some of the test activities done by independent testers increases defect detection effectiveness, which is particularly important for large, complex, or safety-critical systems. Independent testers bring a perspective which is different than that of the work product authors (i.e., business analysts, product owners, designers, and developers), since they have different cognitive biases from the authors.

Summary

- There are psychological factors that can influence the success of testing
- Developers and testers have different mind-sets

End of section exercise

- Attempt the following Questions from Sample Exam B
- Questions 1 to 8
- Please mark your own (See Sample Exam B Answers) and read the answer justifications as needed

You have 12 minutes

Learning Objectives for Fundamentals of Testing

In order to complete this section, you should satisfy yourself that you are able to understand or perform the following items to the standard indicated by the corresponding “K - learning level”. For an explanation of “K levels”, please see the course introduction section – “Learning Objectives and Levels of Knowledge”.

Keywords

coverage, debugging, defect, error, failure, quality, quality assurance, root cause, test analysis, test basis, test case, test completion, test condition, test control, test data, test design, test execution, test implementation, test monitoring, test object, test objective, test oracle, test planning, test procedure, test process, test suite, testing, testware, traceability, validation, verification

Learning Objectives for Fundamentals of Testing:

1.1 What is Testing?

FL-1.1.1 (K1) Identify typical objectives of testing
FL-1.1.2 (K2) Differentiate testing from debugging

1.2 Why is Testing Necessary?

FL-1.2.1 (K2) Give examples of why testing is necessary
FL-1.2.2 (K2) Describe the relationship between testing and quality assurance and give examples of how testing contributes to higher quality
FL-1.2.3 (K2) Distinguish between error, defect, and failure
FL-1.2.4 (K2) Distinguish between the root cause of a defect and its effects

1.3 Seven Testing Principles

FL-1.3.1 (K2) Explain the seven testing principles

1.4 Test Process

FL-1.4.1 (K2) Explain the impact of context on the test process
FL-1.4.2 (K2) Describe the test activities and respective tasks within the test process
FL-1.4.3 (K2) Differentiate the work products that support the test process
FL-1.4.4 (K2) Explain the value of maintaining traceability between the test basis and test work products

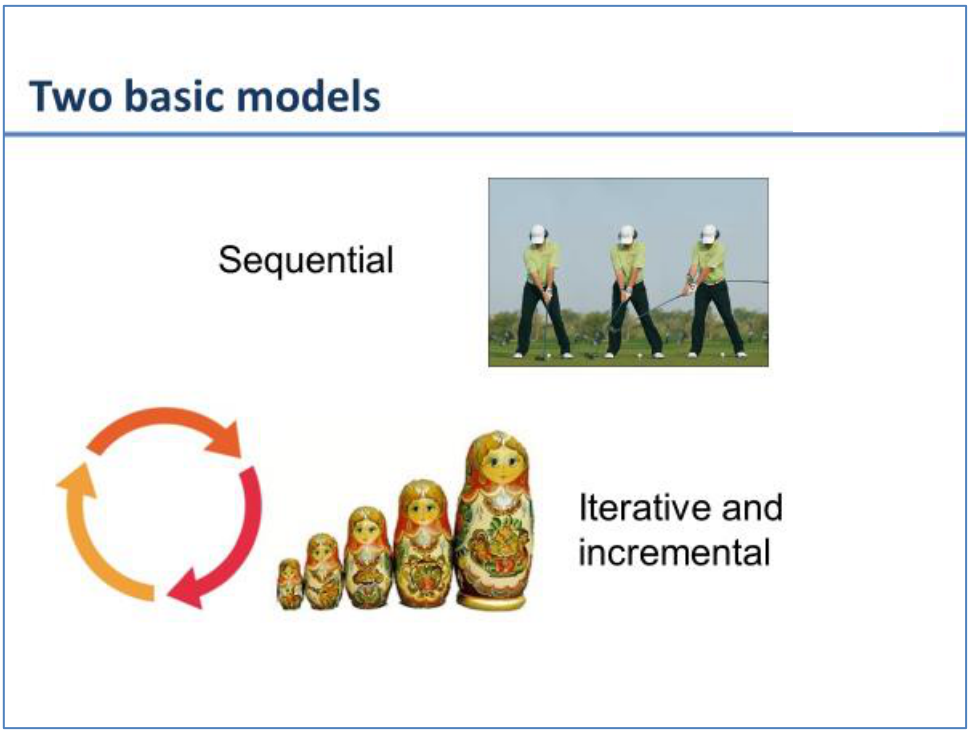
1.5 The Psychology of Testing

FL-1.5.1 (K1) Identify the psychological factors that influence the success of testing
FL-1.5.2 (K2) Explain the difference between the mindset required for test activities and the mindset required for development activities

Testing Throughout the Software Lifecycle

Table of Contents

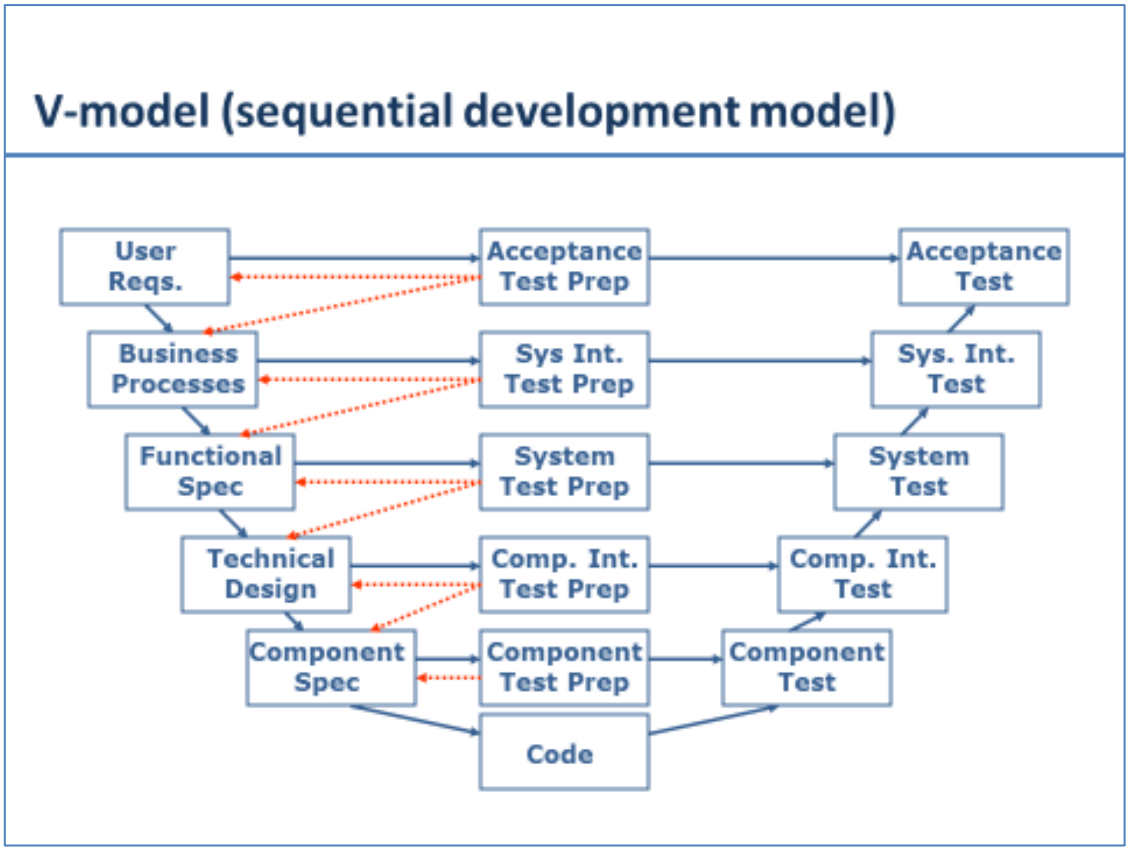
Section 2.1 - Software Development Lifecycle Models



A software development lifecycle model describes the types of activity performed at each stage in a software development project, and how the activities relate to one another logically and chronologically. There are a number of different software development lifecycle models, each of which requires different approaches to testing.

This syllabus categorizes common software development lifecycle models as follows:

- Sequential development models
- Iterative and incremental development models

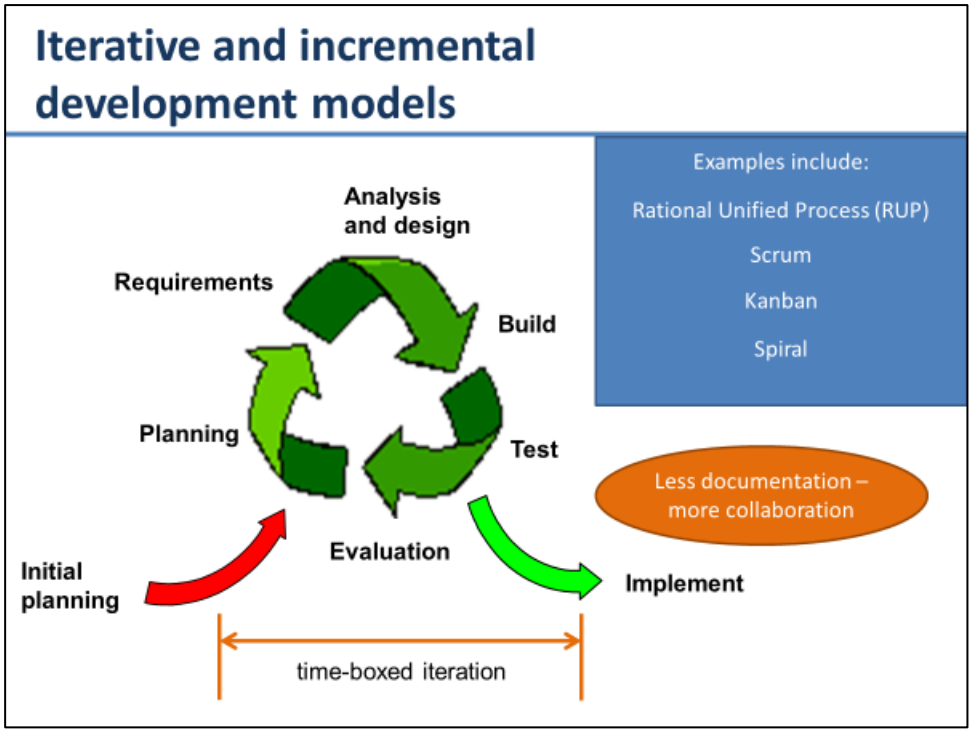


A sequential development model describes the software development process as a linear, sequential flow of activities. This means that any phase in the development process should begin when the previous phase is complete. In theory, there is no overlap of phases, but in practice, it is beneficial to have early feedback from the following phase.

In the Waterfall model, the development activities (e.g., requirements analysis, design, coding, testing) are completed one after another. In this model, test activities only occur after all other development activities have been completed.

Unlike the Waterfall model, the V-model integrates the test process throughout the development process, implementing the principle of early testing. Further, the V-model includes test levels associated with each corresponding development phase, which further supports early testing (see section 2.2 for a discussion of test levels). In this model, the execution of tests associated with each test level proceeds sequentially, but in some cases overlapping occurs.

Sequential development models deliver software that contains the complete set of features, but typically require months or years for delivery to stakeholders and users.



Incremental development involves establishing requirements, designing, building, and testing a system in pieces, which means that the software's features grow incrementally. The size of these feature increments varies, with some methods having larger pieces and some smaller pieces. The feature increments can be as small as a single change to a user interface screen or new query option.

Iterative development occurs when groups of features are specified, designed, built, and tested together in a series of cycles, often of a fixed duration. Iterations may involve changes to features developed in earlier iterations, along with changes in project scope. Each iteration delivers working software which is a growing subset of the overall set of features until the final software is delivered or development is stopped.

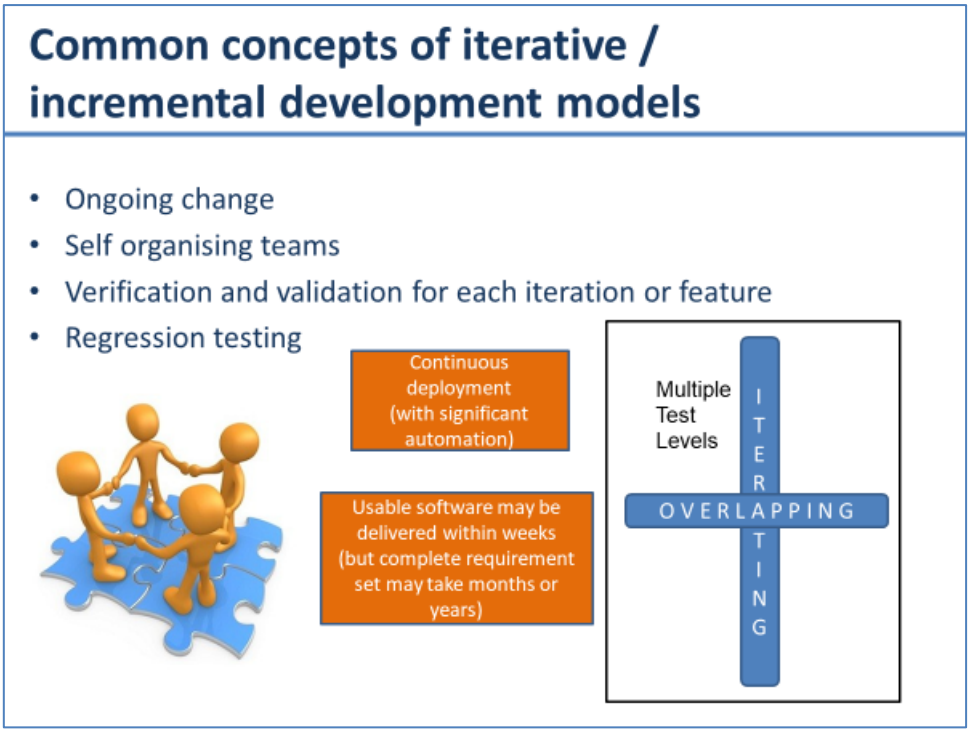
Examples include:

- Rational Unified Process: Each iteration tends to be relatively long (e.g., two to three months), and the feature increments are correspondingly large, such as two or three groups of related features

- Scrum: Each iteration tends to be relatively short (e.g., hours, days, or a few weeks), and the feature increments are correspondingly small, such as a few enhancements and/or two or three new features

- Kanban: Implemented with or without fixed-length iterations, which can deliver either a single enhancement or feature upon completion, or can group features together to release at once

- Spiral: Involves creating experimental increments, some of which may be heavily re-worked or even abandoned in subsequent development work

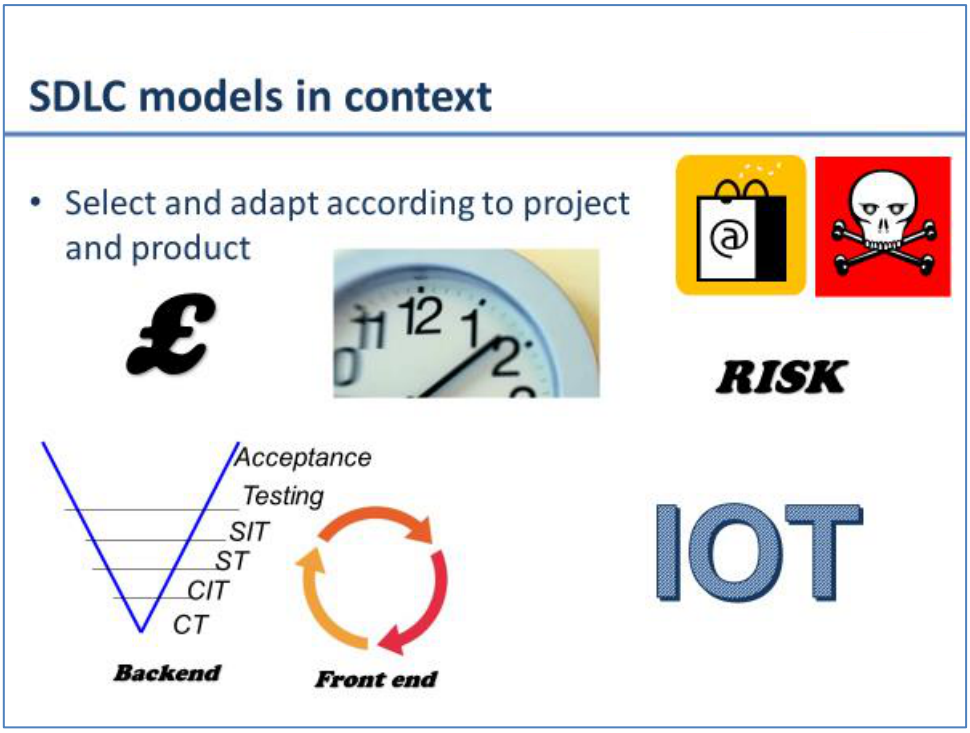


Components or systems developed using these methods often involve overlapping and iterating test levels throughout development. Ideally, each feature is tested at several test levels as it moves towards delivery. In some cases, teams use continuous delivery or continuous deployment, both of which involve significant automation of multiple test levels as part of their delivery pipelines. Many development efforts using these methods also include the concept of self-organizing teams, which can change the way testing work is organized as well as the relationship between testers and developers.

These methods form a growing system, which may be released to end-users on a feature-by-feature basis, on an iteration-by-iteration basis, or in a more traditional major-release fashion. Regardless of whether the software increments are released to end-users, regression testing is increasingly important as the system grows.

In contrast to sequential models, iterative and incremental models may deliver usable software in weeks or even days, but may only deliver the complete set of requirements product over a period of months or even years.

For more information on software testing in the context of Agile development, see ISTQB-CTFL-AT, Black 2017, Crispin 2008, and Gregory 2015.



Software development lifecycle models must be selected and adapted to the context of project and product characteristics. An appropriate software development lifecycle model should be selected and adapted based on the project goal, the type of product being developed, business priorities (e.g., time-to market), and identified product and project risks. For example, the development and testing of a minor internal administrative system should differ from the development and testing of a safety-critical system such as an automobile's brake control system. As another example, in some cases organizational and cultural issues may inhibit communication between team members, which can impede iterative development.

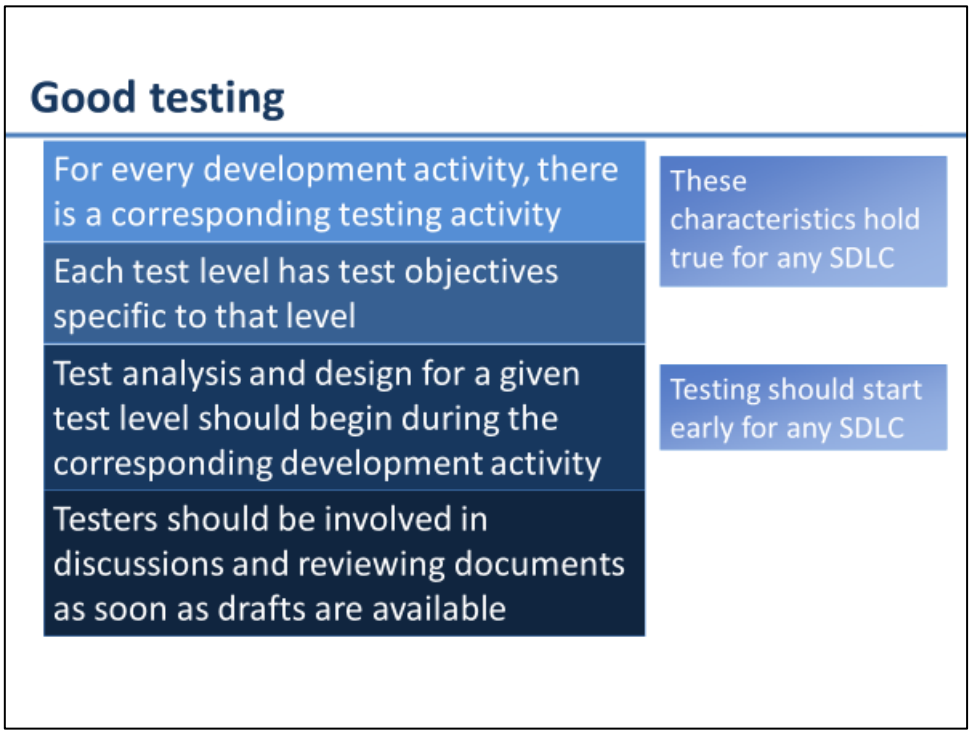
Depending on the context of the project, it may be necessary to combine or reorganize test levels and/or test activities. For example, for the integration of a commercial off-the-shelf (COTS) software product into a larger system, the purchaser may perform interoperability testing at the system integration test level (e.g., integration to the infrastructure and other systems) and at the acceptance test level (functional and non-functional, along with user acceptance testing and operational acceptance testing). See section 2.2 for a discussion of test levels and section 2.3 for a discussion of test types.

In addition, software development lifecycle models themselves may be combined. For example, a V-model may be used for the development and testing of the backend systems and their integrations, while an Agile development model may be used to develop and test the front-end user interface (UI) and functionality. Prototyping may be used early in a project, with an incremental development model adopted once the experimental phase is complete.

Internet of Things (IoT) systems, which consist of many different objects, such as devices, products, and services, typically apply separate software development lifecycle models for each object. This presents a particular challenge for the development of Internet of Things system versions. Additionally the software development lifecycle of such objects places stronger emphasis on the later phases of the software development lifecycle after they have been introduced to operational use (e.g., operate, update, and decommission phases).

Reasons why software development models must be adapted to the context of project and product characteristics can be:

- Difference in product risks of systems (complex or simple project)
- Many business units can be part of a project or program (combination of sequential and agile development)
- Short time to deliver a product to the market (merge of test levels and/or integration of test types in test levels)



It is an important part of a tester's role to be familiar with the common software development lifecycle models so that appropriate test activities can take place.

In any software development lifecycle model, there are several characteristics of good testing:

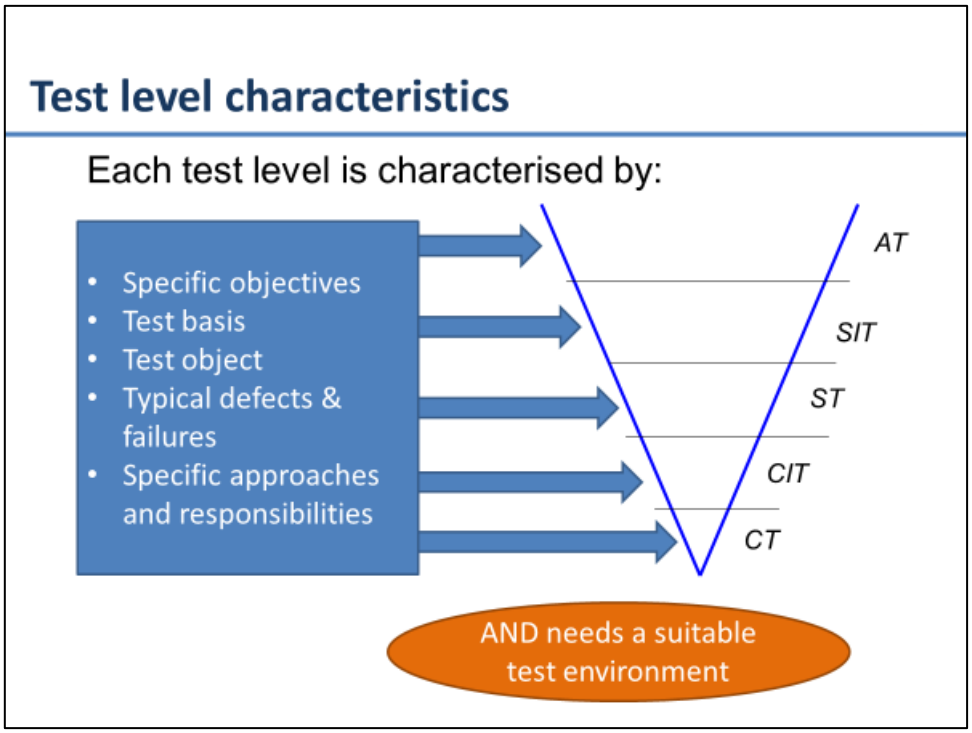
- For every development activity, there is a corresponding test activity
- Each test level has test objectives specific to that level
- Test analysis and design for a given test level begin during the corresponding development activity
- Testers participate in discussions to define and refine requirements and design, and are involved in reviewing work products (e.g., requirements, design, user stories, etc.) as soon as drafts are available

No matter which software development lifecycle model is chosen, test activities should start in the early stages of the lifecycle, adhering to the testing principle of early testing.

Summary

- The development life cycle model defines the relationship between development, test activities and work products
- Software development models must be adapted to the context of the project and product characteristics
- There are characteristics of good testing that are applicable to any lifecycle model

Section 2.2 - Test Levels



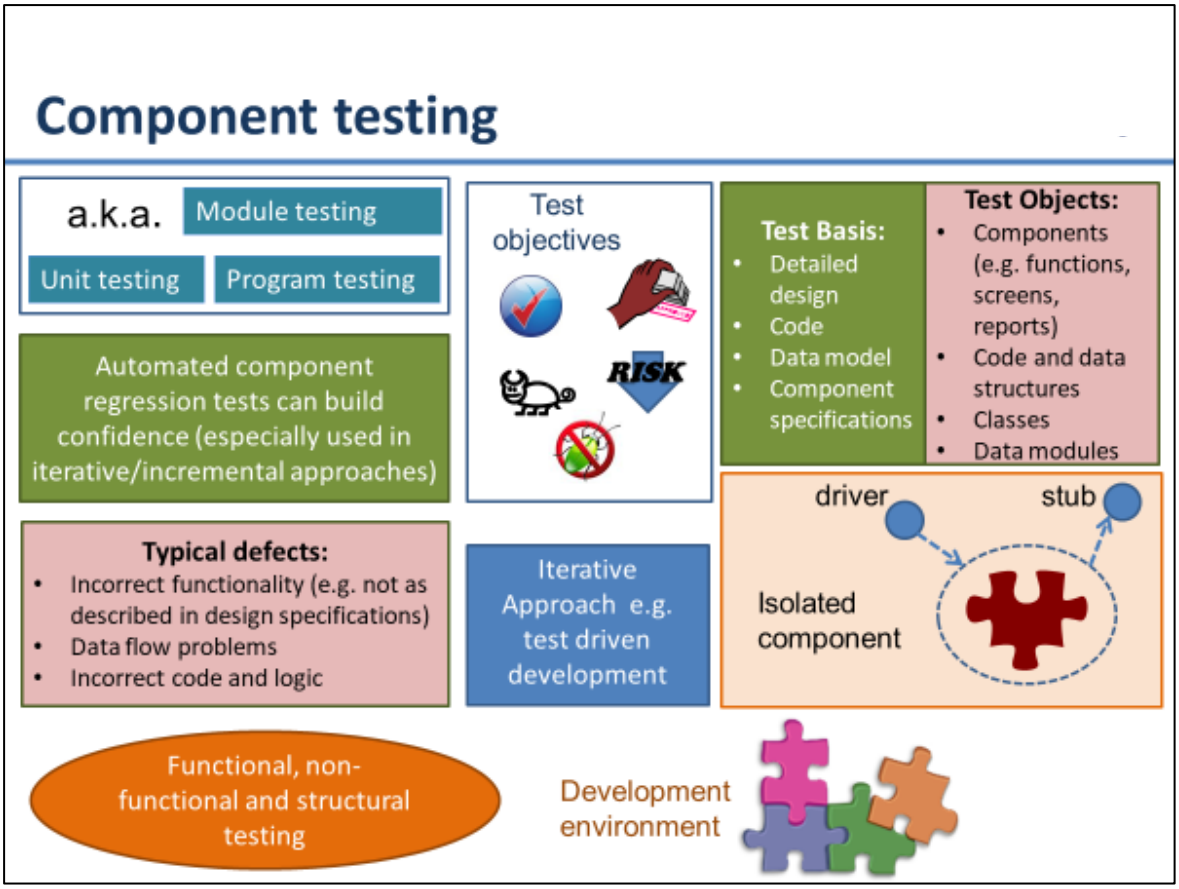
Test levels are groups of test activities that are organized and managed together. Each test level is an instance of the test process, consisting of the activities described in section 1.4, performed in relation to software at a given level of development, from individual units or components to complete systems or, where applicable, systems of systems. Test levels are related to other activities within the software development lifecycle. The test levels used in this syllabus are:

- Component testing
- Integration testing
- System testing
- Acceptance testing

Test levels are characterized by the following attributes:

- Specific objectives
- Test basis, referenced to derive test cases
- Test object (i.e., what is being tested)
- Typical defects and failures
- Specific approaches and responsibilities

For every test level, a suitable test environment is required. In acceptance testing, for example, a production-like test environment is ideal, while in component testing the developers typically use their own development environment.



Objectives of component testing
Component testing (also known as unit or module testing) focuses on components that are separately testable. Objectives of component testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviors of the component are as designed and specified
- Building confidence in the component's quality
- Finding defects in the component
- Preventing defects from escaping to higher test levels

In some cases, especially in incremental and iterative development models (e.g., Agile) where code changes are ongoing, automated component regression tests play a key role in building confidence that changes have not broken existing components.

Component testing is often done in isolation from the rest of the system, depending on the software development lifecycle model and the system, which may require mock objects, service virtualization, harnesses, stubs, and drivers. Component testing may cover functionality (e.g., correctness of calculations), non-functional characteristics (e.g., searching for memory leaks), and structural properties (e.g., decision testing).

Test basis
Examples of work products that can be used as a test basis for component testing include:

- Detailed design
- Code
- Data model
- Component specifications

(continued on next page)

Test objects
Typical test objects for component testing include:

- Components, units or modules
- Code and data structures
- Classes
- Database modules

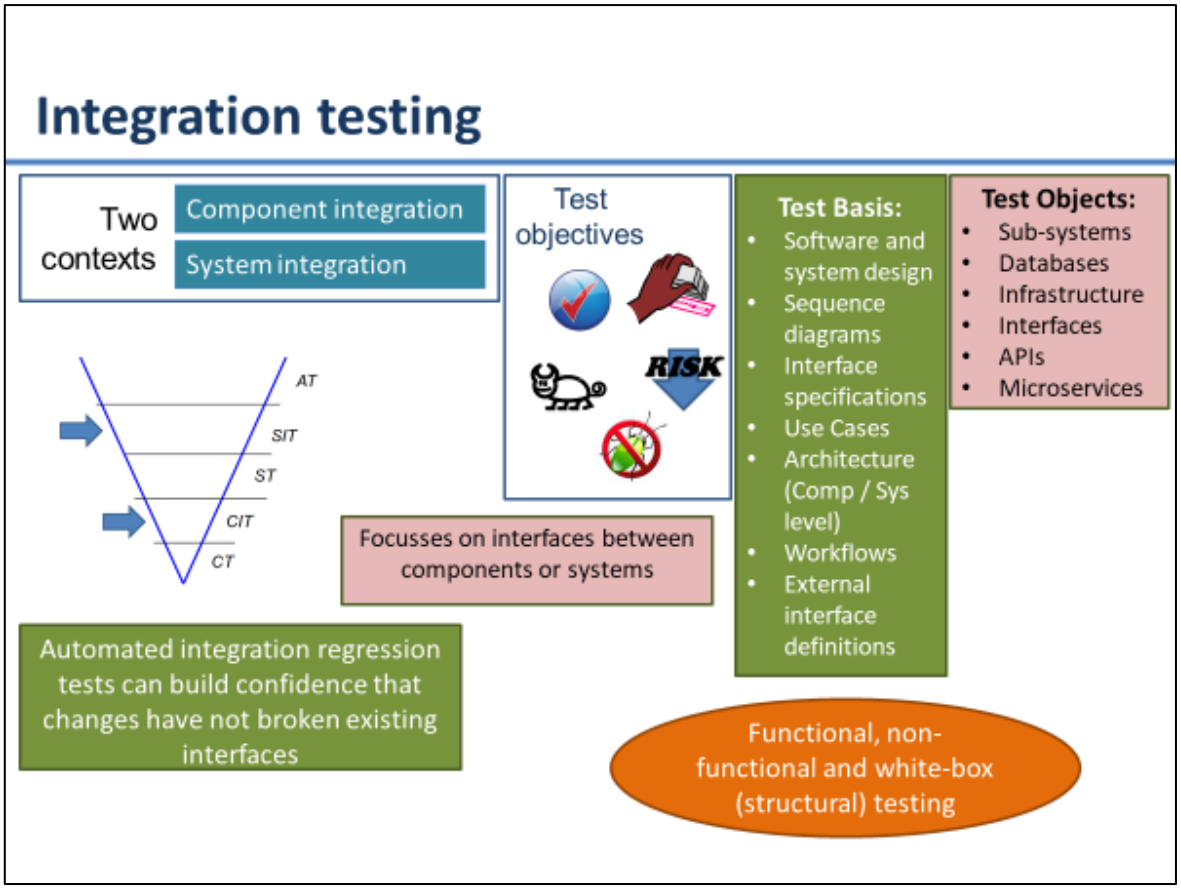
Typical defects and failures
Examples of typical defects and failures for component testing include:

- Incorrect functionality (e.g., not as described in design specifications)
- Data flow problems
- Incorrect code and logic

Defects are typically fixed as soon as they are found, often with no formal defect management. However, when developers do report defects, this provides important information for root cause analysis and process improvement.

Specific approaches and responsibilities
Component testing is usually performed by the developer who wrote the code, but it at least requires access to the code being tested. Developers may alternate component development with finding and fixing defects. Developers will often write and execute tests after having written the code for a component. However, in Agile development especially, writing automated component test cases may precede writing application code.

For example, consider test driven development (TDD). Test driven development is highly iterative and is based on cycles of developing automated test cases, then building and integrating small pieces of code, then executing the component tests, correcting any issues, and re-factoring the code. This process continues until the component has been completely built and all component tests are passing. Test driven development is an example of a test-first approach. While test driven development originated in eXtreme Programming (XP), it has spread to other forms of Agile and also to sequential lifecycles (see ISTQBCTFL-AT).



Objectives of integration testing
Integration testing focuses on interactions between components or systems. Objectives of integration testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviors of the interfaces are as designed and specified
- Building confidence in the quality of the interfaces
- Finding defects (which may be in the interfaces themselves or within the components or systems)
- Preventing defects from escaping to higher test levels

As with component testing, in some cases automated integration regression tests provide confidence that changes have not broken existing interfaces, components, or systems.

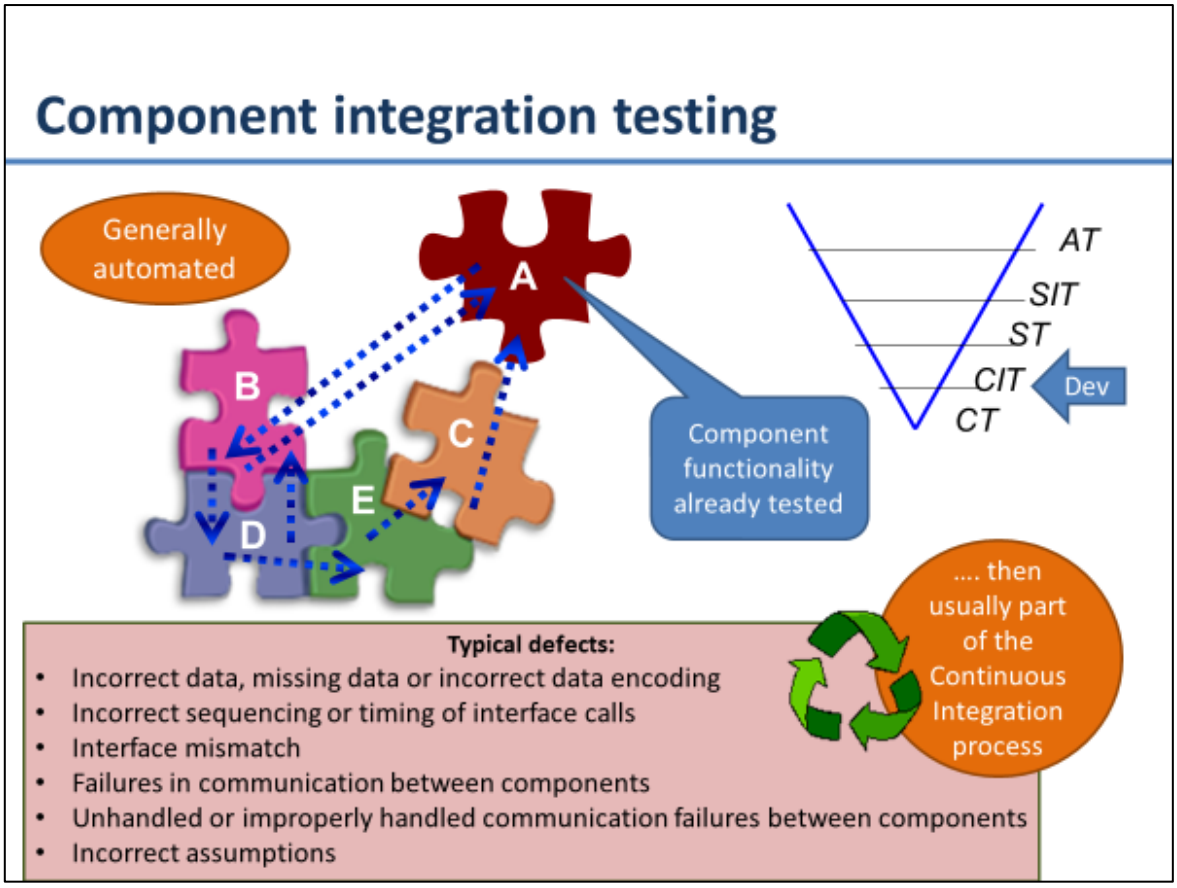
There are two different levels of integration testing described in this syllabus, which may be carried out on test objects of varying size as shown on the following slides.

Test basis
Examples of work products that can be used as a test basis for integration testing include:

- Software and system design
- Sequence diagrams
- Interface and communication protocol specifications
- Use cases
- Architecture at component or system level
- Workflows
- External interface definitions

Test objects
Typical test objects for integration testing include:

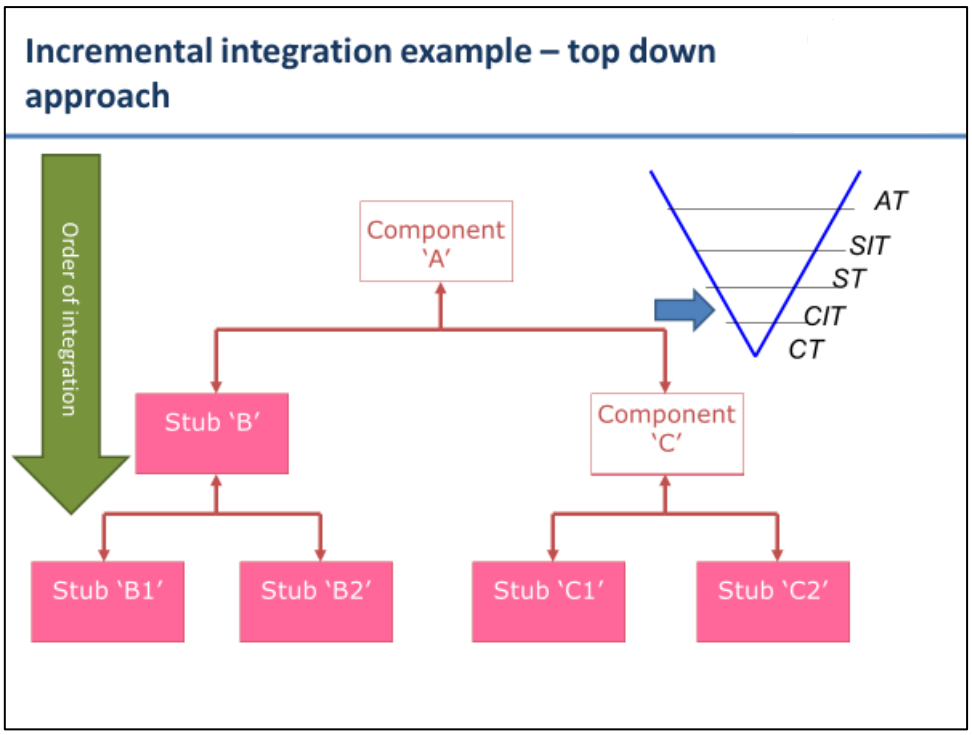
- Subsystems
- Databases
- Infrastructure
- Interfaces
- APIs
- Microservices



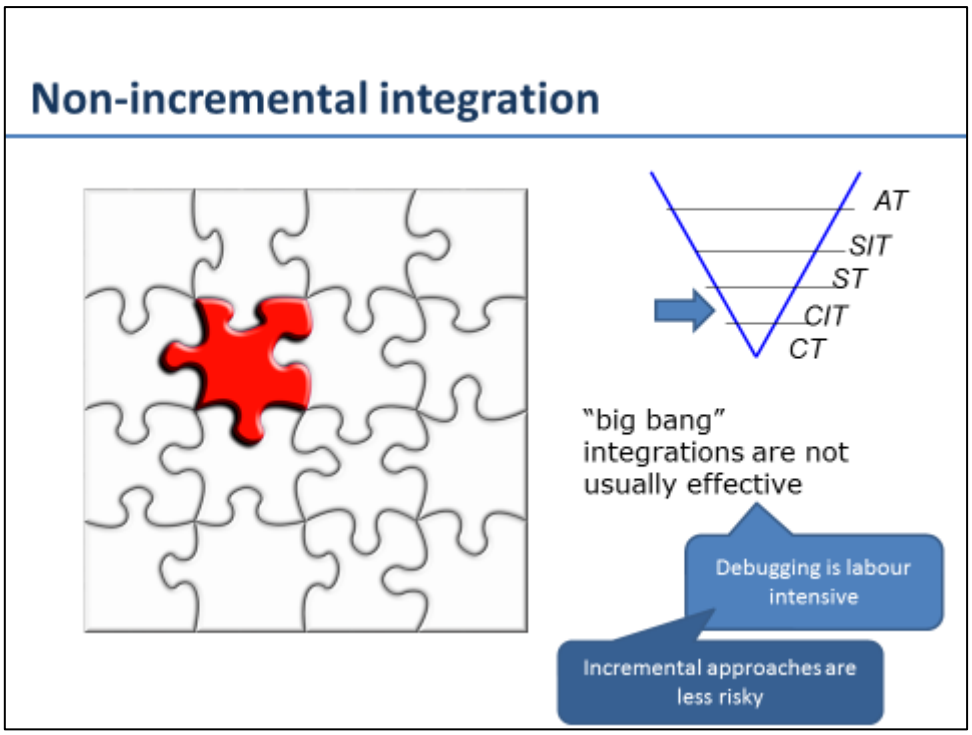
Component integration testing focuses on the interactions and interfaces between integrated components. Component integration testing is performed after component testing, and is generally automated. In iterative and incremental development, component integration tests are usually part of the continuous integration process.

Examples of typical defects and failures for component integration testing include:

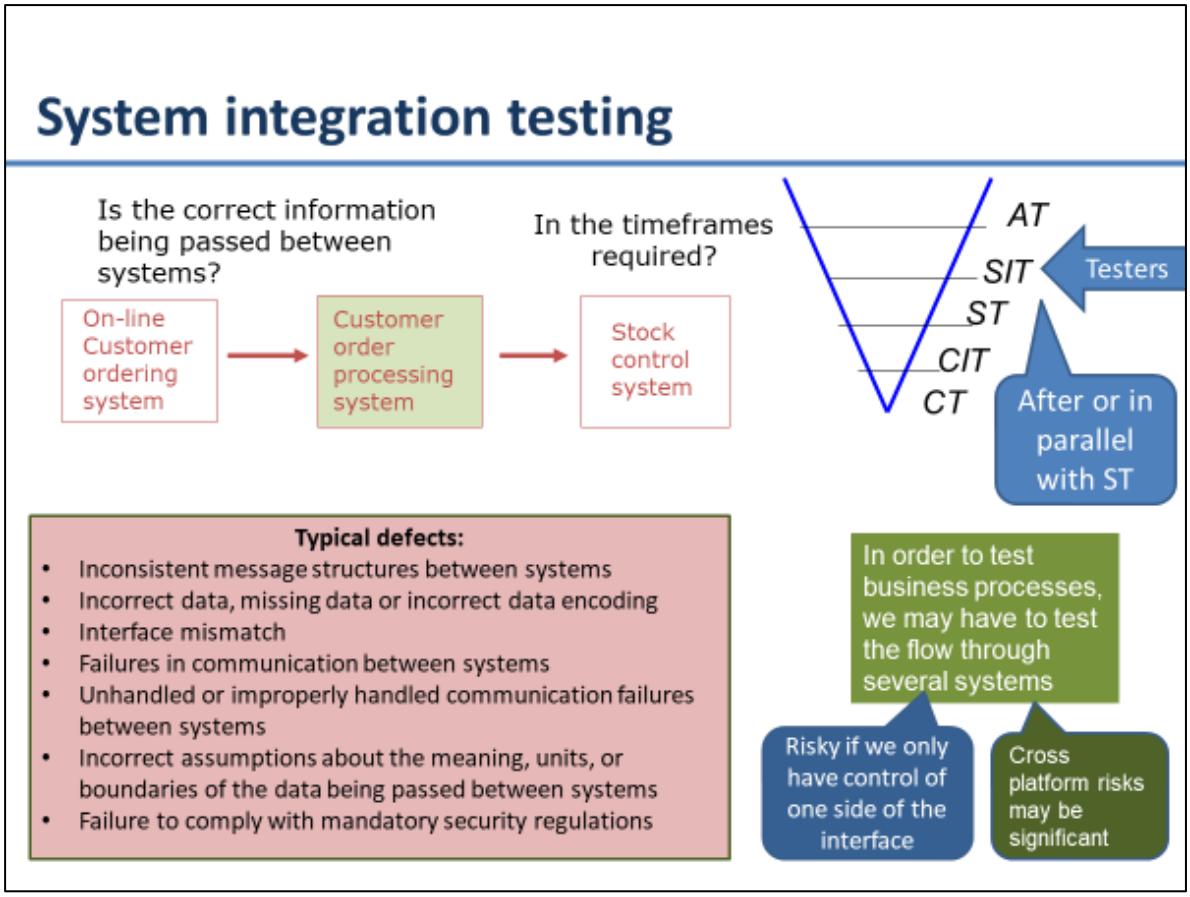
- Incorrect data, missing data, or incorrect data encoding
- Incorrect sequencing or timing of interface calls
- Interface mismatch
- Failures in communication between components
- Unhandled or improperly handled communication failures between components
- Incorrect assumptions



If integration tests and the integration strategy are planned before components or systems are built, those components or systems can be built in the order required for most efficient testing. Systematic integration strategies may be based on the system architecture (e.g., top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components.



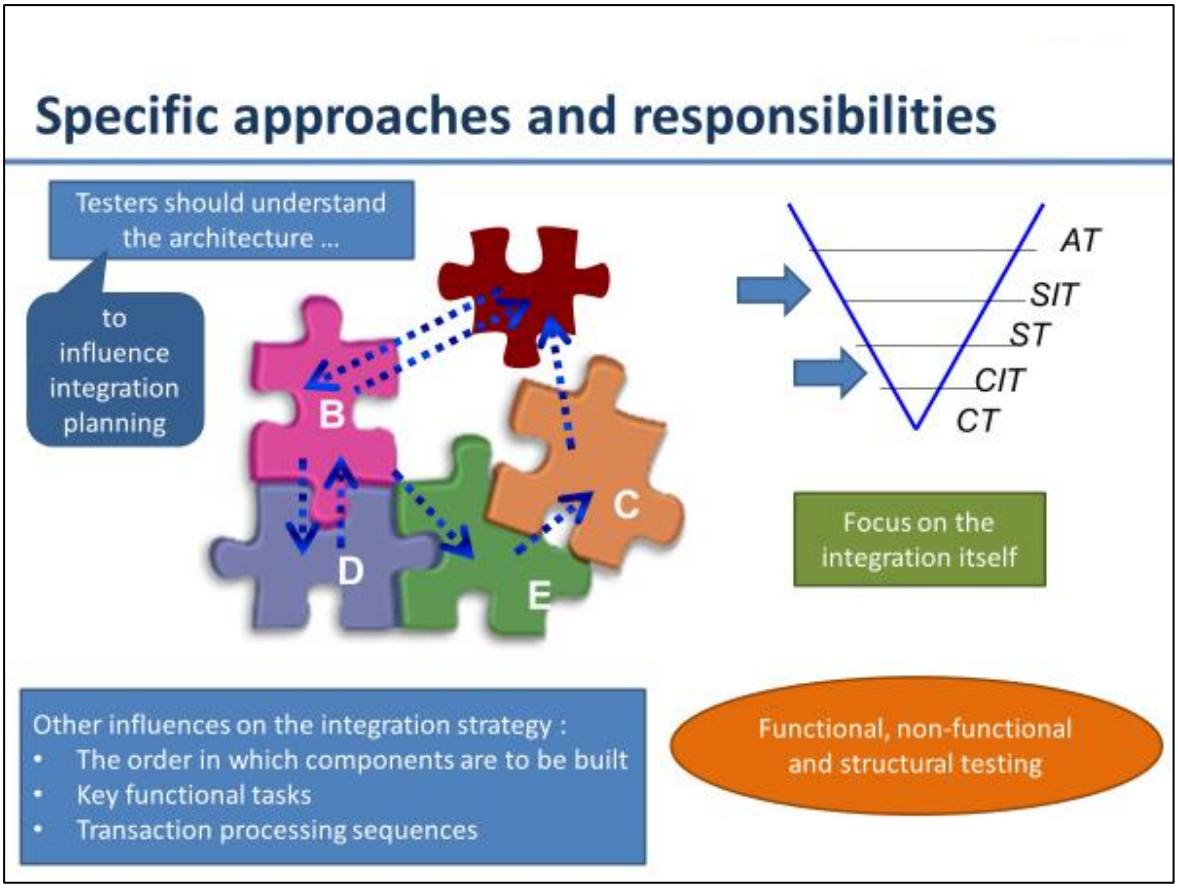
In order to simplify defect isolation and detect defects early, integration should normally be incremental (i.e., a small number of additional components or systems at a time) rather than “big bang” (i.e., integrating all components or systems in one).



System integration testing focuses on the interactions and interfaces between systems, packages, and microservices. System integration testing can also cover interactions with, and interfaces provided by, external organizations (e.g., web services). In this case, the developing organization does not control the external interfaces, which can create various challenges for testing (e.g., ensuring that test-blocking defects in the external organization's code are resolved, arranging for test environments, etc.). System integration testing may be done after system testing or in parallel with ongoing system test activities (in both sequential development and iterative and incremental development).

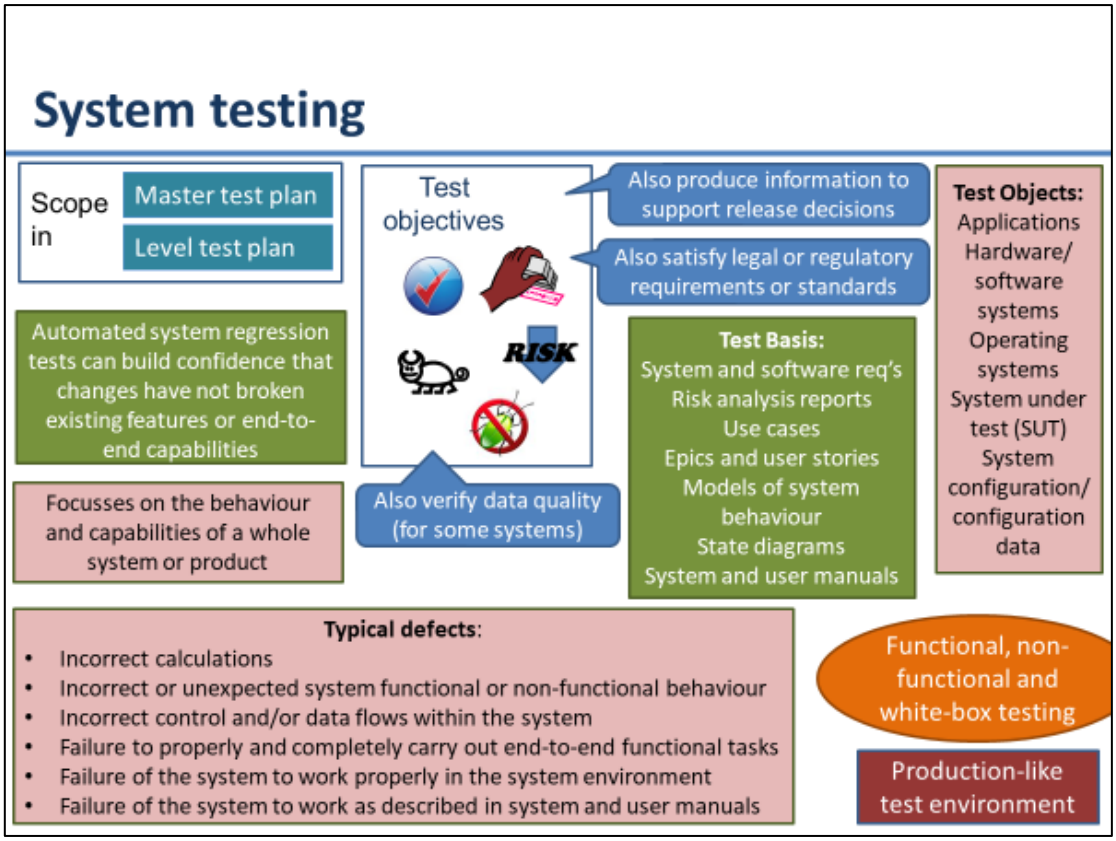
Examples of typical defects and failures for system integration testing include:

- Inconsistent message structures between systems
- Incorrect data, missing data, or incorrect data encoding
- Interface mismatch
- Failures in communication between systems
- Unhandled or improperly handled communication failures between systems
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between systems
- Failure to comply with mandatory security regulations



Component integration tests and system integration tests should concentrate on the integration itself. For example, if integrating module A with module B, tests should focus on the communication between the modules, not the functionality of the individual modules, as that should have been covered during component testing. If integrating system X with system Y, tests should focus on the communication between the systems, not the functionality of the individual systems, as that should have been covered during system testing. Functional, non-functional, and structural test types are applicable.

Component integration testing is often the responsibility of developers. System integration testing is generally the responsibility of testers. Ideally, testers performing system integration testing should understand the system architecture, and should have influenced integration planning.



System testing focuses on the behavior and capabilities of a whole system or product, often considering the end-to-end tasks the system can perform and the non-functional behaviors it exhibits while performing those tasks. Objectives of system testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviors of the system are as designed and specified
- Validating that the system is complete and will work as expected
- Building confidence in the quality of the system as a whole
- Finding defects
- Preventing defects from escaping to higher test levels or production

For certain systems, verifying data quality may also be an objective. As with component testing and integration testing, in some cases automated system regression tests provide confidence that changes have not broken existing features or end-to-end capabilities. System testing often produces information that is used by stakeholders to make release decisions. System testing may also satisfy legal or regulatory requirements or standards.

The test environment should ideally correspond to the final target or production environment.

Test basis
Examples of work products that can be used as a test basis for system testing include:

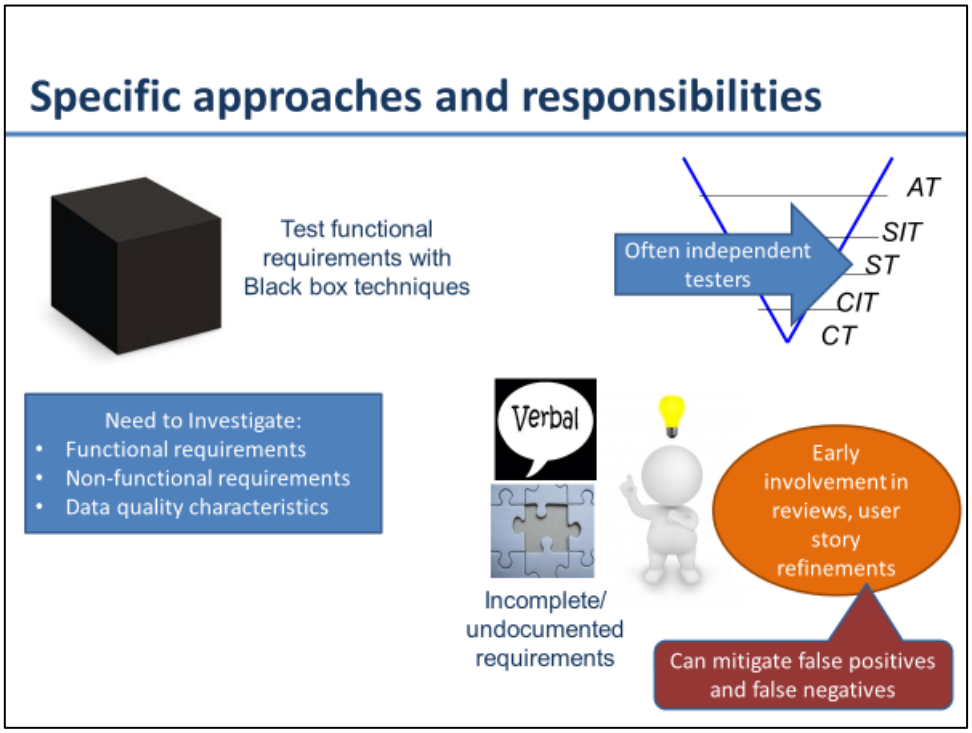
- System and software requirement specifications (functional and non-functional)
- Risk analysis reports
- Use cases
- Epics and user stories
- Models of system behavior
- State diagrams
- System and user manuals

Test objects
Typical test objects for system testing include:

- Applications
- Hardware/software systems
- Operating systems
- System under test (SUT)
- System configuration and configuration data

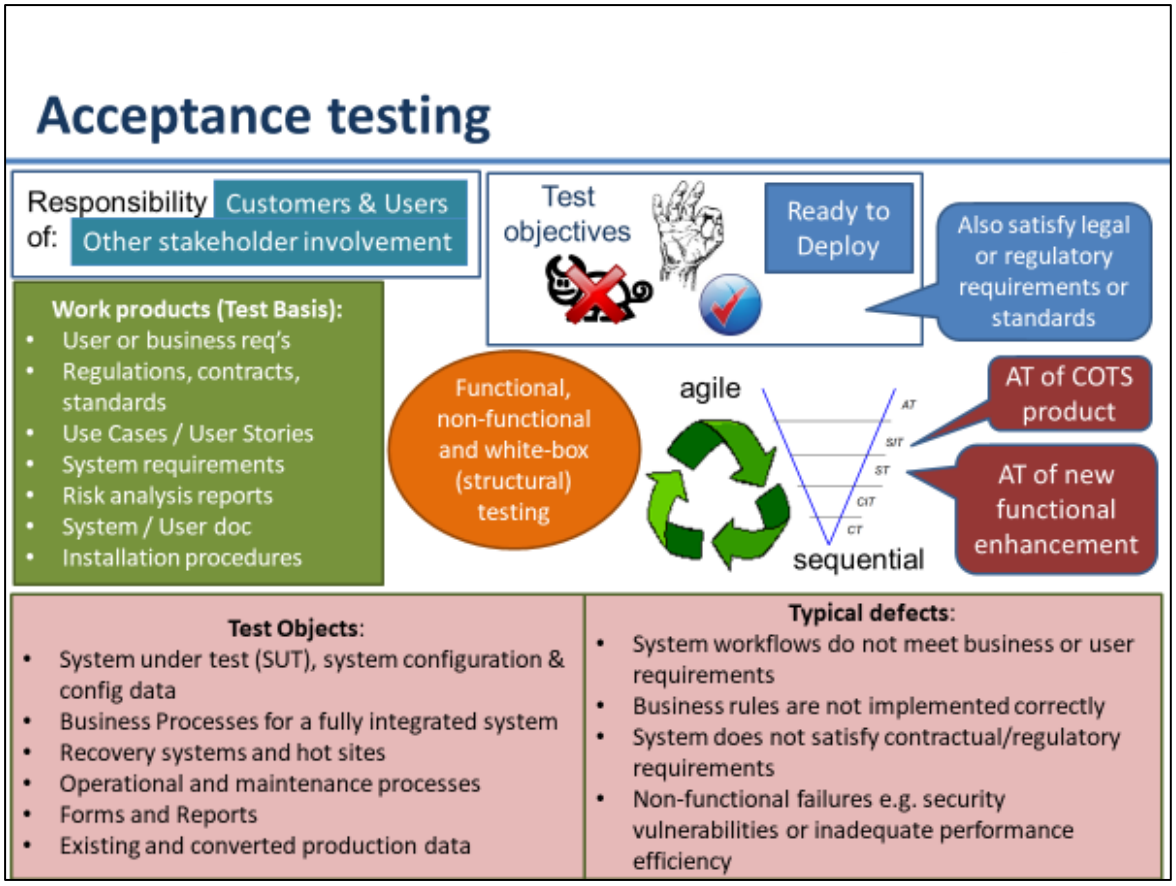
Typical defects and failures
Examples of typical defects and failures for system testing include:

- Incorrect calculations
- Incorrect or unexpected system functional or non-functional behavior
- Incorrect control and/or data flows within the system
- Failure to properly and completely carry out end-to-end functional tasks
- Failure of the system to work properly in the system environment(s)
- Failure of the system to work as described in system and user manuals



System testing should focus on the overall, end-to-end behavior of the system as a whole, both functional and non-functional. System testing should use the most appropriate techniques (see chapter 4) for the aspect(s) of the system to be tested. For example, a decision table may be created to verify whether functional behavior is as described in business rules.

System testing is typically carried out by independent testers who rely heavily on specifications. Defects in specifications (e.g., missing user stories, incorrectly stated business requirements, etc.) can lead to a lack of understanding of, or disagreements about, expected system behaviour. Such situations can cause false positives and false negatives, which waste time and reduce defect detection effectiveness, respectively. Early involvement of testers in user story refinement or static testing activities, such as reviews, helps to reduce the incidence of such situations.



Objectives of acceptance testing
Acceptance testing, like system testing, typically focuses on the behavior and capabilities of a whole system or product. Objectives of acceptance testing include:

- Establishing confidence in the quality of the system as a whole
- Validating that the system is complete and will work as expected
- Verifying that functional and non-functional behaviors of the system are as specified

Acceptance testing may produce information to assess the system's readiness for deployment and use by the customer (end-user). Defects may be found during acceptance testing, but finding defects is often not an objective, and finding a significant number of defects during acceptance testing may in some cases be considered a major project risk. Acceptance testing may also satisfy legal or regulatory requirements or standards.

Typical defects and failures
Examples of typical defects for any form of acceptance testing include:

- System workflows do not meet business or user requirements
- Business rules are not implemented correctly
- System does not satisfy contractual or regulatory requirements
- Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform,

Specific approaches and responsibilities
Acceptance testing is often the responsibility of the customers, business users, product owners, or operators of a system, and other stakeholders may be involved as well.

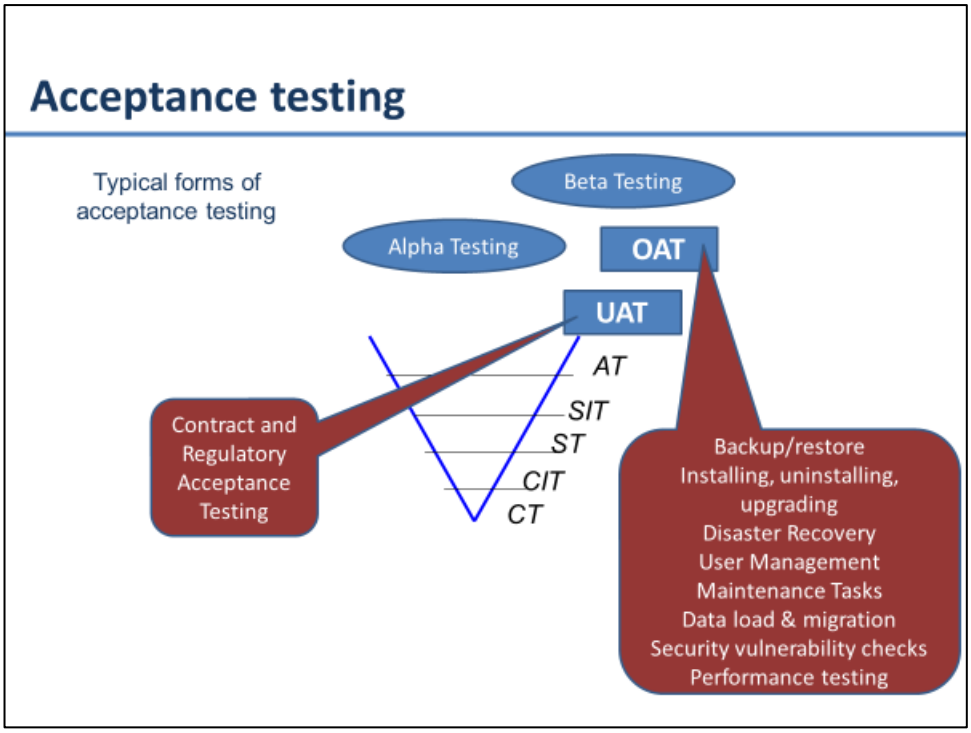
Acceptance testing is often thought of as the last test level in a sequential development lifecycle, but it may also occur at other times, for example:

- Acceptance testing of a COTS software product may occur when it is installed or integrated
- Acceptance testing of a new functional enhancement may occur before system testing

In iterative development, project teams can employ various forms of acceptance testing during and at the end of each iteration, such as those focused on verifying a new feature against its acceptance criteria and those focused on validating that a new feature satisfies the users' needs. In addition, alpha tests and beta tests may occur, either at the end of each iteration, after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contractual acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.

Test basis
Examples of work products that can be used as a test basis for any form of acceptance testing include:

- Business processes
- User or business requirements
- Regulations, legal contracts and standards
- Use cases and/or user stories
- System requirements
- System or user documentation
- Installation procedures
- Risk analysis reports



User acceptance testing (UAT)
User acceptance testing of the system is typically focused on validating the fitness for use of the system by intended users in a real or simulated operational environment. The main objective is building confidence that the users can use the system to meet their needs, fulfill requirements, and perform business processes with minimum difficulty, cost, and risk.

Operational acceptance testing (OAT)
The acceptance testing of the system by operations or systems administration staff is usually performed in a (simulated) production environment. The tests focus on operational aspects, and may include:

- Testing of backup and restore
- Installing, uninstalling and upgrading
- Disaster recovery
- User management
- Maintenance tasks
- Data load and migration tasks
- Checks for security vulnerabilities
- Performance testing

The main objective of operational acceptance testing is building confidence that the operators or system administrators can keep the system working properly for the users in the operational environment, even under exceptional or difficult conditions.

In addition to the work products described earlier for any form of acceptance testing, as a test basis for deriving test cases for operational acceptance testing, one or more of the following work products can be used:

- Backup and restore procedures
- Disaster recovery procedures
- Non-functional requirements
- Operations documentation
- Deployment and installation instructions
- Performance targets
- Database packages
- Security standards or regulations

Contractual and regulatory acceptance testing
Contractual 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. Contractual acceptance testing is often performed by users or by independent testers.

Regulatory acceptance testing is performed against any regulations that must be adhered to, such as government, legal, or safety regulations. Regulatory acceptance testing is often performed by users or by independent testers, sometimes with the results being witnessed or audited by regulatory agencies.

The main objective of contractual and regulatory acceptance testing is building confidence that contractual or regulatory compliance has been achieved.

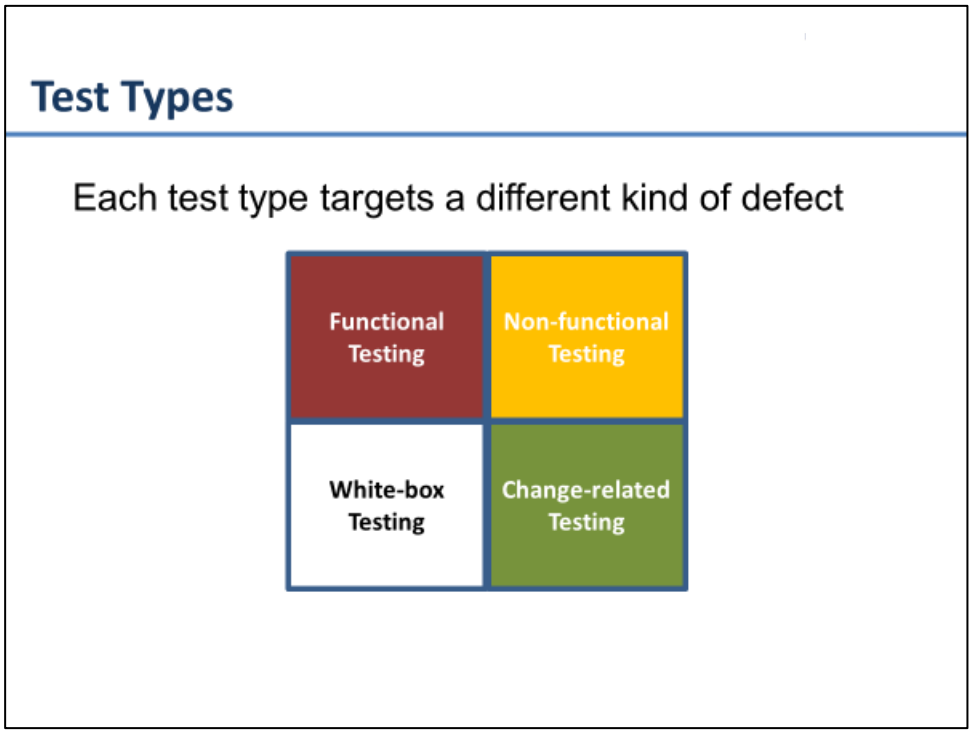
Alpha and beta testing
Alpha and beta testing are typically used by developers of commercial off-the-shelf (COTS) software who want to get feedback from potential or existing users, customers, and/or operators before the software product is put on the market. Alpha testing is performed at the developing organization's site, not by the development team, but by potential or existing customers, and/or operators or an independent test team. Beta testing is performed by potential or existing customers, and/or operators at their own locations. Beta testing may come after alpha testing, or may occur without any preceding alpha testing having occurred.

One objective of alpha and beta testing is building confidence among potential or existing customers, and/or operators that they can use the system under normal, everyday conditions, and in the operational environment(s) to achieve their objectives with minimum difficulty, cost, and risk. Another objective may be the detection of defects related to the conditions and environment(s) in which the system will be used, especially when those conditions and environment(s) are difficult to replicate by the development team.

Summary

- Each test level has its own specific test objectives
- Each test level has its own test basis and test objects
- Developers are less involved in later test levels and sometimes other stakeholders are involved in the testing
- Each test level should be looking for different types of defects and failures
- Functional and non-functional can take place at any test level
- Structural testing is focused on during component and component integration testing

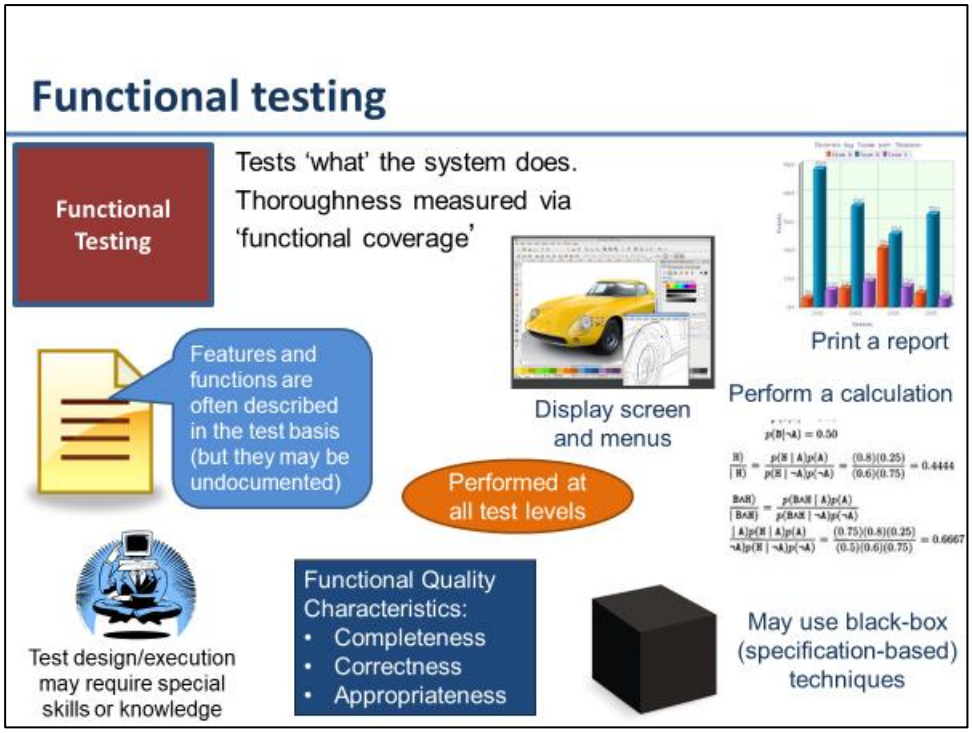
Section 2.3 - Test Types



A test type is a group of test activities aimed at testing specific characteristics of a software system, or a part of a system, based on specific test objectives. Such objectives may include:

- Evaluating functional quality characteristics, such as completeness, correctness, and appropriateness
- Evaluating non-functional quality characteristics, such as reliability, performance efficiency, security, compatibility, and usability
- Evaluating whether the structure or architecture of the component or system is correct, complete, and as specified
- Evaluating the effects of changes, such as confirming that defects have been fixed (confirmation testing) and looking for unintended changes in behavior resulting from software or environment changes (regression testing)

It is possible to perform any of the test types mentioned above at any test level. To illustrate, examples of functional, non-functional, white-box, and change-related tests will be given across all test levels in the following slides. While this section provides examples of every test type across every level, it is not necessary, for all software, to have every test type represented across every level. However, it is important to run applicable test types at each level, especially the earliest level where the test type occurs.



Functional testing of a system involves tests that evaluate functions that the system should perform. Functional requirements may be described in work products such as business requirements specifications, epics, user stories, use cases, or functional specifications, or they may be undocumented. The functions are "what" the system should do.

Functional tests should be performed at all test levels (e.g., tests for components may be based on a component specification), though the focus is different at each level (see section 2.2).

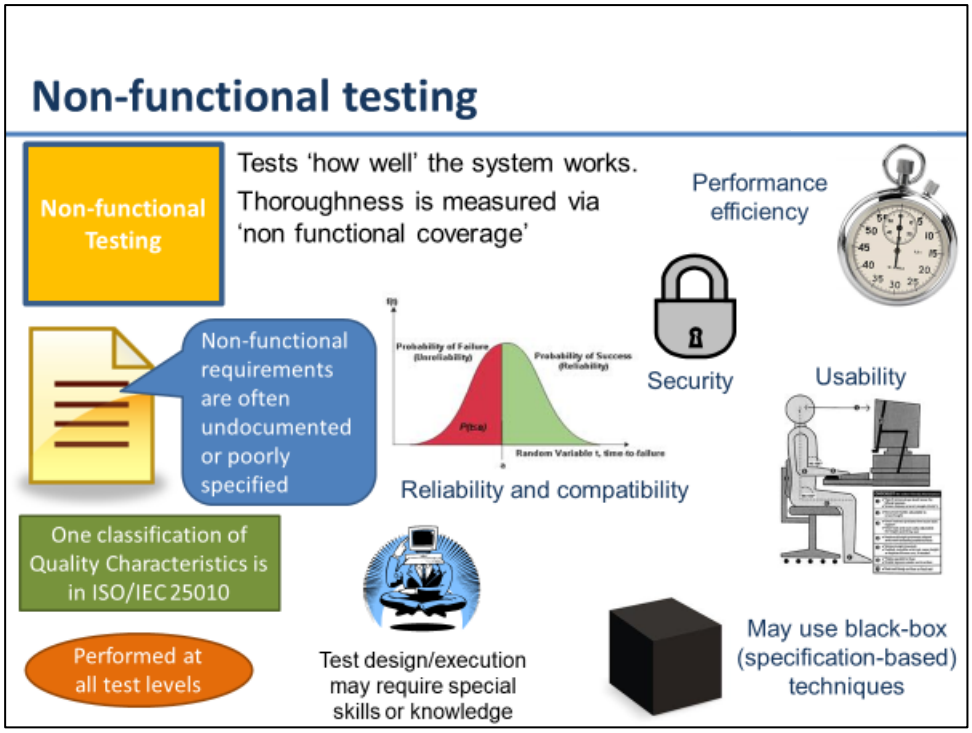
Functional testing considers the behavior of the software, so black-box techniques may be used to derive test conditions and test cases for the functionality of the component or system (see section 4.2).

The thoroughness of functional testing can be measured through functional coverage. Functional coverage is the extent to which some functionality has been exercised by tests, and is expressed as a percentage of the type(s) of element being covered. For example, using traceability between tests and functional requirements, the percentage of these requirements which are addressed by testing can be calculated, potentially identifying coverage gaps.

Functional test design and execution may involve special skills or knowledge, such as knowledge of the particular business problem the software solves (e.g., geological modelling software for the oil and gas industries).

The following are examples of functional tests for a banking application:

- For component testing, tests are designed based on how a component should calculate compound interest.
- For component integration testing, tests are designed based on how account information captured at the user interface is passed to the business logic.
- For system testing, tests are designed based on how account holders can apply for a line of credit on their checking accounts.
- For system integration testing, tests are designed based on how the system uses an external microservice to check an account holder's credit score.
- For acceptance testing, tests are designed based on how the banker handles approving or declining a credit application.



Non-functional testing of a system evaluates characteristics of systems and software such as usability, performance efficiency or security. Refer to ISO standard (ISO/IEC 25010) for a classification of software product quality characteristics. Non-functional testing is the testing of "how well" the system behaves.

Contrary to common misperceptions, non-functional testing can and often should be performed at all test levels, and done as early as possible. The late discovery of non-functional defects can be extremely dangerous to the success of a project.

Black-box techniques (see section 4.2) may be used to derive test conditions and test cases for non-functional testing. For example, boundary value analysis can be used to define the stress conditions for performance tests.

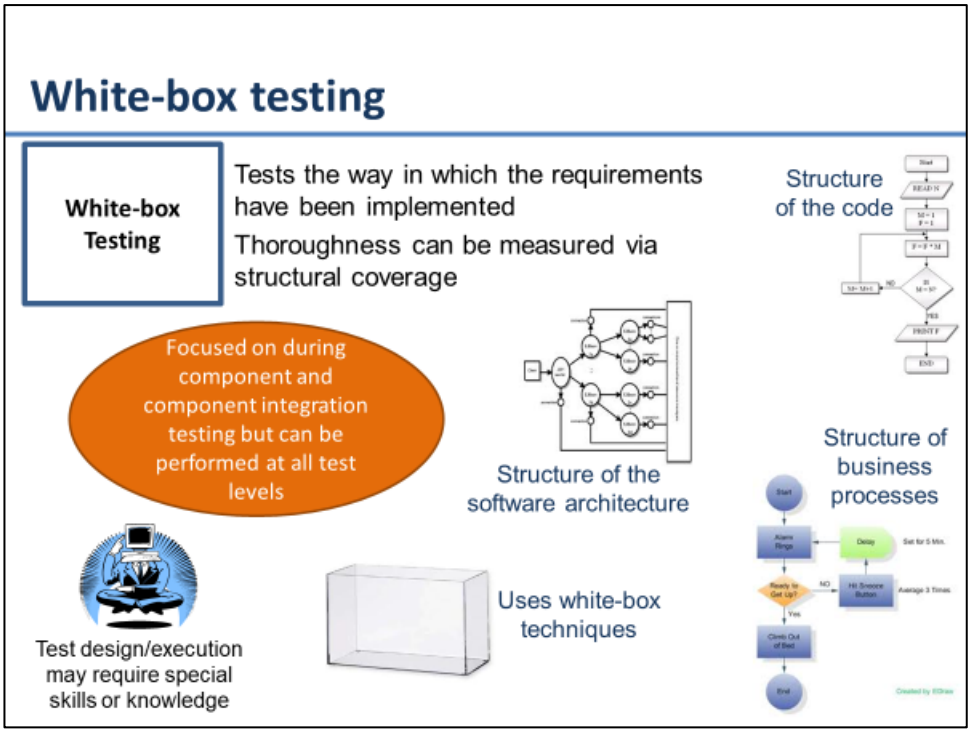
The thoroughness of non-functional testing can be measured through non-functional coverage. Non-functional coverage is the extent to which some type of non-functional element has been exercised by tests, and is expressed as a percentage of the type(s) of element being covered. For example, using traceability between tests and supported devices for a mobile application, the percentage of devices which are addressed by compatibility testing can be calculated, potentially identifying coverage gaps.

Non-functional test design and execution may involve special skills or knowledge, such as knowledge of the inherent weaknesses of a design or technology (e.g., security vulnerabilities associated with particular programming languages) or the particular user base (e.g., the personas of users of healthcare facility management systems).

Refer to ISTQB-CTAL-TA, ISTQB-CTAL-TTA, ISTQB-CTAL-SEC, and other ISTQB® specialist modules for more details regarding the testing of non-functional quality characteristics.

The following are examples of non-functional tests for a banking application:

- For component testing, performance tests are designed to evaluate the number of CPU cycles required to perform a complex total interest calculation.
- For component integration testing, security tests are designed for buffer overflow vulnerabilities due to data passed from the user interface to the business logic.
- For system testing, portability tests are designed to check whether the presentation layer works on all supported browsers and mobile devices.
- For system integration testing, reliability tests are designed to evaluate system robustness if the credit score microservice fails to respond.
- For acceptance testing, usability tests are designed to evaluate the accessibility of the banker's credit processing interface for people with disabilities.



White-box testing derives tests based on the system's internal structure or implementation. Internal structure may include code, architecture, work flows, and/or data flows within the system (see section 4.3).

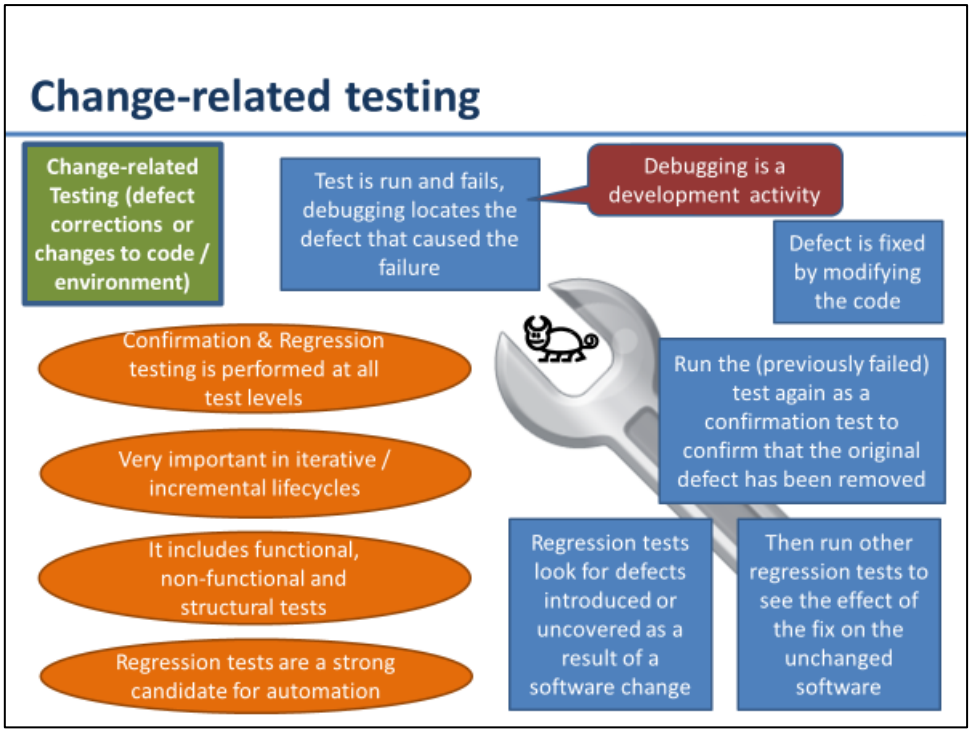
The thoroughness of white-box testing can be measured through structural coverage. Structural coverage is the extent to which some type of structural element has been exercised by tests, and is expressed as a percentage of the type of element being covered.

At the component testing level, code coverage is based on the percentage of component code that has been tested, and may be measured in terms of different aspects of code (coverage items) such as the percentage of executable statements tested in the component, or the percentage of decision outcomes tested. These types of coverage are collectively called code coverage. At the component integration testing level, white-box testing may be based on the architecture of the system, such as interfaces between components, and structural coverage may be measured in terms of the percentage of interfaces exercised by tests.

White-box test design and execution may involve special skills or knowledge, such as the way the code is built, how data is stored (e.g., to evaluate possible database queries), and how to use coverage tools and to correctly interpret their results.

The following are examples of structural tests for a banking application:

- For component testing, tests are designed to achieve complete statement and decision coverage (see section 4.3) for all components that perform financial calculations.
- For component integration testing, tests are designed to exercise how each screen in the browser interface passes data to the next screen and to the business logic.
- For system testing, tests are designed to cover sequences of web pages that can occur during a credit line application.
- For system integration testing, tests are designed to exercise all possible inquiry types sent to the credit score microservice.
- For acceptance testing, tests are designed to cover all supported financial data file structures and value ranges for bank-to-bank transfers



When changes are made to a system, either to correct a defect or because of new or changing functionality, testing should be done to confirm that the changes have corrected the defect or implemented the functionality correctly, and have not caused any unforeseen adverse consequences.

- Confirmation testing: After a defect is fixed, the software may be tested with all test cases that failed due to the defect, which should be re-executed on the new software version. The software may also be tested with new tests to cover changes needed to fix the defects. At the very least, the steps to reproduce the failure(s) caused by the defect must be re-executed on the new software version. The purpose of a confirmation test is to confirm whether the original defect has been successfully fixed.

- Regression testing: It is possible that a change made in one part of the code, whether a fix or another type of change, may accidentally affect the behavior of other parts of the code, whether within the same component, in other components of the same system, or even in other systems. Changes may include changes to the environment, such as a new version of an operating system or database management system. Such unintended side-effects are called regressions. Regression testing involves running tests to detect such unintended side-effects.

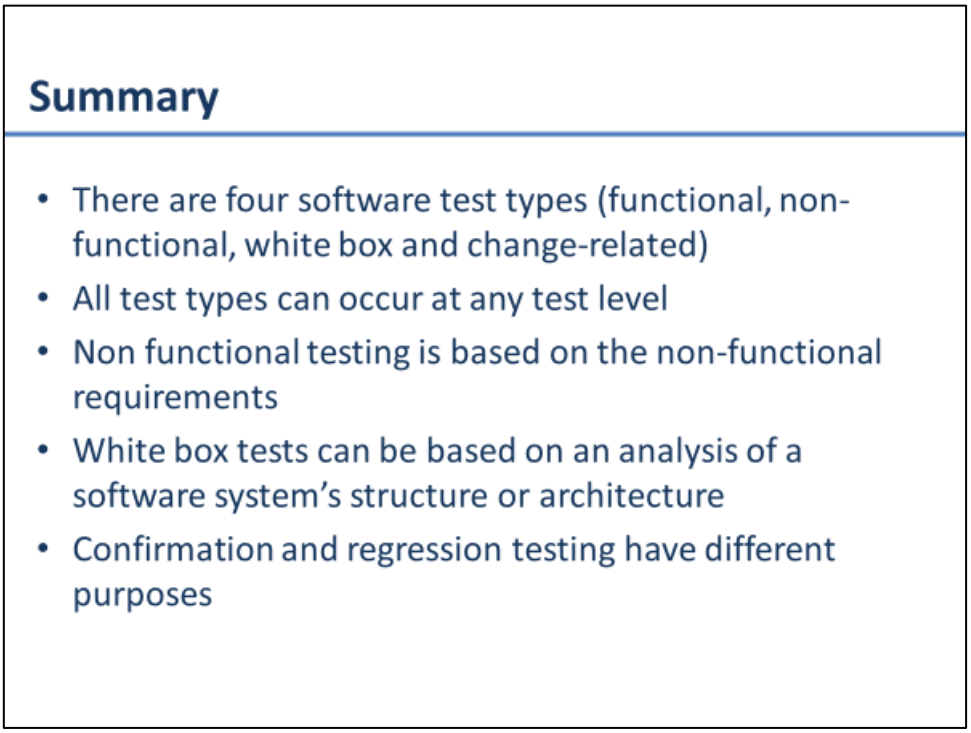
Confirmation testing and regression testing are performed at all test levels.

Especially in iterative and incremental development lifecycles (e.g., Agile), new features, changes to existing features, and code refactoring result in frequent changes to the code, which also requires change-related testing. Due to the evolving nature of the system, confirmation and regression testing are very important. This is particularly relevant for Internet of Things systems where individual objects (e.g., devices) are frequently updated or replaced.

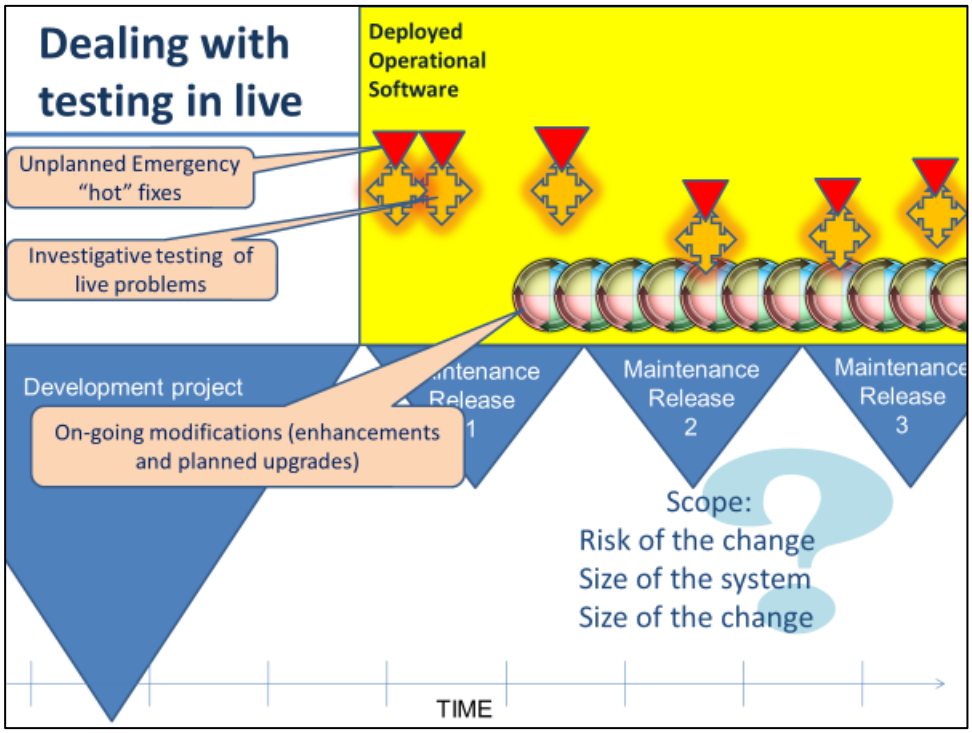
Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation. Automation of these tests should start early in the project (see chapter 6).

The following are examples of change-related tests for a banking application:

- For component testing, automated regression tests are built for each component and included within the continuous integration framework.
- For component integration testing, tests are designed to confirm fixes to interface-related defects as the fixes are checked into the code repository.
- For system testing, all tests for a given workflow are re-executed if any screen on that workflow changes.
- For system integration testing, tests of the application interacting with the credit scoring microservice are re-executed daily as part of continuous deployment of that microservice.
- For acceptance testing, all previously-failed tests are re-executed after a defect found in acceptance testing is fixed.



Section 2.4 - Maintenance Testing

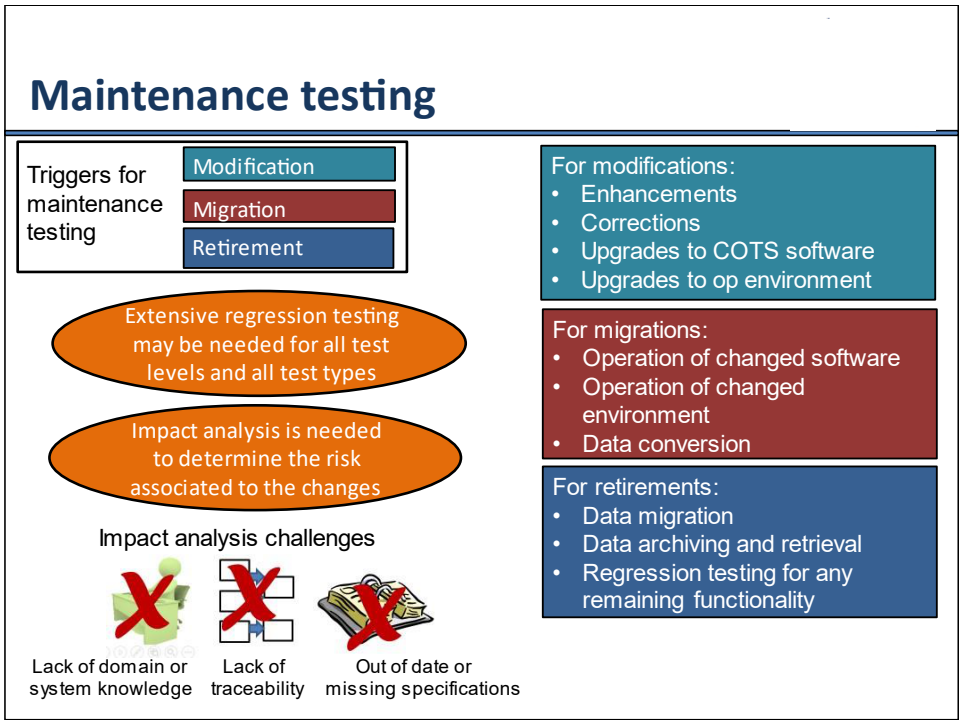


Once deployed to production environments, software and systems need to be maintained. Changes of various sorts are almost inevitable in delivered software and systems, either to fix defects discovered in operational use, to add new functionality, or to delete or alter already-delivered functionality. Maintenance is also needed to preserve or improve non-functional quality characteristics of the component or system over its lifetime, especially performance efficiency, reliability, security, and portability.

When any changes are made as part of maintenance, maintenance testing should be performed, both to evaluate the success with which the changes were made and to check for possible side-effects (e.g., regressions) in parts of the system that remain unchanged (which is usually most of the system). Maintenance can involve planned releases and unplanned releases (hot fixes).

A maintenance release may require maintenance testing at multiple test levels, using various test types, based on its scope. The scope of maintenance testing depends on:

- The degree of risk of the change, for example, the degree to which the changed area of software communicates with other components or systems
- The size of the existing system
- The size of the change



There are several reasons why software maintenance, and thus maintenance testing, takes place, both for planned and unplanned changes.

We can classify the triggers for maintenance as follows:

- Modification, such as planned enhancements (e.g., release-based), corrective and emergency changes, changes of the operational environment (such as planned operating system or database upgrades), upgrades of COTS software, and patches for defects and vulnerabilities
- Migration, such as from one platform to another, which can require operational tests of the new environment as well as of the changed software, or tests of data conversion when data from another application will be migrated into the system being maintained

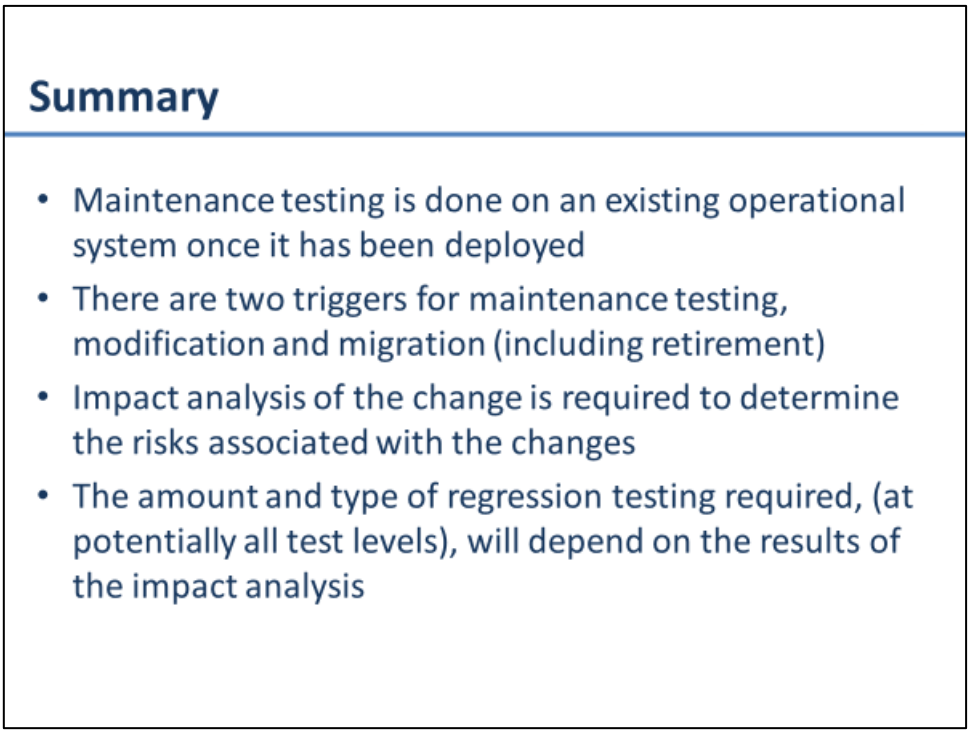
- Retirement, such as when an application reaches the end of its life. When an application or system is retired, this can require testing of data migration or archiving if long data retention periods are required. Testing restore/retrieve procedures after archiving for long retention periods may also be needed. Regression testing may be needed to ensure that any functionality that remains in service still works

For Internet of Things systems, maintenance testing may be triggered by the introduction of completely new or modified things, such as hardware devices and software services, into the overall system. The maintenance testing for such systems places particular emphasis on integration testing at different levels (e.g., network level, application level) and on security aspects, in particular those relating to personal data.

Impact analysis evaluates the changes that were made for a maintenance release to identify the intended consequences as well as expected and possible side effects of a change, and to identify the areas in the system that will be affected by the change. Impact analysis can also help to identify the impact of a change on existing tests. The side effects and affected areas in the system need to be tested for regressions, possibly after updating any existing tests affected by the change.

Impact analysis may be done before a change is made, to help decide if the change should be made, based on the potential consequences in other areas of the system. Impact analysis can be difficult if:

- Specifications (e.g., business requirements, user stories, architecture) are out of date or missing
- Test cases are not documented or are out of date
- Bi-directional traceability between tests and the test basis has not been maintained
- Tool support is weak or non-existent
- The people involved do not have domain and/or system knowledge
- Insufficient attention has been paid to the software's maintainability during development



End of section exercise

- Attempt the following Questions from Sample Exam B
- Questions 9 to 13
- Please mark your own (See Sample Exam B Answers) and read the answer justifications as needed

You have 7.5 minutes

Learning Objectives for Testing Throughout the Software Lifecycle

In order to complete this section, you should satisfy yourself that you are able to understand or perform the following items to the standard indicated by the corresponding “K - learning level”. For an explanation of “K levels”, please see the course introduction section – “Learning Objectives and Levels of Knowledge”.

Keywords

acceptance testing, alpha testing, beta testing, change-related testing, commercial off-the-shelf (COTS), component integration testing, component testing, confirmation testing, contractual acceptance testing, functional testing, impact analysis, integration testing, maintenance testing, non-functional testing, operational acceptance testing, regression testing, regulatory acceptance testing, sequential development model, system integration testing, system testing, test basis, test case, test environment, test level, test object, test objective, test type, user acceptance testing, white-box testing

Learning Objectives for Testing Throughout the Software Development Lifecycle

2.1 Software Development Lifecycle Models

FL-2.1.1 (K2) Explain the relationships between software development activities and test activities in the software development lifecycle

FL-2.1.2 (K1) Identify reasons why software development lifecycle models must be adapted to the context of project and product characteristics

2.2 Test Levels

FL-2.2.1 (K2) Compare the different test levels from the perspective of objectives, test basis, test objects, typical defects and failures, and approaches and responsibilities

2.3 Test Types

FL-2.3.1 (K2) Compare functional, non-functional, and white-box testing

FL-2.3.2 (K1) Recognize that functional, non-functional, and white-box tests occur at any test level

FL-2.3.3 (K2) Compare the purposes of confirmation testing and regression testing

2.4 Maintenance Testing

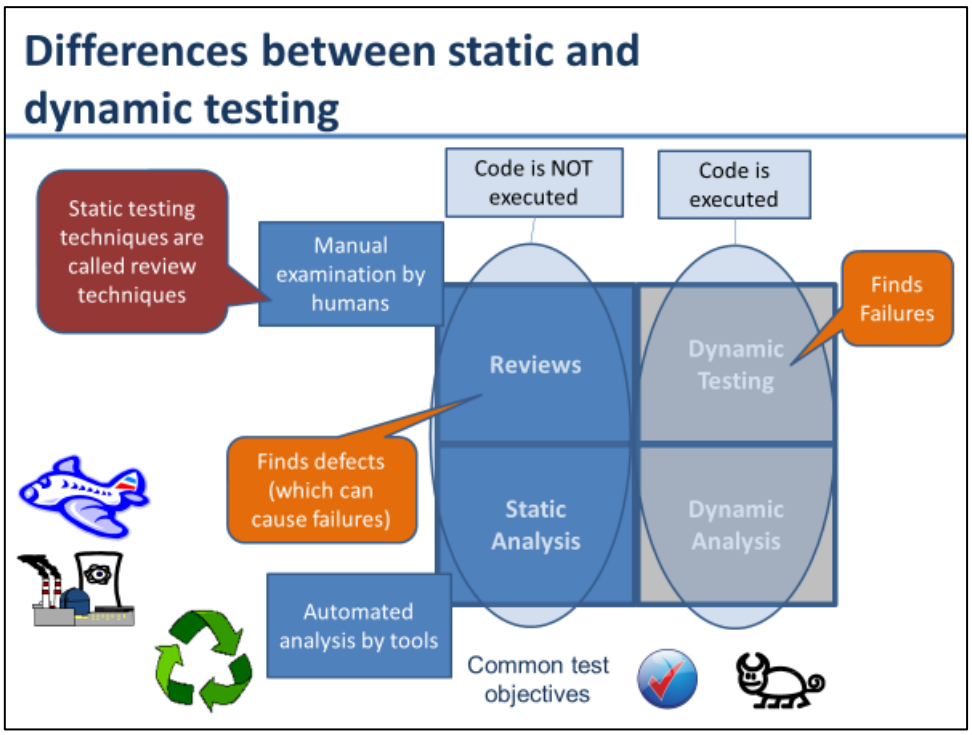
FL-2.4.1 (K2) Summarize triggers for maintenance testing

FL-2.4.2 (K2) Describe the role of impact analysis in maintenance testing

Static Techniques

Table of Contents

Section 3.1 - Static Testing basics



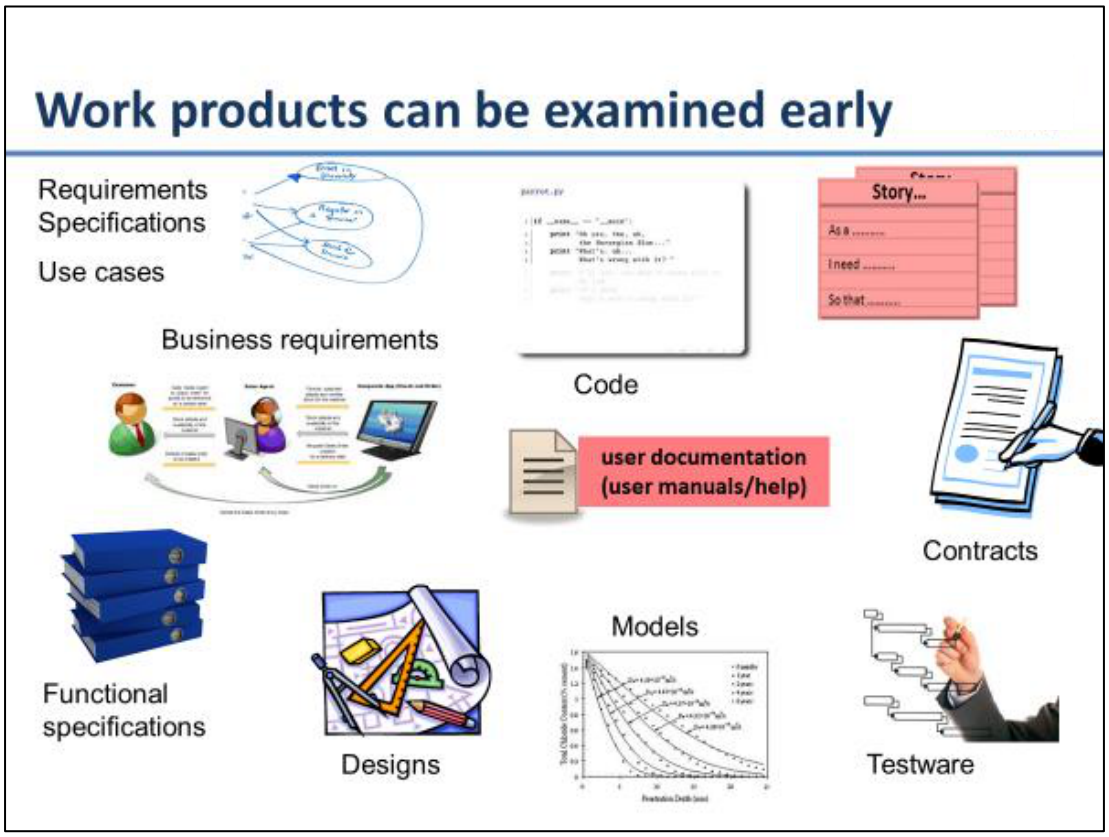
In contrast to dynamic testing, which requires the execution of the software being tested, static testing relies on the manual examination of work products (i.e., reviews) or tool-driven evaluation of the code or other work products (i.e., static analysis). Both types of static testing assess the code or other work product being tested without actually executing the code or work product being tested.

Static analysis is important for safety-critical computer systems (e.g., aviation, medical, or nuclear software), but static analysis has also become important and common in other settings. For example, static analysis is an important part of security testing. Static analysis is also often incorporated into automated software build and distribution tools, for example in Agile development, continuous delivery and continuous deployment.

Static testing and dynamic testing can have the same objectives (see section 1.1.1), such as providing an assessment of the quality of the work products and identifying defects as early as possible. Static and dynamic testing complement each other by finding different types of defects.

One main distinction is that static testing finds defects in work products directly rather than identifying failures caused by defects when the software is run. A defect can reside in a work product for a very long time without causing a failure. The path where the defect lies may be rarely exercised or hard to reach, so it will not be easy to construct and execute a dynamic test that encounters it. Static testing may be able to find the defect with much less effort.

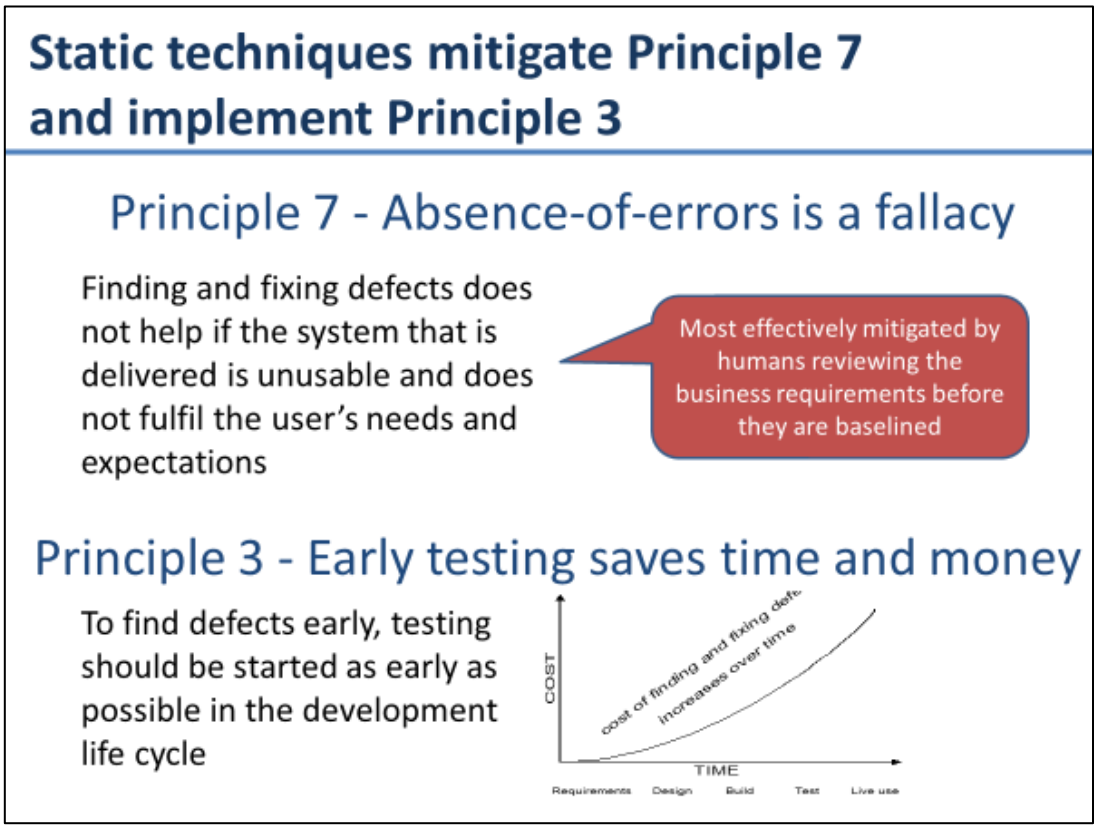
Another distinction is that static testing can be used to improve the consistency and internal quality of work products, while dynamic testing typically focuses on externally visible behaviors.



Almost any work product can be examined using static testing (reviews and/or static analysis), for example:

- Specifications, including business requirements, functional requirements, and security requirements
- Epics, user stories, and acceptance criteria
- Architecture and design specifications
- Code
- Testware, including test plans, test cases, test procedures, and automated test scripts
- User guides
- Web pages
- Contracts, project plans, schedules, and budget planning
- Configuration set up and infrastructure set up
- Models, such as activity diagrams, which may be used for Model-Based testing (see ISTQBCTFL-MBT and Kramer 2016)

Reviews can be applied to any work product that the participants know how to read and understand. Static analysis can be applied efficiently to any work product with a formal structure (typically code or models) for which an appropriate static analysis tool exists. Static analysis can even be applied with tools that evaluate work products written in natural language such as requirements (e.g., checking for spelling, grammar, and readability).



Static testing techniques provide a variety of benefits. When applied early in the software development lifecycle, static testing enables the early detection of defects before dynamic testing is performed (e.g., in requirements or design specifications reviews, backlog refinement, etc.). Defects found early are often much cheaper to remove than defects found later in the lifecycle, especially compared to defects found after the software is deployed and in active use.

Using static testing techniques to find defects and then fixing those defects promptly is almost always much cheaper for the organization than using dynamic testing to find defects in the test object and then fixing them, especially when considering the additional costs associated with updating other work products and performing confirmation and regression testing.

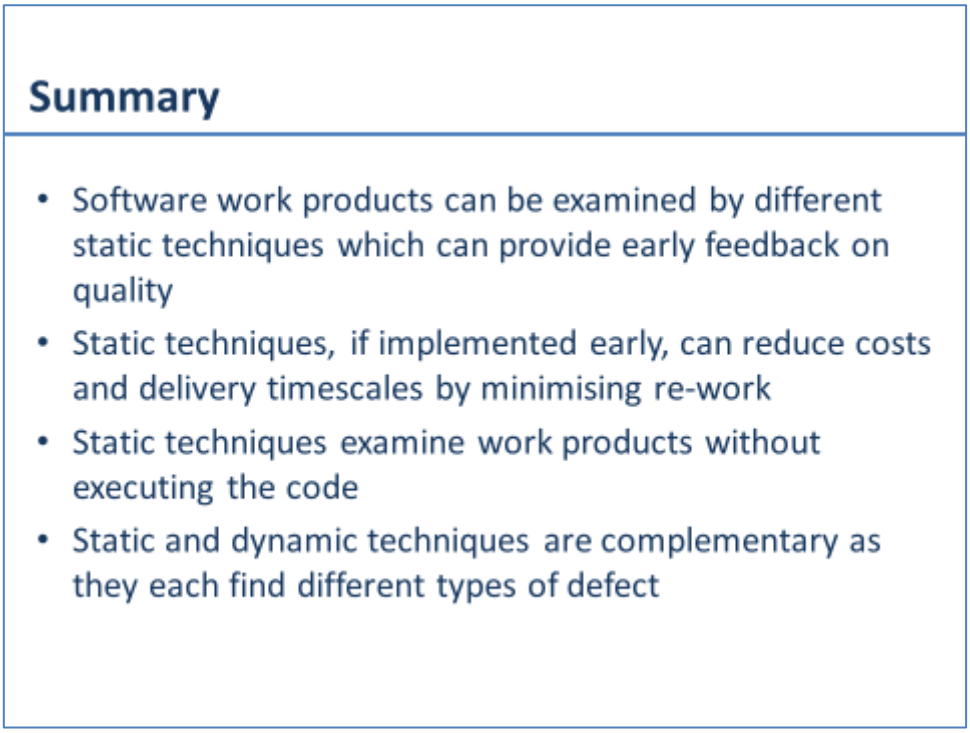
Additional benefits of static testing may include:

- Detecting and correcting defects more efficiently, and prior to dynamic test execution
- Identifying defects which are not easily found by dynamic testing
- Preventing defects in design or coding by uncovering inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies in requirements
- Increasing development productivity (e.g., due to improved design, more maintainable code)
- Reducing development cost and time
- Reducing testing cost and time
- Reducing total cost of quality over the software's lifetime, due to fewer failures later in the lifecycle or after delivery into operation
- Improving communication between team members in the course of participating in reviews

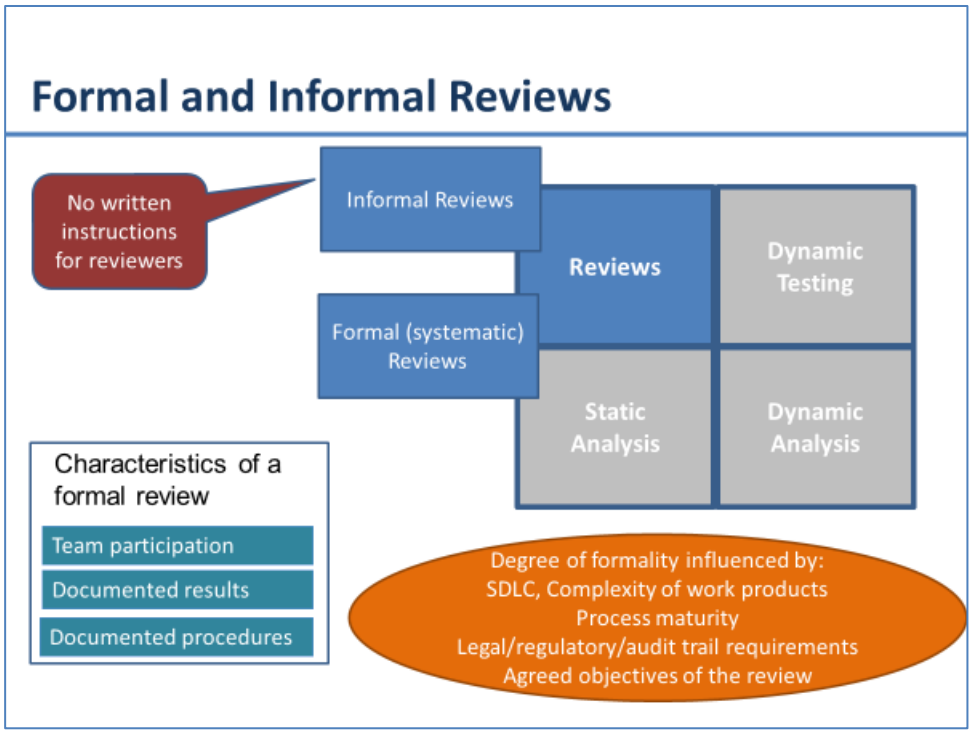
Compared with dynamic testing, typical defects that are easier and cheaper to find and fix through static testing include:

- Requirement defects (e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies)
- Design defects (e.g., inefficient algorithms or database structures, high coupling, low cohesion)
- Coding defects (e.g., variables with undefined values, variables that are declared but never used, unreachable code, duplicate code)
- Deviations from standards (e.g., lack of adherence to coding standards)
- Incorrect interface specifications (e.g., different units of measurement used by the calling system than by the called system)
- Security vulnerabilities (e.g., susceptibility to buffer overflows)
- Gaps or inaccuracies in test basis traceability or coverage (e.g., missing tests for an acceptance criterion)

Moreover, most types of maintainability defects can only be found by static testing (e.g., improper modularization, poor reusability of components, code that is difficult to analyze and modify without introducing new defects).



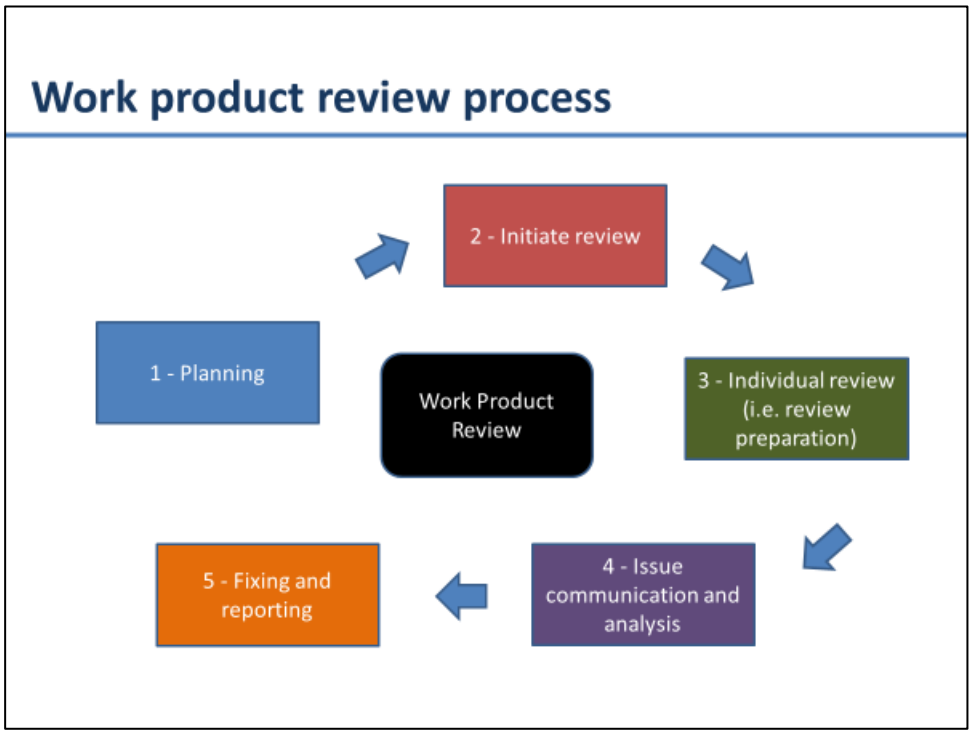
Section 3.2 - Review Process



Reviews vary from informal to formal. Informal reviews are characterized by not following a defined process and not having formal documented output. Formal reviews are characterized by team participation, documented results of the review, and documented procedures for conducting the review. The formality of a review process is related to factors such as the software development lifecycle model, the maturity of the development process, the complexity of the work product to be reviewed, any legal or regulatory requirements, and/or the need for an audit trail.

The focus of a review depends on the agreed objectives of the review (e.g., finding defects, gaining understanding, educating participants such as testers and new team members, or discussing and deciding by consensus).

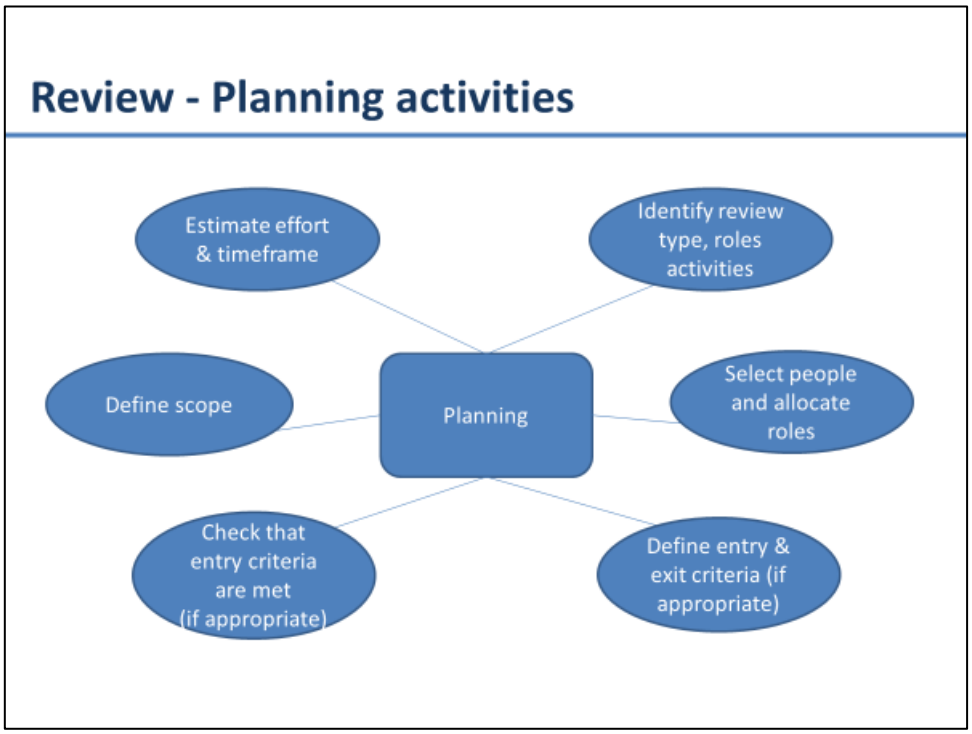
ISO standard (ISO/IEC 20246) contains more in-depth descriptions of the review process for work products, including roles and review techniques.



The review process comprises the following main activities (covered in detail in the following slides):

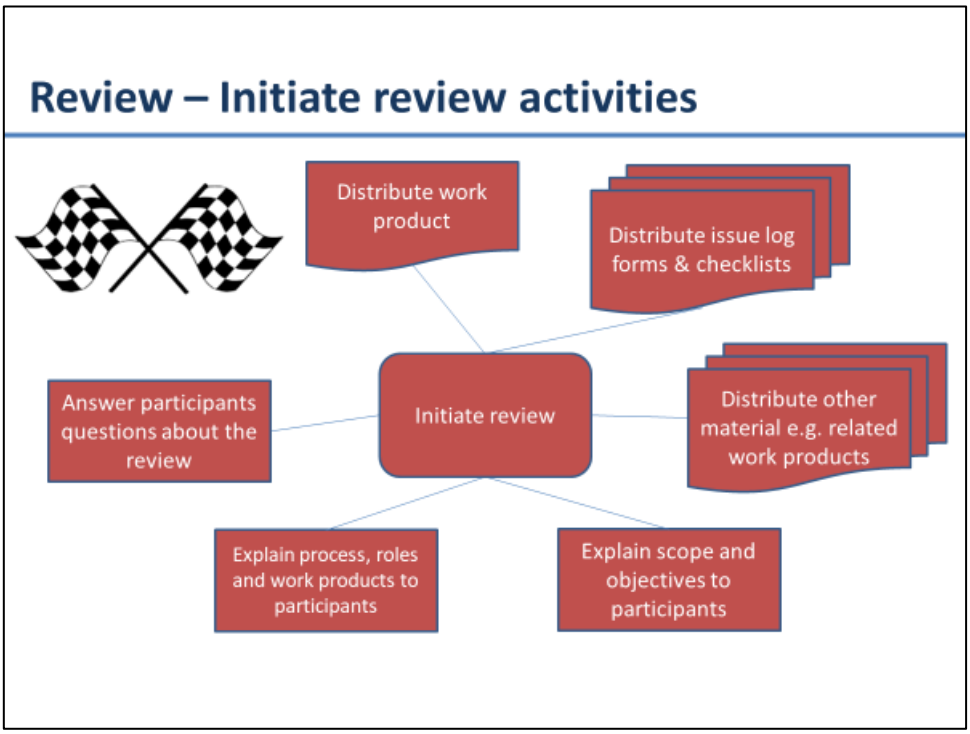
- Planning
- Initiate review
- Individual review (i.e., individual preparation)
- Issue communication and analysis
- Fixing and reporting

The results of a work product review vary, depending on the review type and formality, as described in section 3.2.3.



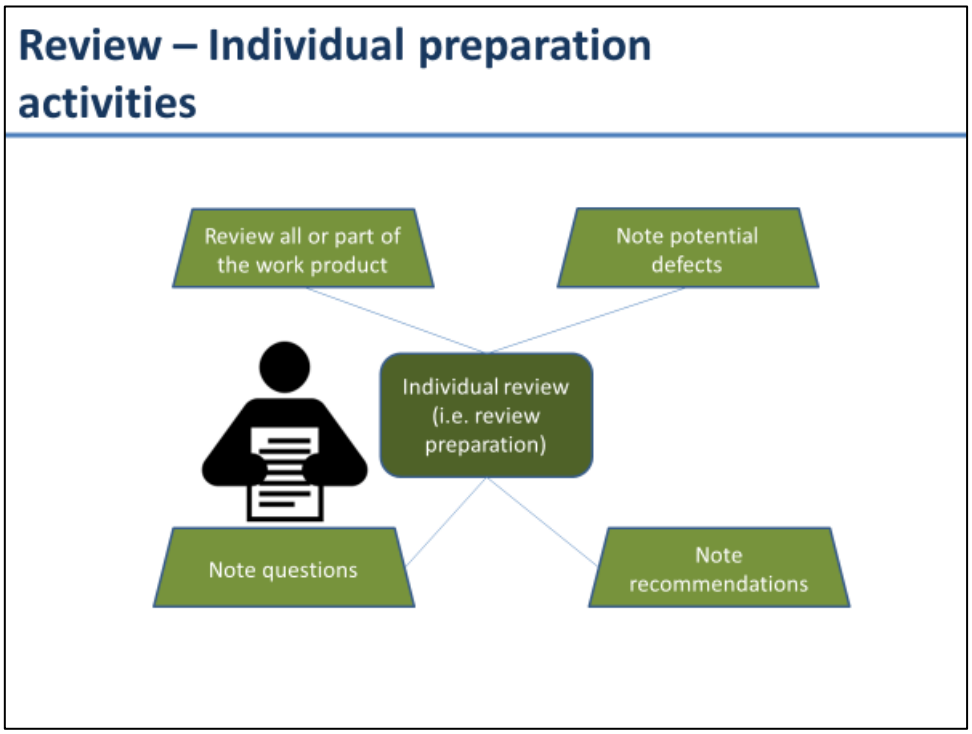
Planning

- Defining the scope, which includes the purpose of the review, what documents or parts of documents to review, and the quality characteristics to be evaluated
- Estimating effort and timeframe
- Identifying review characteristics such as the review type with roles, activities, and checklists
- Selecting the people to participate in the review and allocating roles
- Defining the entry and exit criteria for more formal review types (e.g., inspections)
- Checking that entry criteria are met (for more formal review types)



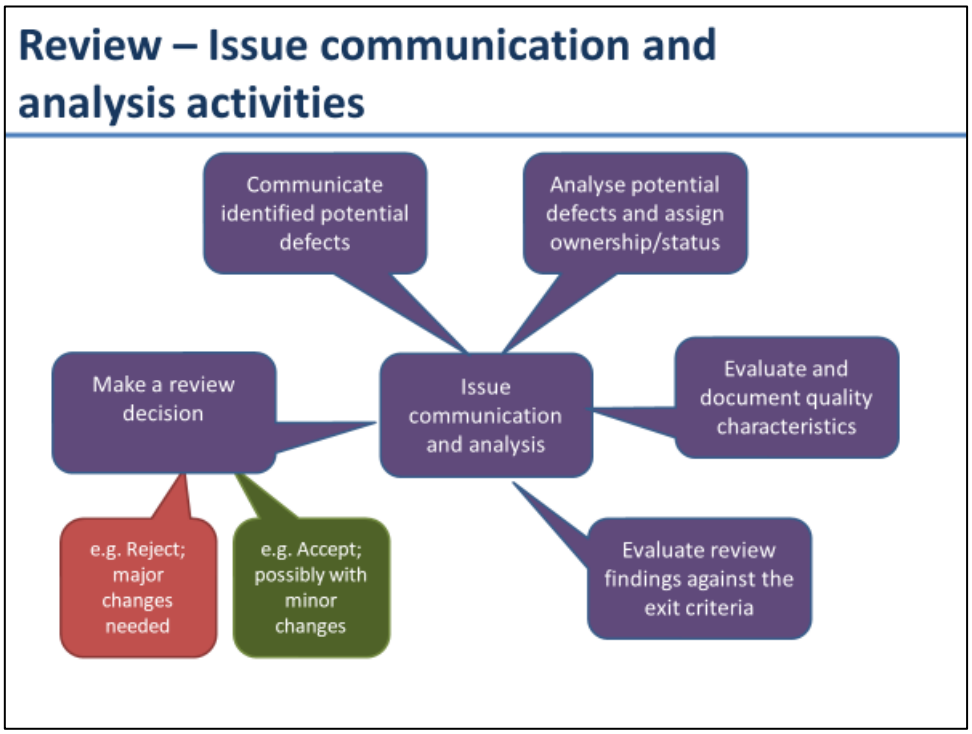
Initiate review

- Distributing the work product (physically or by electronic means) and other material, such as issue log forms, checklists, and related work products
- Explaining the scope, objectives, process, roles, and work products to the participants
- Answering any questions that participants may have about the review



Individual review (i.e., individual preparation)

- Reviewing all or part of the work product
- Noting potential defects, recommendations, and questions



Issue communication and analysis

- Communicating identified potential defects (e.g., in a review meeting)
- Analyzing potential defects, assigning ownership and status to them
- Evaluating and documenting quality characteristics
- Evaluating the review findings against the exit criteria to make a review decision (reject; major changes needed; accept, possibly with minor changes)

Fixing and reporting

- Creating defect reports for those findings that require changes to a work product
- Fixing defects found (typically done by the author) in the work product reviewed
- Communicating defects to the appropriate person or team (when found in a work product related to the work product reviewed)
- Recording updated status of defects (in formal reviews), potentially including the agreement of the comment originator
- Gathering metrics (for more formal review types)
- Checking that exit criteria are met (for more formal review types)
- Accepting the work product when the exit criteria are reached

A typical formal review will include the roles below:

Author

- Creates the work product under review
- Fixes defects in the work product under review (if necessary)

Management

- Is responsible for review planning
- Decides on the execution of reviews
- Assigns staff, budget, and time
- Monitors ongoing cost-effectiveness
- Executes control decisions in the event of inadequate outcomes

Facilitator (often called moderator)

- Ensures effective running of review meetings (when held)
- Mediates, if necessary, between the various points of view
- Is often the person upon whom the success of the review depends

Review leader

- Takes overall responsibility for the review
- Decides who will be involved and organizes when and where it will take place

Reviewers

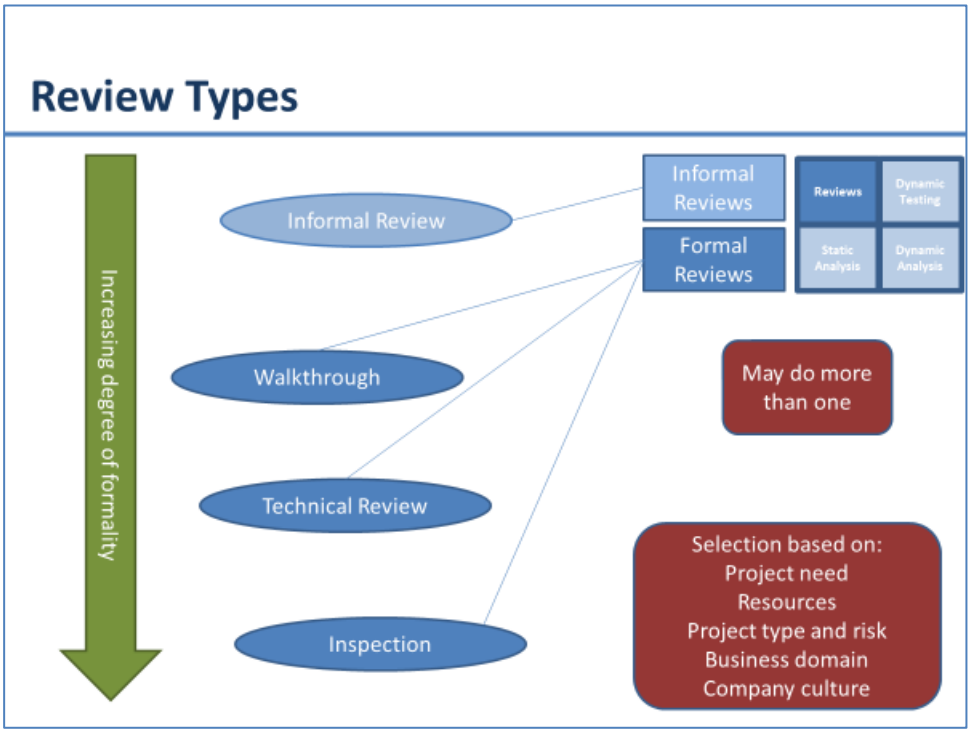
- May be subject matter experts, persons working on the project, stakeholders with an interest in the work product, and/or individuals with specific technical or business backgrounds
- Identify potential defects in the work product under review
- May represent different perspectives (e.g., tester, developer, user, operator, business analyst, usability expert, etc.)

Scribe (or recorder)

- Collates potential defects found during the individual review activity
- Records new potential defects, open points, and decisions from the review meeting (when held)

In some review types, one person may play more than one role, and the actions associated with each role may also vary based on review type. In addition, with the advent of tools to support the review process, especially the logging of defects, open points, and decisions, there is often no need for a scribe.

Further, more detailed roles are possible, as described in ISO standard (ISO/IEC 20246).

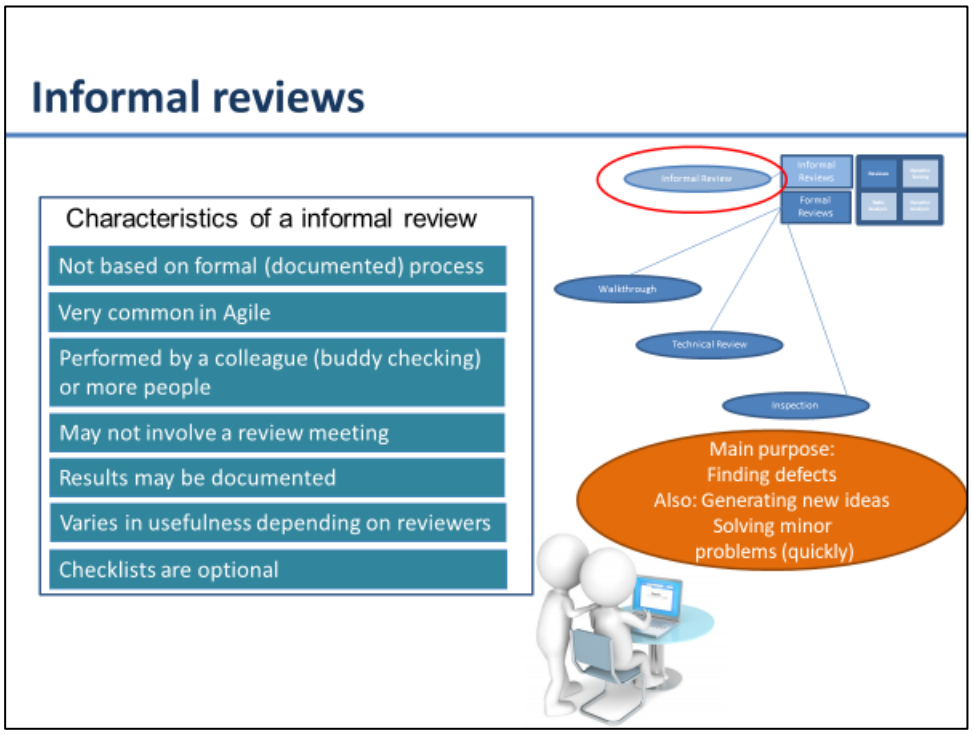


Although reviews can be used for various purposes, one of the main objectives is to uncover defects. All review types can aid in defect detection, and the selected review type should be based on the needs of the project, available resources, product type and risks, business domain, and company culture, among other selection criteria.

A single work product may be the subject of more than one type of review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review to ensure the work product is ready for a technical review.

The types of reviews described below can be done as peer reviews, i.e., done by colleagues qualified to do the same work.

The types of defects found in a review vary, depending especially on the work product being reviewed. (See section 3.1.3 for examples of defects that can be found by reviews in different work products, and Gilb 1993 for information on formal inspections). Reviews can be classified according to various attributes. The following lists the four most common types of reviews and their associated attributes.

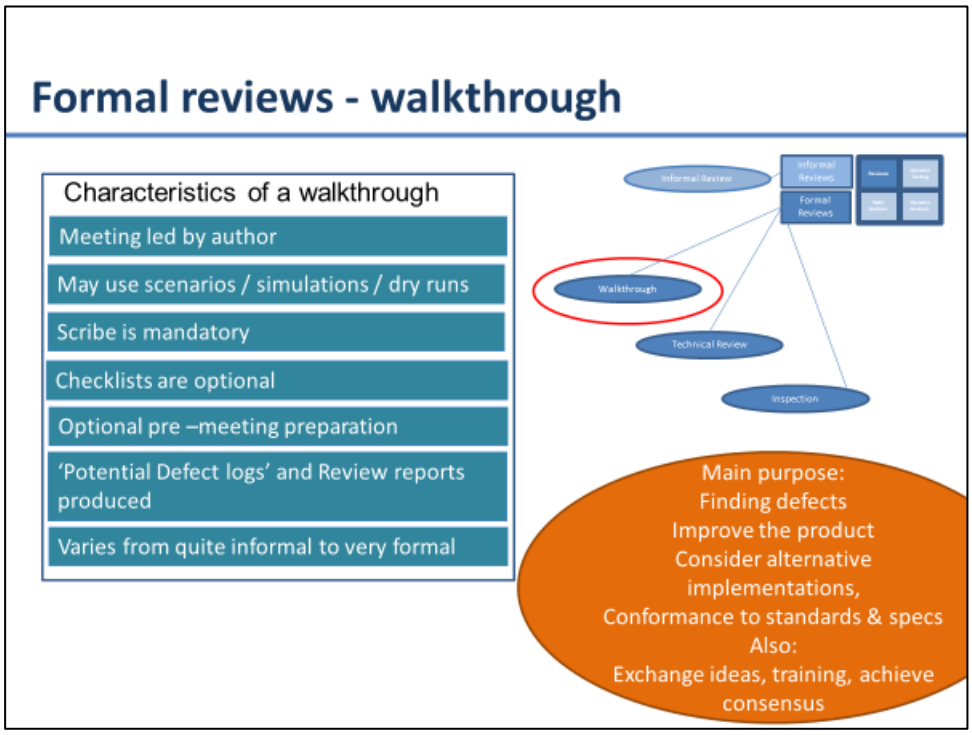


Informal review e.g., buddy check, pairing, pair review

Main purpose: detecting potential defects

Possible additional purposes: generating new ideas or solutions, quickly solving minor problems

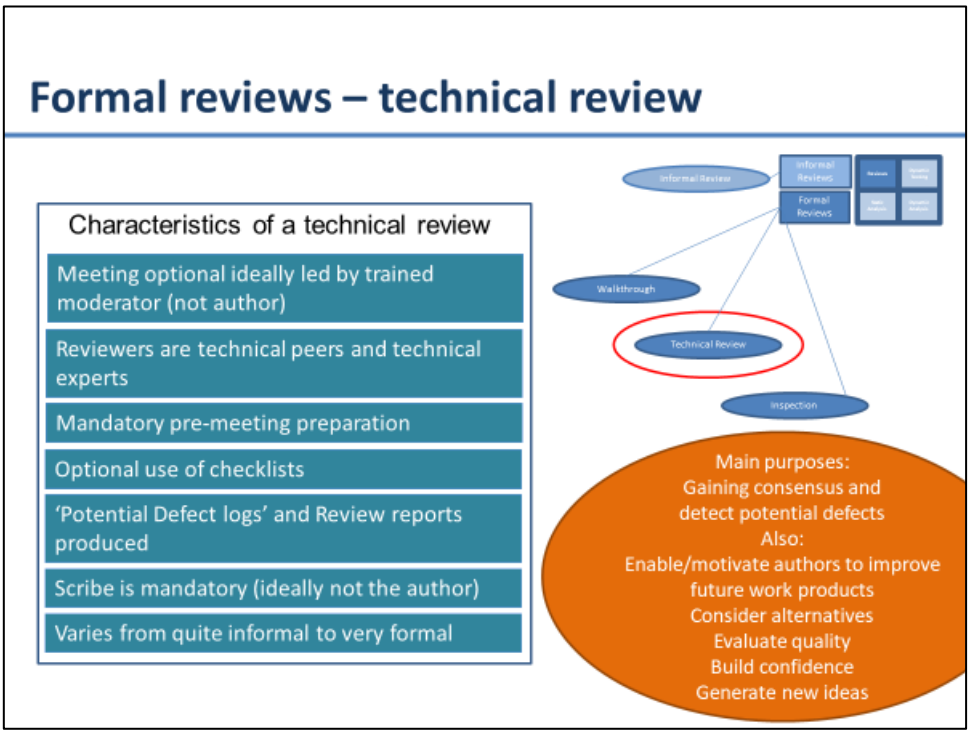
- Not based on a formal (documented) process
- May not involve a review meeting
- May be performed by a colleague of the author (buddy check) or by more people
- Results may be documented
- Varies in usefulness depending on the reviewers
- Use of checklists is optional
- Very commonly used in Agile development



Main purposes: find defects, improve the software product, consider alternative implementations, evaluate conformance to standards and specifications

Possible additional purposes: exchanging ideas about techniques or style variations, training of participants, achieving consensus

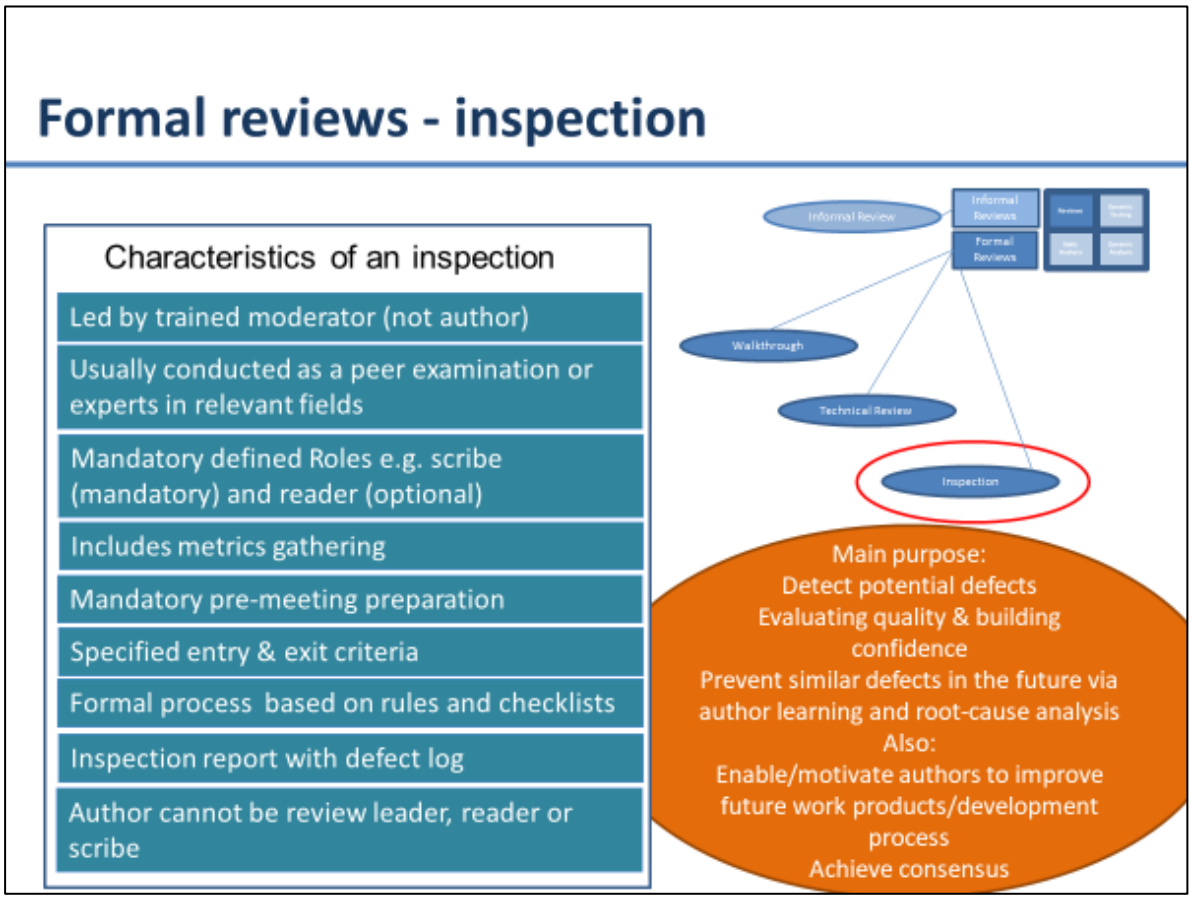
- Individual preparation before the review meeting is optional
- Review meeting is typically led by the author of the work product
- Scribe is mandatory
- Use of checklists is optional
- May take the form of scenarios, dry runs, or simulations
- Potential defect logs and review reports are produced
- May vary in practice from quite informal to very formal



Main purposes: gaining consensus, detecting potential defects

Possible further purposes: evaluating quality and building confidence in the work product, generating new ideas, motivating and enabling authors to improve future work products, considering alternative implementations

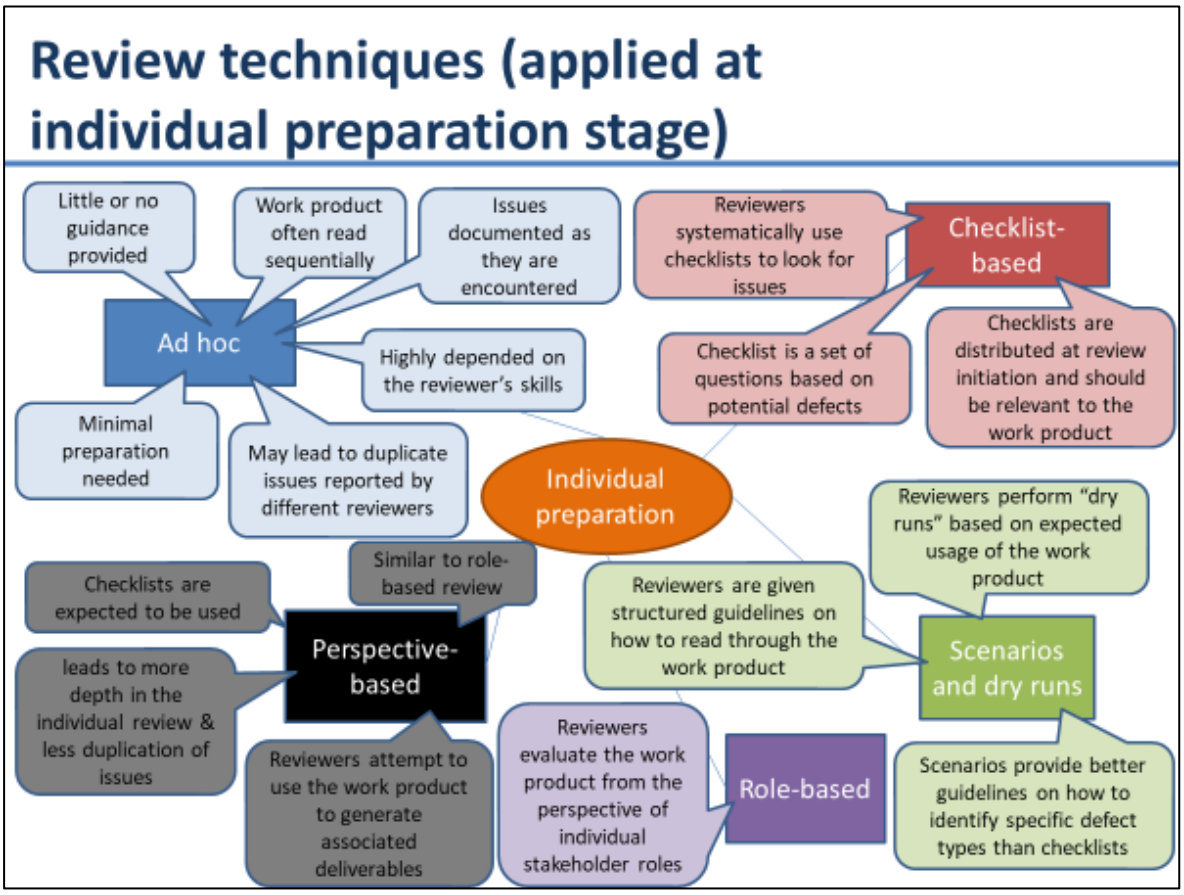
- Reviewers should be technical peers of the author, and technical experts in the same or other disciplines
- Individual preparation before the review meeting is required
- Review meeting is optional, ideally led by a trained facilitator (typically not the author)
- Scribe is mandatory, ideally not the author
- Use of checklists is optional
- Potential defect logs and review reports are produced



Main purposes: detecting potential defects, evaluating quality and building confidence in the work product, preventing future similar defects through author learning and root cause analysis

Possible further purposes: motivating and enabling authors to improve future work products and the software development process, achieving consensus

- Follows a defined process with formal documented outputs, based on rules and checklists
- Uses clearly defined roles, such as those specified in section 3.2.2 which are mandatory, and may include a dedicated reader (who reads the work product aloud often paraphrase, i.e. describes it in own words, during the review meeting)
- Individual preparation before the review meeting is required
- Reviewers are either peers of the author or experts in other disciplines that are relevant to the work product
- Specified entry and exit criteria are used
- Scribe is mandatory
- Review meeting is led by a trained facilitator (not the author)
- Author cannot act as the review leader, reader, or scribe
- Potential defect logs and review report are produced
- Metrics are collected and used to improve the entire software development process, including the inspection process



There are a number of review techniques that can be applied during the individual review (i.e., individual preparation) activity to uncover defects. These techniques can be used across the review types described above. The effectiveness of the techniques may differ depending on the type of review used. Examples of different individual review techniques for various review types are listed below.

Ad hoc

In an ad hoc review, reviewers are provided with little or no guidance on how this task should be performed. Reviewers often read the work product sequentially, identifying and documenting issues as they encounter them. Ad hoc reviewing is a commonly used technique needing little preparation. This technique is highly dependent on reviewer skills and may lead to many duplicate issues being reported by different reviewers.

Checklist-based

A checklist-based review is a systematic technique, whereby the reviewers detect issues based on checklists that are distributed at review initiation (e.g., by the facilitator). A review checklist consists of a set of questions based on potential defects, which may be derived from experience. Checklists should be specific to the type of work product under review and should be maintained regularly to cover issue types missed in previous reviews. The main advantage of the checklist-based technique is a systematic coverage of typical defect types. Care should be taken not to simply follow the checklist in individual reviewing, but also to look for defects outside the checklist.

Scenarios and dry runs
In a scenario-based review, reviewers are provided with structured guidelines on how to read through the work product. A scenario-based review supports reviewers in performing “dry runs” on the work product based on expected usage of the work product (if the work product is documented in a suitable format such as use cases). These scenarios provide reviewers with better guidelines on how to identify specific defect types than simple checklist entries. As with checklist-based reviews, in order not to miss other defect types (e.g., missing features), reviewers should not be constrained to the documented scenarios.

Perspective-based
In perspective-based reading, similar to a role-based review, reviewers take on different stakeholder viewpoints in individual reviewing. Typical stakeholder viewpoints include end user, marketing, designer, tester, or operations. Using different stakeholder viewpoints leads to more depth in individual reviewing with less duplication of issues across reviewers.

In addition, perspective-based reading also requires the reviewers to attempt to use the work product under review to generate the product they would derive from it. For example, a tester would attempt to generate draft acceptance tests if performing a perspective-based reading on a requirements specification to see if all the necessary information was included. Further, in perspective-based reading, checklists are expected to be used.

Empirical studies have shown perspective-based reading to be the most effective general technique for reviewing requirements and technical work products. A key success factor is including and weighing different stakeholder viewpoints appropriately, based on risks. See Shul 2000 for details on perspective based reading, and Sauer 2000 for the effectiveness of different review techniques.

Role-based
A role-based review is a technique in which the reviewers evaluate the work product from the perspective of individual stakeholder roles. Typical roles include specific end user types (experienced, inexperienced, senior, child, etc.), and specific roles in the organization (user administrator, system administrator, performance tester, etc.). The same principles apply as in perspective-based reading because the roles are similar.

Review Exercise

You have been asked to participate in a checklist based review

You have been provided with an excerpt from the design specification (Reader page 24)

The first point on your checklist says:

"are there any potential risks in the design"

Which ONE of the following comments are you LEAST LIKELY to make:

A. Customers with very long names may get their details truncated by the client app
B. Mobile phone numbers may get stored as home phone numbers on the server
C. Email campaigns based on server data may not reach all the customers collected by the client
D. Data protection legislation risks have been adequately mitigated on both client and server

You have 5 minutes

See following page for details

A checklist-based review is being conducted of the following excerpt from the design specification for a client-server application that will be used for international marketing.

The app will allow promotional canvassers to collect information on potential customers for follow up email marketing campaigns, which are subject to data protection (DP) legislation. This legislation requires customers to give consent for their details to be stored on a computer system.
The marketing app (Client) will collect the following information for transmission to the marketing database:

- All entry fields on the app will store up to a maximum of 20 characters (Chrs) per line
- First Name (Mandatory)
- Surname (Mandatory)
- Address (Optional) up to 4 lines, including house number or house name)
- Post Code (Optional)
- Contact number (Mandatory)
- Email address (optional)

The marketing database (Server) consists of the following fields:

Field Name	Field Length	Field Type	Validation Rules	Mandatory Field Y/N
First Name	20 Chrs	Alphanumeric	1-20 Chrs (No Special Chrs)	Y
Surname	20 Chrs	Alphanumeric	1-20 Chrs (No Special Chrs)	Y
House Number	6 Digit	Numeric	1-6 Digits	Y
Street	20 Chrs	Alphanumeric	0-20 Chrs (No Special Chrs)	N
Town	20 Chrs	Alphanumeric	0-20 Chrs (No Special Chrs)	N
County/City	20 Chrs	Alphanumeric	0-20 Chrs (No Special Chrs)	N
Post Code	8 Characters	Alphanumeric	0-20 Chrs (No Special Chrs)	N
Home Tel No	16 Digit	Numeric	1 – 16 Digits	Y
Mobile Tel No	16 Digit	Numeric	0 – 16 Digits	N
Email	20 Characters	Alphanumeric	1-20 Chrs (No Special Chrs)	Y
DP Consent	1 Character	Alphanumeric	Y or N only	N

Which ONE of the following comments is LEAST LIKELY to have been made by the reviewer with the checklist entry to review the design for potential risks:

A. Customers with very long names may get their details truncated by the client app
B. Mobile phone numbers may get stored as home phone numbers on the server
C. Email campaigns based on server data may not reach all the customers collected by the client
D. Data protection legislation risks have been adequately mitigated on both client and server

Answer on following page

Answer justification

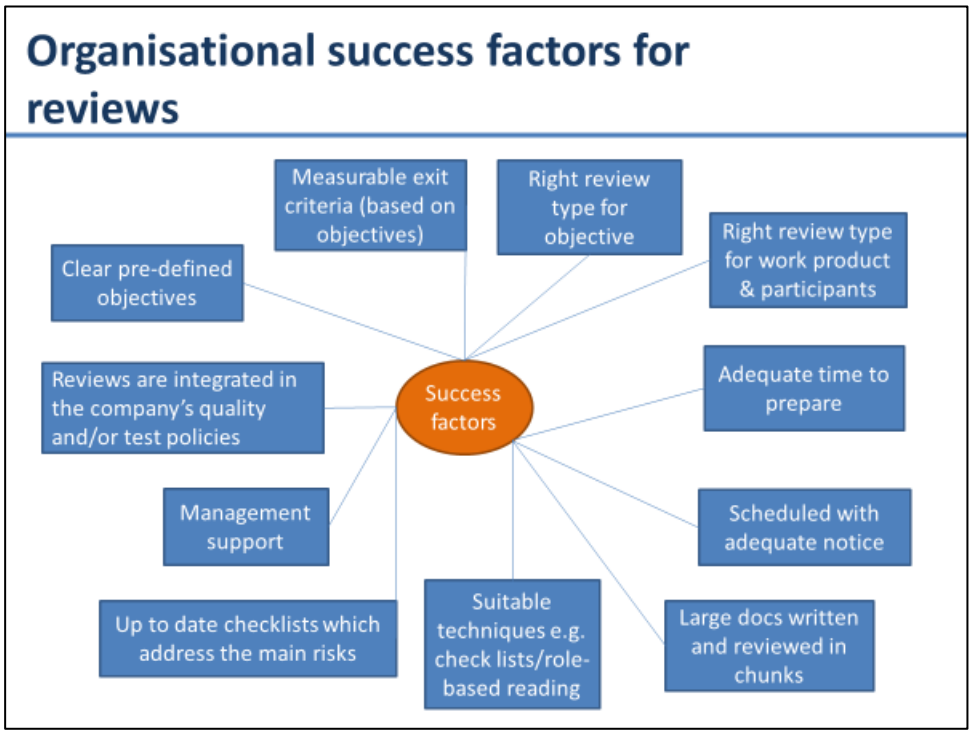
The CORRECT answer is option D

A is INCORRECT – this comment is LIKELY to be made because the application is intended for international marketing and in some languages and cultures forenames or surnames may exist which are potentially longer than 20 characters (also some names may not be easily categorised/separated into discrete forename or surname entries).

B is INCORRECT – this comment is LIKELY to be made because the client does not specifically categorise the contact number, also as home number is a mandatory field on the server; a customer record with only one number supplied may automatically be written to the (mandatory) home number field, even if that number represents the customer's mobile number.

C is INCORRECT – this comment is LIKELY to be made because the email address is an optional field on the client (even though the purpose of the application is to facilitate email campaigns). Also email records are unlikely to be correctly written to the database as the validation rules do not permit special characters (which means that the "@" character in an email address is forbidden).

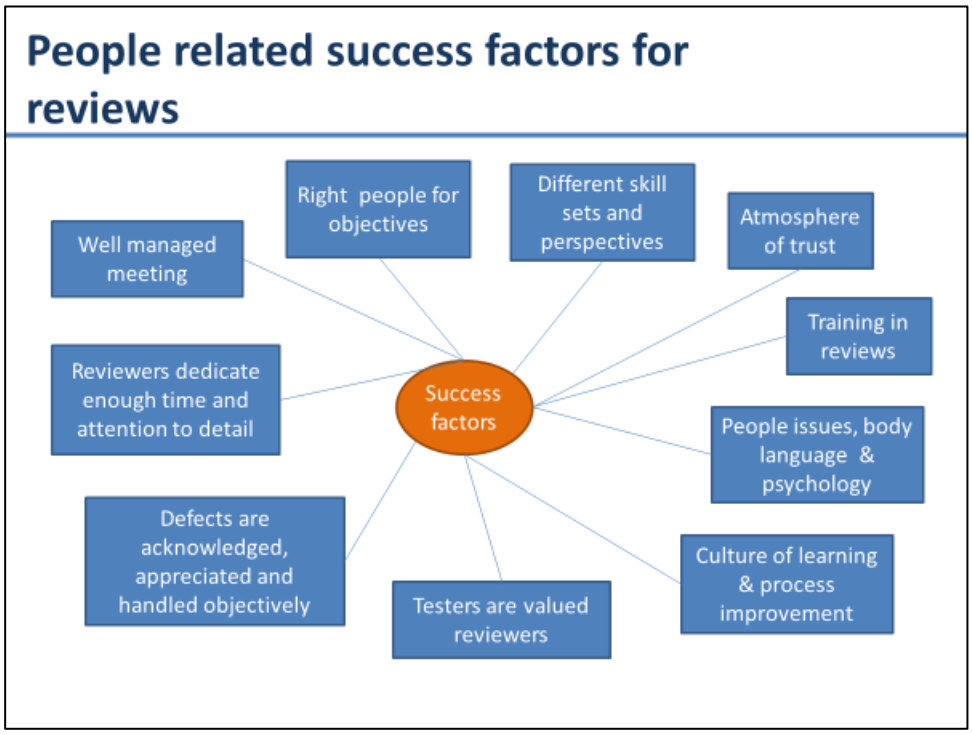
D is CORRECT – this comment is UNLIKELY to be made, because regulatory risks exist on both the server (although the database contains a field for DP content is not mandatory) and the client (client does not capture any consent details).



In order to have a successful review, the appropriate type of review and the techniques used must be considered. In addition, there are a number of other factors that will affect the outcome of the review.

Organizational success factors for reviews include:

- Each review has clear objectives, defined during review planning, and used as measurable exit criteria
- Review types are applied which are suitable to achieve the objectives and are appropriate to the type and level of software work products and participants
- Any review techniques used, such as checklist-based or role-based reviewing, are suitable for effective defect identification in the work product to be reviewed
- Any checklists used address the main risks and are up to date
- Large documents are written and reviewed in small chunks, so that quality control is exercised by providing authors early and frequent feedback on defects
- Participants have adequate time to prepare
- Reviews are scheduled with adequate notice
- Management supports the review process (e.g., by incorporating adequate time for review activities in project schedules)
- Reviews are integrated in the company's quality and/or test policies



People-related success factors for reviews include:

- The right people are involved to meet the review objectives, for example, people with different skill sets or perspectives, who may use the document as a work input
- Testers are seen as valued reviewers who contribute to the review and learn about the work product, which enables them to prepare more effective tests, and to prepare those tests earlier
- Participants dedicate adequate time and attention to detail
- Reviews are conducted on small chunks, so that reviewers do not lose concentration during individual review and/or the review meeting (when held)
- Defects found are acknowledged, appreciated, and handled objectively
- The meeting is well-managed, so that participants consider it a valuable use of their time
- The review is conducted in an atmosphere of trust; the outcome will not be used for the evaluation of the participants
- Participants avoid body language and behaviors that might indicate boredom, exasperation, or hostility to other participants
- Adequate training is provided, especially for more formal review types such as inspections
- A culture of learning and process improvement is promoted

See Gilb 1993, Wiegers 2002, and van Veenendaal 2004 for more on successful reviews.

Summary

- Formal reviews follow a 5 stage process
- There are 6 key roles in a review team
- Formal and informal reviews have different characteristics
- There are 3 types of formal review
- There are many factors that can influence the successful performance of reviews

End of section exercise

- Attempt the following Questions from Sample Exam B
- Questions 14 to 18
- Please mark your own (See Sample Exam B Answers) and read the answer justifications as needed

You have 7.5 minutes

Homework – Day 1

- Attempt the following Questions from Sample Exam C
- Section 1
  - Questions 1 to 8
- Section 2
  - Questions 9 to 13
- Section 3
  - Questions 14 to 18
- Please mark your own (See Sample Exam C Answers) and read the answer justifications as needed
- Read the Foundation Syllabus Sections 1, 2 and 3

Learning Objectives for Static Techniques

In order to complete this section, you should satisfy yourself that you are able to understand or perform the following items to the standard indicated by the corresponding “K - learning level”. For an explanation of “K levels”, please see the course introduction section – “Learning Objectives and Levels of Knowledge”.

Keywords

ad hoc review, checklist-based review, dynamic testing, formal review, informal review, inspection, perspective-based reading, review, role-based review, scenario-based review, static analysis, static testing, technical review, walkthrough

Learning Objectives for Static Testing

3.1 Static Testing Basics

FL-3.1.1 (K1) Recognize types of software work product that can be examined by the different static testing techniques
FL-3.1.2 (K2) Use examples to describe the value of static testing
FL-3.1.3 (K2) Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software lifecycle

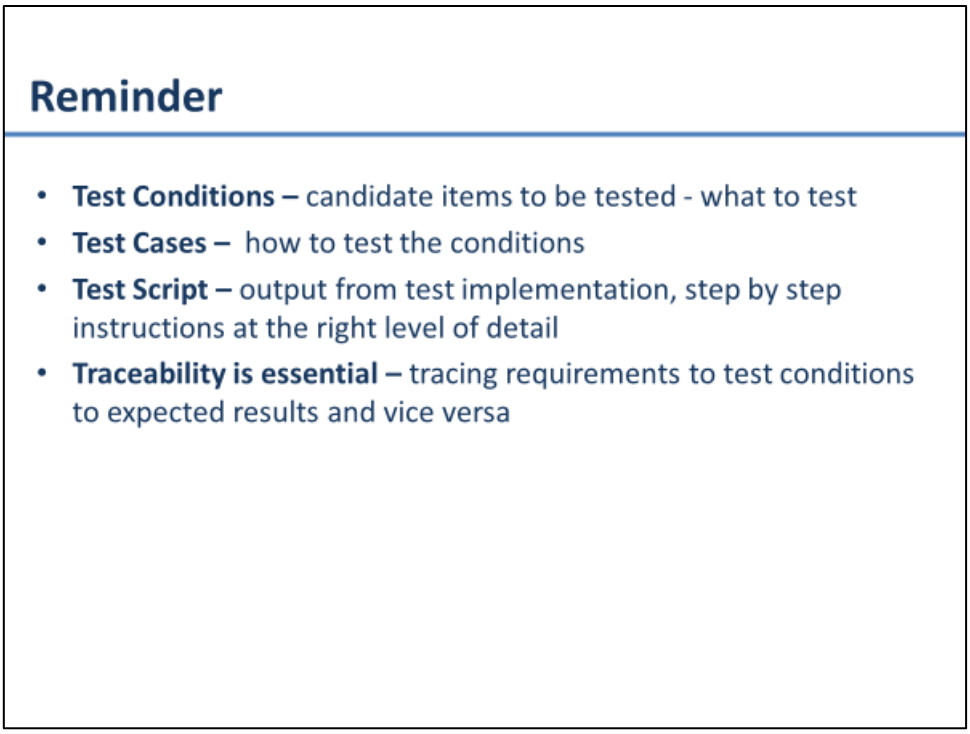
3.2 Review Process

FL-3.2.1 (K2) Summarize the activities of the work product review process
FL-3.2.2 (K1) Recognize the different roles and responsibilities in a formal review
FL-3.2.3 (K2) Explain the differences between different review types: informal review, walkthrough, technical review, and inspection
FL-3.2.4 (K3) Apply a review technique to a work product to find defects
FL-3.2.5 (K2) Explain the factors that contribute to a successful review

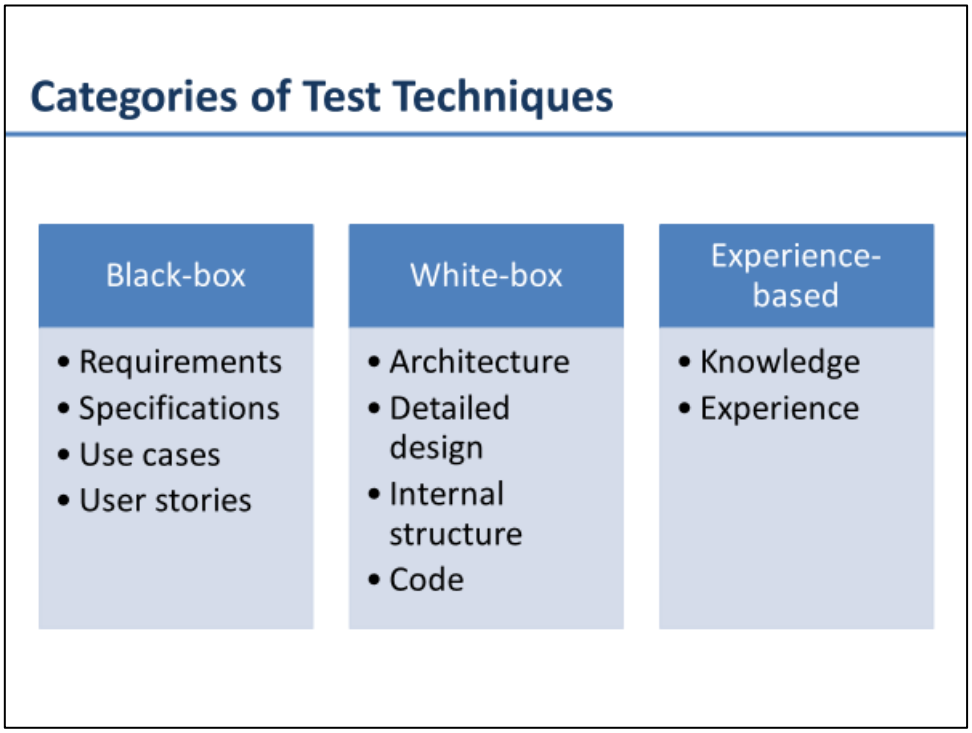
Test Techniques

Table of Contents

Section 4.1 – Categories of Test Techniques



The purpose of a test technique, including those discussed in this section, is to help in identifying test conditions, test cases, and test data.



In this syllabus, test techniques are classified as black-box, white-box or experience-based.

Common characteristics of black-box test techniques include the following:

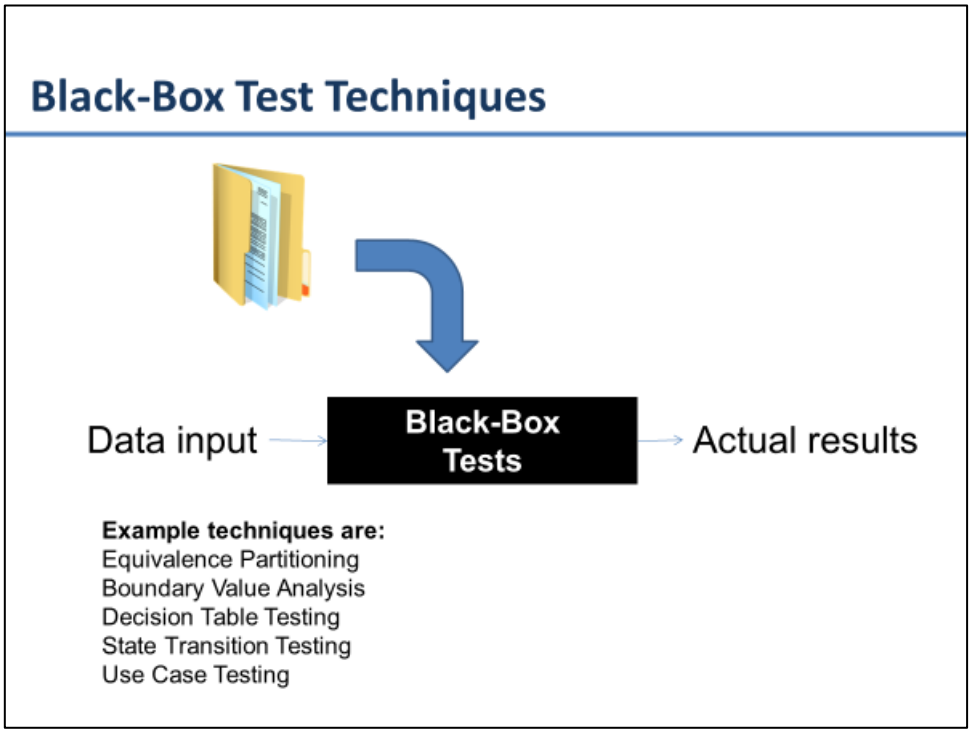
- Test conditions, test cases, and test data are derived from a test basis that may include software requirements, specifications, use cases, and user stories
- Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements
- Coverage is measured based on the items tested in the test basis and the technique applied to the test basis

Common characteristics of white-box test techniques include:

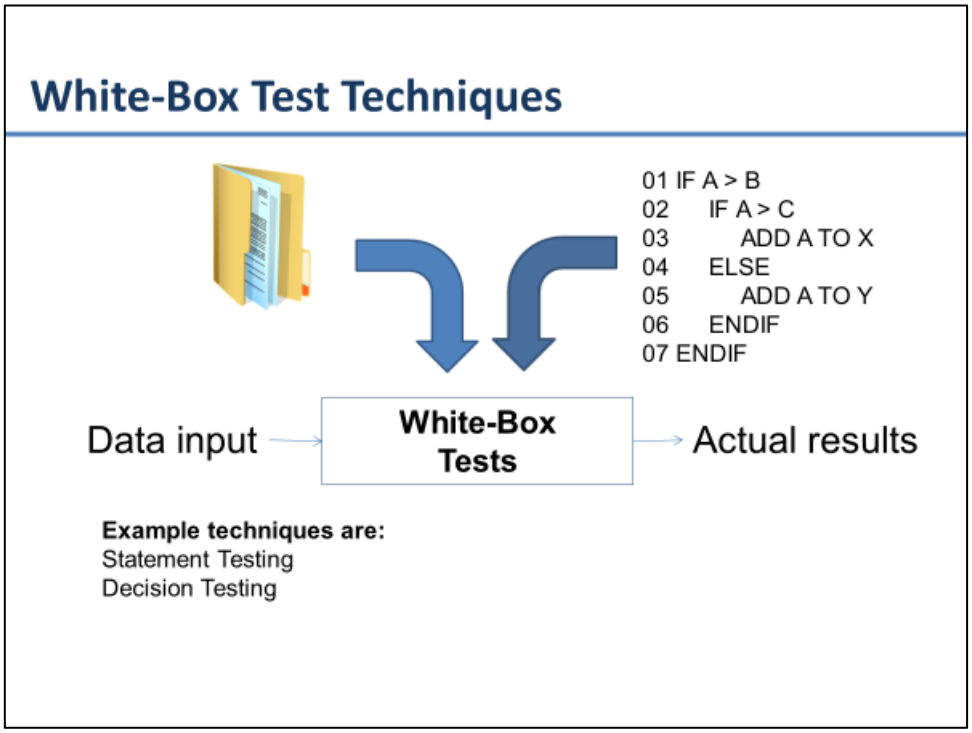
- Test conditions, test cases, and test data are derived from a test basis that may include code, software architecture, detailed design, or any other source of information regarding the structure of the software
- Coverage is measured based on the items tested within a selected structure (e.g., the code or interfaces) and the technique applied to the test basis

Common characteristics of experience-based test techniques include:

- Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders
- This knowledge and experience includes expected use of the software, its environment, likely defects, and the distribution of those defects

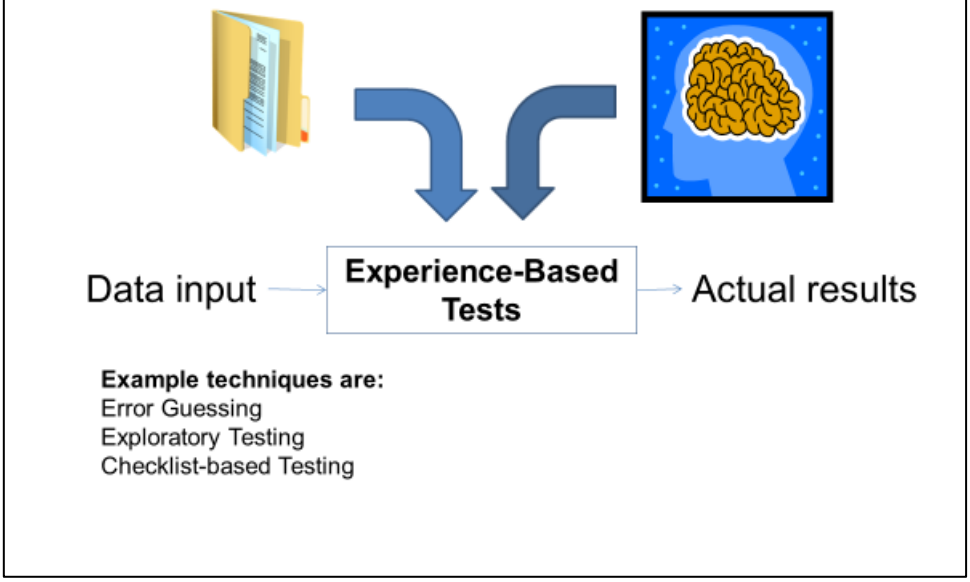


Black-box test techniques (also called behavioral or behavior-based techniques) are based on an analysis of the appropriate test basis (e.g., formal requirements documents, specifications, use cases, user stories, or business processes). These techniques are applicable to both functional and non-functional testing. Black-box test techniques concentrate on the inputs and outputs of the test object without reference to its internal structure.



White-box test techniques (also called structural or structure-based techniques) are based on an analysis of the architecture, detailed design, internal structure, or the code of the test object. Unlike black-box test techniques, white-box test techniques concentrate on the structure and processing within the test object.

Experience-Based Test Techniques

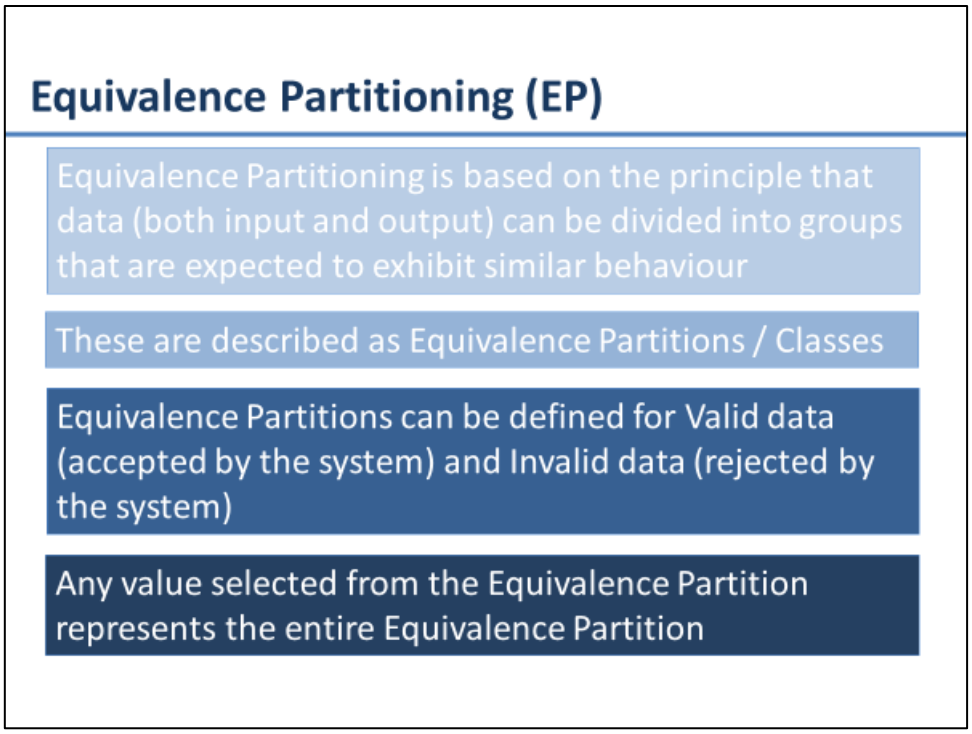


Example techniques are:
Error Guessing
Exploratory Testing
Checklist-based Testing

Experience-based test techniques leverage the experience of developers, testers and users to design, implement and execute tests. These techniques are often combined with black-box and white-box test techniques.

Summary

- **Black-box Techniques**
  - Find errors in the behaviours of the system
  - Use requirements, specifications, user stories, both formal and informal
  - Includes EP, BVA, Decision Table Testing, State Transition Testing and Use Case Testing
- **White-box Techniques**
  - Find errors in the construction of the system
  - Use the code and detailed designs
  - Includes Statement Testing and Decision Testing
- **Experience-based Techniques**
  - Find errors not easily found by other techniques
  - Use the knowledge and experience of the tester and the project team
  - Includes Error Guessing, Exploratory Testing and Checklist-based Testing
- **Techniques are complementary**
  - Techniques can be (and often are) used alongside each other



Equivalence partitioning divides data into partitions (also known as equivalence classes) in such a way that all the members of a given partition are expected to be processed in the same way (see Kaner 2013 and Jorgensen 2014). There are equivalence partitions for both valid and invalid values.

- Valid values are values that should be accepted by the component or system. An equivalence partition containing valid values is called a “valid equivalence partition.”
- Invalid values are values that should be rejected by the component or system. An equivalence partition containing invalid values is called an “invalid equivalence partition.”
- Partitions can be identified for any data element related to the test object, including inputs, 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).
- Any partition may be divided into sub partitions if required.
- Each value must belong to one and only one equivalence partition.

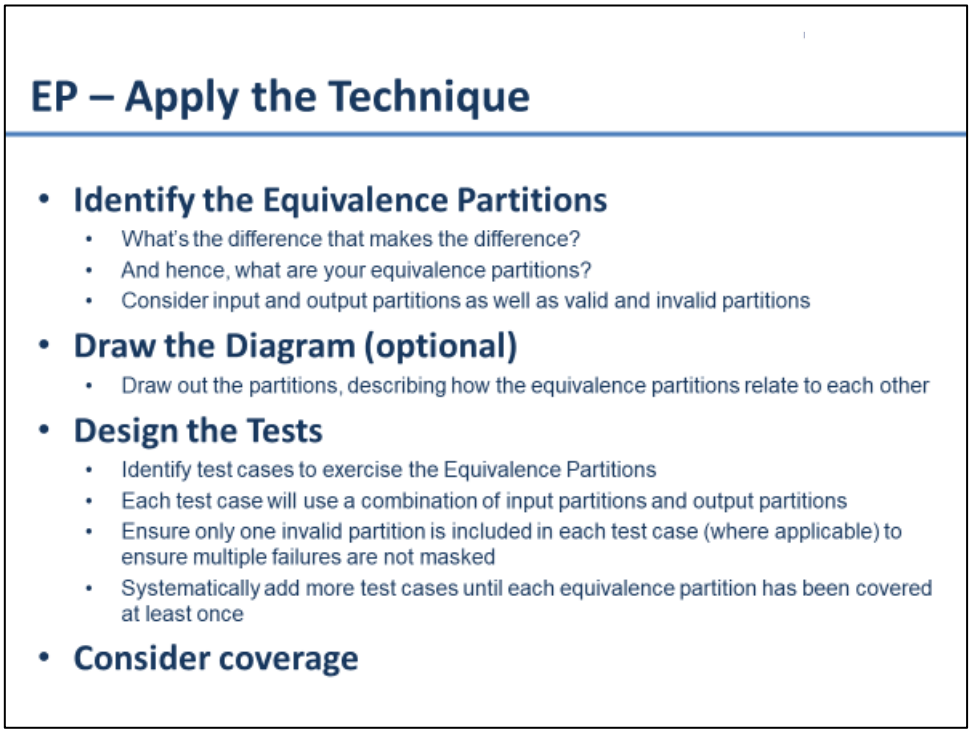
e.g. All airline passengers pay for their tickets.

“All Airline Passengers” can be subdivided into the following “Equivalence Partitions”:

- “Infants”
- “Children”
- “Adults”

each of whom will pay a different fare from each other.

However, each individual within each equivalence partition will pay the same fare for the same journey. ANY Infant can be selected for the partition “Infants” - they will all give the same result.



When invalid equivalence partitions are used in test cases, they should be tested individually, i.e., not combined with other invalid equivalence partitions, to ensure that failures are not masked. Failures can be masked when several failures occur at the same time but only one is visible, causing the other failures to be undetected.

To achieve 100% coverage with this technique, test cases must cover all identified partitions (including invalid partitions) by using a minimum of one value from each partition. Coverage is measured as the number of equivalence partitions tested by at least one value, divided by the total number of identified equivalence partitions, normally expressed as a percentage. Equivalence partitioning is applicable at all test levels.

EP – Worked Example

Requirements

- A customer is eligible for car insurance if they are at least 17, but no older than 75 years of age
- Customers younger than 17 and older than 75 years of age are not eligible for car insurance
- Age may be entered in the range 0 to 999 years inclusive

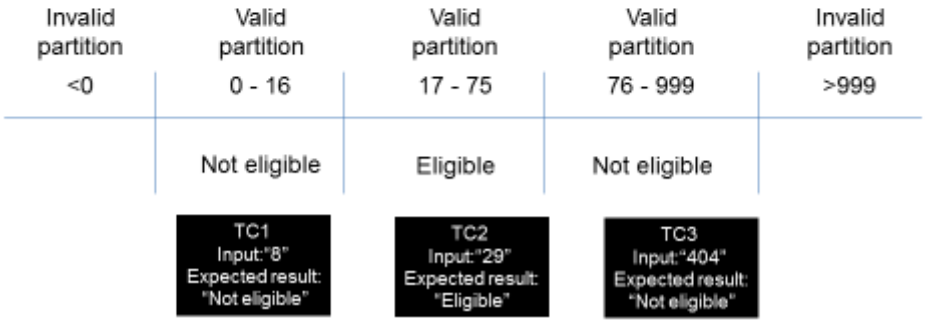
Exercise

- Create the equivalence partition diagram which describes the above requirements
- Identify test cases which will exercise the above requirements fully via Equivalence Partitioning

EP – Worked Example Answer

Requirements

- A customer is eligible for car insurance if they are at least 17, but no older than 75 years of age
- Customers younger than 17 and older than 75 years of age are not eligible for car insurance
- Age may be entered in the range 0 to 999 years inclusive



EP – Class Exercise

Requirements

- Users may input an item quantity
- Any item quantity may be ordered up to 999, but the minimum order is for 5 items
- Price is calculated based on quantity - orders of 100 items or more receive a 20% discount

Exercise

- Create the equivalence partition diagram which describes the above requirements
- Identify test cases which to achieve full Equivalence Partition coverage

Space for you to note down your answer

Page intentionally left blank

EP – Class Exercise Answer

Requirements

- Users may input an item quantity
- Any item quantity may be ordered up to 999, but the minimum order is for 5 items
- Price is calculated based on quantity - orders of 100 items or more receive a 20% discount

Invalid partition	Invalid partition	Valid partition	Valid partition	Invalid partition
Order Qty <0	Order Qty 0 - 4	Order Qty 5 - 99	Order Qty 100 - 999	Order Qty >999
Order rejected	Order rejected	Order accepted full price	Order accepted 20% discount	Order rejected

Remember:

- The two most obvious invalid partitions are those adjacent to the two valid partitions. These are invalid because the specification states the minimum and maximum values to be accepted by the system, thus implying that other inputs should be rejected.
- There could be other invalid partitions because no lower and upper boundaries have been specified for the invalid partitions.
- We have shown one additional invalid partition that may be relevant as it also coincides with another data type (i.e. negative values).
- The choice of which invalid partitions to include depends on risk.

Class Exercise (optional)

- A trial is planned to assess the impact of fitness level (very low, low, moderate, high, very high) and height (under 150 cm, 150-159 cm, 160 – 169 cm, 170 – 179 cm, over 179 cm) on climbing ability
- Bob (low fitness and 175 cm), Simon (moderate fitness and 168 cm) and Valerie (moderate fitness and 155 cm) have volunteered
- What is the minimum number of additional volunteers needed to achieve 100% Equivalence Partitioning coverage?

In the previous example, there are three requirements, all using the same variable – item quantity. This means that the requirements can be represented using one diagram (see previous answer). However, a requirement may contain different variables meaning that separate diagrams have to be drawn.

Class Exercise Answer (optional)

- A trial is planned to assess the impact of fitness level (very low, low, moderate, high, very high) and height (under 150 cm, 150-159 cm, 160 – 169 cm, 170 – 179 cm, over 179 cm) on climbing ability.
- Bob (low fitness and 175 cm), Simon (moderate fitness and 168 cm) and Valerie (moderate fitness and 155 cm) have volunteered.
- How many more volunteers are needed to achieve 100% Equivalence Partitioning coverage?

Very Low	Low	Moderate	High	Very High
	Bob	Simon Valerie		

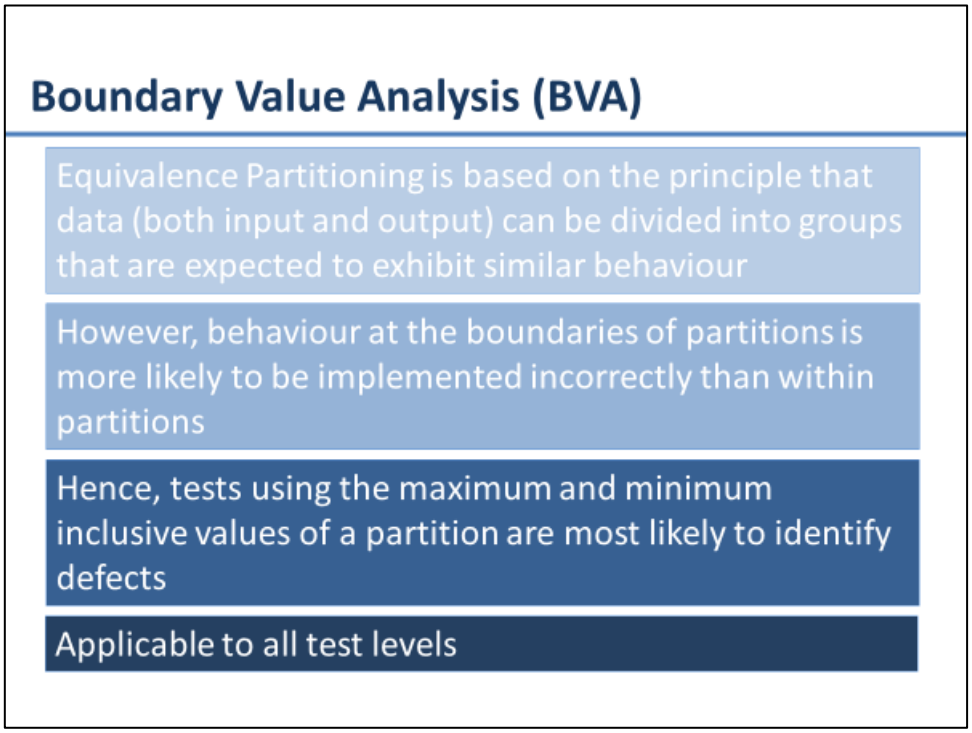
Under 150 cm	150 – 159 cm	160 – 169 cm	170 – 179 cm	Over 179 cm
	Valerie	Simon	Bob	

3 more volunteers are needed – the 'very low', 'high', and 'very high' partitions haven't been covered for fitness level along with the under 150 cm and over 180 cm partitions. These could be covered by selecting the right 3 volunteers to achieve 100% Equivalence coverage.

EP – Summary

Equivalence Partitioning

- A single value from each equivalence partition can be used to test how the system will deal with all the values represented by the Equivalence Partition
- The technique is applicable at any test level – Component to Acceptance
- Full coverage is achieved by systematically creating test cases until each of the equivalence partitions identified have been exercised at least once



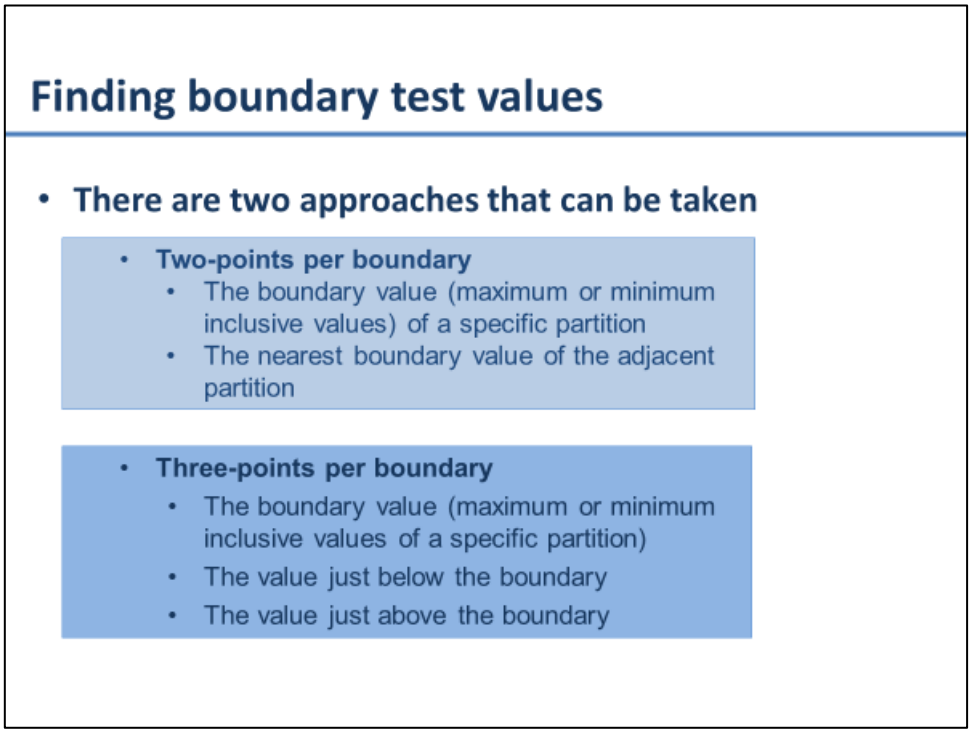
Boundary value analysis (BVA) is an extension of equivalence partitioning, but can only be used when the partition is ordered, consisting of numeric or sequential data. The minimum and maximum values (or first and last values) of a partition are its boundary values (see Beizer 1990).

For example, while it is easy to see the differences among the following lines of code:

```
X > 15;
X >= 16
X NOT < 16,
```

it may not be so easy to recognise the differences when only one of these lines of code has been used to implement a requirement, and that line of code has been buried inside hundreds, perhaps thousands of lines of code in a system.

Boundary value analysis can be applied at all test levels. This technique is generally used to test requirements that call for a range of numbers (including dates and times). Boundary coverage for a partition is measured as the number of boundary values tested, divided by the total number of identified boundary test values, normally expressed as a percentage.



Behavior at the boundaries of equivalence partitions is more likely to be incorrect than behavior within the partitions. It is important to remember that both specified and implemented boundaries may be displaced to positions above or below their intended positions, may be omitted altogether, or may be supplemented with unwanted additional boundaries. Boundary value analysis and testing will reveal almost all such defects by forcing the software to show behaviors from a partition other than the one to which the boundary value should belong.

BVA – Apply the Technique

- **Identify the Equivalence Partitions and Boundaries**
  - What's the difference that makes the difference?
  - And hence, what are your data partitions (input, output, valid, invalid)?
  - Confirm that you are dealing with ordered sets
  - Identify the minimum and maximum inclusive values for each partition and hence the boundaries.
- **Draw the Diagram (optional)**
  - Draw out the equivalence partitions that relate to the ordered set(s) with minimum and maximum inclusive values identified (i.e. the boundaries).
- **Design the Tests**
  - For each boundary, create a test case, with expected results
  - Systematically add new test cases until each boundary value has been tested at least once
  - The number of tests required depends on whether the two-point or three-point approach is used

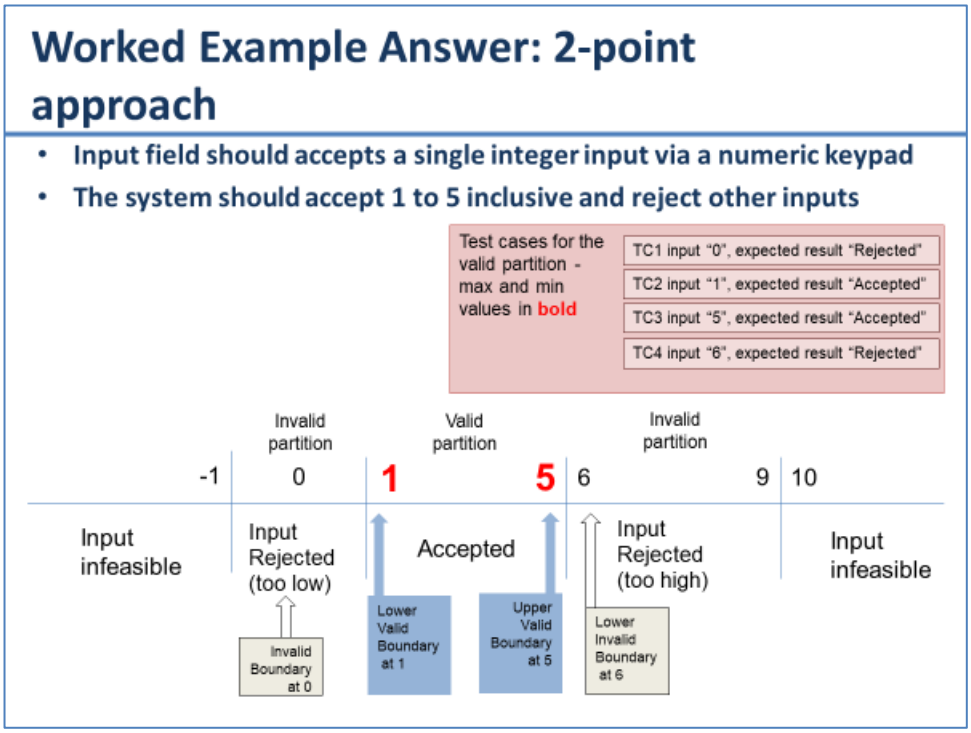
BVA – Worked Example

Requirements

- **Input field should accept a single digit integer**
- **Input is via a numeric keypad**
- **The system should accept 1 to 5 inclusive and reject other inputs**

Exercise

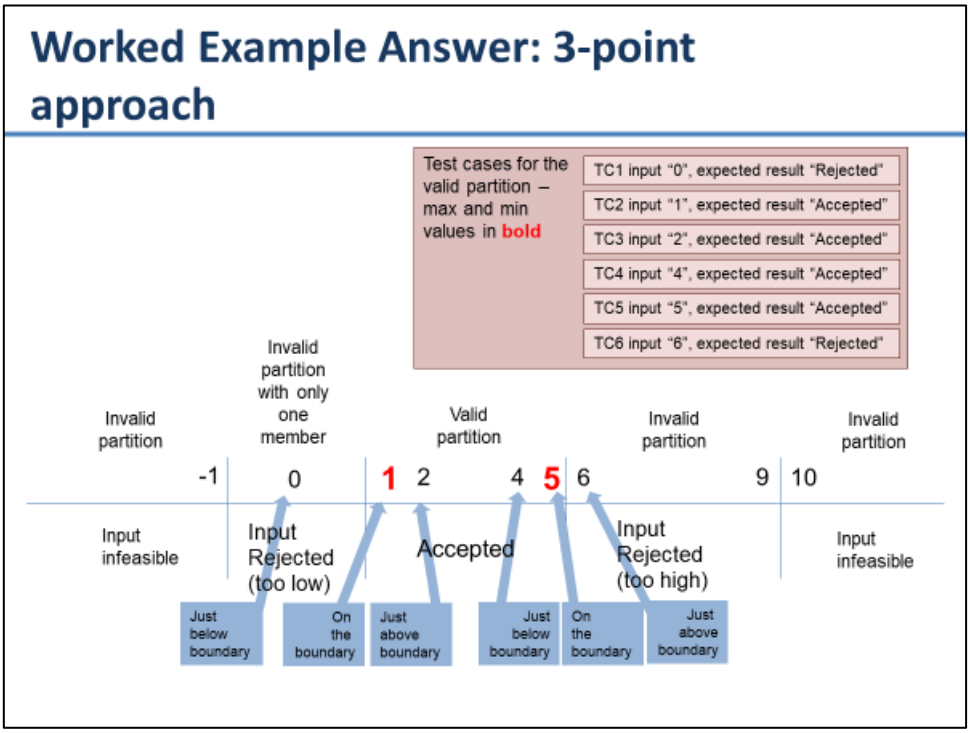
- Create the diagram which describes the equivalence partitions and boundaries
- Identify the boundary tests to cover the valid partitions using the 2-point and 3-point approaches



For example, suppose an input field accepts a single integer value as an input, using a keypad to limit inputs so that non-integer inputs are impossible. The valid range is from 1 to 5, inclusive. So, there are three equivalence partitions: invalid (too low); valid; invalid (too high). For the valid equivalence partition, the boundary values are 1 and 5. For the invalid (too high) partition, the boundary value is 6. For the invalid (too low) partition, there is only one boundary value, 0, because this is a partition with only one member.

In the example above, we identify two boundary values per boundary. The boundary between invalid (too low) and valid gives the test values 0 and 1. The boundary between valid and invalid (too high) gives the test values 5 and 6.

Other invalid partitions relating to infeasible inputs may also be identified (together with their associated boundaries). The choice of whether to include these in the test approach will depend on the level of risk.



Some variations of this technique identify three boundary values per boundary: the values before, at, and just over the boundary. In the previous example, using three-point boundary values, the lower boundary test values are 0, 1, and 2, and the upper boundary test values are 4, 5, and 6 (see Jorgensen 2014).

BVA – Class Exercise

Requirements

- People under the age of 16 cannot buy lotto tickets
- People under the age of 18 cannot buy cigarettes
- Age can be entered in the range 0-999

Exercise

- Create the diagram which describes the above data
- Identify the boundary tests which will exercise the above requirements fully via Boundary Value Analysis
  - First apply the 2 value approach
  - Then apply the 3 value approach

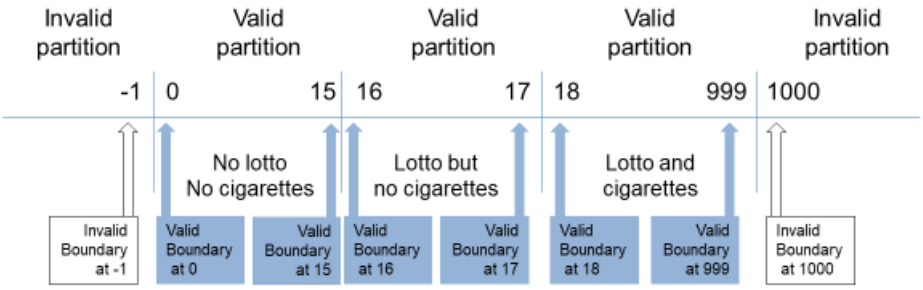
Space for you to note down your answer.

Page intentionally left blank

Class Exercise Answer – 2-point BVA

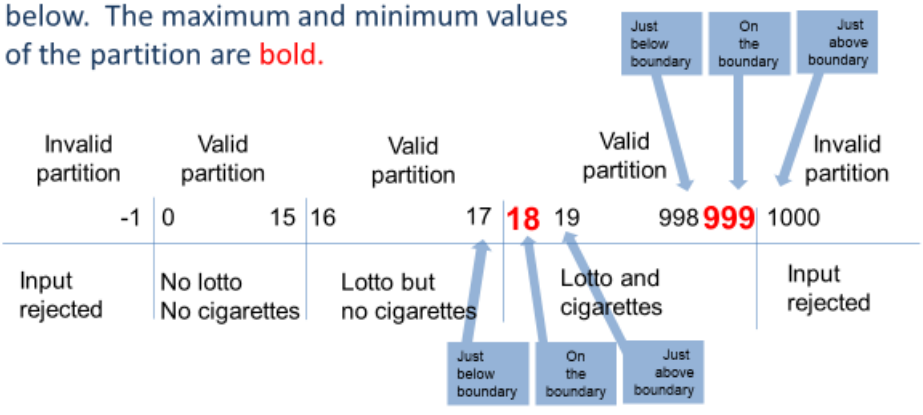
Requirements

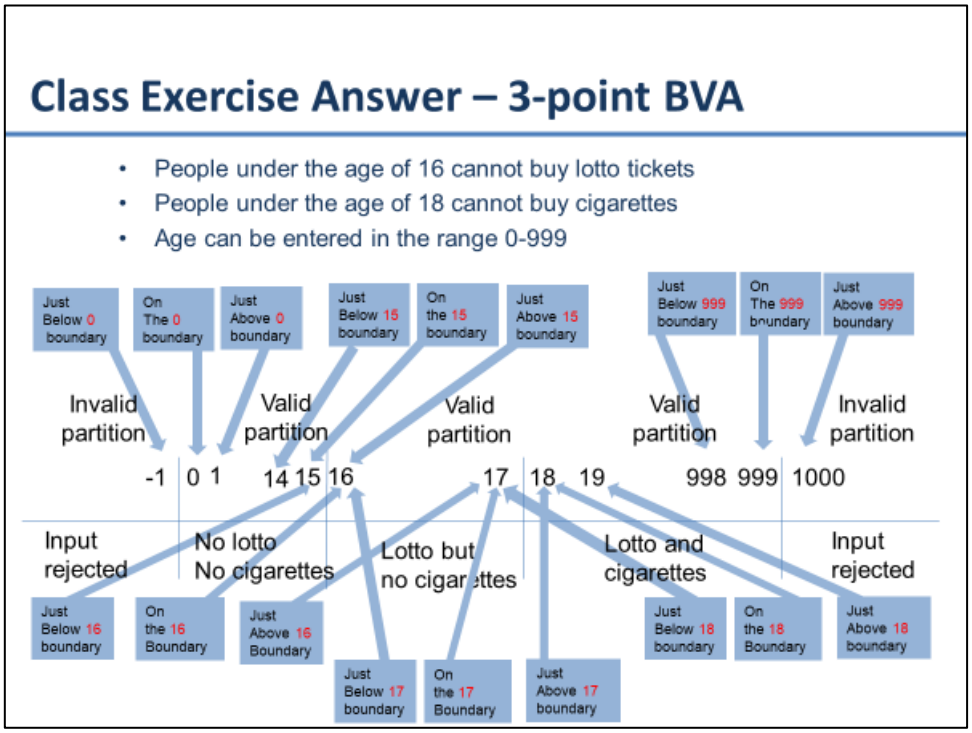
- People under the age of 16 cannot buy lotto tickets
- People under the age of 18 cannot buy cigarettes
- Age can be entered in the range 0-999



Class Exercise Answer – 3-point BVA

Only the values for 3-point BVA for the 'lotto and cigarettes' partition are illustrated below. The maximum and minimum values of the partition are bold.





Here is the complete answer, also fully represented in the table below. This addresses 3 point BVA for the valid partitions.

Partition	Boundary	Test data	Eliminate duplication
0-15	0	-1, 0, 1	-1, 0, 1
0-15	15	14, 15, 16	14, 15, 16
16-17	16	15, 16, 17	~~15, 16,~~ 17
16-17	17	16, 17, 18	~~16, 17,~~ 18
18-999	18	17, 18, 19	~~17, 18,~~ 19
18-999	999	998, 999, 1000	998, 999, 1000

EP & BVA – Class Exercise

Requirements

- Tax is charged in four bands
- The first £4,499 attracts 10% tax
- The next £12,500 attracts 15% tax
- The next £15,000 attracts 20% tax
- Any additional income attracts 40% tax

The corporate test strategy states that 2-point BVA must be used

Exercise

Part 1 - Which one of the following tests would be the result of generating a valid boundary test?

a) £12,500 b) £17,000 c) £15,000 d) £4,498

Part 2 - Which one of the following test inputs exercise a valid equivalence partition?

a) 5M b) 28,274 c) !"$^ d) forty pounds

Space for you to note down your answer

Part 1 –

Part 2 –

Page intentionally left blank

BVA – Class Exercise Answer – Part 1

Requirements

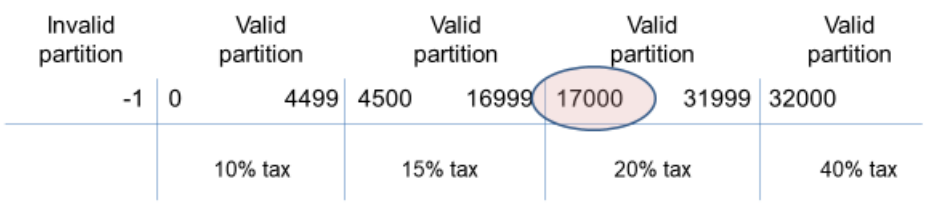
- Tax is charged in four bands
- The first £4,499 attracts 10% tax
- The next £12,500 attracts 15% tax
- The next £15,000 attracts 20% tax
- Any additional income attracts 40% tax
- The corporate test strategy states that 2-point BVA must be used

a) £12500

b) £17000

c) £15000

d) £4498

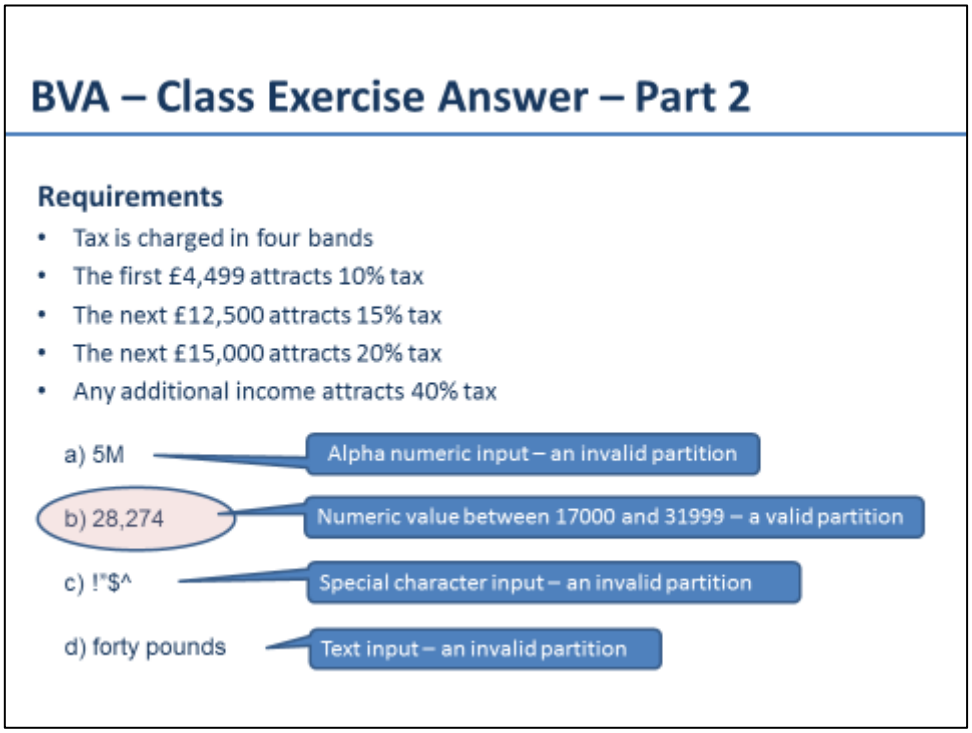


Some points to consider:

- In this example, the boundaries are cumulative.
- Although the upper input limit for the system has been not quantified in the system specifications, from a business process perspective all positive income values should be accepted so the partition for incomes over £31999 is treated as a valid partition.
- Although £4498 would also be a valid test case using the 3-point approach, the corporate test strategy has stated that the 2-point approach must be used, so option b is the correct answer.

However:

- The specification provides no lower limit to the input income range, so while it is reasonable to assume that numbers less than zero are invalid, this should be confirmed with the designers
- The specification provides no upper limit to the input income range. Equally, it is reasonable to assume that at least in theory, there is no technical limit to the size of someone's income. However, from a technical implementation perspective, there will be a practical limit to the income figure, either defined by the number of characters which can be input on the screen or defined by the number of characters which can be stored internally. Exactly what this limit is should be confirmed with the designers.



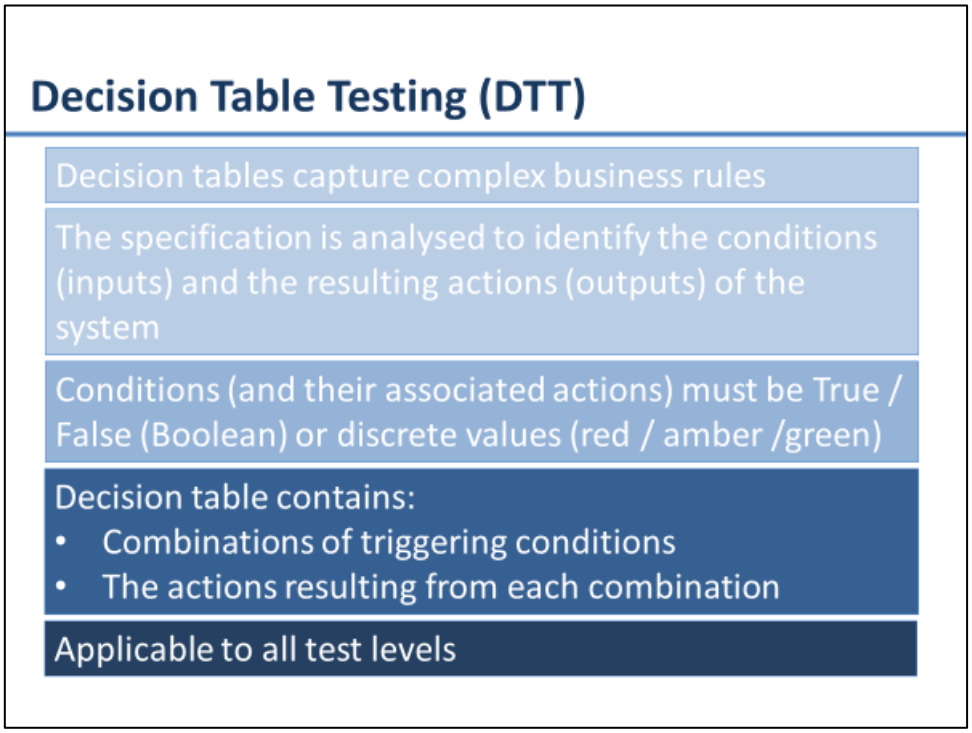
In considering your answer to Part 2 above:

- Three of the possible answers involve invalid partitions
- These tests are examples of typical negative tests implied from the limited details available in the requirement
- The inputs in options a, c and d have all been treated as invalid partitions. Although the specification does not describe what the system response should be, it is reasonable to assume that they should be rejected by the system, but this should be confirmed with the designers.

BVA – Summary

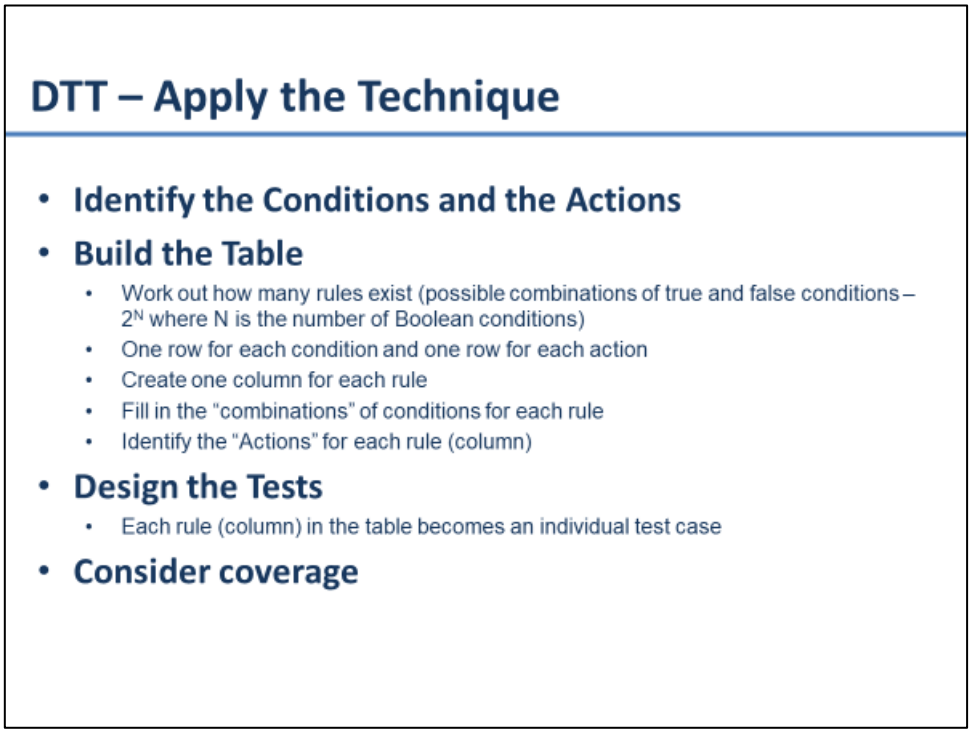
Boundary Value Analysis

- "Extends" Equivalence Partitioning in so far as it recognises that errors tend to occur in the handling of boundary conditions
- Data items selected on the boundaries data partitions can be used both to test for how the system will deal with all such data items, and to identify defects where they are most likely to occur
- The technique is applicable at any test level – Component to Acceptance
- Full coverage is achieved by running tests for each boundary value identified



Decision tables are a good way to record complex business rules that a system must implement. When creating decision tables, the tester identifies conditions (often inputs) and the resulting actions (often outputs) of the system. These form the rows of the table, usually with the conditions at the top and the actions at the bottom. Each column corresponds to a decision rule that defines a unique combination of conditions which results in the execution of the actions associated with that rule. The values of the conditions and actions are usually shown as Boolean values (true or false) or discrete values (e.g., red, green, blue), but can also be numbers or ranges of numbers. These different types of conditions and actions might be found together in the same table.

The strength of decision table testing is that it helps to identify all the important combinations of conditions, some of which might otherwise be overlooked. It also helps in finding any gaps in the requirements. It may be applied to all situations in which the behavior of the software depends on a combination of conditions, at any test level.



The common notation in decision tables is as follows:

For conditions:

- Y means the condition is true (may also be shown as T or 1)
- N means the condition is false (may also be shown as F or 0)
- — means the value of the condition doesn't matter (may also be shown as N/A)

For actions:

- X means the action should occur (may also be shown as Y or T or 1)
- Blank means the action should not occur (may also be shown as – or N or F or 0)

The common minimum coverage standard for decision table testing is to have at least one test case per decision rule in the table. This typically involves covering all combinations of conditions. Coverage is measured as the number of decision rules tested by at least one test case, divided by the total number of decision rules, normally expressed as a percentage.

DTT – Worked Example

Requirements

- The sale of cigarettes and lotto tickets is subject to age restrictions
  - People under the age of 16 are not allowed to buy lotto tickets
  - People under the age of 18 are not allowed to buy cigarettes
  - Proof of age must be provided before a sale can be made

Exercise

- Build the decision table which describes all combinations of possible conditions described above
- Identify the actions arising from each combination
- Identify the tests which will exercise the above requirements fully via a Decision Table

DTT – Worked Answer

Build the Table – Step 1 – What are the conditions and actions?

Condition
C1 - Age < 16
C2 - Age < 18
C3 - Proof of age provided
Actions
A1 - Sell lotto
A2 - Sell cigs

DTT – Worked Answer

Build the Table – Step 2 – How many combinations?

- Three Boolean conditions – so – 2^3 combinations = 8 columns

Condition	1	2	3	4	5	6	7	8
C1 - Age < 16								
C2 - Age < 18								
C3 - Proof of age provided								
Actions								
A1 - Sell lotto								
A2 - Sell cigs								

Remember: You will always have 2^N combinations of conditions where "N" is the number of Boolean conditions, so in this case, with three conditions, you will have 2^3 combinations – that is, 2 x 2 x 2 combinations – and hence, eight columns.

DTT – Worked Answer

Build the Table – Step 3 – Fill in the combinations of conditions

Hence eight tests for complete coverage

Condition	1	2	3	4	5	6	7	8
C1 - Age < 16	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
C2 - Age < 18	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
C3 - Proof of age provided	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Actions								
A1 - Sell lotto								
A2 - Sell cigs								

Remember: In creating "all combinations"

- In the first row, the first half of the row is "true", then the second half is false
- In the next row, the first half of the first row "trues" are "true", the second half are "false", the first half of the first row "falses" are "true", the second half are "false" And so on.

DTT – Worked Answer

Build the Table – Step 4 – complete the actions for each rule

Condition	1	2	3	4	5	6	7	8
C1 - Age < 16	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
C2 - Age < 18	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
C3 - Proof of age provided	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Actions								
A1 - Sell lotto	N	N	?	?	Y	N	Y	N
A2 - Sell cigs	N	N	?	?	N	N	Y	N

And now, add the actions:

- Take each column in turn
- Work out from the specification what the system will do with each combination

Remember:
There's no "mechanical" method to this part – just follow each column in turn.

DTT – Worked Answer

Build the Table – Step 5 – Are these tests feasible?

Hence maybe only 6 rules can become test cases

Condition	1	2	3	4	5	6	7	8
C1 - Age < 16	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
C2 - Age < 18	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
C3 - Proof of age provided	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Actions								
A1 - Sell lotto	N	N	?	?	Y	N	Y	N
A2 - Sell cigs	N	N	?	?	N	N	Y	N

Depending on risk, the table can be collapsed further. These tests contain combinations that do not affect the outcome. Age is irrelevant until there is Proof of age. One 'false' should be included – 8 is the best choice.

A full decision table has enough columns (test cases) to cover every combination of conditions. By deleting columns that do not affect the outcome, the number of test cases can decrease considerably. For example by removing impossible combinations of conditions. For more information on how to collapse decision tables. (see ISTQB-CTAL-AT).

Remember:

- Certain combinations of events may be illogical – in this case, being under sixteen but **not** being under 18 is clearly impossible!
- Infeasible rules are of special interest for negative testing
- Whether these test cases are feasible will depend on the input constraints of the system. For example, if the conditions are represented by a single numeric value, then rules 3 and 4 are definitely infeasible. If however they were input via checkboxes then they may be feasible

DTT – Class Exercise

Requirements

An airline is implementing new rules for how benefits are allocated under their Frequent Flier programme:

- Emerald cards are awarded to any passenger with between 50 and 500 points inclusive
- Diamond cards are awarded to any passenger with over 500 points
- Emerald card holders receive an enhanced baggage allowance
- Diamond card holders receive an enhanced baggage allowance and airline lounge access
- Diamond card holders flying long-haul receive an enhanced baggage allowance, airline lounge access and a one-class cabin upgrade

These rules have been implemented in the decision table on the next slide

Passengers in the frequent flyer program receive a card which accumulates points. The name and benefits of the card change with the number of points.

DTT – Class Exercise

	1	2	3	4	5	6	7	8
C1 – Emerald card holder	T	T	T	T	F	F	F	F
C2 – Diamond card holder	T	T	F	F	T	T	F	F
C3 – Flying long-haul	T	F	T	F	T	F	T	F
A1 – Enhanced baggage	Y	Y	Y	Y	Y	Y	N	N
A2 – Lounge access	Y	Y	N	N	Y	Y	N	N
A3 – One-class upgrade	Y	N	N	N	Y	N	N	N

Part 1
What are the expected results for:
a) Steve – who has 68 points and is flying short-haul
b) Mary – who has 700 points and is flying short-haul
c) Frank – who has 240 points and is flying short-haul

Part 2
a) Which rules are potentially infeasible/illogical?
b) Building on the answers to part 1, how many more tests are needed to cover the feasible tests?

Part 1 –

a)

b)

c)

Part 2 –

a)

b)

Page intentionally left blank

DTT – Class Exercise Answers - Part 1

	1	2	3	4	5	6	7	8
C1 – Emerald card holder	T	T	T	T	F	F	F	F
C2 – Diamond card holder	T	T	F	F	T	T	F	F
C3 – Flying long-haul	T	F	T	F	T	F	T	F
A1 – Enhanced baggage	Y	Y	Y	Y	Y	Y	N	N
A2 – Lounge access	Y	Y	N	N	Y	Y	N	N
A3 – One-class upgrade	Y	N	N	N	Y	N	N	N

Part 1
What are the expected results for:
a) Steve – who has 68 points and is flying short-haul
b) Mary – who has 700 points and is flying short-haul
c) Frank – who has 240 points and is flying short-haul

Part 1 answers
a) Enhanced baggage allowance (rule 4)
b) Enhanced baggage allowance plus lounge access (rule 6)
c) Enhanced baggage allowance (rule 4)

DTT – Class Exercise Answers - Part 2

	1	2	3	4	5	6	7	8
C1 – Emerald card holder	T	T	T	T	F	F	F	F
C2 – Diamond card holder	T	T	F	F	T	T	F	F
C3 – Flying long-haul	T	F	T	F	T	F	T	F
A1 – Enhanced baggage	Y	Y	Y	Y	Y	Y	N	N
A2 – Lounge access	Y	Y	N	N	Y	Y	N	N
A3 – One-class upgrade	Y	N	N	N	Y	N	N	N

Part 2
a) Which rules are potentially infeasible/illogical?
b) How many more tests are needed for full coverage

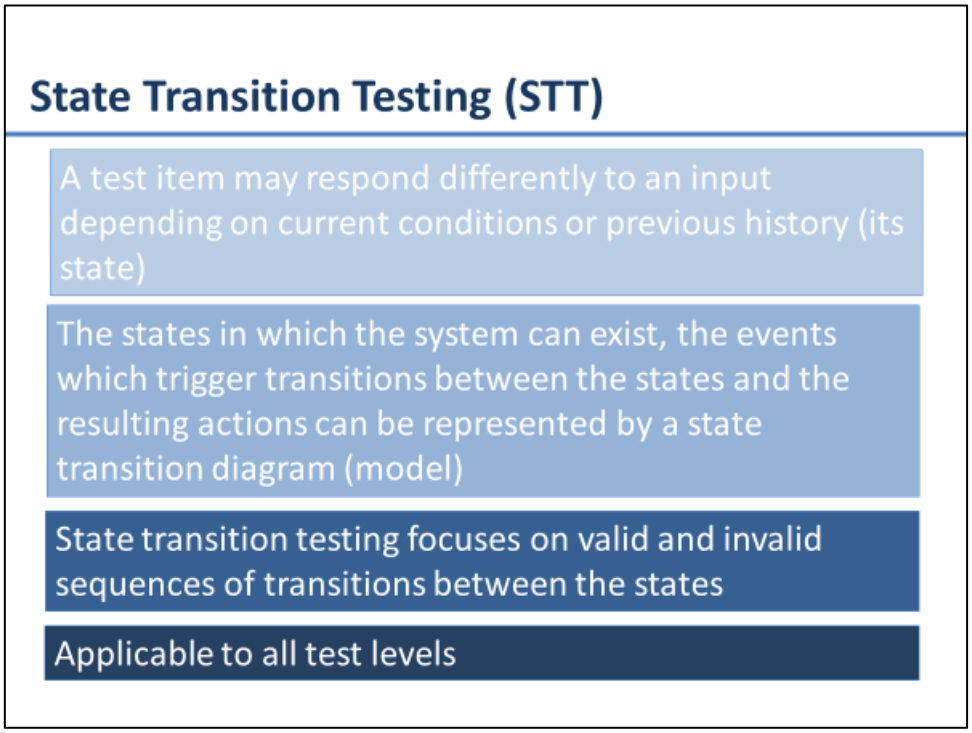
Part 2 answers
a) Rules 1 and 2
b) Four additional tests, covering rules 3, 5, 7 and 8 if we omit the infeasible rules (Rules 1 and 2)

Although it is not possible to make a definitive decision without checking the logic with the business analysts, it is logical to assume that combinations 1 and 2 are likely to be invalid.

DTT – Summary

Decision Table Testing

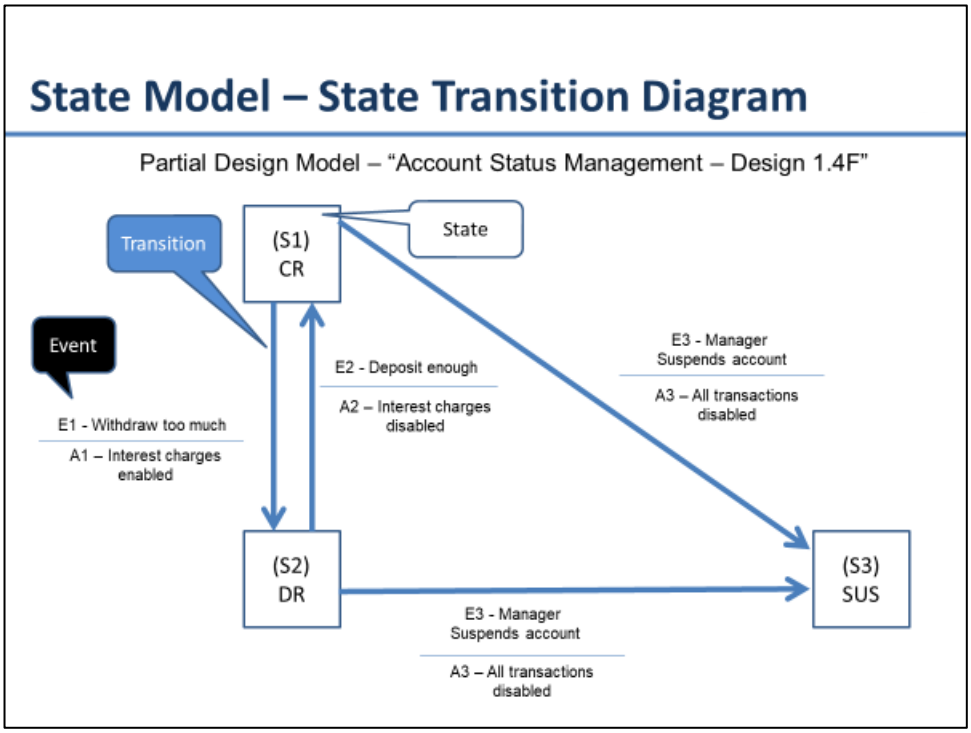
- Tests the outcome of applying two or more “conditions” simultaneously
- Each “condition” must lead to a “Yes / No” decision
- May highlight the situation where the outcome of applying two conditions is different from the “simple addition” of the effects of each condition
- The technique is applicable at any test level – Component to Acceptance
- Full coverage is achieved by running one test for every rule - 2^N tests where “N” is the number of Boolean conditions being considered



Components or systems may respond differently to an event depending on current conditions or previous history (e.g., the events that have occurred since the system was initialized). The previous history can be summarized using the concept of states.

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

State transition testing is used for menu-based applications and is widely used within the embedded software industry. The technique is also suitable for modeling a business scenario having specific states or for testing screen navigation. The concept of a state is abstract – it may represent a few lines of code or an entire business process.

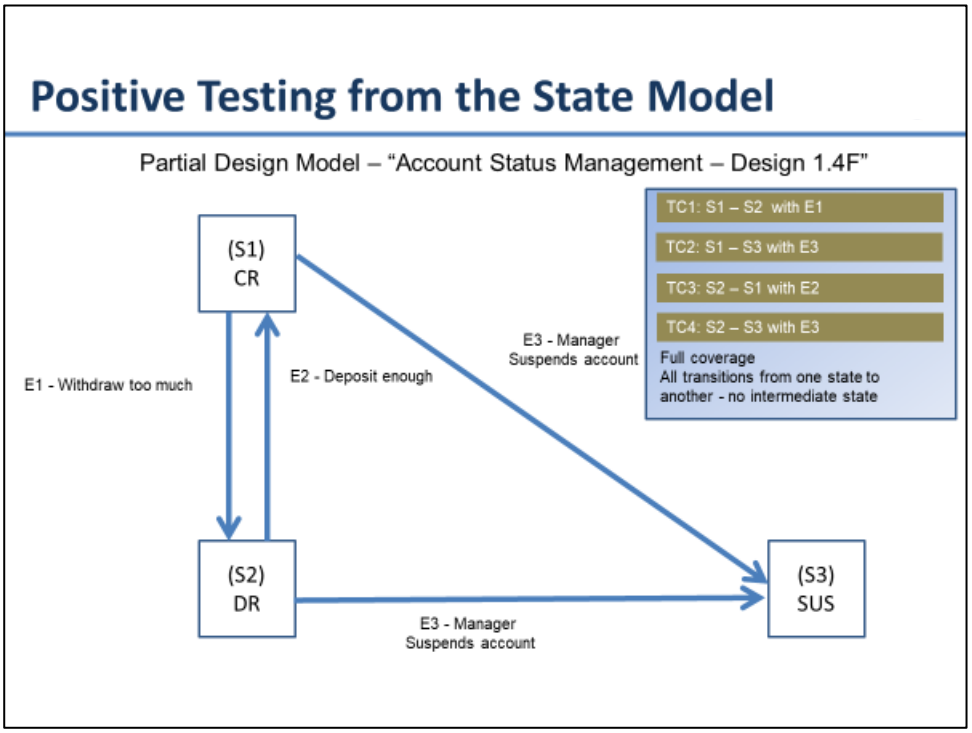


A state transition diagram shows the possible software states, as well as how the software enters, exits, and transitions between states. A transition is initiated by an event (e.g., user input of a value into a field). The event results in a transition. The same event can result in two or more different transitions from the same state. The state change may result in the software taking an action (e.g., outputting a calculation or error message).

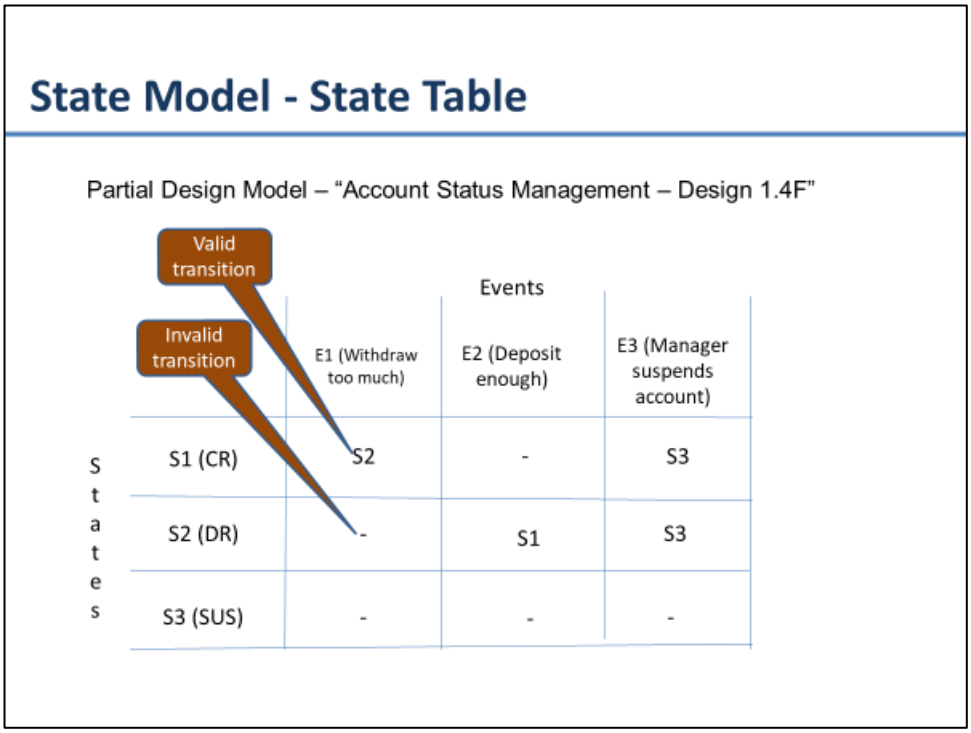
Tests can be designed to cover a typical sequence of states, to exercise all states, to exercise every transition, to exercise specific sequences of transitions, or to test invalid transitions.

Remember:

- Positive tests can be constructed to exercise all valid transitions in any combination
- Negative tests can be constructed to confirm that ONLY valid transitions are possible
- The diagram highlights internal and external actions
  - Internal - those changes "visible" only within the system itself
  - External - those changes observable from outside the system
  - For example, that a nasty letter is produced is an external action, observable outside the system, whereas that the account flag status and the charging of interest will require internal investigation to be disclosed.



Taking each state as a "start state" in turn, it then becomes clear that full coverage will produce four individual tests.



This diagram shows another way in which single transitions can be represented – in this case, as a "State table", also known as a "State Transition Table".

A state transition table shows all valid transitions and potentially invalid transitions between states, as well as the events, and resulting actions for valid transitions. State transition diagrams normally show only the valid transitions and exclude the invalid transitions.

Remember:
In reading this table:

- Find the "starting state" in the left hand column – so defining which row of the table should be read
- Then go along the row to the column corresponding to the event being applied
- If there's a state in that cell, then:
    - It's a valid transition - That's the state the system will be in when that event has been applied
- If there's anything else in that cell – for example, a space, "Null" or "-"
    - It's an invalid transition! The system should "respond" by ignoring the event and making no change

Alternative notations for how valid and invalid transitions might be represented in this table in a state table include:

- "Finish State/Action" for a valid transition
- "Start State/Null" for an invalid transition

Positive Testing from the State Model

TC1: S1 – S2 with E1
TC2: S1 – S3 with E3
TC3 S2 – S1 with E2
TC4: S2 – S3 with E3

States	Events: E1 (Withdraw too much)	E2 (Deposit enough)	E3 (Manager suspends account)
S1 (CR)	S2	-	S3
S2 (DR)	-	S1	S3
S3 (SUS)	-	-	-

So by interpreting the above table to identify all the tests required for Positive testing of the state model, we would get four test cases, as shown above.

Negative Testing from the State Model

TC6: Attempt E2 from S1
TC7: Attempt E1 from S2
TC8: Attempt E1 from S3
TC9: Attempt E2 from S3
TC10: Attempt E3 from S3

States	Events: E1 (Withdraw too much)	E2 (Deposit enough)	E3 (Manager suspends account)
S1 (CR)	S2	-	S3
S2 (DR)	-	S1	S3
S3 (SUS)	-	-	-

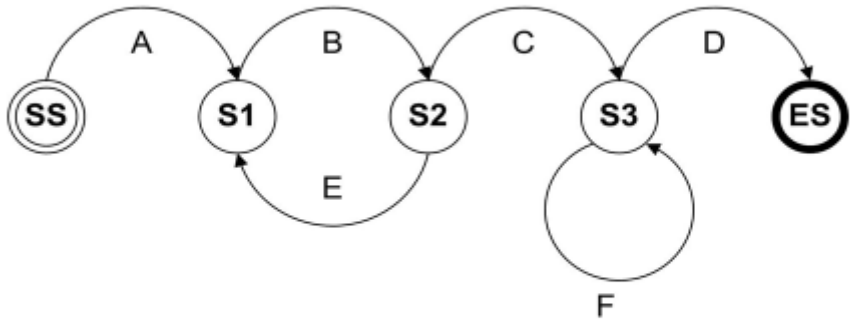
Similarly, by interpreting the above table to identify all the tests required for Negative testing of the state model, we would get five test cases, as shown above.

STT – Apply the Technique

- **Build the state model**
  - Recognise the states, events and transitions as represented in the specifications, state transition diagrams and state tables.
  - Use the state transition diagram to identify the valid transitions
  - Use the state table to identify the potential invalid transitions (optional)
- **Design the Tests**
  - Each state transition test case is a valid transition between one state and another
  - For negative testing try to induce all the invalid transitions implied by the state table
- **Consider coverage**

Coverage is commonly measured as the number of identified states or transitions tested, divided by the total number of identified states or transitions in the test object, normally expressed as a percentage. For more information on coverage criteria for state transition testing, (see ISTQB-CTAL-AT).

STT – Class Exercise 1



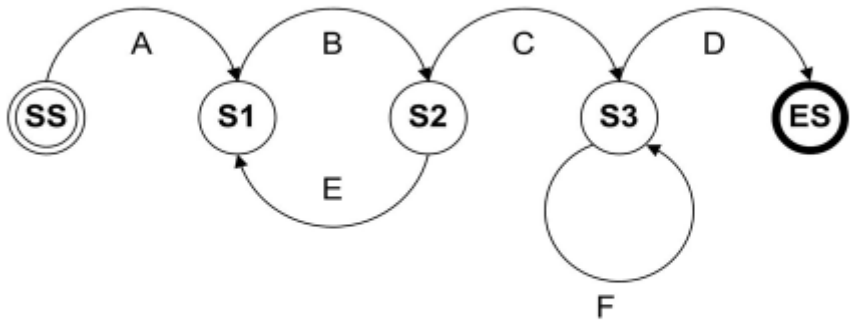
Which one of the following tests is a valid positive test:

a) SS – S3
b) S3 – S3
c) S3 – S2
d) S1 – SS

Space for you to note down your answer

Page intentionally left blank

STT – Class Exercise 1 Answer

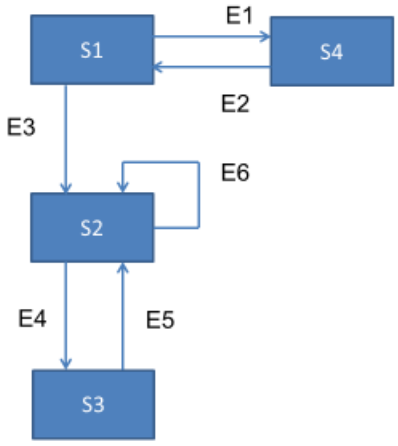


Which one of the following tests is a valid positive test:

- ✗ a) SS – S3
- ✓ b) S3 – S3
- ✗ c) S3 – S2
- ✗ d) S1 – SS

a) In this case, there's no direct transition from state SS to S3
b) Correct answer
c) In this case, there's no transition from state S3 to S2, only from S2 to S3
d) In this case, there's no transition from state S1 to SS, only from SS to S1

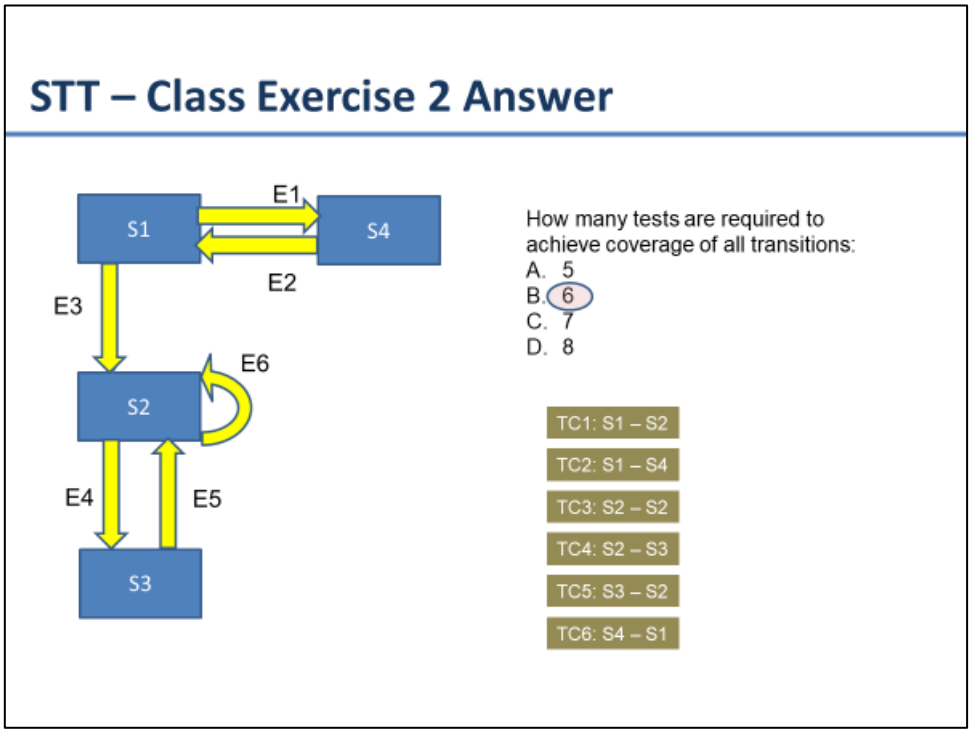
STT – Class Exercise 2



How many tests are required to achieve coverage of all transitions:

A. 5
B. 6
C. 7
D. 8

Space for you to note down your answer

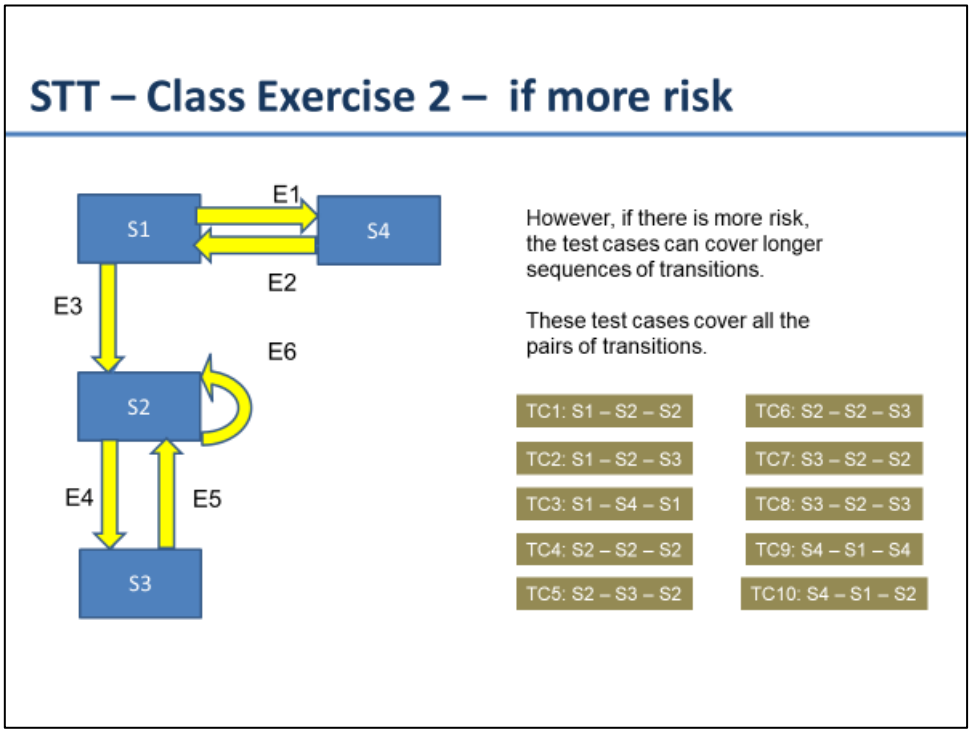


A set of test cases that cover all the possible individual transitions in a state transition model will achieve 0-switch coverage (also known as Chow 0 coverage).

A typical 0-switch test case usually consists of a single transition between a start state and a finish state. Additional test cases are added until every possible transition has been covered. The set then can be seen to achieve 100% 0-switch coverage.

Test Case ID	TC1	TC2	TC3	TC4	TC5	TC6
Start State	State 1	State 1	State 2	State 2	State 3	State 4
Event	Event 3	Event 1	Event 6	Event 4	Event 5	Event 2
Action*	Action 2	Action 4	Action 2	Action 3	Action 2	Action 1
Finish State	State 2	State 4	State 2	State 3	State 2	State 1

*Note that the actions have not been explicitly shown in the state transition diagram above but are added to this test case table for completeness on the basis that each state has a corresponding action with the same ID number.



A set of test cases that cover all the possible pairs of transitions in a state transition model will achieve 1-switch coverage (also known as Chow 1 coverage).

A typical 1-switch test case usually consists of sequence of 2 transitions between a start state, a next (or middle) state and a finish state. Additional test cases are added until every possible pair of transition has been covered. The set then can be seen to achieve 100% 1-switch coverage.

Test Case ID	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10
Start State	S1	S1	S1	S2	S2	S2	S3	S3	S4	S4
Event	E3	E3	E1	E6	E4	E6	E5	E5	E2	E2
Action*	A2	A2	A4	A2	A3	A2	A2	A2	A1	A1
Next State	S2	S2	S4	S2	S3	S2	S2	S2	S1	S1
Event	E6	E4	E2	E6	E5	E4	E6	E4	E1	E3
Action*	A2	A3	A1	A2	A2	A3	A2	A3	A4	A2
Finish State	S2	S3	S1	S2	S2	S3	S2	S3	S4	S2

*Note that the actions have not been explicitly shown in the state transition diagram above but are added to this test case table for completeness (on the basis that each state has a corresponding action with the same ID number)

STT – Class Exercise 3

	E1	E2	E3	E4	E5	E6
S1	S4		S2			
S2				S3		S2
S3					S2	
S4		S1				

Pick **two** from a-e which exercise invalid transitions:

a) E1 from S1, E5 from S3
b) E4 from S3, E6 from S1
c) E1 from S4, E3 from S1
d) E4 from S1, E5 from S4
e) E4 from S2, E6 from S2

Space for you to note down your answer

Page intentionally left blank

STT – Class Exercise 3 Answer

	E1	E2	E3	E4	E5	E6
S1	S4		S2			
S2				S3		S2
S3					S2	
S4		S1				

Pick **two** from a-e which exercise invalid transitions:

- ✗ a) E1 from S1, E5 from S3
- ✓ b) E4 from S3, E6 from S1
- ✗ c) E1 from S4, E3 from S1
- ✓ d) E4 from S1, E5 from S4
- ✗ e) E4 from S2, E6 from S2

b) and d) are the correct answers

a) Event E1 applied to state S1 moves the system to state S4
Event E5 applied to state S3 moves the system to state S2

c) Event E1 applied to state S4 exercises an invalid transition
Event E3 applied to state S1 moves the system to state S2

e) Event E4 applied to state S2 moves the system to state S3
Event E6 applied to state S2 moves the system to state S2

STT– Summary

State Transition Testing

- Tests the system's transitions from one "state" to another or itself, confirming that:
  - all valid transitions are possible
  - only valid transitions are possible
- State models may be represented by State Transition Diagrams and State Tables
- Positive testing is measured using coverage
- Negative tests are derived from the State Table
- The technique is applicable at any test level – Component to Acceptance

Use Case Testing

- Use cases:
  - A way to capture requirements and design interactions with software items
  - Have a primary path and may have multiple alternative paths, each describing a process flow
  - Use case testing looks for defects in process flows and can also disclose integration issues, for example, between components and systems
  - Can include possible variations of its basic behaviour including exceptions and error handling
  - Coverage is measured as a percentage of use case behaviours (basic, exceptional/alternative and error handling)

Tests can be derived from use cases, which are a specific way of designing interactions with software items. They incorporate requirements for the software functions. Use cases are associated with actors (human users, external hardware, or other components or systems) and subjects (the component or system to which the use case is applied).

A use case can include possible variations of its basic behavior, including exceptional behavior and error handling (system response and recovery from programming, application and communication errors, e.g., resulting in an error message). Tests are designed to exercise the defined behaviors (basic, exceptional or alternative, and error handling). Coverage can be measured by the number of use case behaviors tested divided by the total number of use case behaviors, normally expressed as a percentage.

For more information on coverage criteria for state transition testing, (see ISTQB-CTAL-AT).

Use Case Testing

- Actors
  - Can be people, external hardware or other components/systems
- Subject
  - The system under test (SUT) to which the use case is applied
- Interactions between actors and the subject
  - Will exercise defined behaviours (basic, exceptional/alternative and error handling)
  - May result in state changes to the SUT

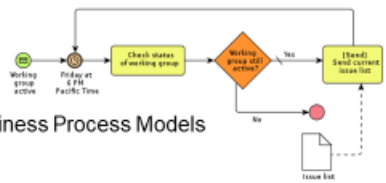
Each use case specifies some behavior that a subject can perform in collaboration with one or more actors (UML 2.5.1 2017). A use case can be described by interactions and activities, as well as preconditions, postconditions and natural language where appropriate. Interactions between the actors and the subject may result in changes to the state of the subject. Interactions may be represented graphically by work flows, activity diagrams, or business process models.

Use Case Testing

- Use cases:
  - Interactions between actors and the SUT can be represented in different ways:

Graphical work flows

Business Process Models



Activity diagrams

Use Case Testing – Summary

Use Case Testing

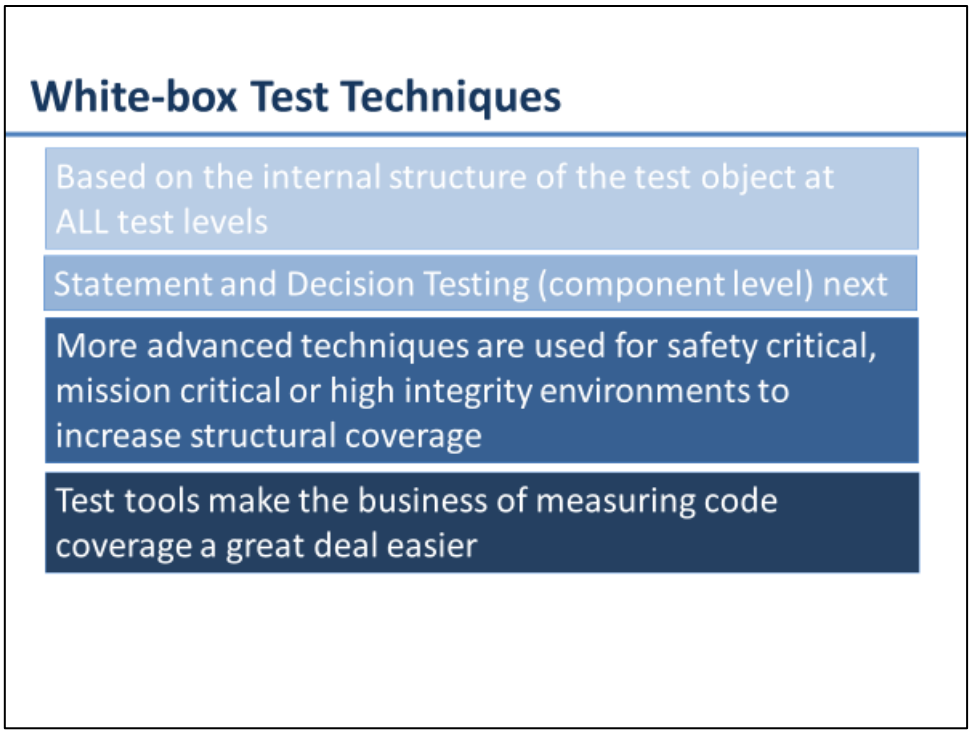
- Use Cases describe the intended behaviours of a system
- Coverage is measured as a percentage of use case behaviours (basic, exceptional/alternative and error handling)
- The technique can be combined with other techniques

Summary

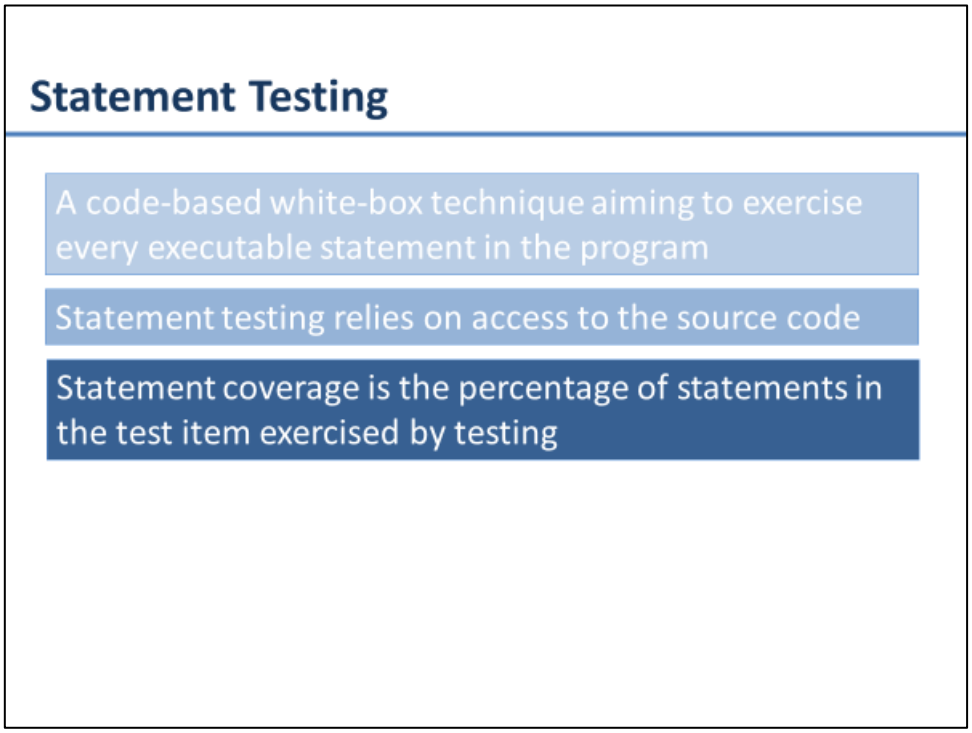
Black-Box Test Techniques

- Include:
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Decision Tables
  - State Transition Diagrams / Tables
  - Use Case Testing
- Rely on an understanding of the intended behaviours of the system
- Require no knowledge of how the system is implemented

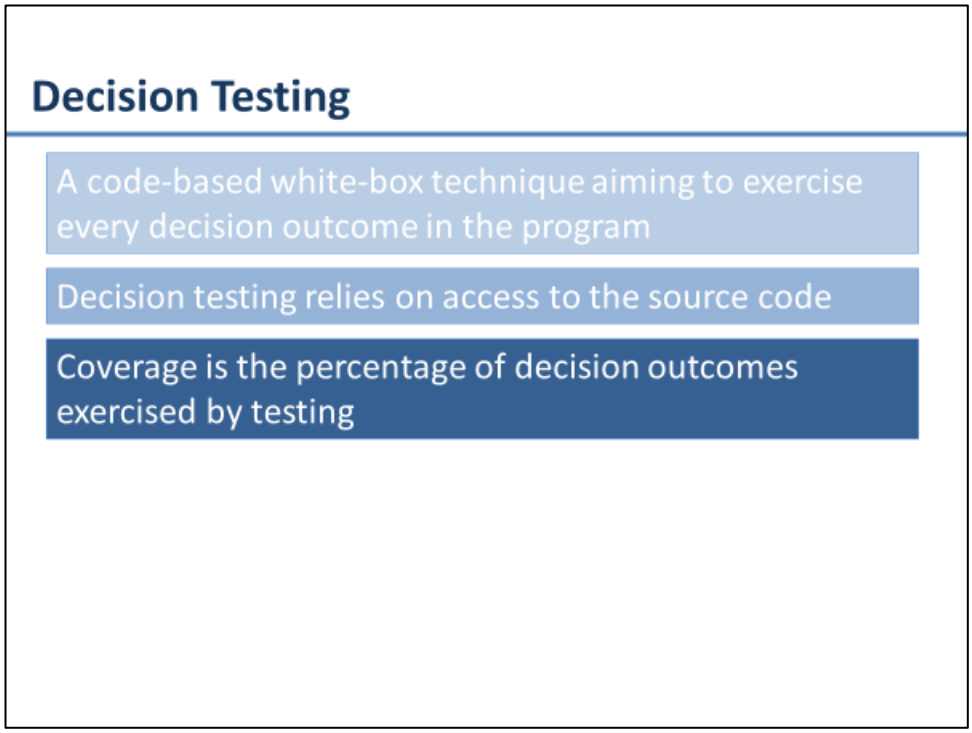
Section 4.3 - White-box Test Techniques



White-box testing is based on the internal structure of the test object. White-box test techniques can be used at all test levels, but the two code-related techniques discussed in this section are most commonly used at the component test level. There are more advanced techniques that are used in some safety-critical, mission-critical, or high integrity environments to achieve more thorough coverage, but those are not discussed here. For more information on such techniques, see the ISTQB-CTAL-TTA.



Statement testing exercises the potential executable statements in the code. Coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage.

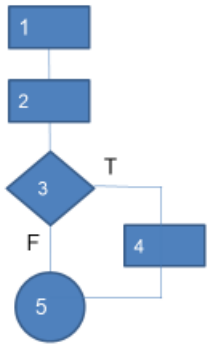


Decision testing exercises the decisions in the code and tests the code that is executed based on the decision outcomes. To do this, the test cases follow the control flows that occur from a decision point (e.g., for an IF statement, one for the true outcome and one for the false outcome; for a CASE statement, test cases would be required for all the possible outcomes, including the default outcome). Coverage is measured as the number of decision outcomes executed by the tests divided by the total number of decision outcomes in the test object, normally expressed as a percentage.

Statement Coverage vs Decision Coverage

Consider the code

```
01 READ A
02 READ B
03 IF A > B THEN
04     PRINT "A IS BIG"
05 ENDIF
```



When 100% statement coverage is achieved, it ensures that all executable statements in the code have been tested at least once, but it does not ensure that all decision logic has been tested. Of the two white-box techniques discussed in this syllabus, statement testing may provide less coverage than decision testing.

When 100% decision coverage is achieved, it executes all decision outcomes, which includes testing the true outcome and also the false outcome, even when there is no explicit false statement (e.g., in the case of an IF statement without an else in the code). Statement coverage helps to find defects in code that was not exercised by other tests. Decision coverage helps to find defects in code where other tests have not taken both true and false outcomes.

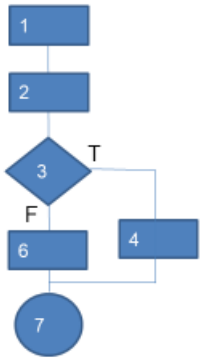
Achieving 100% decision coverage guarantees 100% statement coverage (but not vice versa).

In the example above, only the path covered by the “True” decision outcome includes executable statements, so only one test is needed to achieve 100% statement coverage. However two tests are needed to ensure both the “True” and “False” decision outcomes are covered.

Statement Coverage vs Decision Coverage

Consider the code

```
01 READ A
02 READ B
03 IF A > B THEN
04     PRINT "A IS BIG"
05 ELSE
06     PRINT "B IS BIG"
07 ENDIF
```



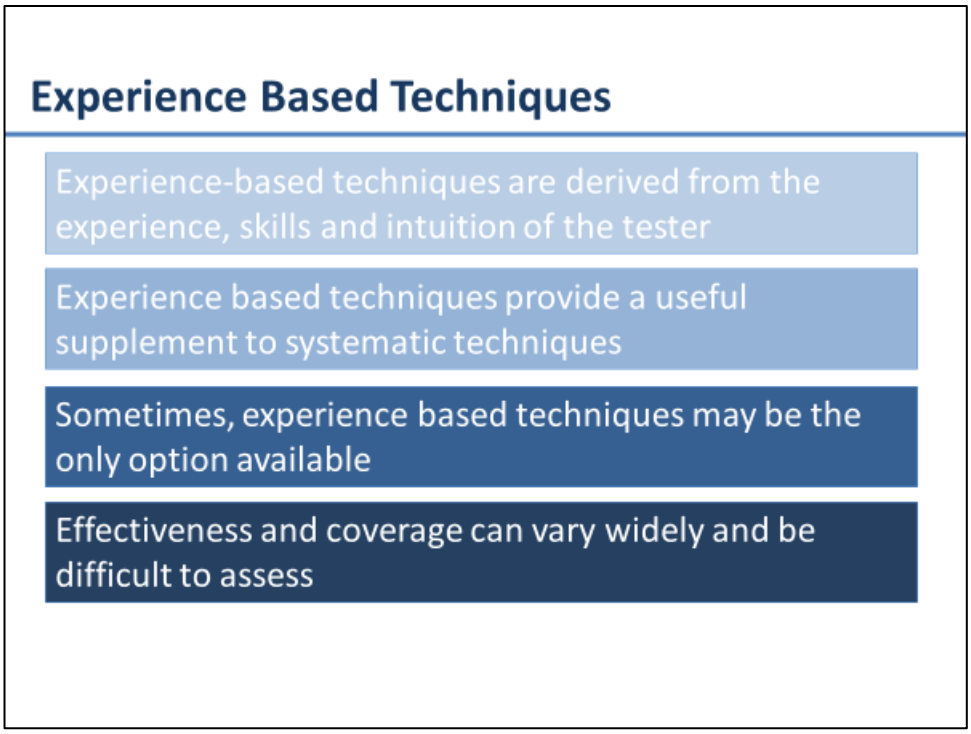
In the example above, the paths covered by each decision outcome include executable statements, so the same number of tests are needed for both statement and decision coverage.

Summary

White-box Test Techniques

- Include:
  - Statement Coverage (Testing)
  - Decision Coverage (Testing)
- Rely on access to the source code
- Decision Coverage is more rigorous than Statement Coverage
- Coverage metrics can be calculated for code-based techniques, describing the percentage of statements and decision outcomes actually executed during testing

Section 4.4 – Experience-based Test Techniques



When applying experience-based test techniques, the test cases are derived from the tester's skill and intuition, and their experience with similar applications and technologies. These techniques can be helpful in identifying tests that were not easily identified by other more systematic techniques. Depending on the tester's approach and experience, these techniques may achieve widely varying degrees of coverage and effectiveness. Coverage can be difficult to assess and may not be measurable with these techniques.

Commonly used experience-based techniques are discussed in the following sections.

Remember:
Experience Based techniques are not without a process altogether – but the processes have not been formally defined. These techniques operate on guidelines rather than rules and rely heavily on the experience of the individual tester for how these guidelines are applied.

Error Guessing

- A negative testing approach
- Testers anticipate failures based on:
  - past experience of application
  - typical errors made
  - failures in other applications

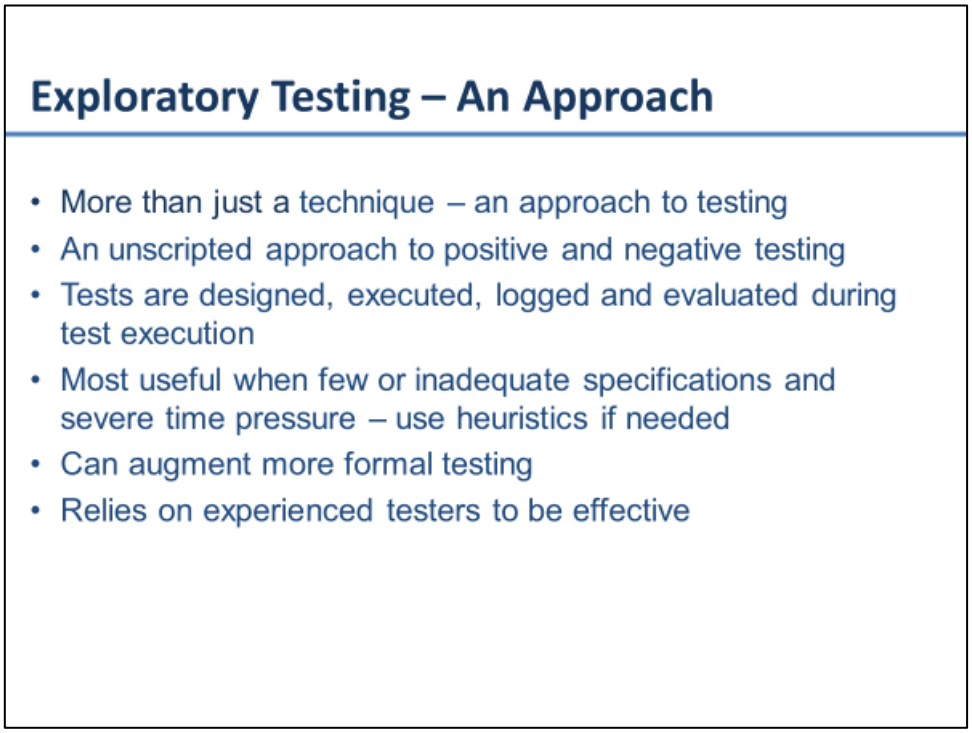
Error guessing is a technique used to anticipate the occurrence of errors, defects, and failures, based on the tester's knowledge, including:

- How the application has worked in the past
- What kind of errors tend to be made
- Failures that have occurred in other applications

Error Guessing – a Methodical Approach

- Create a list of possible errors, defects and failures based on:
  - Experience of similar systems and projects
  - Past history of defects and failures identified in the system
  - Commonly held knowledge and experience of how and why systems fail
- Design tests to expose these possible failures

A methodical approach to the error guessing technique is to create a list of possible errors, defects, and failures, and design tests that will expose those failures and the defects that caused them. These error, defect, failure lists can be built based on experience, defect and failure data, or from common knowledge about why software fails.



In exploratory testing, informal (not pre-defined) tests are designed, executed, logged, and evaluated dynamically during test execution. The test results are used to learn more about the component or system, and to create tests for the areas that may need more testing.

Exploratory testing is most useful when there are few or inadequate specifications or significant time pressure on testing. Exploratory testing is also useful to complement other more formal testing techniques.

Exploratory testing is strongly associated with reactive test strategies (see section 5.2.2). Exploratory testing can incorporate the use of other black-box, white-box, and experience-based techniques.

Exploratory Testing – Session-Based approach

- Identify test charter
  - These are the test objectives for the exploratory test session
- Decide the time-box duration
- Start the session
  - Design and run the initial tests
  - Observe what happens and use observations to
    - Log the test result
    - Design the next test based on heuristics
- Stop when the time-box limit is reached
- Debrief developer/test leader on what was found in the session and how much of the charter was covered

Exploratory testing is sometimes conducted using session-based testing to structure the activity. In session-based testing, exploratory testing is conducted within a defined time-box, and the tester uses a test charter containing test objectives to guide the testing. The tester may use test session sheets to document the steps followed and the discoveries made.

Checklist-Based Testing

- Design tests to exercise checklists - existing or new
- Based on:




- Functional and non-functional
- Can provide guidelines and a degree of consistency when no detailed test cases exist
- High-level lists can introduce some variations, so potentially greater coverage but less repeatability

In checklist-based testing, testers design, implement, and execute tests to cover test conditions found in a checklist. As part of analysis, testers create a new checklist or expand an existing checklist, but testers may also use an existing checklist without modification. Such checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails.

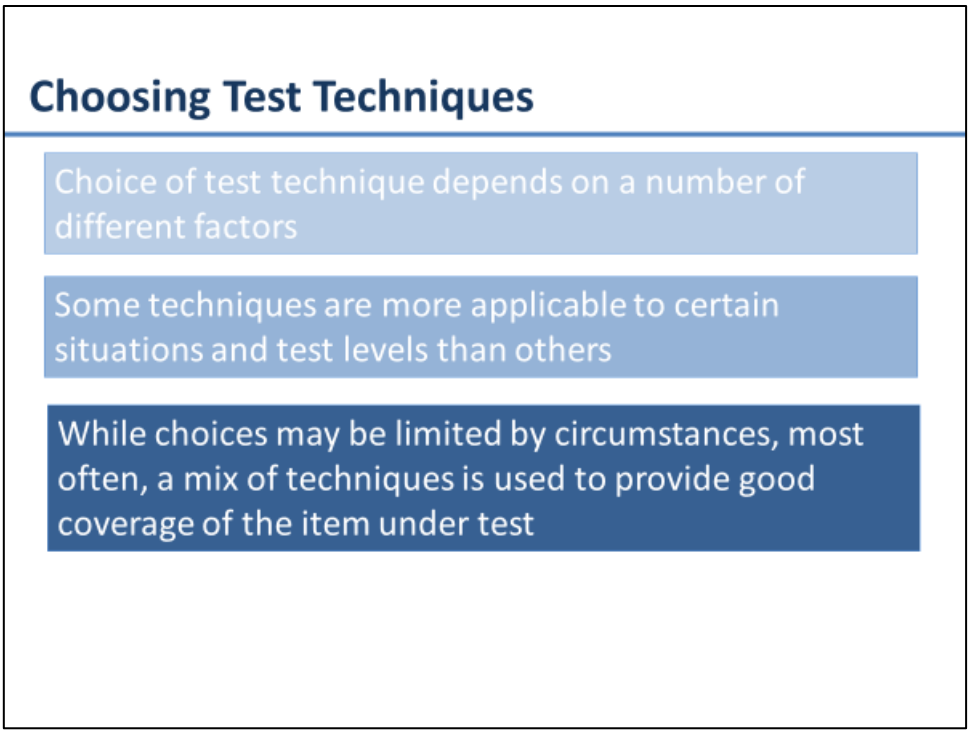
Checklists can be created to support various test types, including functional and non-functional testing. In the absence of detailed test cases, checklist-based testing can provide guidelines and a degree of consistency. As these are high-level lists, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

Summary

Experience-Based Techniques

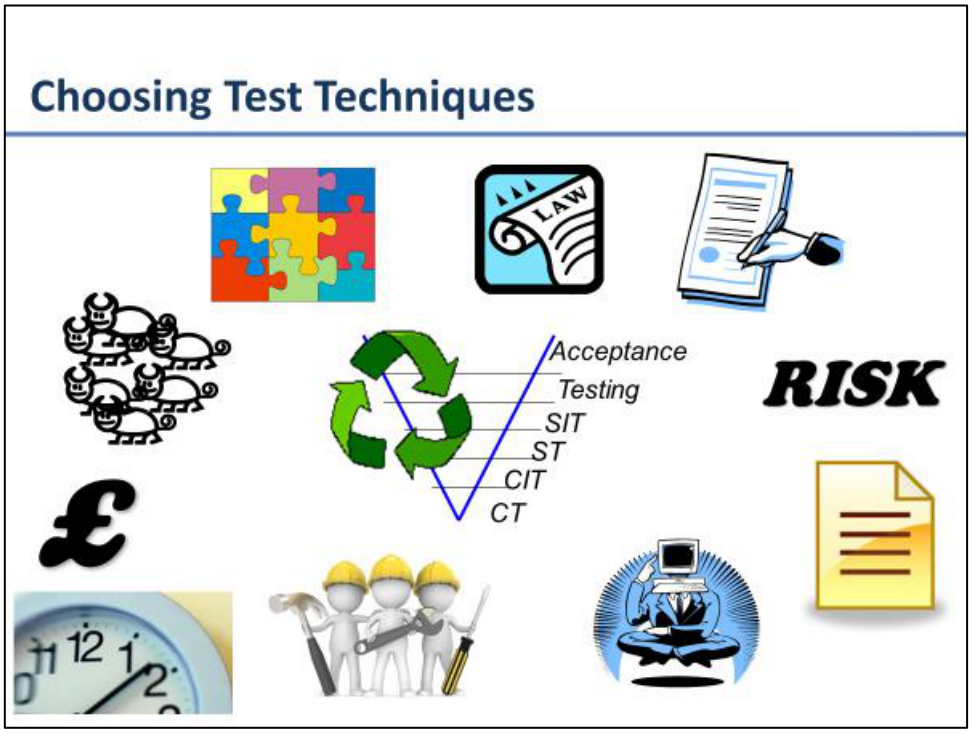
- Include:
  - Error Guessing
  - Exploratory Testing
  - Checklist-based Testing
- Can be used to supplement the other test techniques
- Can be used when no documentation available
- Can be used when limited time available
- Relies on testers' experience to be effective

Section 4.5 - Choosing Test Techniques



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 to achieve the best results from the test effort.

The use of test techniques in the test analysis, test design, and test implementation activities can range from very informal (little to no documentation) to very formal. The appropriate level of formality depends on the context of testing, including the maturity of test and development processes, time constraints, safety or regulatory requirements, the knowledge and skills of the people involved, and the software development lifecycle model being followed.



The purpose of a test technique, including those discussed in this section, is to help in identifying test conditions, test cases, and test data. The choice of which test techniques to use depends on a number of factors, including:

- Component or system complexity
- Regulatory standards
- Customer or contractual requirements
- Risk levels and types
- Available documentation
- Tester knowledge and skills
- Available tools
- Time and budget
- Software development lifecycle model
- The types of defects expected in the component or system

Summary

The choice of test techniques is influenced by many factors

It is essential to ensure that the right combination of test techniques is selected in each case

End of section exercise

- Attempt the following Questions from Sample Exam B
- Questions 19 to 29
- Please mark your own (See Sample Exam B Answers) and read the answer justifications as needed

You have 16.5 minutes

Learning Objectives for Test Techniques

In order to complete this section, you should satisfy yourself that you are able to understand or perform the following items to the standard indicated by the corresponding “K - learning level”. For an explanation of “K levels”, please see the course introduction section – “Learning Objectives and Levels of Knowledge”.

Keywords

black-box test technique, boundary value analysis, checklist-based testing, coverage, decision coverage, decision table testing, error guessing, equivalence partitioning, experience-based test technique, exploratory testing, state transition testing, statement coverage, test technique, use case testing, white-box test technique

Learning Objectives for Test Techniques

4.1 Categories of Test Techniques

FL-4.1.1 (K2) Explain the characteristics, commonalities, and differences between black-box test techniques, white-box test techniques, and experience-based test techniques

4.2 Black-box Test Techniques

FL-4.2.1 (K3) Apply equivalence partitioning to derive test cases from given requirements
FL-4.2.2 (K3) Apply boundary value analysis to derive test cases from given requirements
FL-4.2.3 (K3) Apply decision table testing to derive test cases from given requirements
FL-4.2.4 (K3) Apply state transition testing to derive test cases from given requirements
FL-4.2.5 (K2) Explain how to derive test cases from a use case

4.3 White-box Test Techniques

FL-4.3.1 (K2) Explain statement coverage
FL-4.3.2 (K2) Explain decision coverage
FL-4.3.3 (K2) Explain the value of statement and decision coverage

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

Test Management

Table of Contents

5.1 Test Organization

Exercise – Test Independence

Working in pairs, arrange these approaches as a list with the most independent option first and the least independent last

A. Independent testers from the business/user community reporting to the development manager

B. Outsourced independent testers

C. No independent testers, developers test their own code

D. Independent testers / developers testing within the development teams

E. Independent test group reporting to executive management

F. Independent test specialists for usability testing reporting to the project manager

Suggested answer (for discussion) comes next...

You have 2 minutes

Space for you to note down your answers

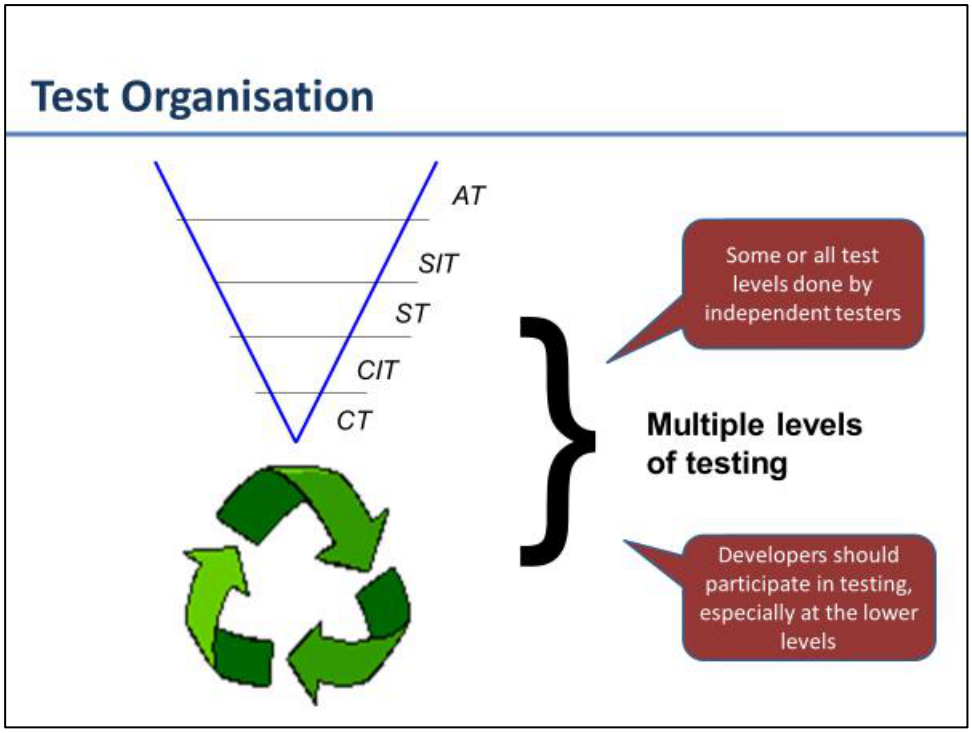
Page intentionally left blank

Suggested answer for discussion purposes:

B. Outsourced independent testers
E. Independent test group reporting to executive management
F. Independent test specialists for usability testing reporting to the project manager
A. Independent testers from the business/user community reporting to the development manager
D. Independent testers / developers testing within the development teams
C. No independent testers, developers test their own code

Degrees of independence in testing include the following (from low level of independence to high level):

- No independent testers; the only form of testing available is developers testing their own code
- Independent developers or testers within the development teams or the project team; this could be developers testing their colleagues' products
- Independent test team or group within the organization, reporting to project management or executive management
- Independent testers from the business organization or user community, or with specializations in specific test types such as usability, security, performance, regulatory/compliance, or portability
- Independent testers external to the organization, either working on-site (in-house) or off-site (outsourcing



Testing tasks may be done by people in a specific testing role, or by people in another role (e.g., customers). A certain degree of independence often makes the tester more effective at finding defects due to differences between the author's and the tester's cognitive biases (see section 1.5). Independence is not, however, a replacement for familiarity, and developers can efficiently find many defects in their own code.

For most types of projects, it is usually best to have multiple test levels, with some of these levels handled by independent testers. Developers should participate in testing, especially at the lower levels, so as to exercise control over the quality of their own work.

The way in which independence of testing is implemented varies depending on the software development lifecycle model. For example, in Agile development, testers may be part of a development team. In some organizations using Agile methods, these testers may be considered part of a larger independent test team as well. In addition, in such organizations, product owners may perform acceptance testing to validate user stories at the end of each iteration.

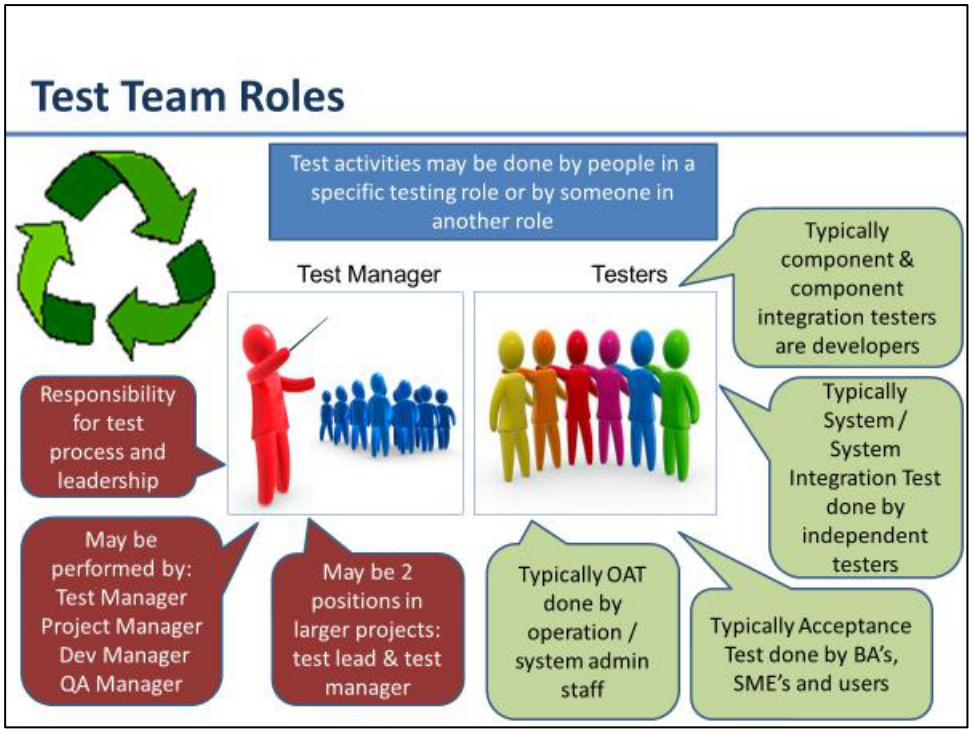
Potential benefits of test independence include:

- Independent testers are likely to recognize different kinds of failures compared to developers because of their different backgrounds, technical perspectives, and biases
- An independent tester can verify, challenge, or disprove assumptions made by stakeholders during specification and implementation of the system
- Independent testers of a vendor can report in an upright and objective manner about the system under test without (political) pressure of the company that hired them

Potential drawbacks of test independence include:

- Isolation from the development team, may lead to a lack of collaboration, delays in providing feedback to the development team, or an adversarial relationship with the development team
- Developers may lose a sense of responsibility for quality
- Independent testers may be seen as a bottleneck
- Independent testers may lack some important information (e.g., about the test object)

Many organizations are able to successfully achieve the benefits of test independence while avoiding the drawbacks.

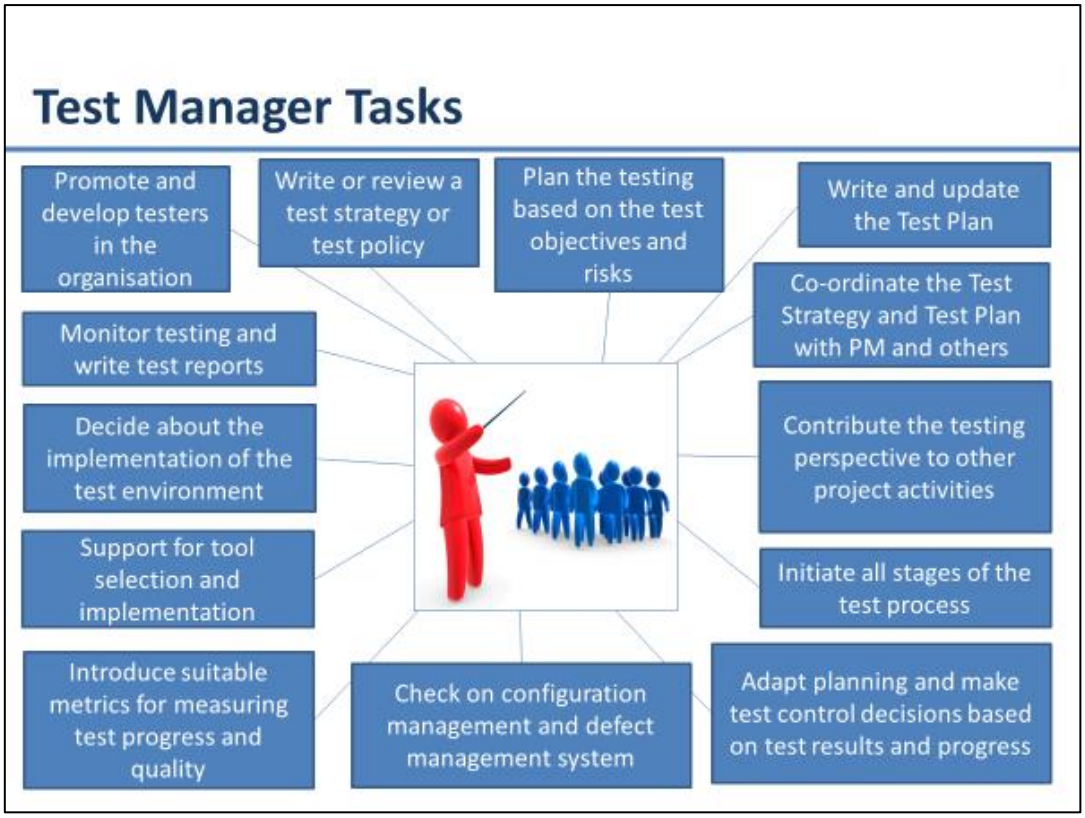


In this syllabus, two test roles are covered, test managers and testers. The activities and tasks performed by these two roles depend on the project and product context, the skills of the people in the roles, and the organization.

The test manager is tasked with overall responsibility for the test process and successful leadership of the test activities. The test management role might be performed by a professional test manager, or by a project manager, a development manager, or a quality assurance manager. In larger projects or organizations, several test teams may report to a test manager, test coach, or test coordinator, each team being headed by a test leader or lead tester.

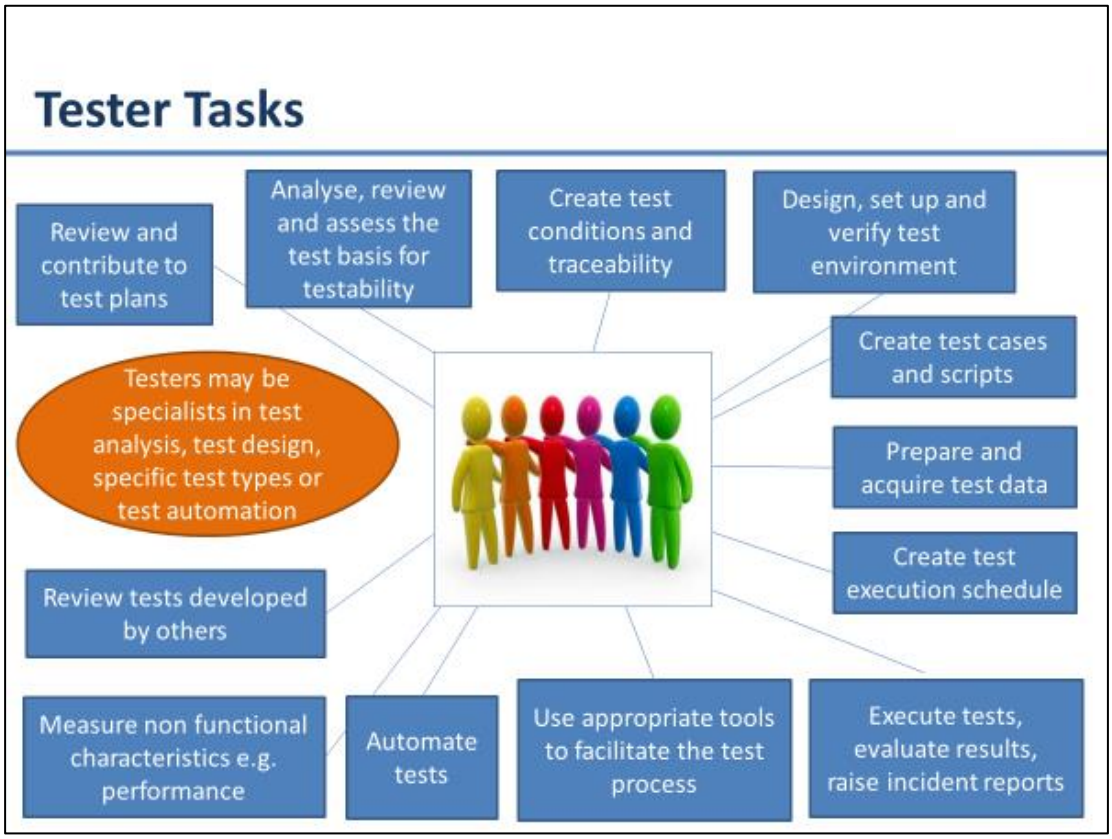
The way in which the test manager role is carried out varies depending on the software development lifecycle. For example, in Agile development, some of the tasks mentioned above are handled by the Agile team, especially those tasks concerned with the day-to-day testing done within the team, often by a tester working within the team. Some of the tasks that span multiple teams or the entire organization, or that have to do with personnel management, may be done by test managers outside of the development team, who are sometimes called test coaches. See Black 2009 for more on managing the test process.

Depending on the risks related to the product and the project, and the software development lifecycle model selected, different people may take over the role of tester at different test levels. For example, at the component testing level and the component integration testing level, the role of a tester is often done by developers. At the acceptance test level, the role of a tester is often done by business analysts, subject matter experts, and users. At the system test level and the system integration test level, the role of a tester is often done by an independent test team. At the operational acceptance test level, the role of a tester is often done by operations and/or systems administration staff.



Typical test manager tasks may include:

- Develop or review a test policy and test strategy for the organization
- Plan the test activities by considering the context, and understanding the test objectives and risks. This may include selecting test approaches, estimating test time, effort and cost, acquiring resources, defining test levels and test cycles, and planning defect management
- Write and update the test plan(s)
- Coordinate the test plan(s) with project managers, product owners, and others
- Share testing perspectives with other project activities, such as integration planning
- Initiate the analysis, design, implementation, and execution of tests, monitor test progress and results, and check the status of exit criteria (or definition of done) and facilitate test completion activities
- Prepare and deliver test progress reports and test summary reports based on the information gathered
- Adapt planning based on test results and progress (sometimes documented in test progress reports, and/or in test summary reports for other testing already completed on the project) and take any actions necessary for test control
- Support setting up the defect management system and adequate configuration management of testware
- Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
- Support the selection and implementation of tools to support the test process, including recommending the budget for tool selection (and possibly purchase and/or support), allocating time and effort for pilot projects, and providing continuing support in the use of the tool(s)
- Decide about the implementation of test environment(s)
- Promote and advocate the testers, the test team, and the test profession within the organization
- Develop the skills and careers of testers (e.g., through training plans, performance evaluations, coaching, etc.)



Typical tester tasks may include:

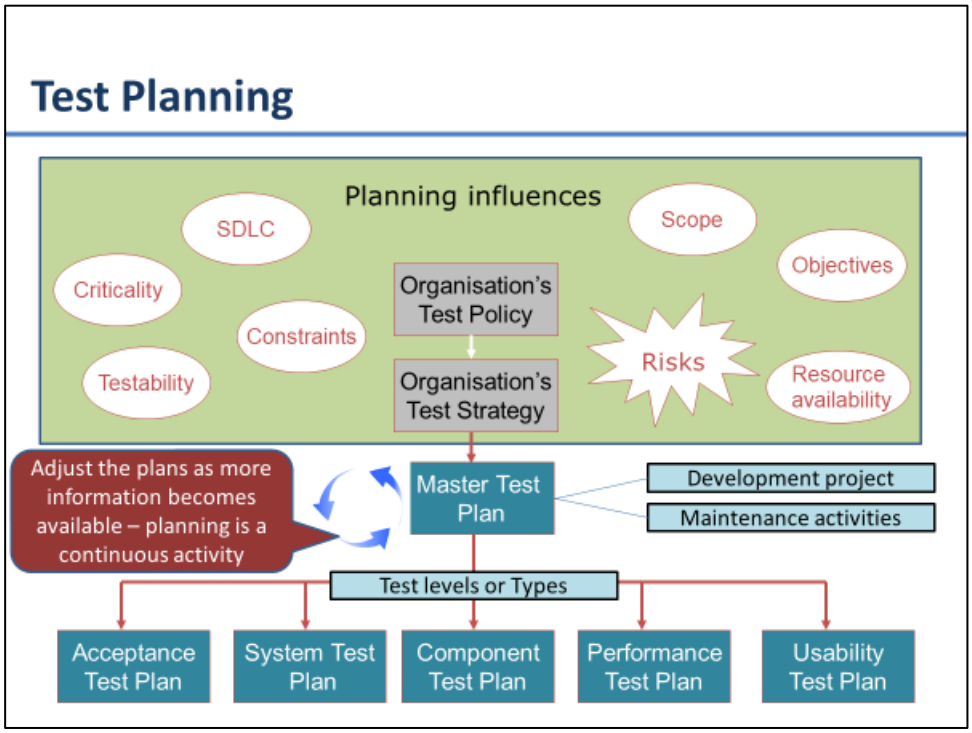
- Review and contribute to test plans
- Analyze, review, and assess requirements, user stories and acceptance criteria, specifications, and models for testability (i.e., the test basis)
- Identify and document test conditions, and capture traceability between test cases, test conditions, and the test basis
- Design, set up, and verify test environment(s), often coordinating with system administration and network management
- Design and implement test cases and test procedures
- Prepare and acquire test data
- Create the detailed test execution schedule
- Execute tests, evaluate the results, and document deviations from expected results
- Use appropriate tools to facilitate the test process
- Automate tests as needed (may be supported by a developer or a test automation expert)
- Evaluate non-functional characteristics such as performance efficiency, reliability, usability, security, compatibility, and portability
- Review tests developed by others

People who work on test analysis, test design, specific test types, or test automation may be specialists in these roles.

Summary

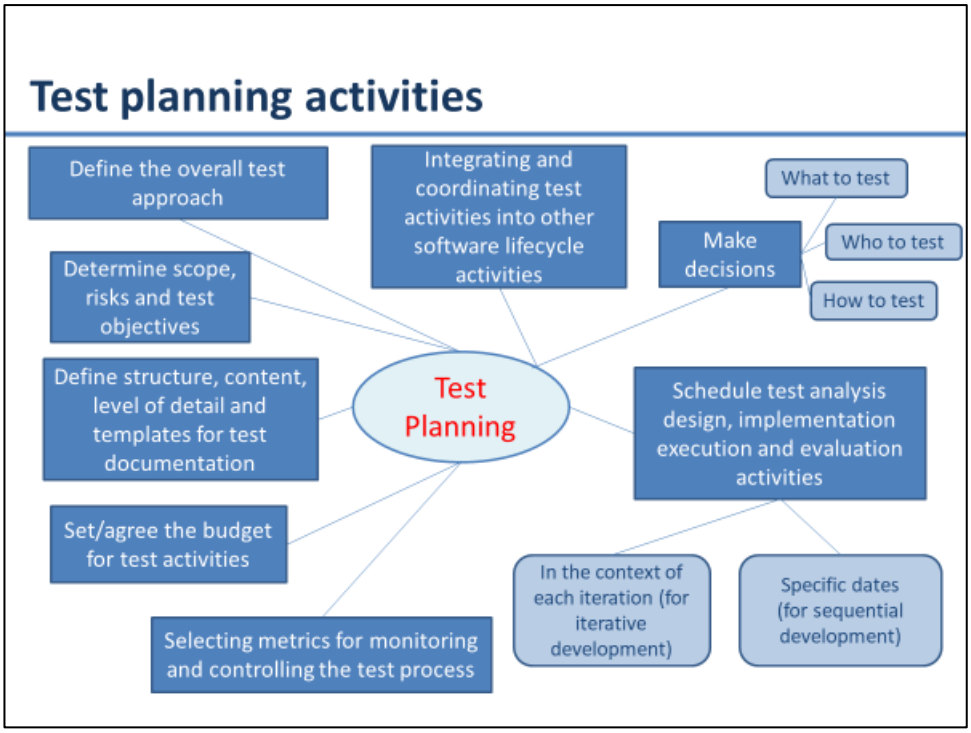
- Independent testing is an important consideration when organising test teams
- Test independence provides significant benefits, but there are some drawbacks that need to be overcome
- A test team is made up of different team members, depending on the test level at which they are working
- Test managers and testers typically perform different testing tasks

5.2 Test Planning and Estimation



A test plan outlines test activities for development and maintenance projects. Planning is influenced by the test policy and test strategy of the organization, the development lifecycles and methods being used (see section 2.1), the scope of testing, objectives, risks, constraints, criticality, testability, and the availability of resources.

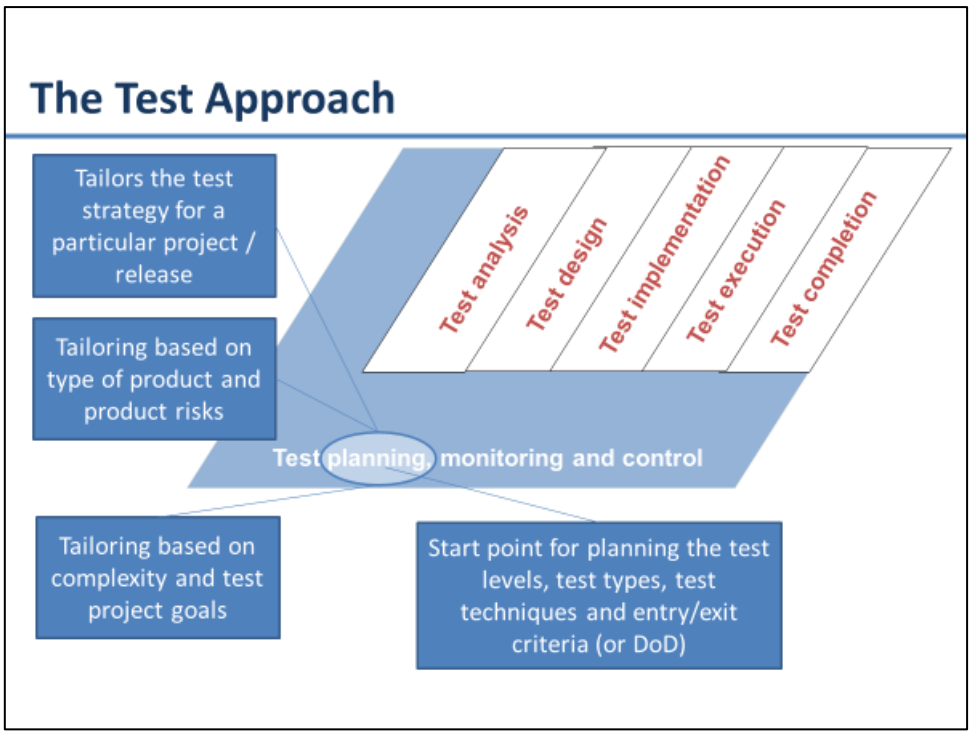
As the project and test planning progress, more information becomes available and more detail can be included in the test plan. Test planning is a continuous activity and is performed throughout the product's lifecycle. (Note that the product's lifecycle may extend beyond a project's scope to include the maintenance phase.) Feedback from test activities should be used to recognize changing risks so that planning can be adjusted. Planning may be documented in a master test plan and in separate test plans for test levels, such as system testing and acceptance testing, or for separate test types, such as usability testing and performance testing.



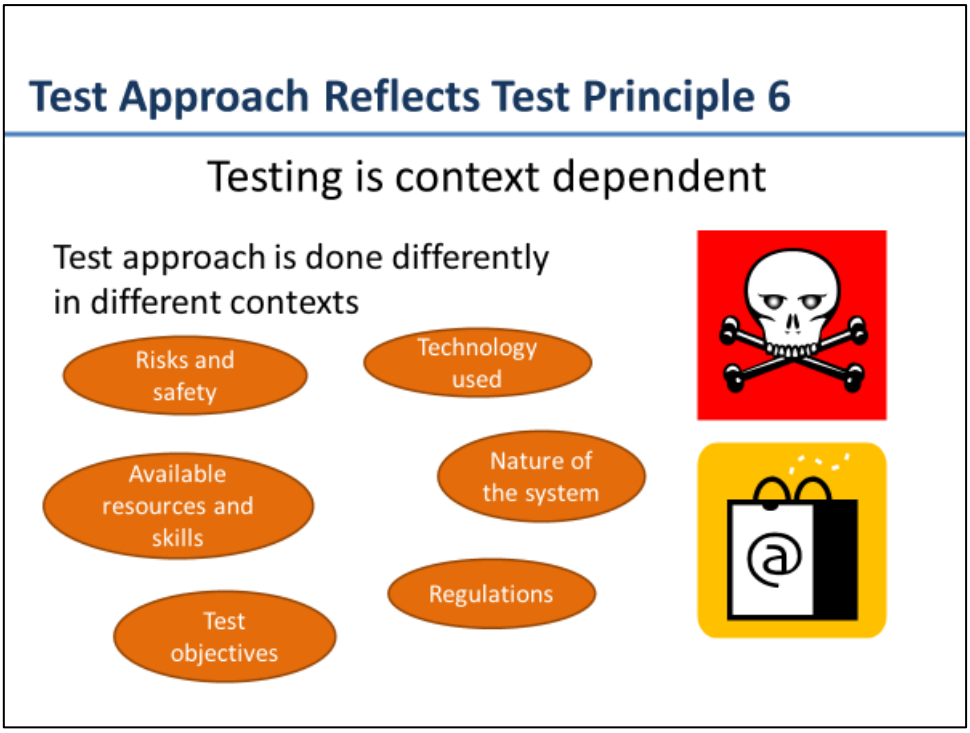
Test planning activities may include the following and some of these may be documented in a test plan:

- Determining the scope, objectives, and risks of testing
- Defining the overall approach of testing
- Integrating and coordinating the test activities into the software lifecycle activities
- Making decisions about what to test, the people and other resources required to perform the various test activities, and how test activities will be carried out
- Scheduling of test analysis, design, implementation, execution, and evaluation activities, either on particular dates (e.g., in sequential development) or in the context of each iteration (e.g., in iterative development)
- Selecting metrics for test monitoring and control
- Budgeting for the test activities
- Determining the level of detail and structure for test documentation (e.g., by providing templates or example documents)

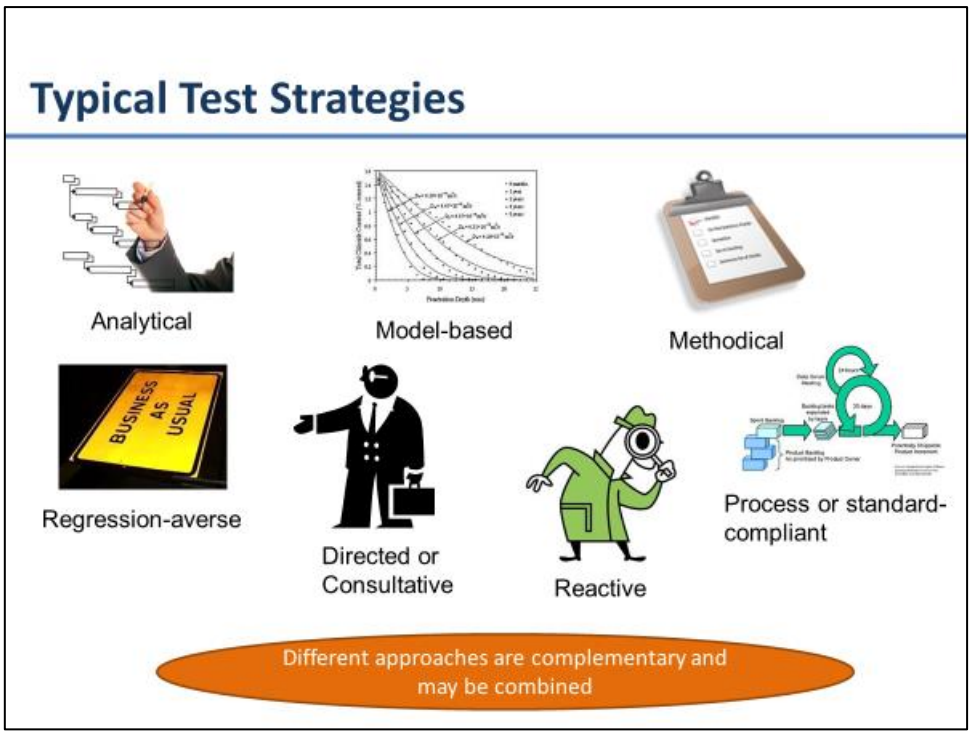
The content of test plans vary, and can extend beyond the topics identified above. A sample test plan structure and a sample test plan can be found in ISO standard (ISO/IEC/IEEE 29119-3).



While the test strategy provides a generalized description of the test process, the test approach tailors the test strategy for a particular project or release. The test approach is the starting point for selecting the test techniques, test levels, and test types, and for defining the entry criteria and exit criteria (or definition of ready and definition of done, respectively). The tailoring of the strategy is based on decisions made in relation to the complexity and goals of the project, the type of product being developed, and product risk analysis.



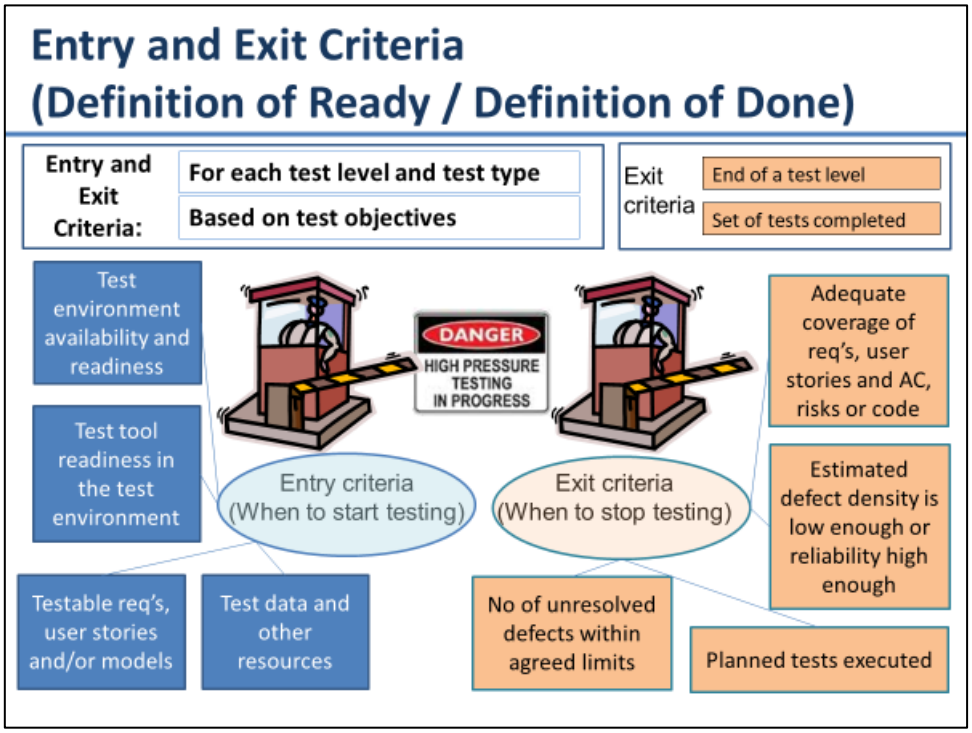
The selected approach depends on the context and may consider factors such as risks, safety, available resources and skills, technology, the nature of the system (e.g., custom-built versus COTS), test objectives, and regulations.



A test strategy provides a generalized description of the test process, usually at the product or organizational level. Common types of test strategies include:

- **Analytical:** This type of test strategy is based on an analysis of some factor (e.g., requirement or risk). Risk-based testing is an example of an analytical approach, where tests are designed and prioritized based on the level of risk.
- **Model-Based:** In this type of test strategy, tests are designed based on some model of some required aspect of the product, such as a function, a business process, an internal structure, or a non-functional characteristic (e.g., reliability). Examples of such models include business process models, state models, and reliability growth models.
- **Methodical:** This type of test strategy relies on making systematic use of some predefined set of tests or test conditions, such as a taxonomy of common or likely types of failures, a list of important quality characteristics, or company-wide look-and-feel standards for mobile apps or web pages.
- **Process-compliant** (or standard-compliant): This type of test strategy involves analyzing, designing, and implementing tests based on external rules and standards, such as those specified by industry-specific standards, by process documentation, by the rigorous identification and use of the test basis, or by any process or standard imposed on or by the organization.
- **Reactive:** In this type of test strategy, testing is reactive to the component or system being tested, and the events occurring during test execution, rather than being pre-planned (as the preceding strategies are). Tests are designed and implemented, and may immediately be executed in response to knowledge gained from prior test results. Exploratory testing is a common technique employed in reactive strategies.
- **Directed** (or consultative): This type of test strategy is driven primarily by the advice, guidance, or instructions of stakeholders, business domain experts, or technology experts, who may be outside the test team or outside the organization itself.
- **Regression-averse:** This type of test strategy is motivated by a desire to avoid regression of existing capabilities. This test strategy includes reuse of existing testware (especially test cases and test data), extensive automation of regression tests, and standard test suites.

An appropriate test strategy is often created by combining several of these types of test strategies. For example, risk-based testing (an analytical strategy) can be combined with exploratory testing (a reactive strategy); they complement each other and may achieve more effective testing when used together.



In order to exercise effective control over the quality of the software, and of the testing, it is advisable to have criteria which define when a given test activity should start and when the activity is complete. Entry criteria (more typically called definition of ready in Agile development) define the preconditions for undertaking a given test activity.

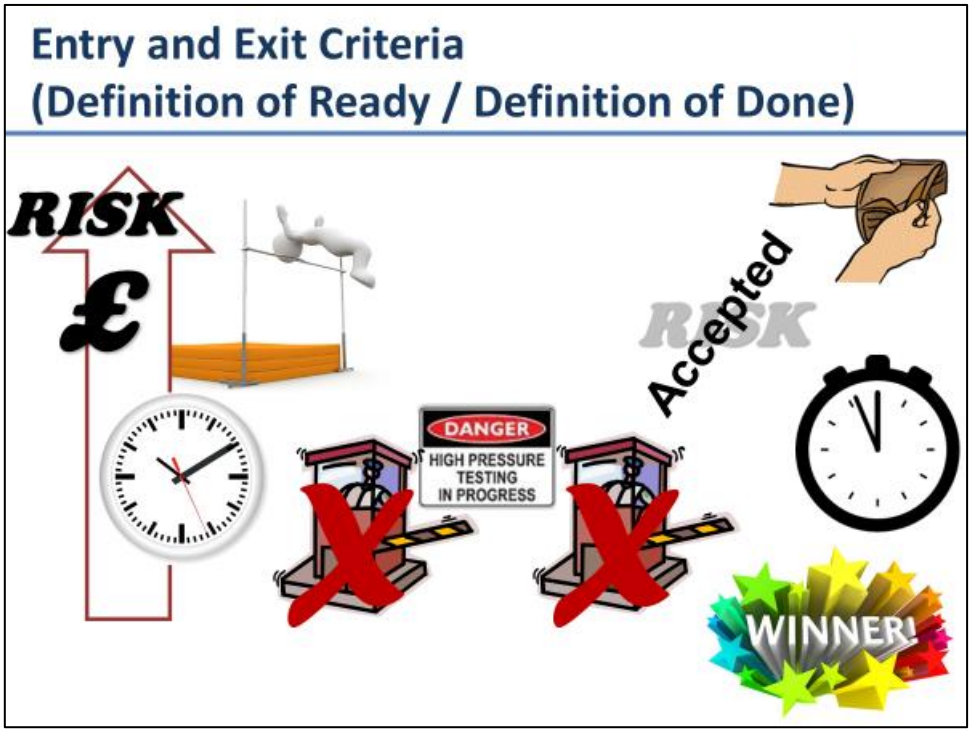
Exit criteria (more typically called definition of done in Agile development) define what conditions must be achieved in order to declare a test level or a set of tests completed. Entry and exit criteria should be defined for each test level and test type, and will differ based on the test objectives.

Typical entry criteria include:

- Availability of testable requirements, user stories, and/or models (e.g., when following a model based testing strategy)
- Availability of test items that have met the exit criteria for any prior test levels
- Availability of test environment
- Availability of necessary test tools
- Availability of test data and other necessary resources

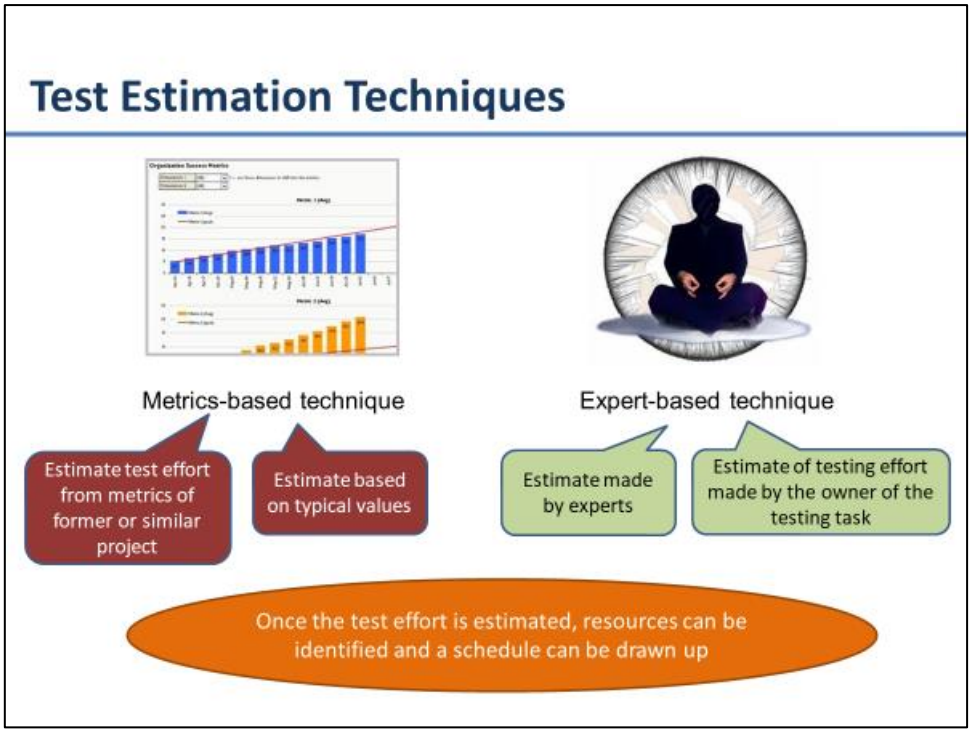
Typical exit criteria include:

- Planned tests have been executed
- A defined level of coverage (e.g., of requirements, user stories, acceptance criteria, risks, code) has been achieved
- The number of unresolved defects is within an agreed limit
- The number of estimated remaining defects is sufficiently low
- The evaluated levels of reliability, performance efficiency, usability, security, and other relevant quality characteristics are sufficient.



If entry criteria are not met, it is likely that the activity will prove more difficult, more time-consuming, more costly, and more risky.

Even without exit criteria being satisfied, it is also common for test activities to be curtailed due to the budget being expended, the scheduled time being completed, and/or pressure to bring the product to market. It can be acceptable to end testing under such circumstances, if the project stakeholders and business owners have reviewed and accepted the risk to go live without further testing.



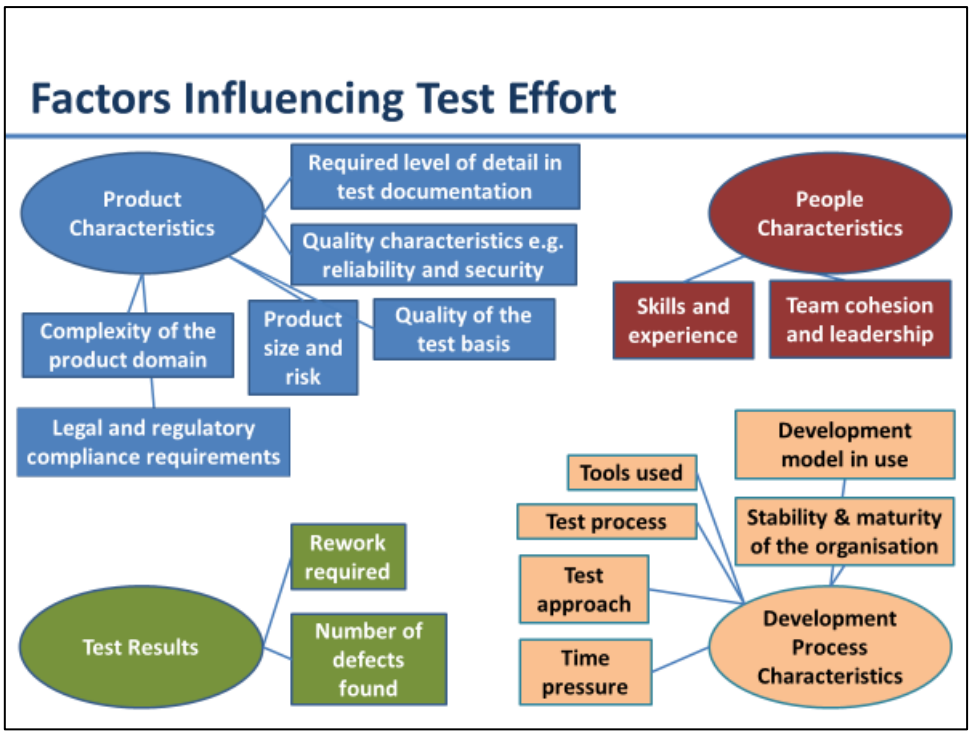
Test effort estimation involves predicting the amount of test-related work that will be needed in order to meet the objectives of the testing for a particular project, release, or iteration.

There are a number of estimation techniques used to determine the effort required for adequate testing. Two of the most commonly used techniques are:

- The metrics-based technique: estimating the test effort based on metrics of former similar projects, or based on typical values
- The expert-based technique: estimating the test effort based on the experience of the owners of the testing tasks or by experts

For example, in Agile development, burndown charts are examples of the metrics-based approach as effort remaining is being captured and reported, and is then used to feed into the team's velocity to determine the amount of work the team can do in the next iteration; whereas planning poker is an example of the expert-based approach, as team members are estimating the effort to deliver a feature based on their experience (ISTQB-CTFL-AT).

Within sequential projects, defect removal models are examples of the metrics-based approach, where volumes of defects and time to remove them are captured and reported, which then provides a basis for estimating future projects of a similar nature; whereas the Wideband Delphi estimation technique is an example of the expert-based approach in which groups of experts provides estimates based on their experience (ISTQB-CTAL-TM).



Factors influencing the test effort may include characteristics of the product, characteristics of the development process, characteristics of the people, and the test results, as shown below.

Product characteristics

- The risks associated with the product
- The quality of the test basis
- The size of the product
- The complexity of the product domain
- The requirements for quality characteristics (e.g., security, reliability)
- The required level of detail for test documentation
- Requirements for legal and regulatory compliance

Development process characteristics

- The stability and maturity of the organization
- The development model in use
- The test approach
- The tools used
- The test process
- Time pressure

People characteristics

- The skills and experience of the people involved, especially with similar projects and products (e.g., domain knowledge)
- Team cohesion and leadership

Test results

- The number and severity of defects found
- The amount of rework required

Once the various test cases and test procedures are produced (with some test procedures potentially automated) and assembled into test suites, the test suites can be arranged in a test execution schedule that defines the order in which they are to be run. The test execution schedule should take into account such factors as prioritization, dependencies, confirmation tests, regression tests, and the most efficient sequence for executing the tests.

Ideally, test cases would be ordered to run based on their priority levels, usually by executing the test cases with the highest priority first. However, this practice may not work if the test cases have dependencies 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. Similarly, if there are dependencies across test cases, they must be ordered appropriately regardless of their relative priorities. Confirmation and regression tests must be prioritized as well, based on the importance of rapid feedback on changes, but here again dependencies may apply.

In some cases, various sequences of tests are possible, with differing levels of efficiency associated with those sequences. In such cases, trade-offs between efficiency of test execution versus adherence to prioritization must be made.

Exercise – test execution schedule

Working in pairs, arrange these test cases into a test execution schedule

- There is no test data set up prior to execution
- The release note states that the modify function will not be delivered
- Female insurance claim processing has the highest priority

TC1 – add male claimant to queue

TC2 – delete male claimant details

TC3 – add female claimant to queue

TC4 – modify male claimant details

TC5 – process claims in queue

TC6 – modify female claimant details

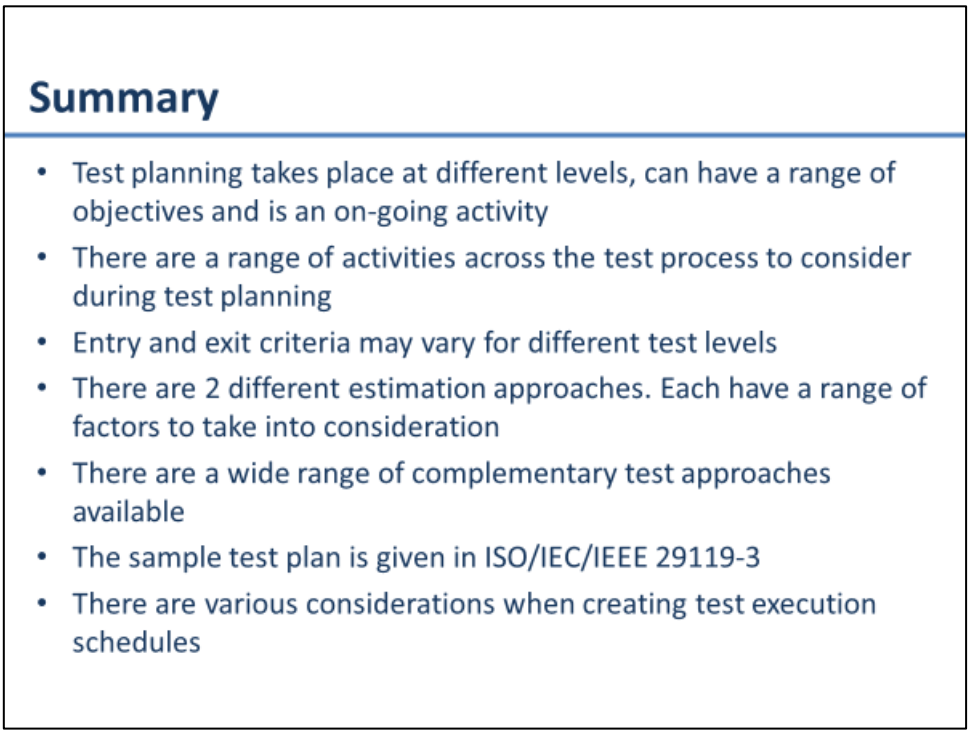
TC7 – delete female claimant details

You have 3 minutes

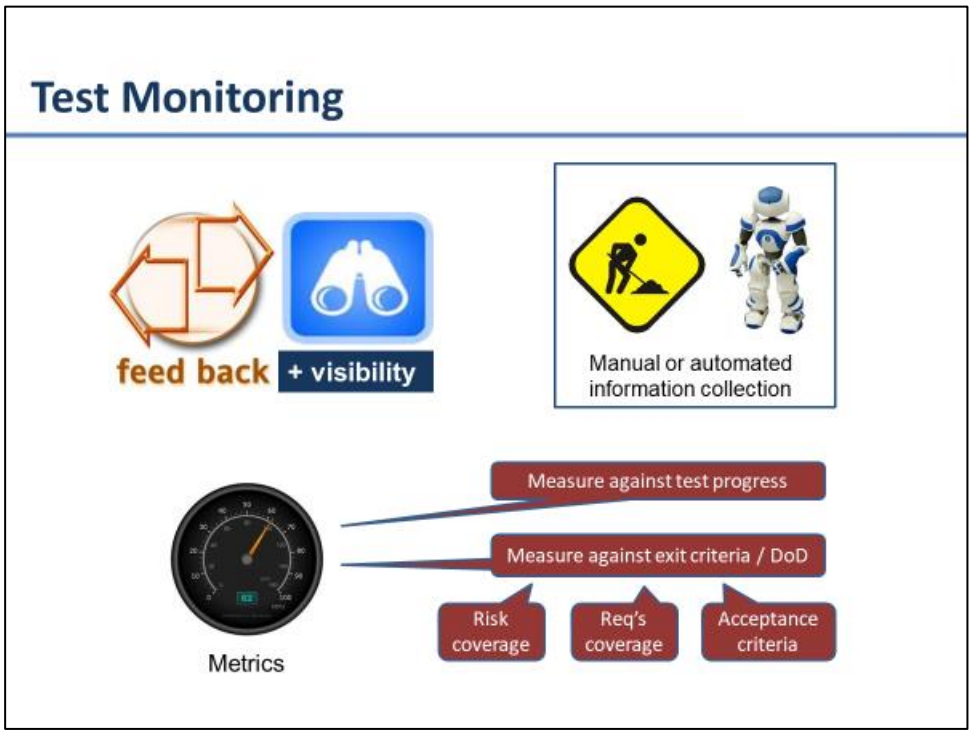
Space for you to note down your answers (the suggested answer is on the following page)

Suggested answer for discussion purposes:

- TC3 - add female claimant (checks that add works for female and creates test data for the delete test)
- TC7 - delete female claimant (checks that delete works for female)
- TC3 - add female claimant (creates test data for process claims test for female – highest priority)
- TC5 - process claims in queue (checks that claim processing works for female – highest priority)
- TC1 - add male claimant (checks that add works for male and creates test data for the delete test)
- TC2 - delete male claimant (checks that delete works for male)
- TC1 – add male claimant (creates test data for process claims test for male – lower priority)
- TC5 - process claims in queue (checks that claim processing works for male– lower priority)
- TC6 and TC4 should not be scheduled as the modify function will not be available



5.3 Test Monitoring and Control



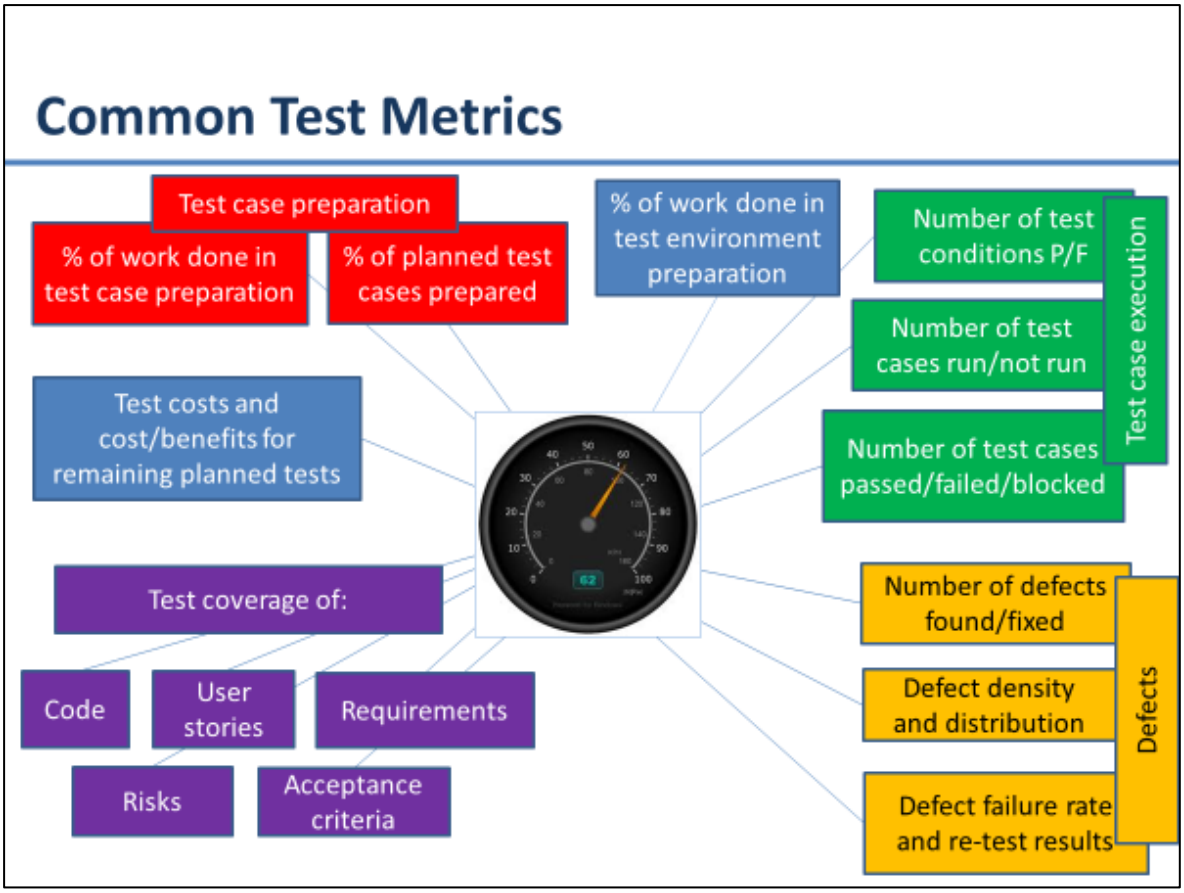
The purpose of test monitoring is to gather information and provide feedback and visibility about test activities. Information to be monitored may be collected manually or automatically and should be used to assess test progress and to measure whether the test exit criteria, or the testing tasks associated with an Agile project's definition of done, are satisfied, such as meeting the targets for coverage of product risks, requirements, or acceptance criteria.

Exercise – what metrics do you collect?

- Working in pairs, write up a list of the metrics that you collect in your test projects

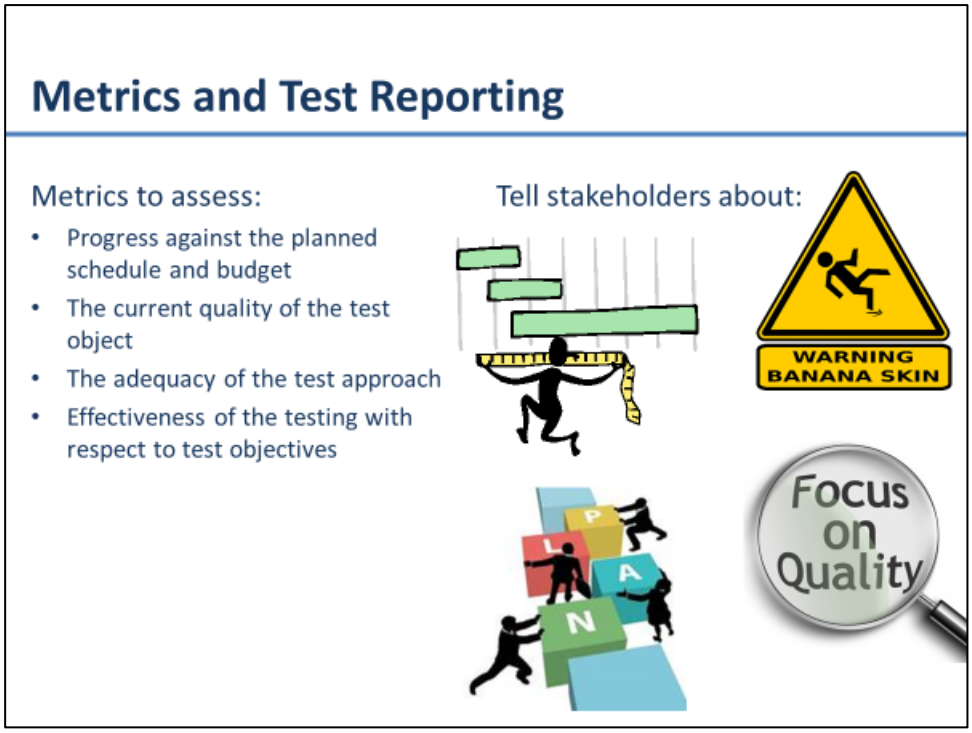
You have 2 minutes

Space for you to note down your answers (the suggested answer is on the following page)



Common test metrics include:

- Percentage of planned work done in test case preparation (or percentage of planned test cases implemented)
- Percentage of planned work done in test environment preparation
- Test case execution (e.g., number of test cases run/not run, test cases passed/failed, and/or test conditions passed/failed)
- Defect information (e.g., defect density, defects found and fixed, failure rate, and confirmation test results)
- Test coverage of requirements, user stories, acceptance criteria, risks, or code
- Task completion, resource allocation and usage, and effort
- Cost of testing, including the cost compared to the benefit of finding the next defect or the cost compared to the benefit of running the next test



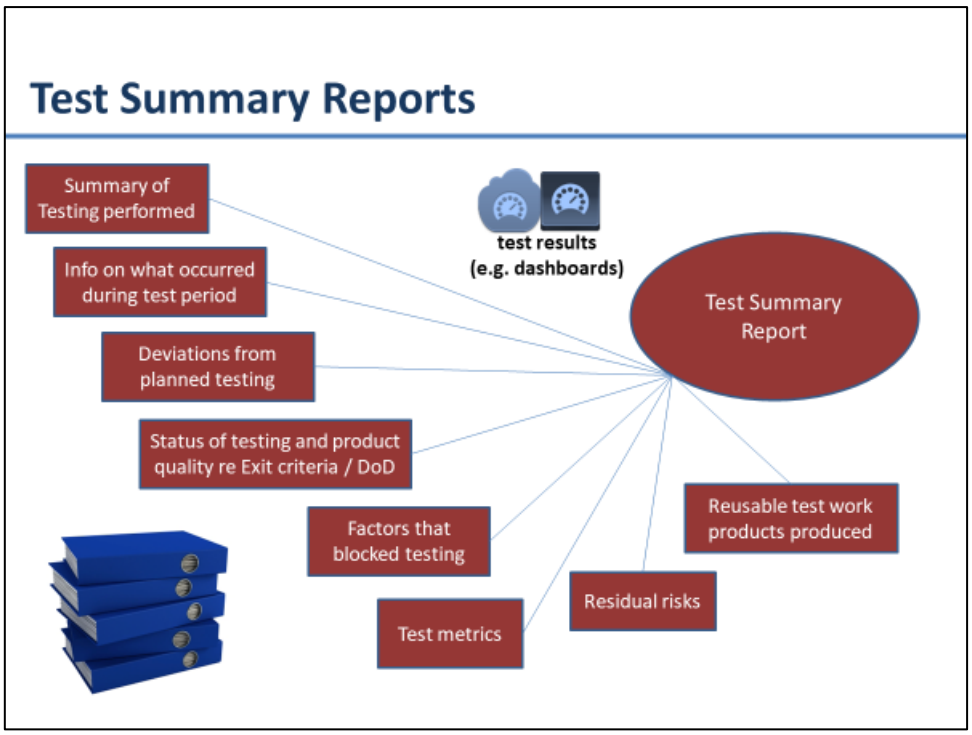
Metrics can be collected during and at the end of test activities in order to assess:

- Progress against the planned schedule and budget
- Current quality of the test object
- Adequacy of the test approach
- Effectiveness of the test activities with respect to the objectives

The purpose of test reporting is to summarize and communicate test activity information, both during and at the end of a test activity (e.g., a test level). The test report prepared during a test activity may be referred to as a test progress report, while a test report prepared at the end of a test activity may be referred to as a test summary report.

During test monitoring and control, the test manager regularly issues test progress reports for stakeholders. In addition to content common to test progress reports and test summary reports, typical test progress reports may also include:

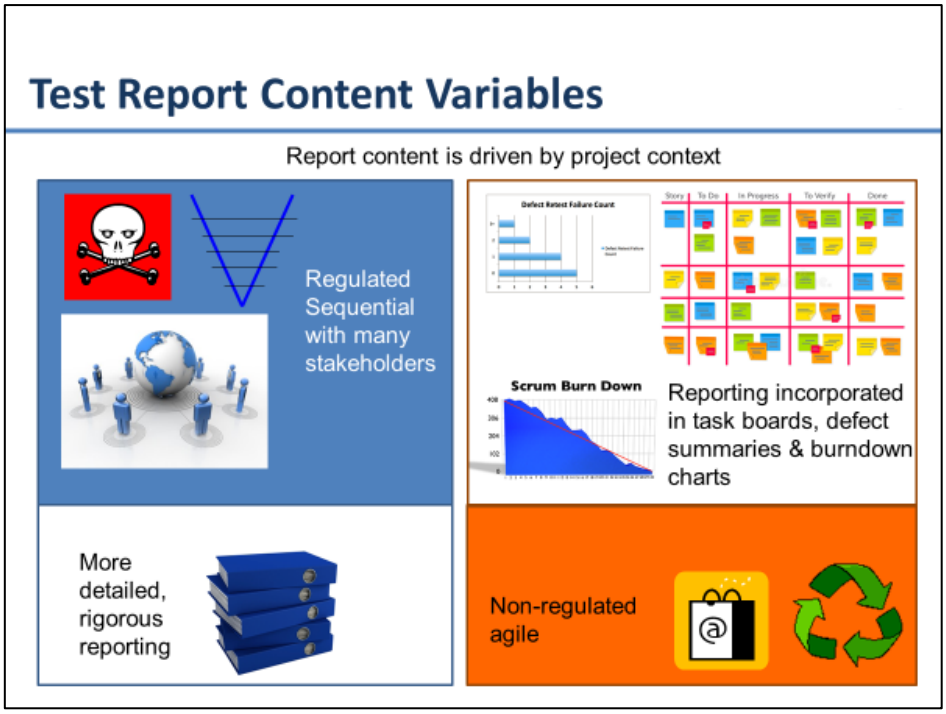
- The status of the test activities and progress against the test plan
- Factors impeding progress
- Testing planned for the next reporting period
- The quality of the test object



When exit criteria are reached, the test manager issues the test summary report. This report provides a summary of the testing performed, based on the latest test progress report and any other relevant information.

Typical test summary reports may include:

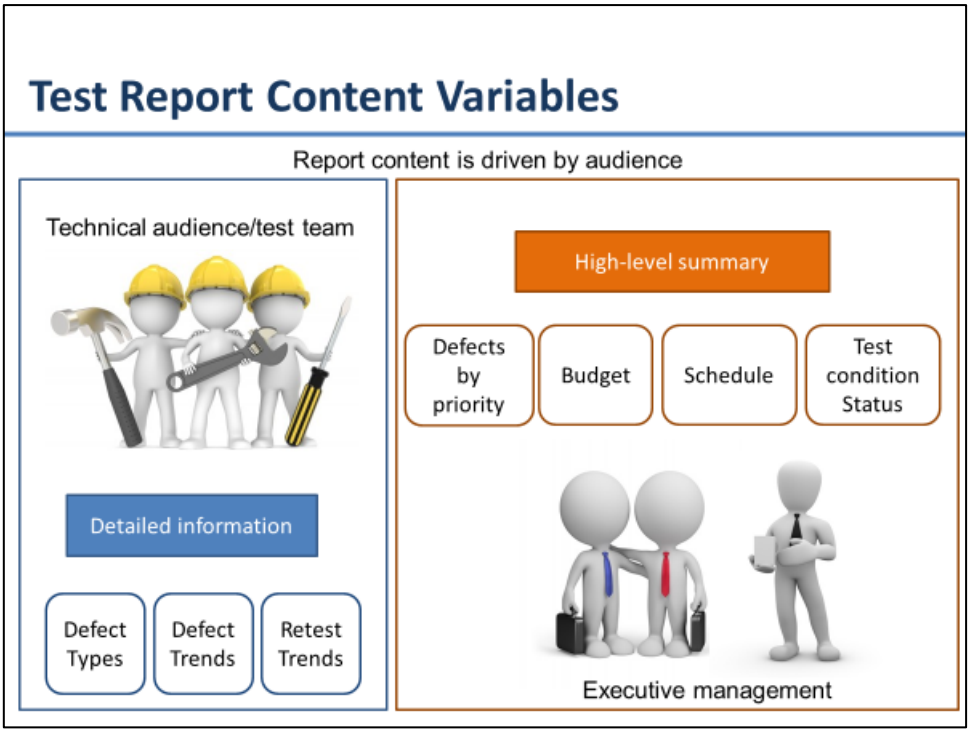
- Summary of testing performed
- Information on what occurred during a test period
- Deviations from plan, including deviations in schedule, duration, or effort of test activities
- Status of testing and product quality with respect to the exit criteria or definition of done
- Factors that have blocked or continue to block progress
- Metrics of defects, test cases, test coverage, activity progress, and resource consumption. (e.g., as described in 5.3.1)
- Residual risks (see section 5.5)
- Reusable test work products produced



The contents of a test report will vary depending on the project, the organizational requirements, and the software development lifecycle.

For example, a complex project with many stakeholders or a regulated project may require more detailed and rigorous reporting than a quick software update.

As another example, in Agile development, test progress reporting may be incorporated into task boards, defect summaries, and burndown charts, which may be discussed during a daily stand-up meeting (see ISTQBCTFL-AT).

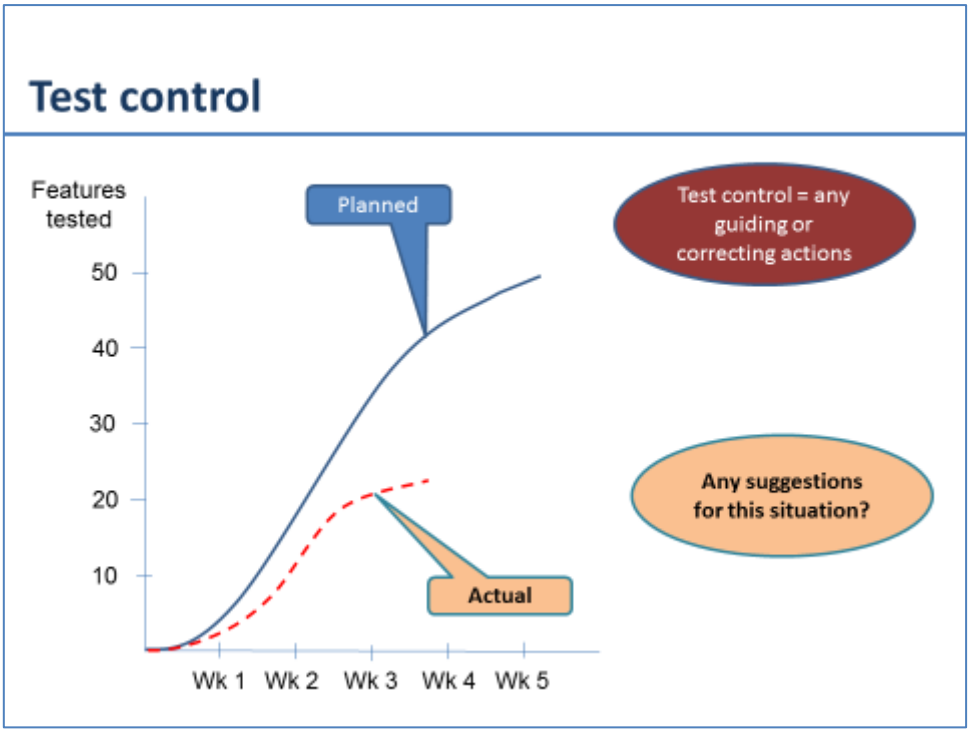


In addition to tailoring test reports based on the context of the project, test reports should be tailored based on the report's audience.

The type and amount of information that should be included for a technical audience or a test team may be different from what would be included in an executive summary report.

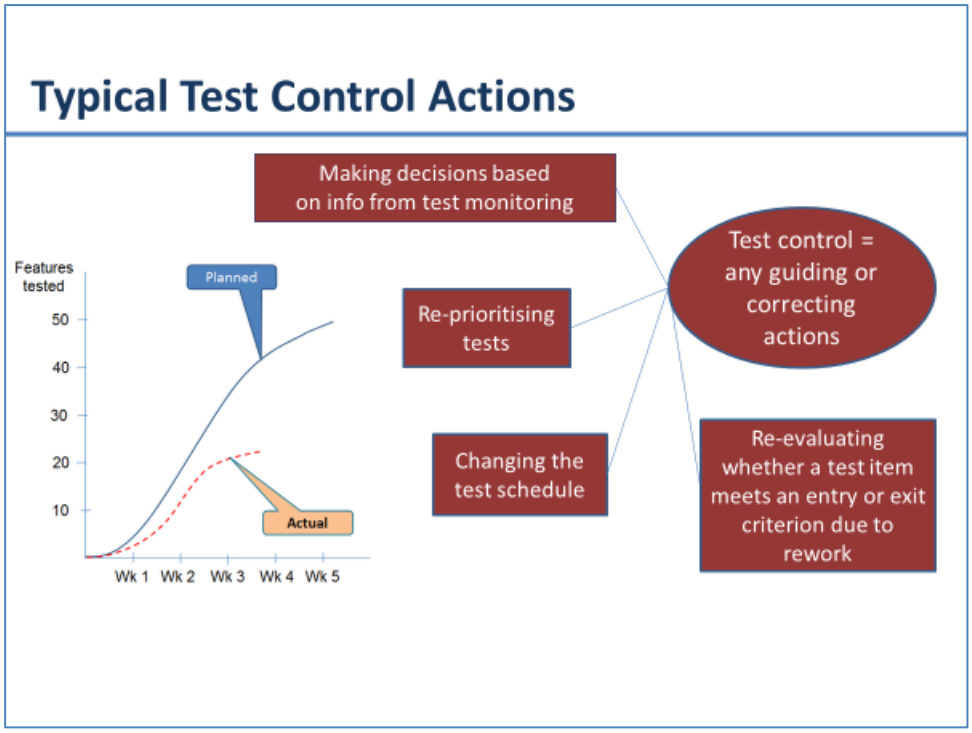
In the first case, detailed information on defect types and trends may be important. In the latter case, a high-level report (e.g., a status summary of defects by priority, budget, schedule, and test conditions passed/failed/not tested) may be more appropriate.

ISO standard (ISO/IEC/IEEE 29119-3) refers to two types of test reports, test progress reports and test completion reports (called test summary reports in this syllabus) and contains structures and examples for each type.



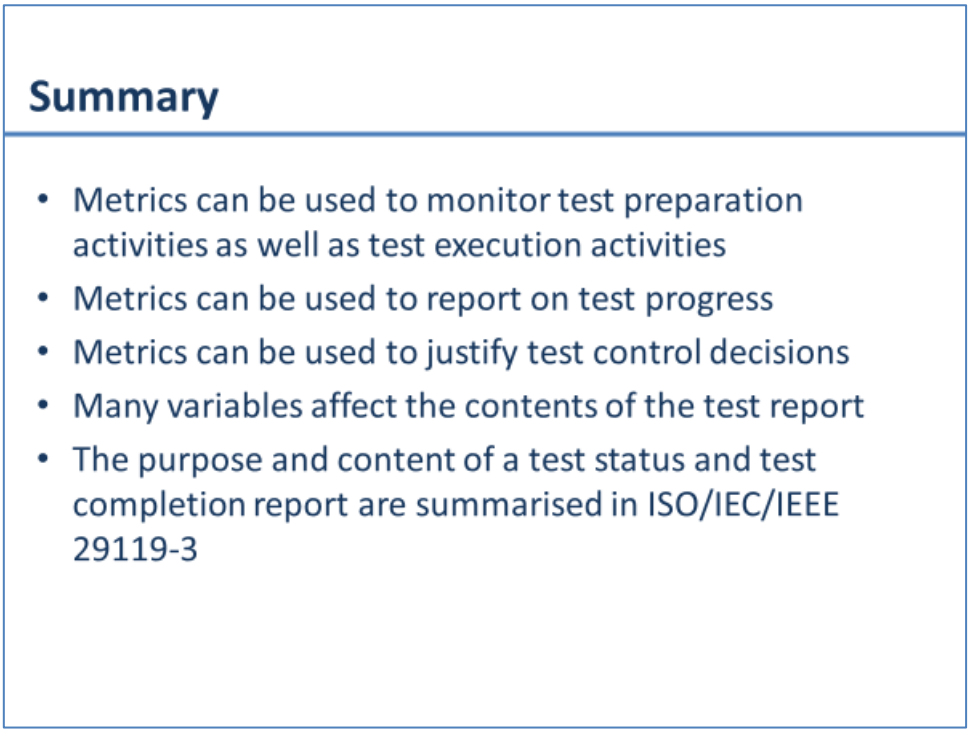
Test control describes any guiding or corrective actions taken as a result of information and metrics gathered and (possibly) reported. Actions may cover any test activity and may affect any other software lifecycle activity.

Space for you to note down your answers (the suggested answer is on the following page).

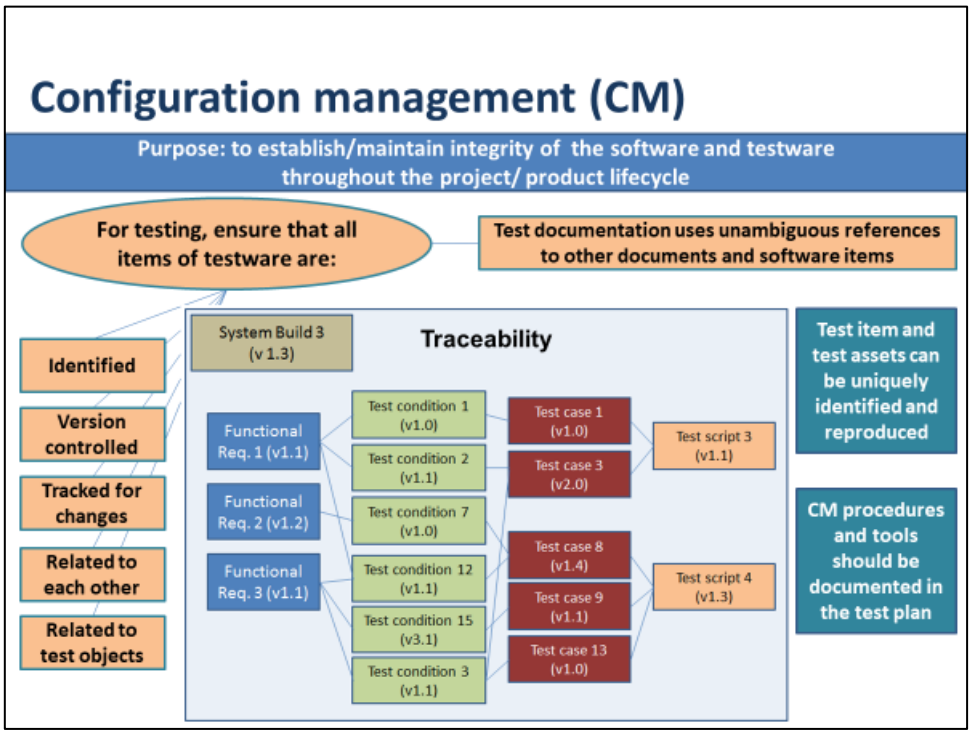


Examples of test control actions include:

- Re-prioritizing tests when an identified risk occurs (e.g., software delivered late)
- Changing the test schedule due to availability or unavailability of a test environment or other resources
- Re-evaluating whether a test item meets an entry or exit criterion due to rework



5.4 Configuration Management



The purpose of configuration management is to establish and maintain the integrity of the component or system, the testware, and their relationships to one another through the project and product lifecycle.

To properly support testing, configuration management may involve ensuring the following:

- All test items are uniquely identified, version controlled, tracked for changes, and related to each other
- All items of testware are uniquely identified, version controlled, tracked for changes, related to each other and related to versions of the test item(s) so that traceability can be maintained throughout the test process
- All identified documents and software items are referenced unambiguously in test documentation

During test planning, configuration management procedures and infrastructure (tools) should be identified and implemented.

Summary

- Configuration management is a general discipline that needs to be in place to support testing
- The test plan needs to ensure that processes and tools are in place to ensure that test assets can be traced to the correct versions of the test basis and the test items

Homework – Day 2

- Attempt the following Questions from Sample Exam C
- Section 4
  - Questions 19 to 29
- Section 5
  - Questions 30 to 34
- Please mark your own (See Sample Exam C Answers) and read the answer justifications as needed
- Read the Foundation Syllabus Sections 4 and 5

Exercise – which is the riskier journey

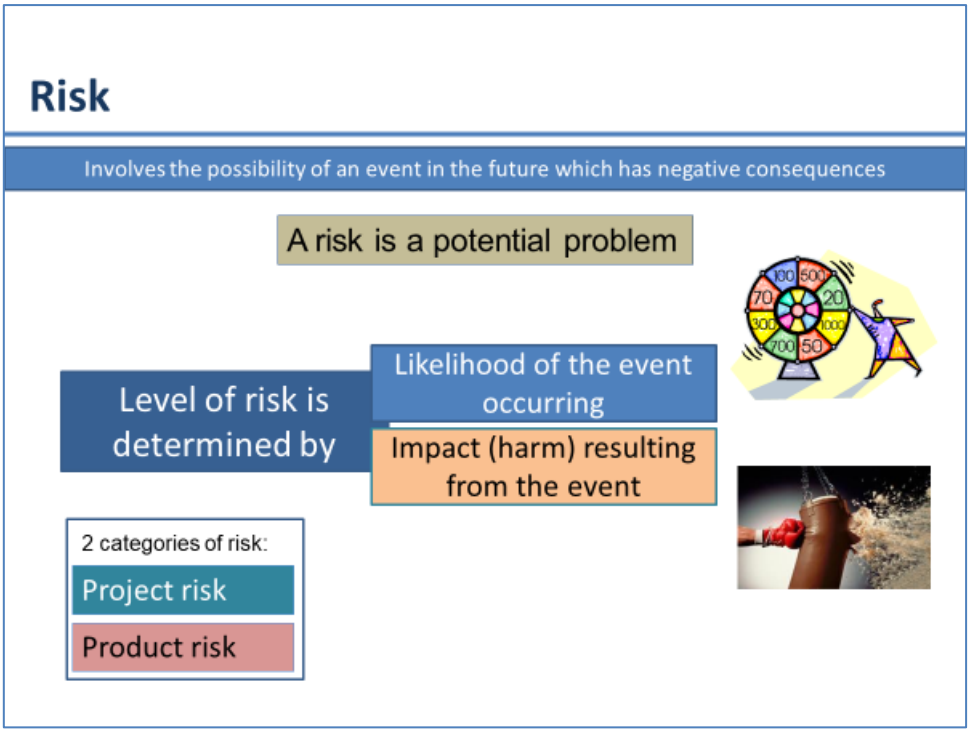
- Working in pairs, decide which is the riskier journey
- Be ready to justify your choice

A - Plymouth to Aberdeen by train

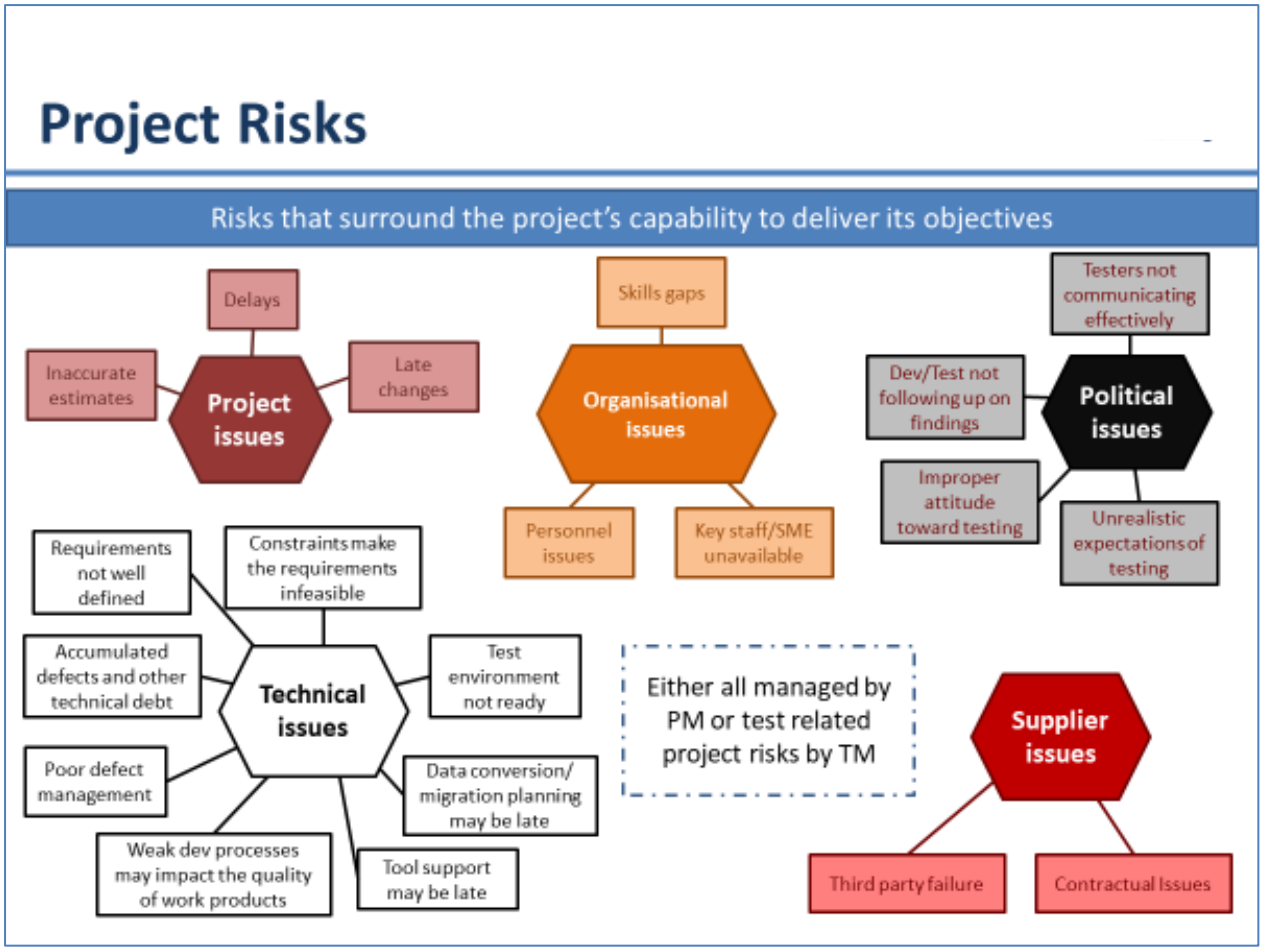
B - London to Brighton by bicycle

You have 2 minutes

Space for you to note down your answers



Risk involves the possibility of an event in the future which has negative consequences. The level of risk is determined by the likelihood of the event and the impact (the harm) from that event.



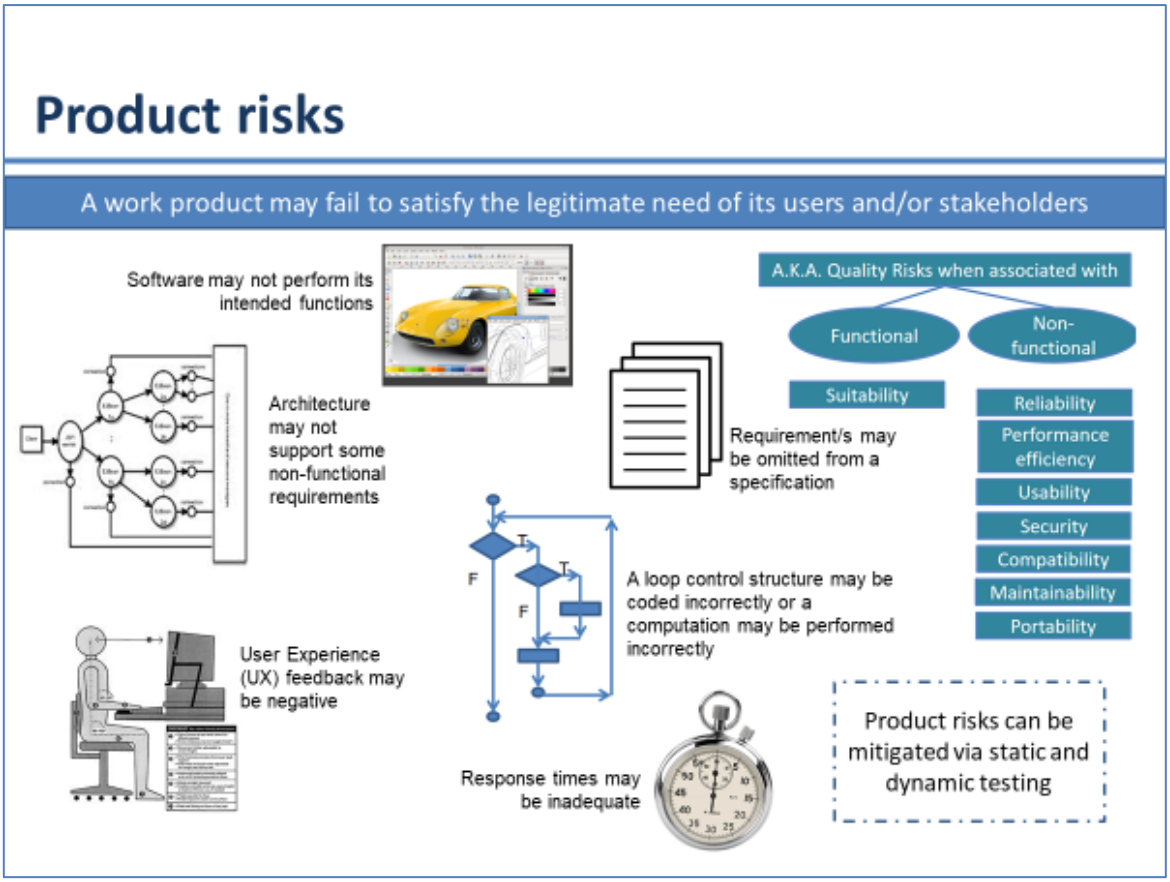
Project risk involves situations that, should they occur, may have a negative effect on a project's ability to achieve its objectives. Examples of project risks include:

- Project issues:
  - Delays may occur in delivery, task completion, or satisfaction of exit criteria or definition of done
  - Inaccurate estimates, reallocation of funds to higher priority projects, or general cost-cutting across the organization may result in inadequate funding
  - Late changes may result in substantial re-work
- Organizational issues:
  - Skills, training, and staff may not be sufficient
  - Personnel issues may cause conflict and problems
  - Users, business staff, or subject matter experts may not be available due to conflicting business priorities
- Political issues:
  - Testers may not communicate their needs and/or the test results adequately
  - Developers and/or testers may fail to follow up on information found in testing and reviews (e.g., not improving development and testing practices)
  - There may be an improper attitude toward, or expectations of, testing (e.g., not appreciating the value of finding defects during testing)

(continued on next page)

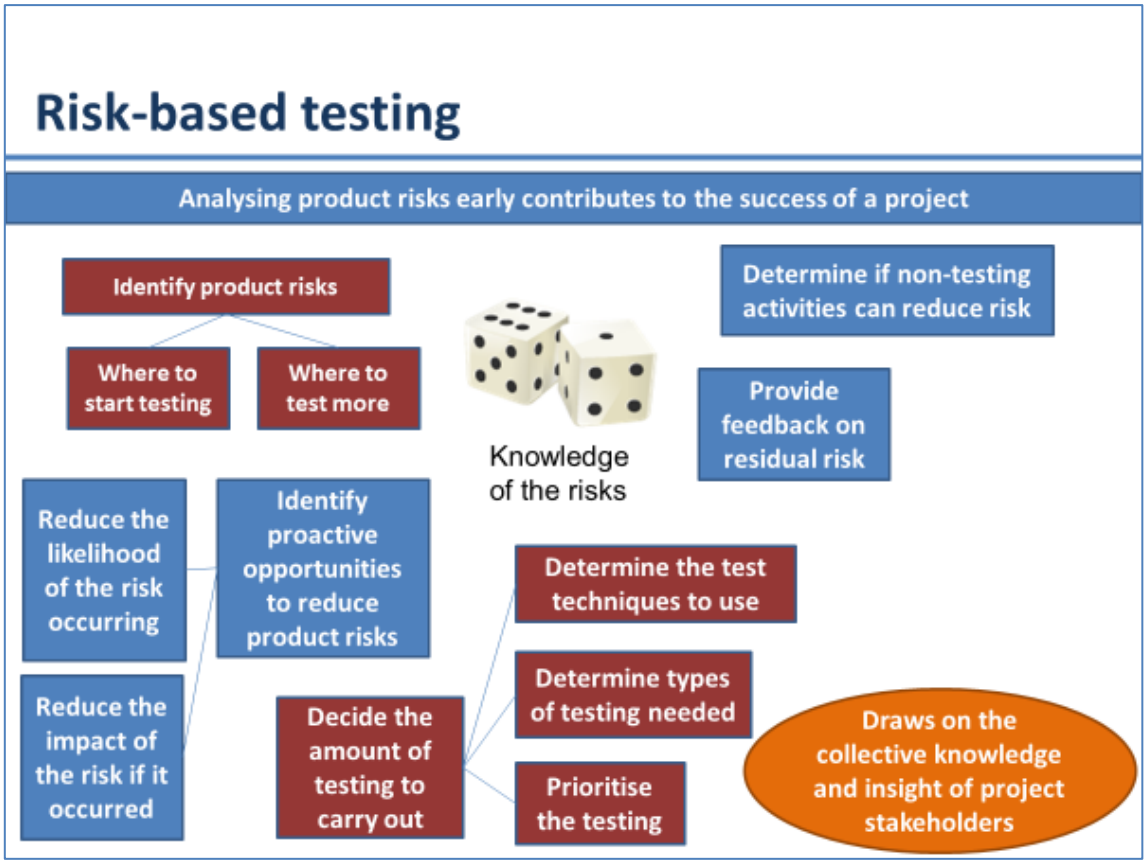
- Technical issues:
  - Requirements may not be defined well enough
  - The requirements may not be met, given existing constraints
  - The test environment may not be ready on time
  - Data conversion, migration planning, and their tool support may be late
  - Weaknesses in the development process may impact the consistency or quality of project work products such as design, code, configuration, test data, and test cases
  - Poor defect management and similar problems may result in accumulated defects and other technical debt
- Supplier issues:
  - A third party may fail to deliver a necessary product or service, or go bankrupt
  - Contractual issues may cause problems to the project

Project risks may affect both development activities and test activities. In some cases, project managers are responsible for handling all project risks, but it is not unusual for test managers to have responsibility for test-related project risks.



Product risk involves the possibility that a work product (e.g., a specification, component, system, or test) may fail to satisfy the legitimate needs of its users and/or stakeholders. When the product risks are associated with specific quality characteristics of a product (e.g., functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability), product risks are also called quality risks. Examples of product risks include:

- Software might not perform its intended functions according to the specification
- Software might not perform its intended functions according to user, customer, and/or stakeholder needs
- A system architecture may not adequately support some non-functional requirement(s)
- A particular computation may be performed incorrectly in some circumstances
- A loop control structure may be coded incorrectly
- Response-times may be inadequate for a high-performance transaction processing system
- User experience (UX) feedback might not meet product expectations

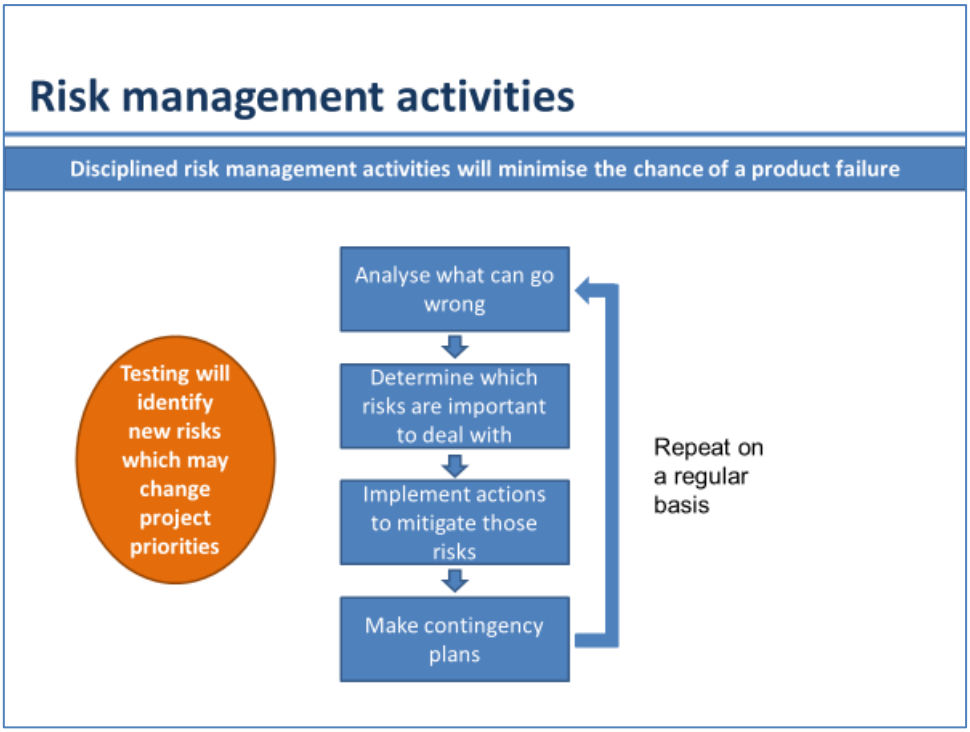


Risk is used to focus the effort required during testing. It is used to decide where and when to start testing and to identify areas that need more attention. Testing is used to reduce the probability of an adverse event occurring, or to reduce the impact of an adverse event. Testing is used as a risk mitigation activity, to provide information about identified risks, as well as providing information on residual (unresolved) risks.

A risk-based approach to testing provides proactive opportunities to reduce the levels of product risk. It involves product risk analysis, which includes the identification of product risks and the assessment of each risk's likelihood and impact. The resulting product risk information is used to guide test planning, the specification, preparation and execution of test cases, and test monitoring and control. Analyzing product risks early contributes to the success of a project.

In a risk-based approach, the results of product risk analysis are used to:

- Determine the test techniques to be employed
- Determine the particular levels and types of testing to be performed (e.g., security testing, accessibility testing)
- Determine the extent of testing to be carried out
- 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 (e.g., providing training to inexperienced designers)



Risk-based testing draws on the collective knowledge and insight of the project stakeholders to carry out product risk analysis. To ensure that the likelihood of a product failure is minimized, risk management activities provide a disciplined approach to:

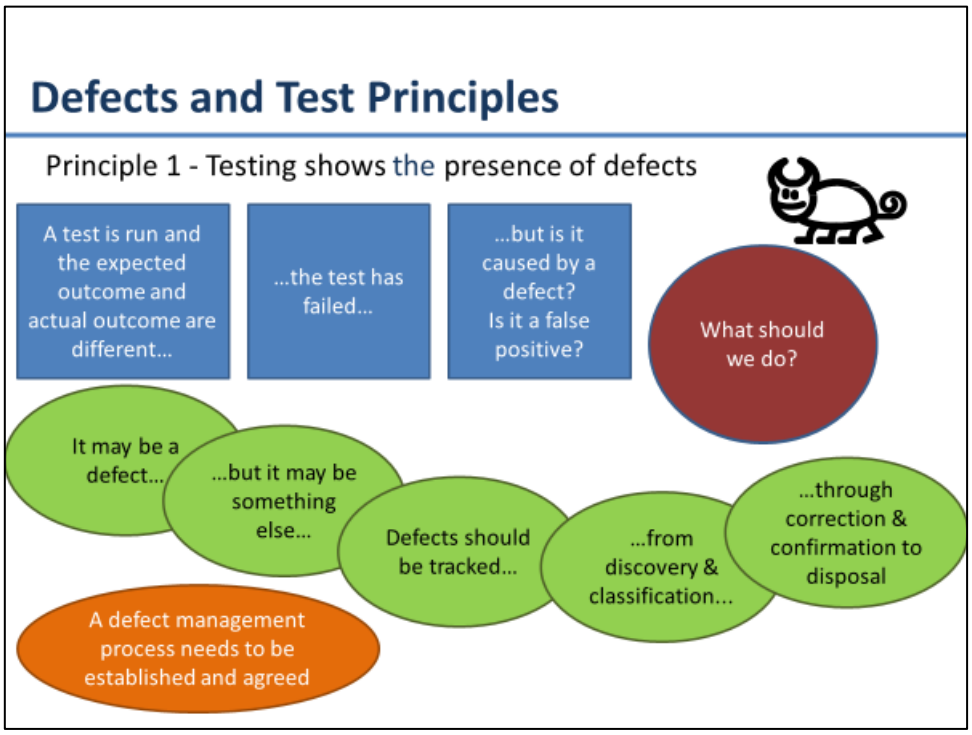
- Analyze (and re-evaluate on a regular basis) what can go wrong (risks)
- Determine which risks are important to deal with
- Implement actions to mitigate those risks
- Make contingency plans to deal with the risks should they become actual events

In addition, testing may identify new risks, help to determine what risks should be mitigated, and lower uncertainty about risks.

Summary

- A risk is a possible problem that would threaten the achievement of one or more stakeholders' project objectives
- The level of risk is determined by a combination of likelihood and impact
- Project risks have an adverse impact on cost or timeframes
- Product risks have an adverse impact on a product's quality attributes
- Risk management activities may be used to set the direction and determine the priorities for test planning

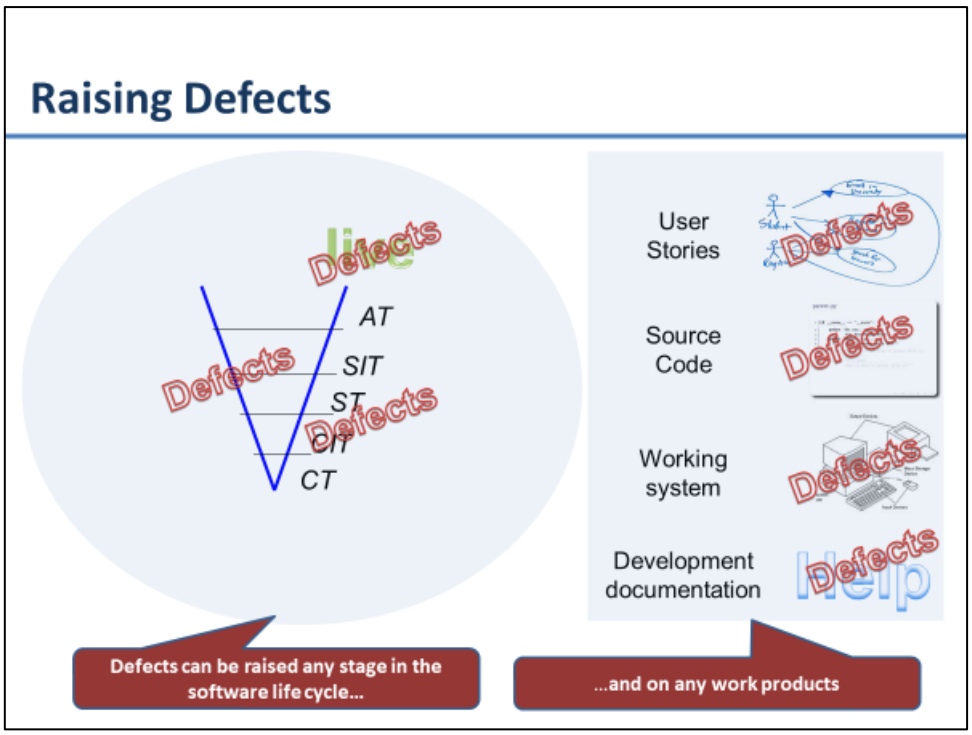
5.6 Defect Management



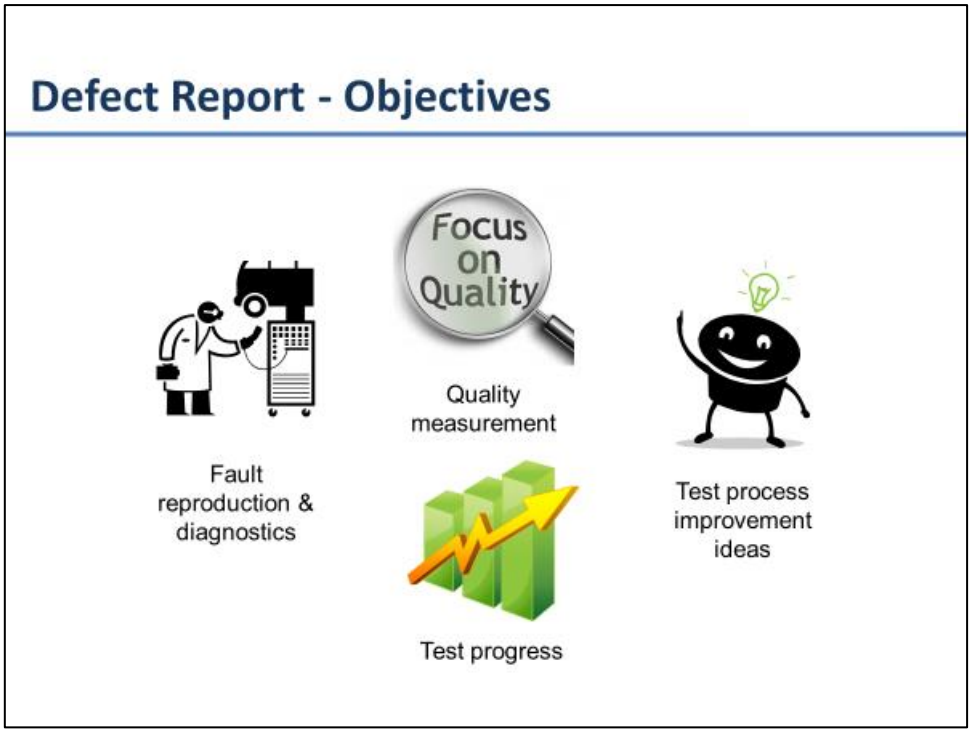
Since one of the objectives of testing is to find defects, defects found during testing should be logged. The way in which defects are logged may vary, depending on the context of the component or system being tested, the test level, and the software development lifecycle model. Any defects identified should be investigated and should be tracked from discovery and classification to their resolution (e.g., correction of the defects and successful confirmation testing of the solution, deferral to a subsequent release, acceptance as a permanent product limitation, etc.).

In order to manage all defects to resolution, an organization should establish a defect management process which includes a workflow and rules for classification. This process must be agreed with all those participating in defect management, including architects, designers, developers, testers, and product owners. In some organizations, defect logging and tracking may be very informal.

During the defect management process, some of the reports may turn out to describe false positives, not actual failures due to defects. For example, a test may fail when a network connection is broken or times out. This behavior does not result from a defect in the test object, but is an anomaly that needs to be investigated. Testers should attempt to minimize the number of false positives reported as defects.



Defects may be reported during coding, static analysis, reviews, or during dynamic testing, or use of a software product. Defects may be reported for issues in code or working systems, or in any type of documentation including requirements, user stories and acceptance criteria, development documents, test documents, user manuals, or installation guides. In order to have an effective and efficient defect management process, organizations may define standards for the attributes, classification, and workflow of defects.



Typical defect reports have the following objectives:

- Provide developers and other parties with information about any adverse event that occurred, to enable them to identify specific effects, to isolate the problem with a minimal reproducing test, and to correct the potential defect(s), as needed or to otherwise resolve the problem
- Provide test managers a means of tracking the quality of the work product and the impact on the testing (e.g., if a lot of defects are reported, the testers will have spent a lot of time reporting them instead of running tests, and there will be more confirmation testing needed)
- Provide ideas for development and test process improvement

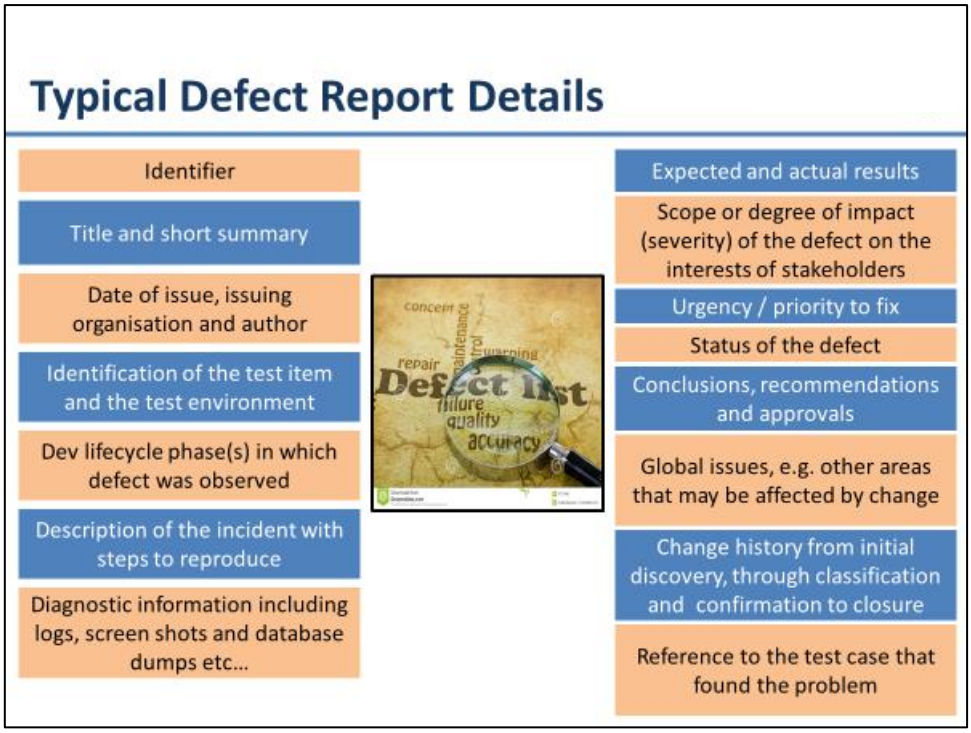
Exercise – Logging a Defect

- Working in pairs, write a list of the details that you would expect to see on a defect report

You have 5 minutes

Space for you to note down your answers (the suggested answer is on the following page)

Page intentionally left blank



A defect report filed during dynamic testing typically includes:

- An identifier
- A title and a short summary of the defect being reported
- Date of the defect report, issuing organization, and author
- Identification of the test item (configuration item being tested) and environment
- The development lifecycle phase(s) in which the defect was observed
- A description of the defect to enable reproduction and resolution, including logs, database dumps, screenshots, or recordings (if found during test execution)
- Expected and actual results
- Scope or degree of impact (severity) of the defect on the interests of stakeholder(s)
- Urgency/priority to fix
- State of the defect report (e.g., open, deferred, duplicate, waiting to be fixed, awaiting confirmation testing, re-opened, closed)
- Conclusions, recommendations and approvals
- Global issues, such as other areas that may be affected by a change resulting from the defect
- Change history, such as the sequence of actions taken by project team members with respect to the defect to isolate, repair, and confirm it as fixed
- References, including the test case that revealed the problem

Some of these details may be automatically included and/or managed when using defect management tools, e.g., automatic assignment of an identifier, assignment and update of the defect report state during the workflow, etc. Defects found during static testing, particularly reviews, will normally be documented in a different way, e.g., in review meeting notes.

An example of the contents of a defect report can be found in ISO standard (ISO/IEC/IEEE 29119-3) (which refers to defect reports as incident reports).

Exercise – Writing a Defect Report

- Working in pairs, identify ways in which the following defect report can be improved

Defect 456 – priority HIGH
I was running test script 6 (buy goods on line) and one of the boundary tests for 16 year olds has failed. I have checked with the business and you should be able to buy goods on your 16th birthday. Steps 1 and 3 (birthday yesterday and birthday tomorrow) both work OK, but I can't buy goods on the day of my birthday. THIS IS REALLY STUPID. Its obvious that the developers haven't looked at the requirements (clause 1.6 v2). I tried this 3 times using the latest versions of Chrome & Safari.

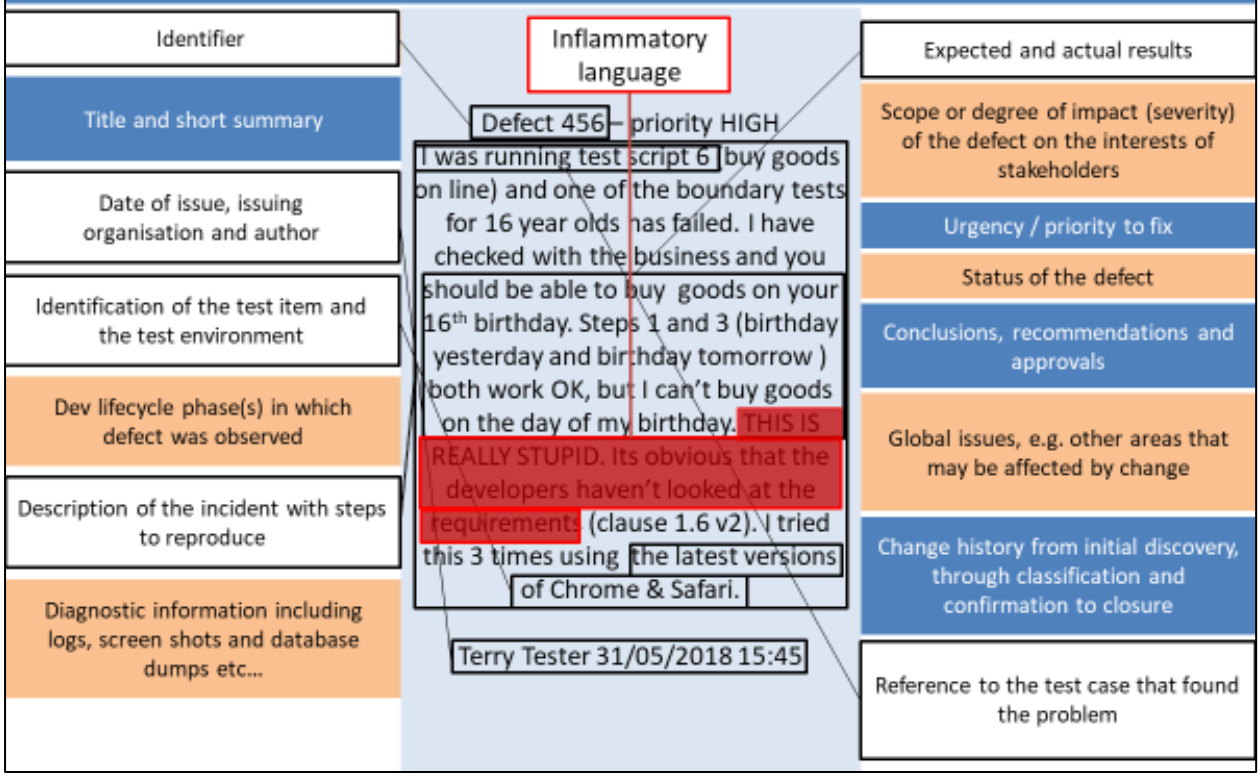
Terry Tester 31/05/2018 15:45

You have 5 minutes

Space for you to note down your answers (the suggested answer is on the following page)

Page intentionally left blank

Suggested Answer



Summary

- Not all testing incidents are defects
- Defects can occur at any stage of the development lifecycle
- Defects can be raised against documents, code and the live system
- A defect report needs to contain sufficient information to allow the developers to reproduce the problem and diagnose (and fix) any underlying defects

End of section exercise

- Attempt the following Questions from Sample Exam B
- Questions 30 to 38
- Please mark your own (See Sample Exam B Answers) and read the answer justifications as needed

You have 12 minutes

Learning Objectives for Test Management

In order to complete this section, you should satisfy yourself that you are able to understand or perform the following items to the standard indicated by the corresponding “K - learning level”. For an explanation of “K levels”, please see the course introduction section – “Learning Objectives and Levels of Knowledge”.

Keywords

configuration management, defect management, defect report, entry criteria, exit criteria, product risk, project risk, risk, risk level, risk-based testing, test approach, test control, test estimation, test manager, test monitoring, test plan, test planning, test progress report, test strategy, test summary report, tester

Learning Objectives for Test Management

5.1 Test Organization

FL-5.1.1 (K2) Explain the benefits and drawbacks of independent testing
FL-5.1.2 (K1) Identify the tasks of a test manager and tester

5.2 Test Planning and Estimation

FL-5.2.1 (K2) Summarize the purpose and content of a test plan
FL-5.2.2 (K2) Differentiate between various test strategies
FL-5.2.3 (K2) Give examples of potential entry and exit criteria
FL-5.2.4 (K3) Apply knowledge of prioritization, and technical and logical dependencies, to schedule test execution for a given set of test cases
FL-5.2.5 (K1) Identify factors that influence the effort related to testing
FL-5.2.6 (K2) Explain the difference between two estimation techniques: the metrics-based technique and the expert-based technique

5.3 Test Monitoring and Control

FL-5.3.1 (K1) Recall metrics used for testing
FL-5.3.2 (K2) Summarize the purposes, contents, and audiences for test reports

5.4 Configuration Management

FL-5.4.1 (K2) Summarize how configuration management supports testing

5.5 Risks and Testing

FL-5.5.1 (K1) Define risk level by using likelihood and impact
FL-5.5.2 (K2) Distinguish between project and product risks
FL-5.5.3 (K2) Describe, by using examples, how product risk analysis may influence the thoroughness and scope of testing

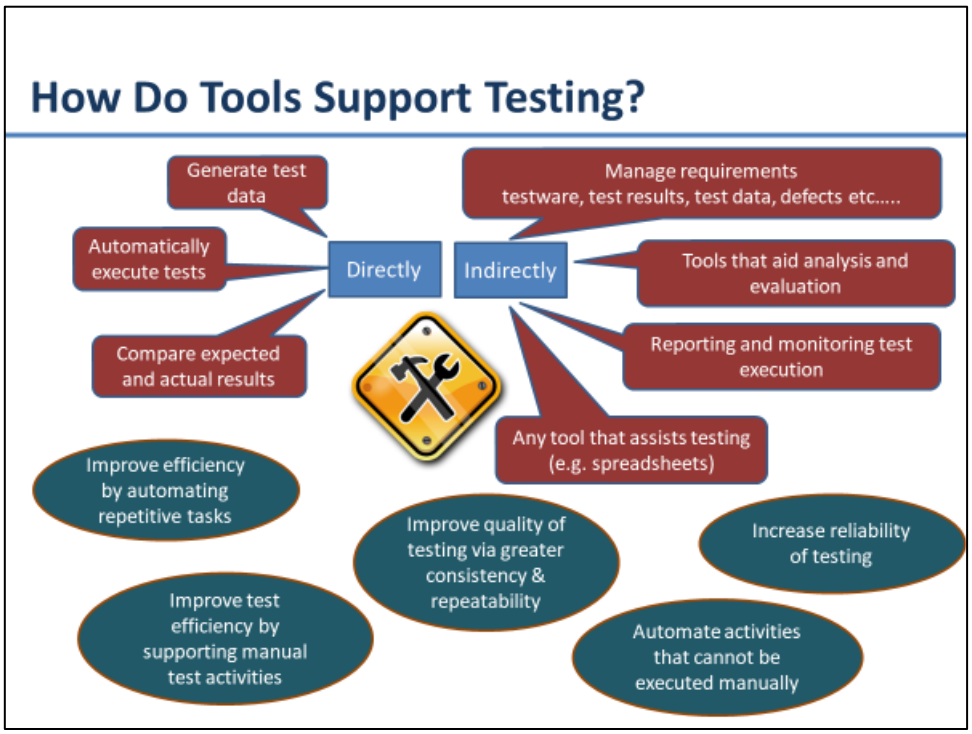
5.6 Defect Management

FL-5.6.1 (K3) Write a defect report, covering defects found during testing

Tool Support for Testing

Table of Contents

6.1 Types of Test Tools

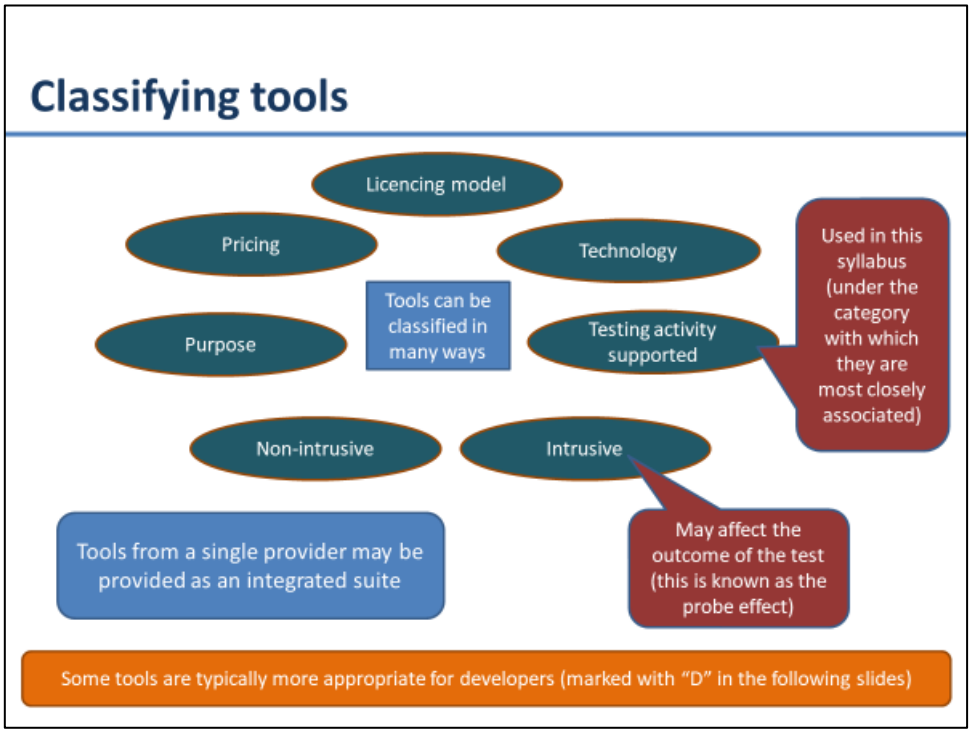


Test tools can be used to support one or more testing activities. Such tools include:

- Tools that are directly used in testing, such as test execution tools and test data preparation tools
- Tools that help to manage requirements, test cases, test procedures, automated test scripts, test results, test data, and defects, and for reporting and monitoring test execution
- Tools that are used for analysis and evaluation
- Any tool that assists in testing (a spreadsheet is also a test tool in this meaning)

Test tools can have one or more of the following purposes depending on the context:

- Improve the efficiency of test activities by automating repetitive tasks or tasks that require significant resources when done manually (e.g., test execution, regression testing)
- Improve the efficiency of test activities by supporting manual test activities throughout the test process (see section 1.4)
- Improve the quality of test activities by allowing for more consistent testing and a higher level of defect reproducibility
- Automate activities that cannot be executed manually (e.g., large scale performance testing)
- Increase reliability of testing (e.g., by automating large data comparisons or simulating behavior)

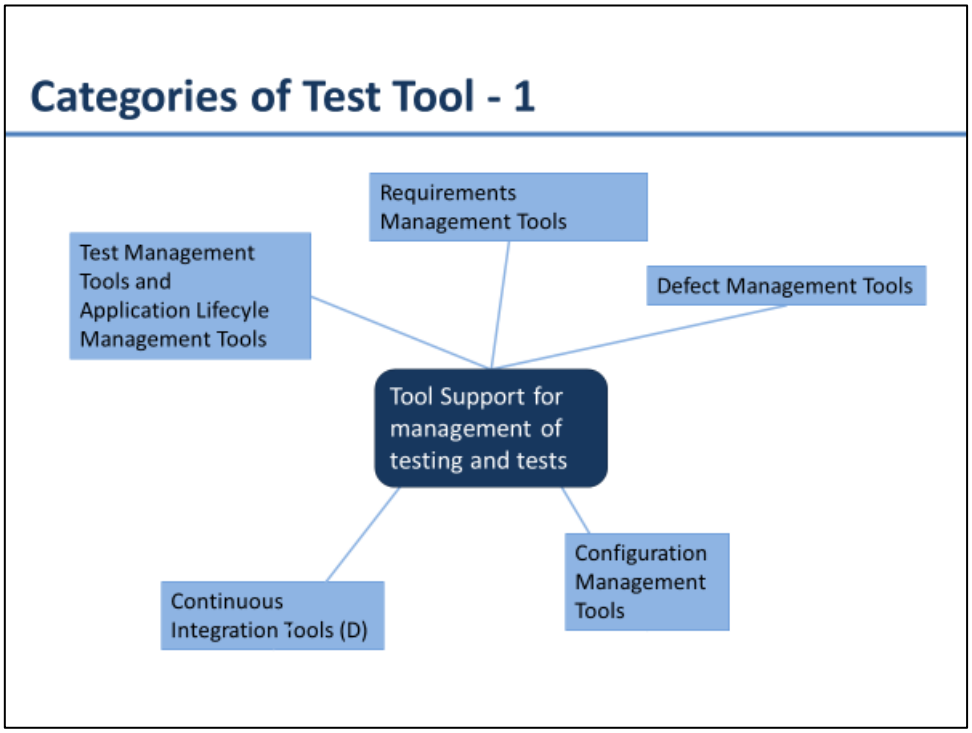


Tools can be classified based on several criteria such as purpose, pricing, licensing model (e.g., commercial or open source), and technology used. Tools are classified in this syllabus according to the test activities that they support.

Some tools clearly support only or mainly one activity; others may support more than one activity, but are classified under the activity with which they are most closely associated. Tools from a single provider, especially those that have been designed to work together, may be provided as an integrated suite.

Some types of test tools can be intrusive, which means that they may affect the actual outcome of the test. For example, the actual response times for an application may be different due to the extra instructions that are executed by a performance testing tool, or the amount of code coverage achieved may be distorted due to the use of a coverage tool. The consequence of using intrusive tools is called the probe effect.

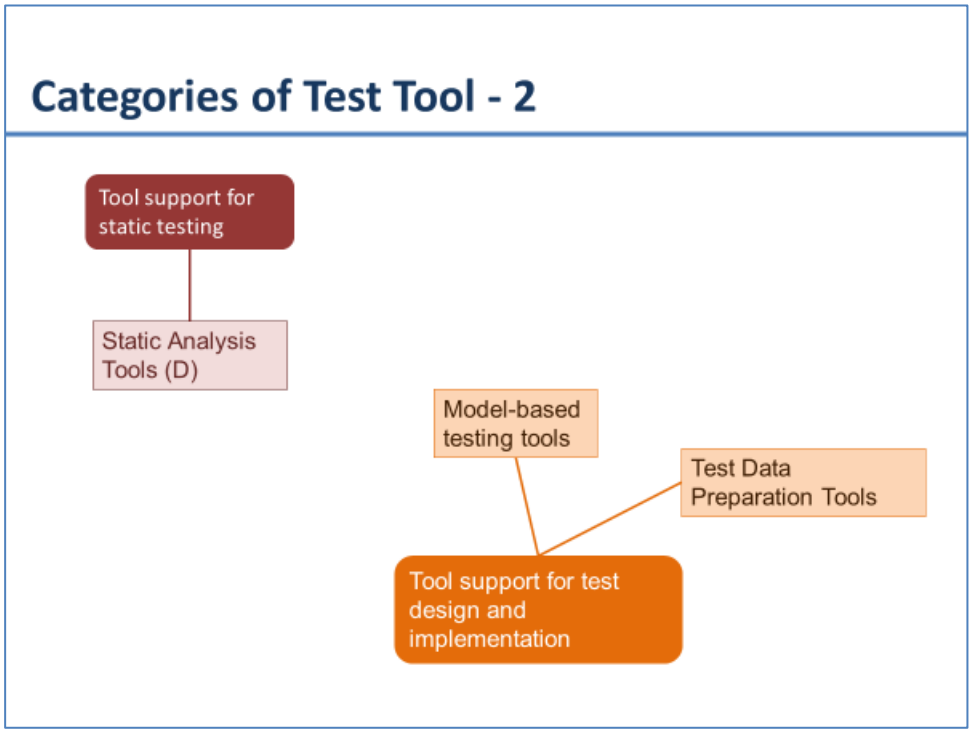
Some tools offer support that is typically more appropriate for developers (e.g., tools that are used during component and integration testing). Such tools are marked with "(D)" in the sections below.



Tool support for management of testing and testware

Management tools may apply to any test activities over the entire software development lifecycle. Examples of tools that support management of testing and testware include:

- Test management tools and application lifecycle management tools (ALM)
- Requirements management tools (e.g., traceability to test objects)
- Defect management tools
- Configuration management tools
- Continuous integration tools (D)



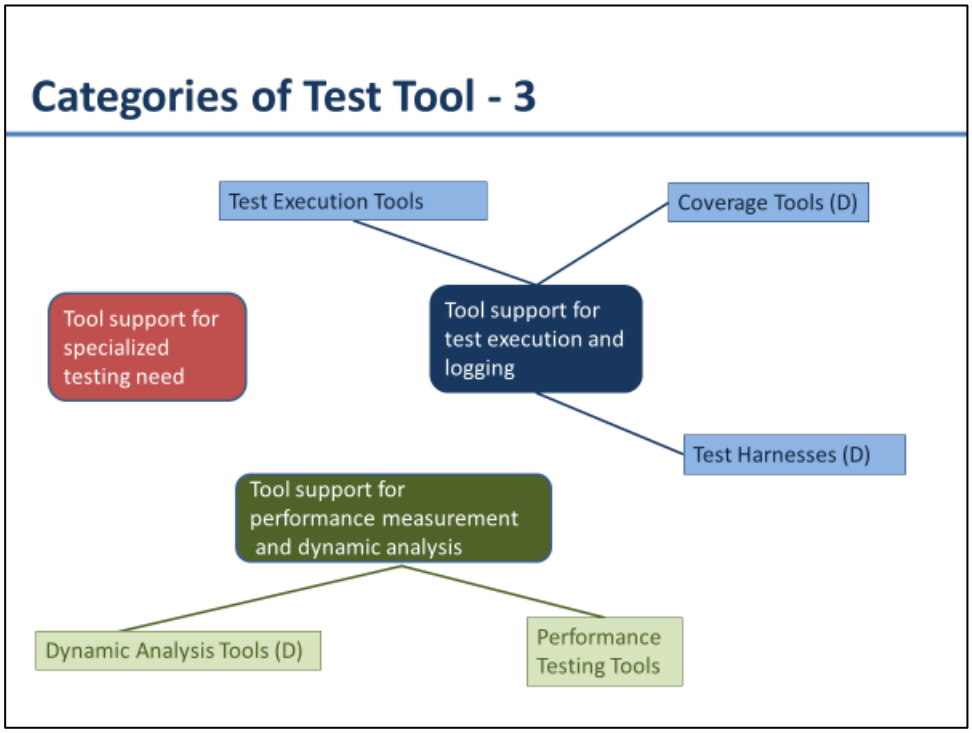
Tool support for static testing
Static testing tools are associated with the activities and benefits described in chapter 3. Examples of such tools include:

- Static analysis tools (D)

Tool support for test design and implementation
Test design tools aid in the creation of maintainable work products in test design and implementation, including test cases, test procedures and test data. Examples of such tools include:

- Model-Based testing tools
- Test data preparation tools

In some cases, tools that support test design and implementation may also support test execution and logging, or provide their outputs directly to other tools that support test execution and logging.



Tool support for test execution and logging
Many tools exist to support and enhance test execution and logging activities. Examples of these tools include:

- Test execution tools (e.g., to run regression tests)
- Coverage tools (e.g., requirements coverage, code coverage (D))
- Test harnesses (D)

Tool support for performance measurement and dynamic analysis
Performance measurement and dynamic analysis tools are essential in supporting performance and load testing activities, as these activities cannot effectively be done manually. Examples of these tools include:

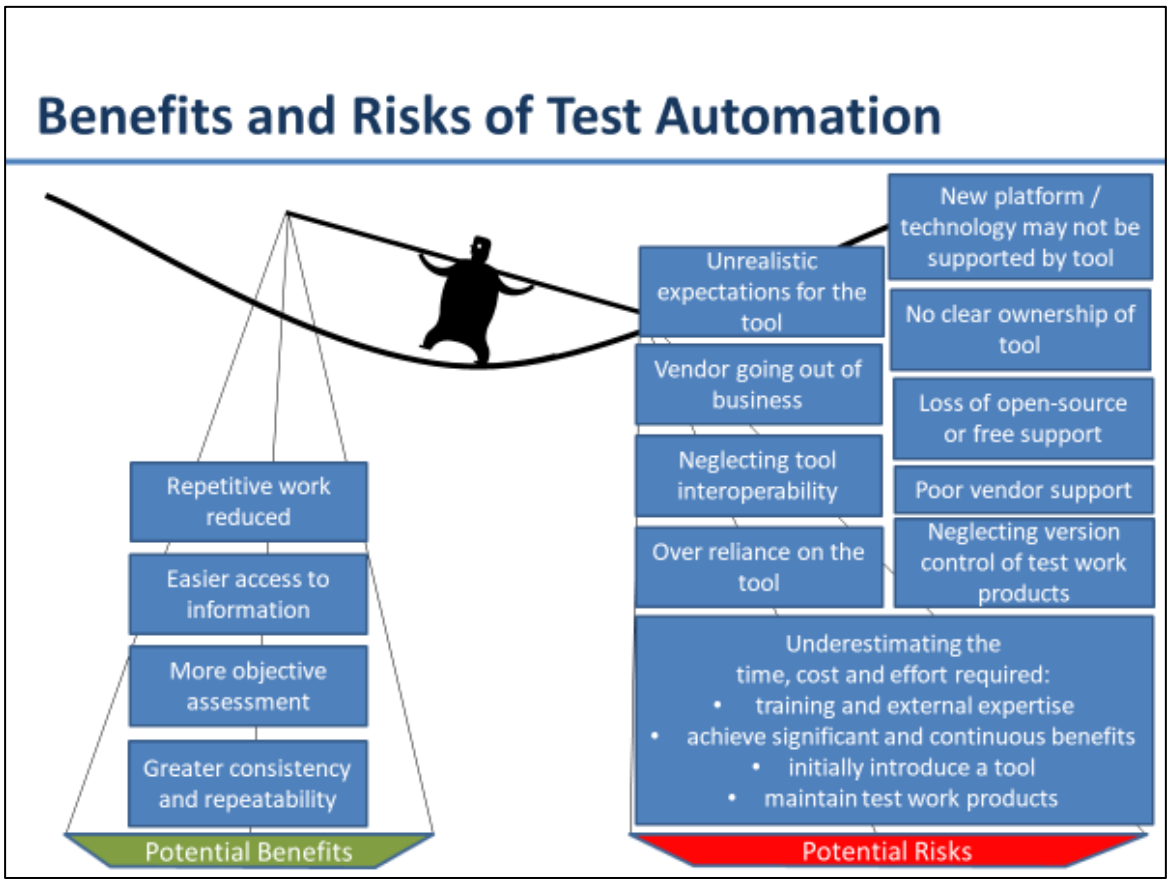
- Performance testing tools
- Dynamic analysis tools (D)

Tool support for specialized testing needs
In addition to tools that support the general test process, there are many other tools that support more specific testing for non-functional characteristics.

Summary

- The terms “test tool” and “test automation” describe any software utility that can be used to support testing activities
- This syllabus categorises tools according to the activity with which they are most closely associated
- Tools are used in different stages of the software development life cycle
- Tools are used in different stages of the test process
- Some tools are more appropriate for developers (D)

6.2 Effective Use of Tools



Simply acquiring a tool does not guarantee success. Each new tool introduced into an organization will require effort to achieve real and lasting benefits. There are potential benefits and opportunities with the use of tools in testing, but there are also risks. This is particularly true of test execution tools (which is often referred to as test automation).

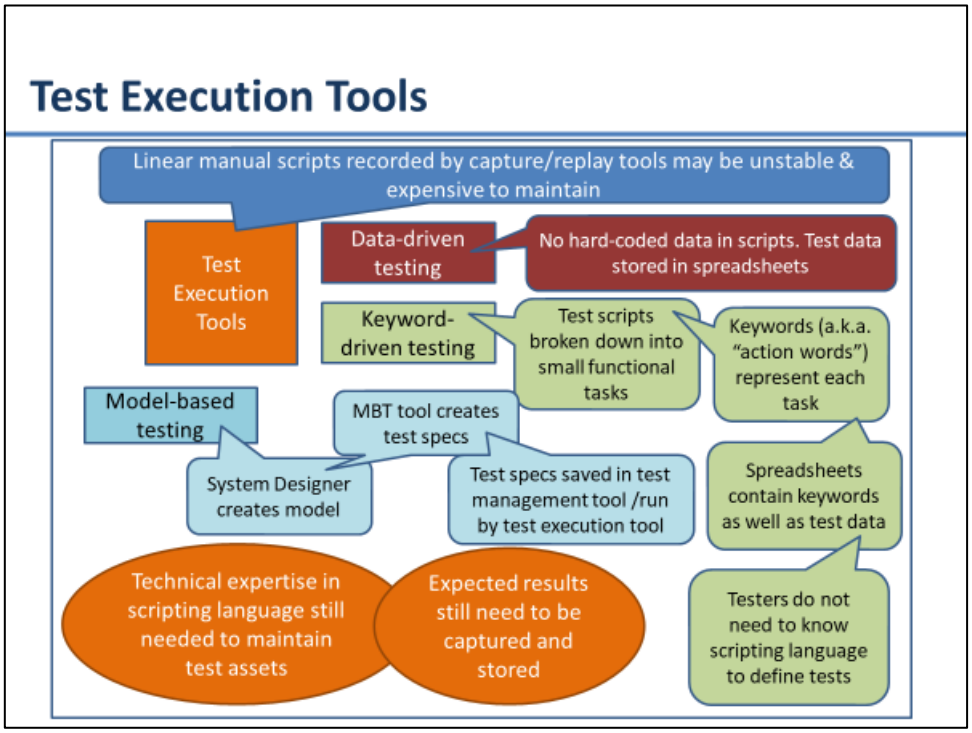
Potential benefits of using tools to support test execution include:

- Reduction in repetitive manual work (e.g., running regression tests, environment set up/tear down tasks, re-entering the same test data, and checking against coding standards), thus saving time
- Greater consistency and repeatability (e.g., test data is created in a coherent manner, tests are executed by a tool in the same order with the same frequency, and tests are consistently derived from requirements)
- More objective assessment (e.g., static measures, coverage)
- Easier access to information about testing (e.g., statistics and graphs about test progress, defect rates and performance)

See potential risks on next page

Potential risks of using tools to support testing include:

- Expectations for the tool may be unrealistic (including functionality and ease of use)
- The time, cost and effort for the initial introduction of a tool may be under-estimated (including training and external expertise)
- The time and effort needed to achieve significant and continuing benefits from the tool may be under-estimated (including the need for changes in the test process and continuous improvement in the way the tool is used)
- The effort required to maintain the test work products generated by the tool may be under-estimated
- The tool may be relied on too much (seen as a replacement for test design or execution, or the use of automated testing where manual testing would be better)
- Version control of test work products may be neglected
- Relationships and interoperability issues between critical tools may be neglected, such as requirements management tools, configuration management tools, defect management tools and tools from multiple vendors
- The tool vendor may go out of business, retire the tool, or sell the tool to a different vendor
- The vendor may provide a poor response for support, upgrades, and defect fixes
- An open source project may be suspended
- A new platform or technology may not be supported by the tool
- There may be no clear ownership of the tool (e.g., for mentoring, updates, etc.)



In order to have a smooth and successful implementation, there are a number of things that ought to be considered when selecting and integrating test execution and test management tools into an organization.

Test execution tools
Test execution tools execute test objects using automated test scripts. This type of tools often requires significant effort in order to achieve significant benefits.

Capturing test approach: Capturing tests by recording the actions of a manual tester seems attractive, but this approach does not scale to large numbers of test scripts. A captured script is a linear representation with specific data and actions as part of each script. This type of script may be unstable when unexpected events occur and require ongoing maintenance as the system's user interface evolves over time.

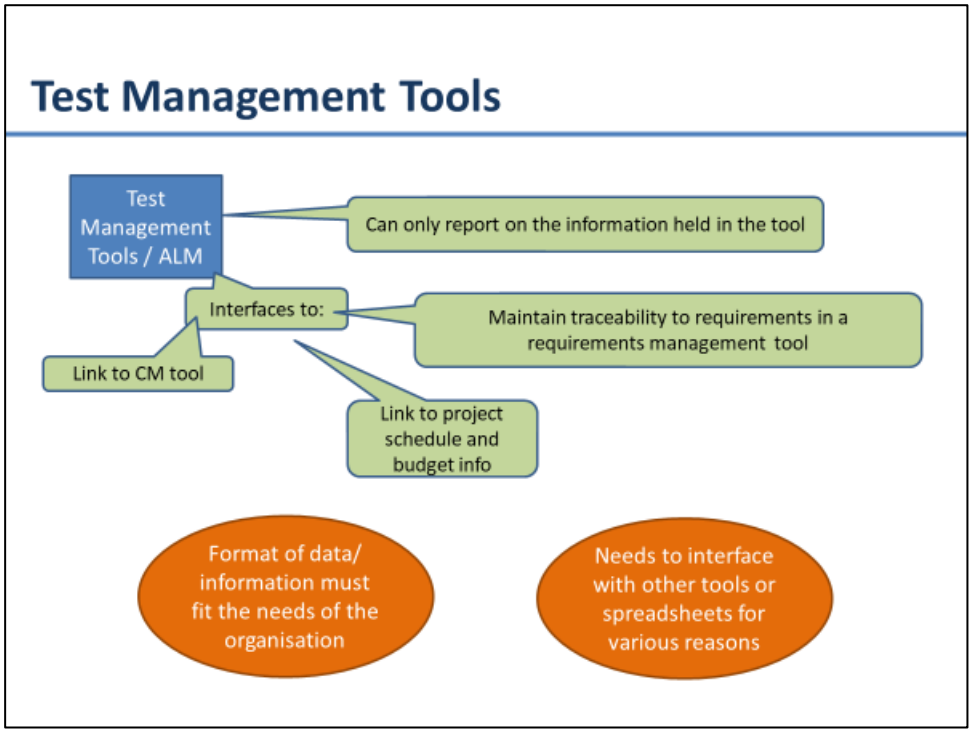
Data-driven test approach:
This test approach separates out the test inputs and expected results, usually into a spreadsheet, and uses a more generic test script that can read the input data and execute the same test script with different data.

Keyword-driven test approach:
This test approach, a generic script processes keywords describing the actions to be taken (also called action words), which then calls keyword scripts to process the associated test data.

The above approaches require someone to have expertise in the scripting language (testers, developers or specialists in test automation). When using data-driven or keyword-driven test approaches testers who are not familiar with the scripting language can also contribute by creating test data and/or keywords for these predefined scripts. Regardless of the scripting technique used, the expected results for each test need to be compared to actual results from the test, either dynamically (while the test is running) or stored for later (post-execution) comparison. Further details

and examples of data-driven and keyword-driven testing approaches are given in ISTQB-CTAL-TAE, Fewster 1999 and Buwalda 2001.

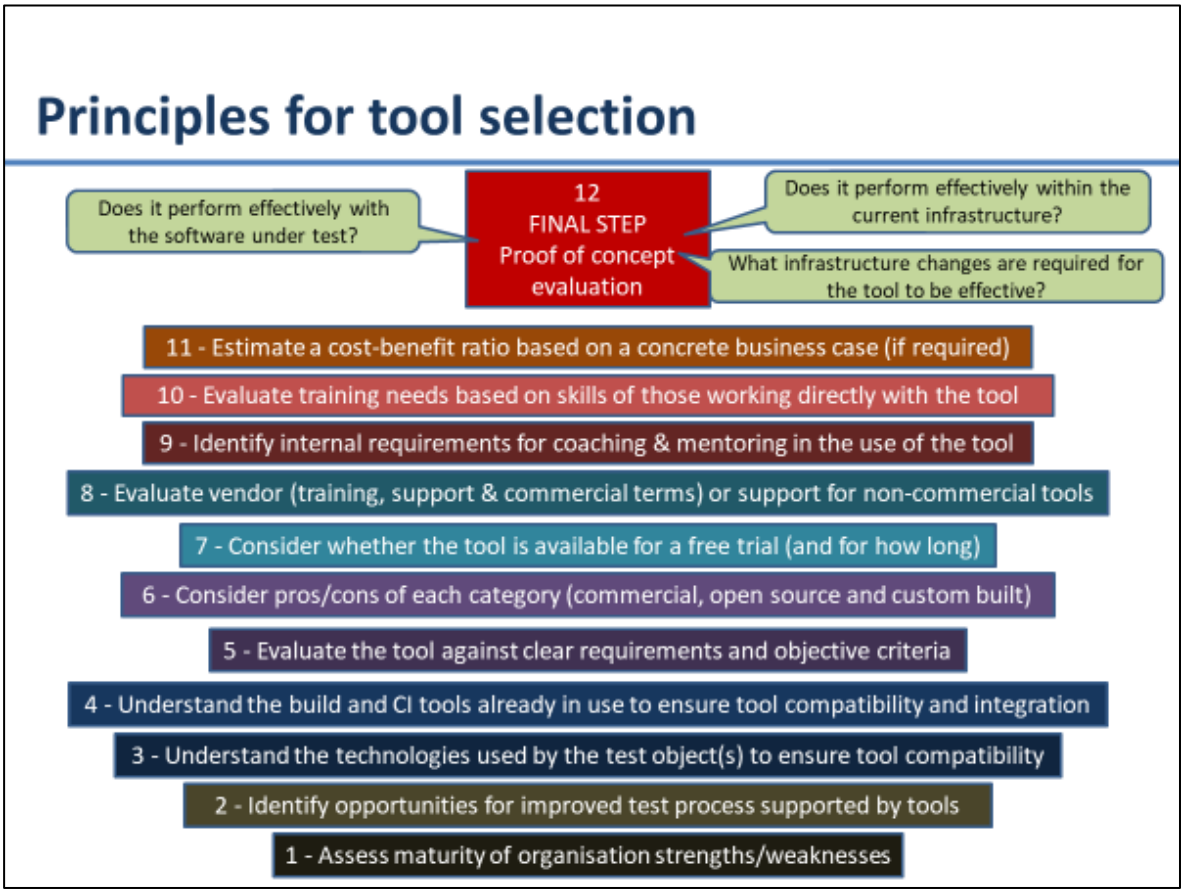
Model-Based testing (MBT) tools:
These enable a functional specification to be captured in the form of a model, such as an activity diagram. This task is generally performed by a system designer. The MBT tool interprets the model in order to create test case specifications which can then be saved in a test management tool and/or executed by a test execution tool (see ISTQB-CTFL-MBT).



Test management tools
Test management tools often need to interface with other tools or spreadsheets for various reasons, including:

- To produce useful information in a format that fits the needs of the organization
- To maintain consistent traceability to requirements in a requirements management tool
- To link with test object version information in the configuration management tool

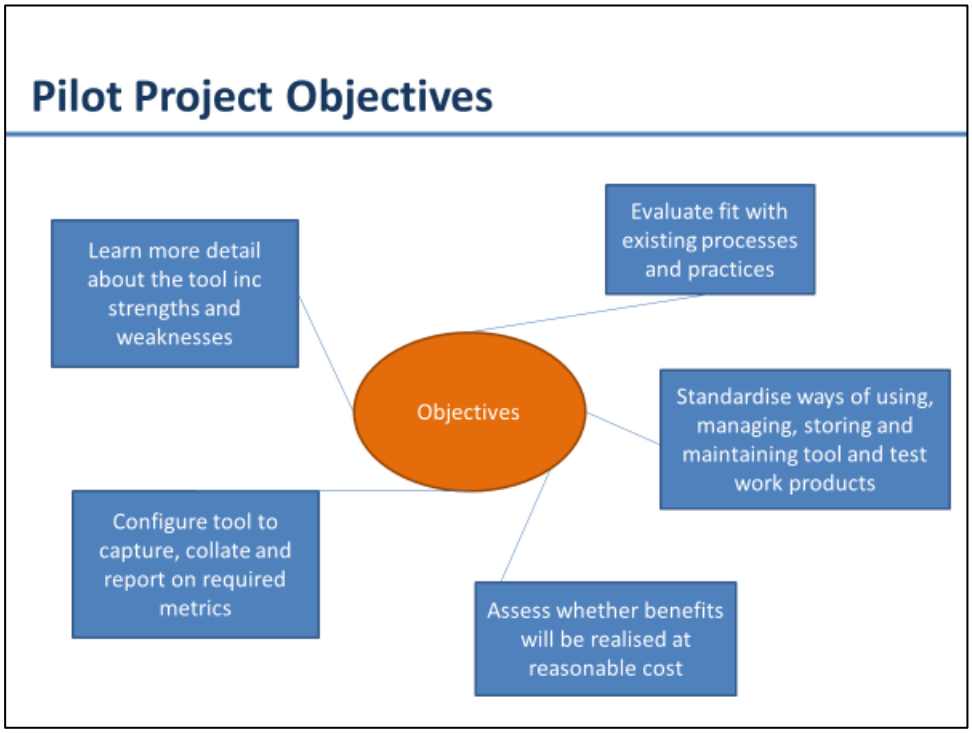
This is particularly important to consider when using an integrated tool (e.g., Application Lifecycle Management), which includes a test management module as well as other modules (e.g., project schedule and budget information) that are used by different groups within an organization.



The main considerations in selecting a tool for an organization include:

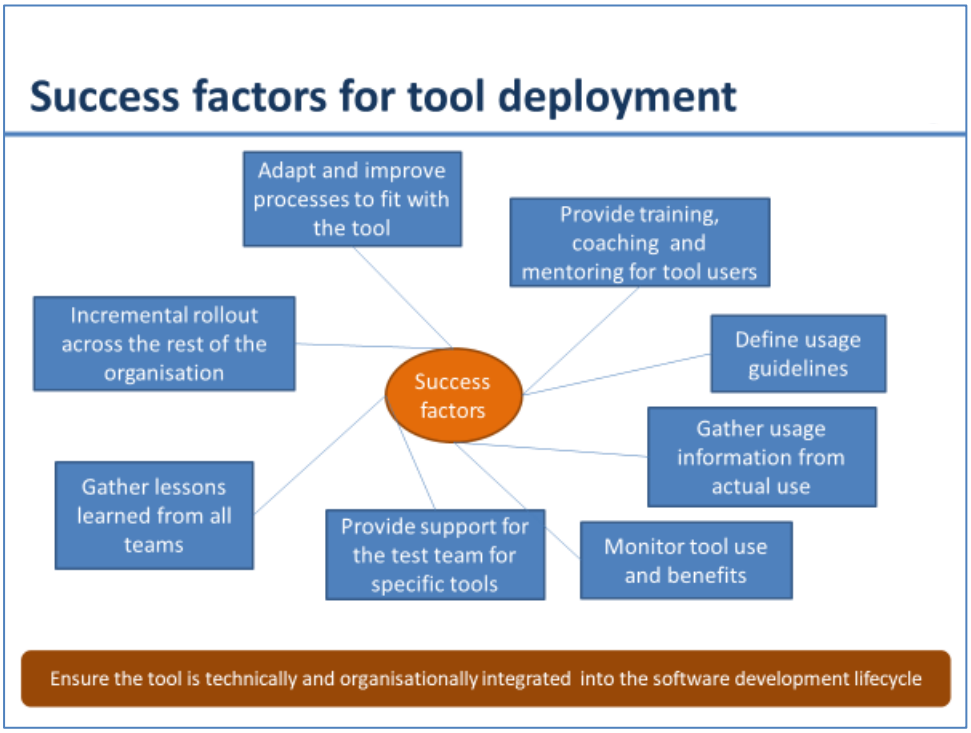
- Assessment of the maturity of the own organization, its strengths and weaknesses
- Identification of opportunities for an improved test process supported by tools
- Understanding of the technologies used by the test object(s), in order to select a tool that is compatible with that technology
- Understanding the build and continuous integration tools already in use within the organization, in order to ensure tool compatibility and integration
- Evaluation of the tool against clear requirements and objective criteria
- Consideration of whether or not the tool is available for a free trial period (and for how long)
- Evaluation of the vendor (including training, support and commercial aspects) or support for noncommercial (e.g., open source) tools
- Identification of internal requirements for coaching and mentoring in the use of the tool
- Evaluation of training needs, considering the testing (and test automation) skills of those who will be working directly with the tool(s)
- Consideration of pros and cons of various licensing models (e.g., commercial or open source)
- Estimation of a cost-benefit ratio based on a concrete business case (if required)

As a final step, a proof-of-concept evaluation should be done to establish whether the tool performs effectively with the software under test and within the current infrastructure or, if necessary, to identify changes needed to that infrastructure to use the tool effectively.



After completing the tool selection and a successful proof-of-concept, introducing the selected tool into an organization generally starts with a pilot project, which has the following objectives:

- Gaining in-depth knowledge about the tool, understanding both its strengths and weaknesses
- Evaluating how the tool fits with existing processes and practices, and determining what would need to change
- Deciding on standard ways of using, managing, storing, and maintaining the tool and the test work products (e.g., deciding on naming conventions for files and tests, selecting coding standards, creating libraries and defining the modularity of test suites)
- Assessing whether the benefits will be achieved at reasonable cost
- Understanding the metrics that you wish the tool to collect and report, and configuring the tool to ensure these metrics can be captured and reported



Success factors for evaluation, implementation, deployment, and on-going support of tools within an organization include:

- Rolling out the tool to the rest of the organization incrementally
- Adapting and improving processes to fit with the use of the tool
- Providing training, coaching, and mentoring for tool users
- Defining guidelines for the use of the tool (e.g., internal standards for automation)
- Implementing a way to gather usage information from the actual use of the tool
- Monitoring tool use and benefits
- Providing support to the users of a given tool
- Gathering lessons learned from all users

It is also important to ensure that the tool is technically and organizationally integrated into the software development lifecycle, which may involve separate organizations responsible for operations and/or third party suppliers.

Summary

- There are specific risks that may prevent tool benefits from being realised
- Data-driven, keyword-driven and model-based testing approaches are special considerations for test execution tools
- Special considerations also relate to test management tools
- Assess the needs of the organisation and select an appropriate tool which meets those needs
- Evaluate the vendor
- Provide that coaching, mentoring and training education
- Ensure that the business case is sound
- Run a "Proof of Concept" exercise
- Pilot introduction of the tool followed by gradual roll-out
- Monitor the continued use of the tool

End of section exercise

- Attempt the following Questions from Sample Exam B
- Questions 39 to 40
- Please mark your own (See Sample Exam B Answers) and read the answer justifications as needed

You have 3 minutes

Learning Objectives for Tool Support for Testing

In order to complete this section, you should satisfy yourself that you are able to understand or perform the following items to the standard indicated by the corresponding “K - learning level”. For an explanation of “K levels”, please see the course introduction section – “Learning Objectives and Levels of Knowledge”.

Keywords

data-driven testing, keyword-driven testing, test automation, test execution tool, test management tool

Learning Objectives for Test Tools

6.1 Test tool considerations

FL-6.1.1 (K2) Classify test tools according to their purpose and the test activities they support
FL-6.1.2 (K1) Identify benefits and risks of test automation
FL-6.1.3 (K1) Remember special considerations for test execution and test management tools

6.2 Effective use of tools

FL-6.2.1 (K1) Identify the main principles for selecting a tool
FL-6.2.2 (K1) Recall the objectives for using pilot projects to introduce tools
FL-6.2.3 (K1) Identify the success factors for evaluation,