IN3240

Chapter 1

False-fail result: A test result in which a defect is reported although no such defect actually exists in the test object.

Error (mistake) A human action that produces an incorrect result.

Defect (bug, fault) A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

Failure Deviation of the component or system from its expected delivery, service or result.

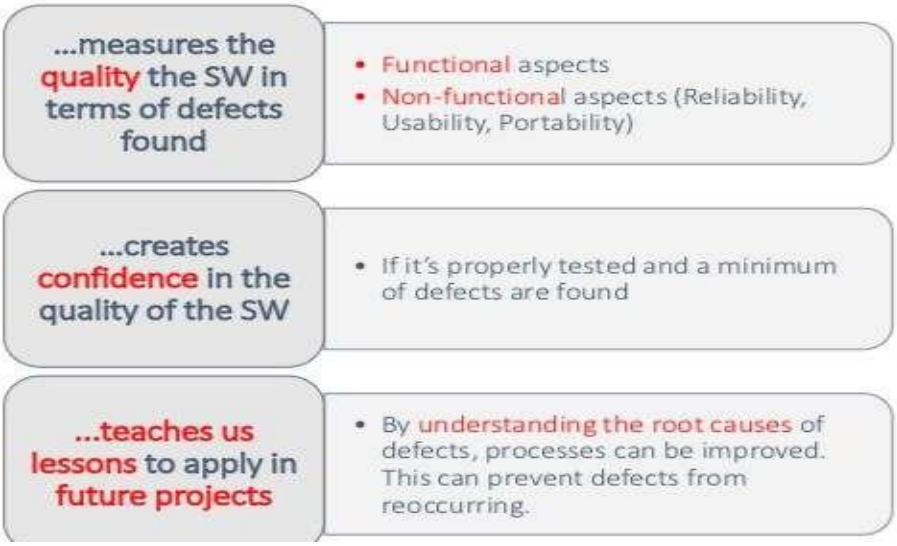
False-pass result: A test result which fails to identify the presence of a defect that is actuallypresent in the test object.

Both causes of errors produce defects (= faults, bugs) in the code.

• Defects, if executed, may result in failures of the SW system (the system will fail to do what it should).

• Failures can affect seriously the users of the SW system, i.e.: • Break pedal not working in some cars

• Miscalculations in financial SW systems



Role of testing:

• To reduce the risk of problems occurring during operation

• To check if the SW system meets:

• legal requirements

• Industry specific standards

To learn more about the SW system

Testing activities:

- **Test planning:** In test planning, we establish (and update) the scope, approach, resources, schedule, and specific tasks in the intended test activities that comprise the rest of the test process. Test planning should identify test items, the features to be tested and not tested, the roles and responsibilities of the participants and stakeholders, the relationship between the testers and the developers of the test items, the extent to which testing is independent of development of these work items (see section 1.5 below), the test environments required, the appropriate test design techniques, entry, and exit criteria, and how we'll handle project risks related to testing. Test planning often produces, as a deliverable work product, a test plan. Test planning is discussed in Chapter 5.
- **Test control:** While test planning is essential, things don't always go according to plan. In test control, we develop and carry out corrective actions to get a test project back on track when we deviate from the plan. Test control is discussed in Chapter 5.
- **Test analysis:** In test analysis, we identify what to test, choosing the test conditions we need to cover. These conditions are any item or event that we can and should verify using one or more

test cases. Test conditions can be functions, transactions, features, quality attributes, quality risks, or structural elements. Test analysis is discussed in Chapter 4.

- **Test design:** In test design, we determine how we will test what we decided to test during test analysis. Using test design techniques, we transform these general test conditions and the general testing objectives from the test plan into tangible test cases at the appropriate level of detail. Test design generally, and specific test design techniques, are discussed in Chapter 4.
- **Test implementation:** In test implementation, we carry out the remaining activities required to be ready for test execution, such as developing and prioritizing our test procedures, creating test data, and setting up test environments. These elements of test implementation are covered in Chapter 4. In many cases, we want to use automated test execution as part of our test process. In such cases, test implementation includes preparing test harnesses and writing automated test scripts. These elements of test implementation are covered in Chapter 6.
- **Test execution:** In test execution, we run our tests against the test object (also called the system under test).
- **Checking results:** As part test execution, we see the actual results of the test case, the consequences and outcomes. These include outputs to screens, changes to data, reports, and communication messages sent out. We must compare these actual results against expected results to determine the pass/fail status of the test. Defining expected results is discussed in Chapter 4, while managing test execution, including checking of results, is discussed in Chapter 5.
- **Evaluating exit criteria:** At a high level, exit criteria are a set of conditions that would allow some part of a process to complete. Exit criteria are usually defined during test planning by working with the project and product stakeholders to balance quality needs against other priorities and various project constraints. Such criteria ensure that everything that should be done has been done before we declare some set of activities finished. Specifically, test exit criteria help us report our results (as we can report progress against the criteria) as well as helping us plan when to stop testing. Establishing and evaluating exit criteria are discussed in Chapter 5. Test results reporting: In test results reporting, we want to report our progress against exit criteria, as described above. This often involves details related to the status of the test project, the test process, and quality of the system under test. We'll discuss test results reporting in Chapter 5.
- **Test closure:** Test closure involves collecting test process data related to the various completed test activities in order to consolidate our experience, re-useable testware, important facts, and relevant metrics. Test closure is discussed in section 1.4 below.

Test design specification A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases.
Test control A test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned.
Test case A set of input values, execution preconditions, expected
results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
Test objective A reason or purpose for designing and executing a test.
Testing The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.
Requirement A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a
system or system component to satisfy a contract, standard, specification, or other formally imposed document.
Review An evaluation of a product or project status to ascertain

discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough.
Debugging The process of finding, analyzing and removing the causes of failures in software.
Confirmation testing (re-testing) Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions. [Note: While the Glossary uses re-testing as the preferred term, this preference is out of step with ordinary usage and with the actual Englishlanguage definition of 're-testing'.]

Seven testing principles

Principle 1:	**Testing shows presence of defects**	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, it is not a proof of correctness.
Principle 2:	**Exhaustive testing is impossible**	Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts.
Principle 3:	**Early testing**	To find defects early, testing activities shall be started as early as possible in the software or system development life cycle, and shall be focused on defined objectives.
Principle 4:	**Defect clustering**	Testing effort shall be focused proportionally to the expected and later observed defect density of modules. 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.
Principle 5:	**Pesticide paradox**	If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new defects. To overcome this 'pesticide paradox', test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to find potentially more defects.
Principle 6:	**Testing is context dependent**	Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.
Principle 7:	**Absence-of-errors fallacy**	Finding and fixing defects does not help if the system built is unusable and does not fulfil the users' needs and expectations.

Fundamental test process

- Planning and control
- Analysis and design
- Implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

A good tester needs:

- Curiosity
- Professional pessimism
- Attention to details
- Good communication skills
- Experience at error guessing
- To communicate defects and failures in a constructive way: fact-focused reports and review of findings

Test strategy A highlevel description of the test levels to be performed and the testing within those levels for an organization or program (one or more projects).

Exhaustive testing (complete testing) A test approach in which the test suite comprises all combinations of input values and preconditions.

Test execution The process of running a test on the component or system under test, producing actual result(s).

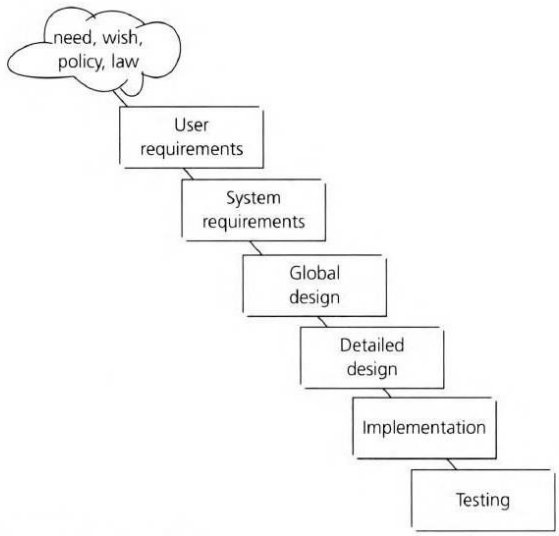
Test approach The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed.

Chapter 2

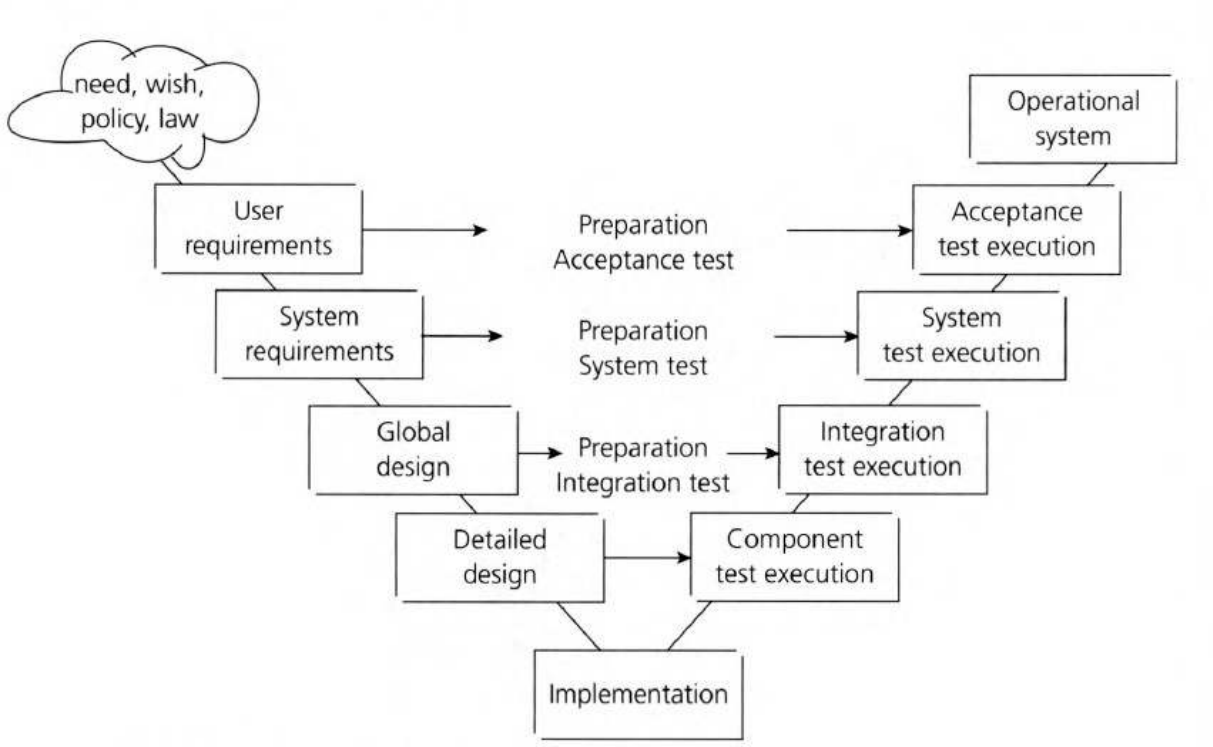
Verification Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.

Validation Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

Waterfall model



V-model A framework to describe the software development lifecycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development lifecycle.



• Testing needs to begin as early as possible in the life cycle.
• Testing can be integrated into each phase of the life cycle.
• Within the V-model, validation testing takes place
especially during:

- the early stages, i.e. reviewing the user requirements
- and late in the life cycle, i.e. during user acceptance testing

The four test levels used, each with their own objectives, are:

• ***component testing***: searches for defects in and verifies the functioning of software components (e.g. modules, programs, objects, classes, etc.) that are separately testable;

Component testing includes testing of functionality and specific non-functional characteristics, such as:

- resource-behavior (e.g. memory leaks)
- robustness testing
- structural testing (e.g. branch coverage).

All materials that are applicable for the component under test:

• Specification of the component • Software design
• The data model
• As well as the code itself

Writing code to test the project code

• Stubs, drivers and simulators may be used.

Stubs:

Code that replaces a called component in order to simulate its purpose (i.e. "hard-coded" data to replace data from a database).

Drivers:

Code that replaces an other software component in order to call the component under test.

Component testing usually involves the programmer who wrote the code.
Defects are fixed as soon as they are found, without formal recording of incidents.
TDD test-driven development
• prepare and automate test cases before coding
• used in XP (extreme programming)

• ***integration testing***: tests interfaces between components, interactions to different parts of a system such as an operating system, file system and hardware or interfaces between systems;

Test interactions with different parts of a system, such as:
• theoperatingsystem
• file system
• hardware
• interfaces between systems

Test basis:
• Software and system design
• The system architecture
• Workflows/use cases

The item under test includes
• builds including some or all component of the system
• Thedatabaseelements
• Systeminfrastructure
• Interface between components or objects
• Systemconfiguration
• Configurationdata

Integration testing-types
Component integration
Tests the interactions between software components is done after component testing
System integration
Tests the interactions between different systems is done after system testing.

Approaches and responsibilities
• Start integration with those components that are expected to cause most problems.
• To reduce the risk of late defect discovery, integration should normally be incremental rather than "big bang".
• Both functional and structural approaches may be used.
• Ideally, testers should understand the architecture and influence integration planning.
• Can be done by the developers or by a separate team.

• ***system testing***: concerned with the behaviour of the whole system/product as defined by the scope of a development project or product. The main focus of system testing is verification against specified requirements;

Testing the behavior of the whole system as defined by the scope of the project.

Test basis:
• System requirements specification, both functional and non-functional
• Business processes

• Risk analysis
• Use cases
• Other high level descriptions of the system behavior, interactions with OS/system resources
• Requirements may exist as text and/or models.
• Testers also need to deal with incomplete or undocumented requirements.

Test objects:
• Theentireintegratedsystem
• Usermanuals
• Operationmanuals
• Systemconfigurationinformation
• Configurationdata

Test environment should correspond to the production environment as much as possible.
- First, the most suited black-box technique
- Then, white-box technique to assess the thoroughness of testing

An independent test team may be responsible for the testing
The level of independence is based on the applicable risk level

• ***acceptance testing***: validation testing with respect to user needs, requirements, and business processes conducted to determine whether or not to accept the system.

The questions to be answered:
• Can the system be released?
• What are the outstanding risks?
• Has development met its obligations?

The goal is to establish confidence in
• Thesystem
• Non-functionalcharacteristicsofthesystem

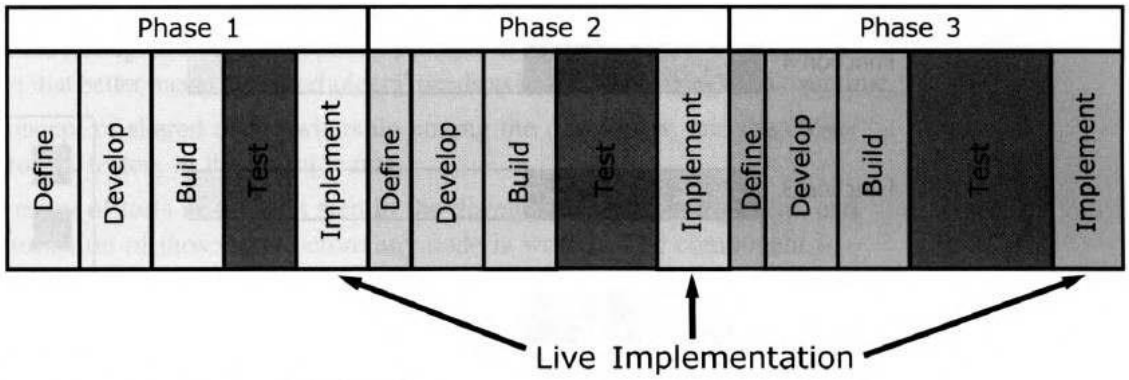
Test basis:
• User requirements specification
• Use cases
• System requirements specification
• Business processes
• Risk analysis

Types:
User acceptance testing – validate the fitness for use of the system by users.
• Operational testing, usually done by the system administrators:
  - testing of backup/restore
  - disaster recovery
  - user management
  - maintenance tasks
  - periodic checks of security vulnerabilities
• Contract and regulation acceptance testing performed against a contract's acceptance criteria (i.e. governmental, legal or safety regulations)
• Alpha and beta testing

• Alpha testing is performed at the developing organization's site.
• Beta testing (field testing), is performed by people at their own locations.

Both are performed by potential customers, not the developers of the product.

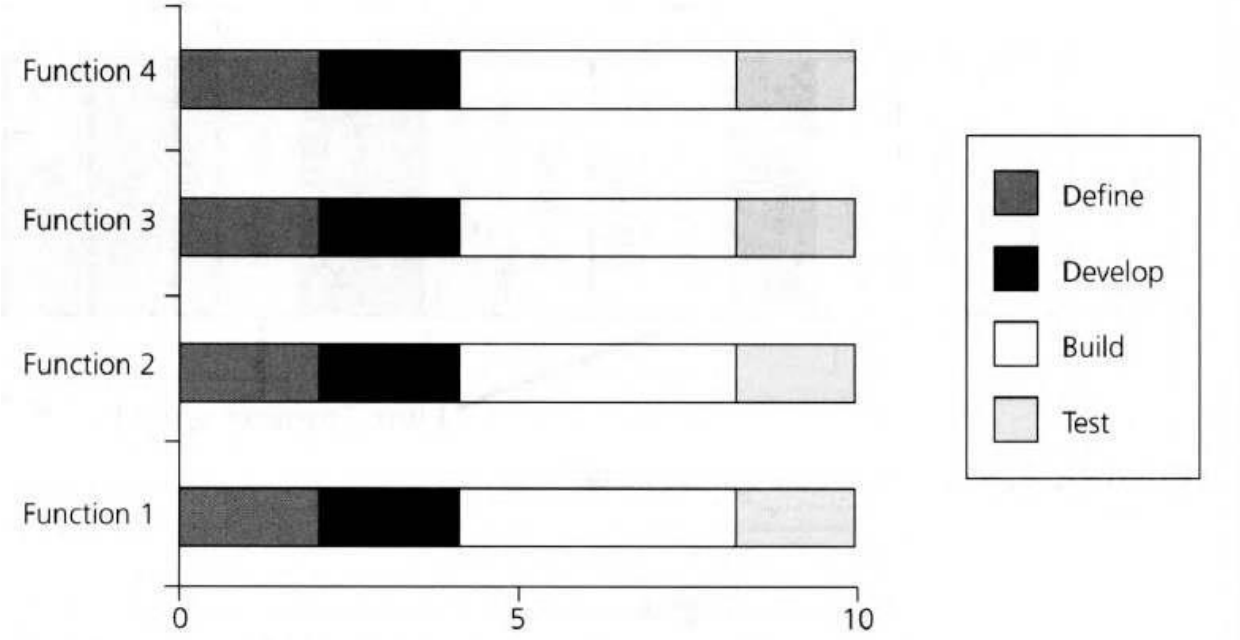
Iterative development model



Incremental development model A development lifecycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this lifecycle model, each subproject follows a 'mini V-model' with its own design, coding and testing phases.

Iterative development model A development lifecycle where a project is broken into a usually large number of iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows from iteration to iteration to become the final product.

Rapid application development



Rapid Application Development (RAD) is formally a parallel development of functions and subsequent integration. An early business-focused solution in the market place gives an early return on investment (ROD and can provide valuable marketing information for the business. Validation with the RAD development process is thus an early and major activity.

Agile development

Agile software development is a group of software development methodologies based on iterative incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

Agile development provides both benefits and challenges for testers. Some of the benefits are:

• the focus on working software and good quality code;
• the inclusion of testing as part of and the starting point of software development (test-driven development);
• accessibility of business stakeholders to help testers resolve questions about expected behaviour of the system;
• self-organizing teams where the whole team is responsible for quality and giving testers more autonomy in their work; and
• simplicity of design that should be easier to test.

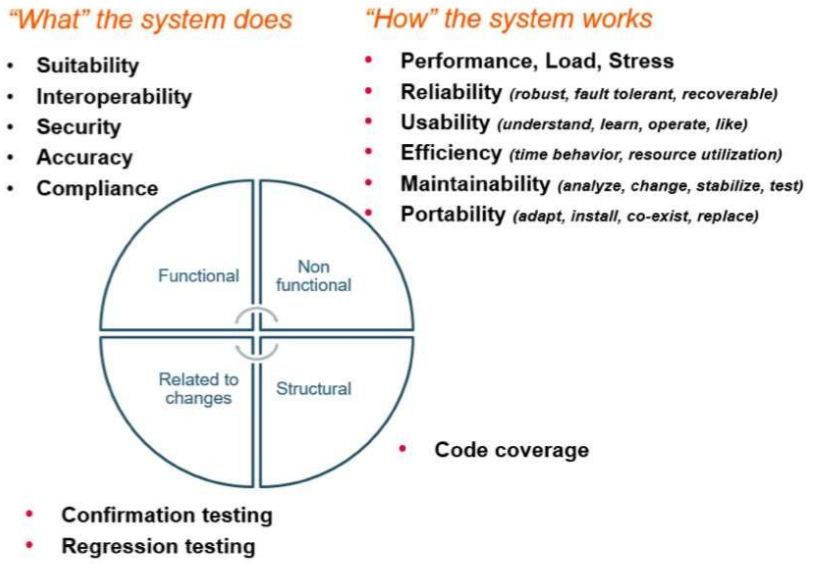
There are also some significant challenges for testers when moving to an agile development approach.

• Testers used to working with well-documented requirements will be designing tests from a different kind of test basis: less formal and subject to change. The manifesto does not say that documentation is no longer necessary or that it has no value, but it is often interpreted that way.
• Because developers are doing more component testing, there may be a perception that testers are not needed. But component testing and confirmationbased acceptance testing by only business representatives may miss major problems. System testing, with its wider perspective and emphasis on nonfunctional testing as well as end to end functional testing is needed, even if it doesn't fit comfortably into a sprint.
• The tester's role is different: since there is less documentation and more personal interaction within an agile team, testers need to adapt to this style of working, and this can be difficult for some testers. Testers may be acting more as coaches in testing to both stakeholders and developers, who may not have a lot of testing knowledge.
• Although there is less to test in one iteration than a whole system, there is also a constant time pressure and less time to think about the testing for the new features.
• Because each increment is adding to an existing working system, regression testing becomes extremely important, and automation becomes more beneficial. However, simply taking existing automated component or component integration tests may not make an adequate regression suite.

In summary, whichever life cycle model is being used, there are several characteristics of good testing:

• for every development activity there is a corresponding testing activity;
• each test level has test objectives specific to that level;
• the analysis and design of tests for a given test level should begin during the corresponding development activity;
• testers should be involved in reviewing documents as soon as drafts are available in the development cycle.

Test types



<u>Functional testing (Black box testing)</u>

<u>Objectives</u>

Test what a system should do and consider the external behavior of the software.

<u>Test levels</u>

May be performed at all test levels

<u>Test basis</u>

The expected behavior description can be found in work products such as:

- requirements specification - business processes
- use cases
- functional specifications
- may be undocumented

<u>Non functional testing</u>

<u>Objectives</u>

Measuring characteristics of software that can be quantified on a varying scale: e.g. response times for performance testing

<u>Test levels</u>

May be performed at all test levels

You can find more about them in 'Software Engineering – Software Product Quality' (ISO 9126).

<u>Structural testing (white box testing)</u>

<u>Objectives</u>

Measuring the thoroughness of testing through assessment of the coverage of a set of structural elements or coverage items.

<u>Test levels</u>

May be performed at all test levels, but especially in component testing and component integration testing.

Test basis

Structural testing is based on the structure of the code as well as the architecture of the system (e.g. a calling hierarchy, a business model or a menu structure)

Approach

Structural techniques are best used after specification- based techniques, in order to help measure the thoroughness of testing.

Tools

Coverage measurement tools assess the percentage of executable elements (e.g. statements or decision outcomes) that have been exercised.

Testing related to changes, confirmation and regression testing

Confirmation testing

After a defect is detected and fixed, the software should be retested to confirm that the original defect has been successfully removed.

Regression testing

The repeated testing of an already tested program, after modification, to discover any defects introduced or uncovered as a result of the change(s).

Objective

To verify that modifications in the software or the environment have not caused unintended side effects and that the system still meets its requirements.

Test levels

May be performed at all test levels, applies to functional, non-functional and structural testing.

Approach

The extent of regression testing is based on the risk of finding defects in software that was working previously.

Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation.

If the regression test suite is very large it may be more appropriate to select a subset for execution.

Maintenance testing

Objectives

Maintenance testing is done on an existing operational system, and is triggered by modifications, migration, or retirement of the software or system.

Types

Modifications

• Planned enhancement changes (e.g. release-based)
• Corrective and emergency changes (patches)
• Changes of environment (operating system or database upgrades)

Migration (e.g. from one platform to another)

• operational tests of the new environment
• tests on the changed software.

Retirement of a system

The testing of data migration or archiving if long data- retention periods are required.

Scope

The scope of maintenance testing is related to:

• the risk of the change
• the size of the existing system
• the size of the change
Test levels
Depending on the changes, maintenance testing may be done at any or all test levels and for any or all test types.
Approach
Determining how the existing system may be affected by changes is called impact analysis, and is used to help decide how much regression testing to do.
Note
Maintenance testing can be difficult if specifications are out of date or missing.

Chapter 3

Dynamic testing - requires the execution of software.

Static testing - manual examination and automated analysis of the code or documentation without execution of the software under test.

Reviews - a way of testing software products (including code) and can be performed well before dynamic test execution.

• Reviews, static analysis and dynamic testing have the same objective: identifying defects.

• They are complementary.

• Compared to dynamic testing, static techniques find causes of failures (defects) rather than the failures themselves.

Reason to make reviews

- Defects detected during reviews early in the life cycle are often cheaper to remove than those detected while running tests.

- Reviews can find defects and omissions, for example, in requirements, which are unlikely to be found in dynamic testing.

Tools (manual + tool support)

The main manual activity is to examine a work product and make comments about it.

Object of reviews

Any software work product can be reviewed, e.g. • requirements specifications

• design specifications

• code

• test plans, test specifications, test cases, test scripts • user guides

• web pages

Benefits

• early defect detection and correction

• development productivity improvements • reduced development timescales

• reduced testing cost and time

• lifetime cost reductions

• fewer defects and improved communication

Typical defects

• deviations from standards

• requirement defects

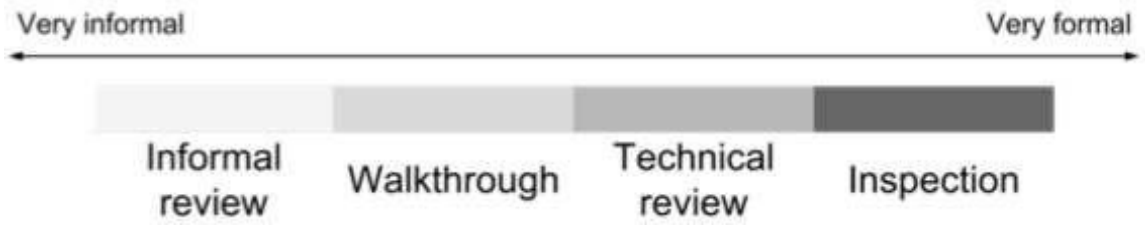
• design defects

• insufficient maintainability

• incorrect interface specifications

These defects are easier to find in reviews than in dynamic testing

Different types of reviews



Different types of reviews vary from:

• very informal(e.g.nowritteninstructionsfor reviewers)

• to very formal (i.e. well structured and regulated)

The formality of a review process is related to factors like:

• Risk

• Size of the project

• the maturity of the development process

• any legal or regulatory requirements

• the need for an audit trail

The way a review is carried out depends on the agreed objective of the review:

• find defects and omissions
• gain understanding
• discussion and decision by consensus

Phases of a formal review

1. Planning
   a. Select the personnel
   b. Allocate roles
   c. Define the entry and exit criteria for more formal review types (e.g. inspection)
   d. Select which parts of documents to look at
2. Kick-off
   a. Distributing documents
   b. Explaining the objectives of the review and the review process
   c. Explaining the documents to the participants
   d. Checking and discuss entry/exit criteria
3. Individual preparation
   Work done by each of the participants on their own before the review meeting, noting potential defects, questions and comments.
   Each participants proposes the severity of the defects. Severity classes: critical, major or minor
4. Review meeting
   • Discussion and logging, with documented results or minutes
   • The meeting participants may simply note defects, make recommendations for handling the defects, or make decisions about the defects. Decisions based on the exit criteria
   • Examining, evaluation and recording
5. Rework
   • fixing defects found, typically done by the author.
6. Follow-up
   • check that defects have been addressed
   • gather metrics, e.g.
   • number of defects found
   • number of defects found per page • time spent checking per page
   • total review effort
   • etc.

The review process

Manager	Moderator	Author	Reviewers	Scribe
• Decides on execution of reviews • Allocates time in project schedules • Determines if review objectives are met	• Leads the review • Plans the review • Runs the meeting • Follow-up after meeting • Mediates between various points of view	• Writer of the documents being reviewed, or • Responsible for the documents being reviewed	• Individuals with specific technical or business background • Identify and describe the findings in product under review	• Documents the entire review meeting • Issues, problems, open points that have been identified

1. Informal review

Purpose

Inexpensive way to get some benefit

Form

Pair reviews; e.g. pair programming or a technical lead reviewing designs and code

Note: No formal process

Note: Optionally may be documented

2. Walkthrough

Purposes

- Learning,
- gaining understanding
- defect finding
- feedback

Form

- meeting led by author
- may vary in practice from quite informal to very formal
- stakeholders may participate

3. Technical review

Purposes

- discuss
- make decisions
- evaluate alternatives
- find defects
- solve technical problems
- check conformance to specifications and standards

Form

May vary from very formal to informal peer review without management participation.

- ideally led by trained moderator
- documented, defined defect-detection process; includes peers and technical experts
- pre-meeting preparation
- optionally the use of checklists, review report, list of findings and management

4. Inspection

Purpose

- Find defects

Form

- Usually peer examination led by trained moderator (not the author)
- Formal process based on rules and checklists with entry and exit criteria
- pre-meeting preparation
- defined roles
- includes metrics
- inspection report,list of findings

Success factors for reviews

1. Objectives

Each review has a clear predefined objective

2. Roles

The right people for the review objectives are involved.

3. Approach

- Defects found are welcomed, and expressed objectively.
- Apply suitable review techniques for the type and level of software products.
- Use checklists or roles if appropriate to increase effectiveness of defect identification.
- Management supports a good review process (e.g. by incorporating adequate time for review activities).

4. Training and learning

- Training is given in review techniques, especially the more formal techniques, such as inspection.
- There is an emphasis on learning and process improvement

Static Analysis by Tools

Static analysis Analysis of software artifacts, e.g. requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.

Objectives of static analysis

Find defects in

- software source code
- software models

Note! Static analysis finds defects rather than failures

Static analysis is performed without actually executing the software being examined by the tool.
Static analysis tools analyze program code, as well as generated output such as HTML and XML.

Typical defects

discovered by static analysis tools include:

- referencing a variable with an undefined value
- inconsistent interface between modules and components • variables that are never used
- unreachable (dead) code
- programming standards violations
- security vulnerabilities
- syntax violations of code and software models

Developers

Use static analysis before and during:

- Component testing
- Integration testing

Designers

Use static analysis during software modeling

Practical side

Static analysis tools may produce a large number of warning messages, which need to be well managed to allow the most effective use of the tool.

Why is static analysis valuable

- Early detection of defects prior to test execution.
- Early warning about suspicious aspects of the code or design, by the calculation of metrics, such as a high complexity measure.

- Identification of defects not easily found by dynamic testing.
- Detecting dependencies and inconsistencies in software models, such as links.
- Improved maintainability of code and design.
- Prevention of defects, if lessons are learned in development.

Coding standards

Recommended that existing standards should be adopted in order to save a lot of effort

- Set of programming rules, i.e. always check boundaries on an array when using it
- Naming conversions, e.g. class name should start with a Capital letter
- Access conversions, e.g. public/private
- Layout specifications, e.g. indents
- Checking tools supports code standards

Code metrics

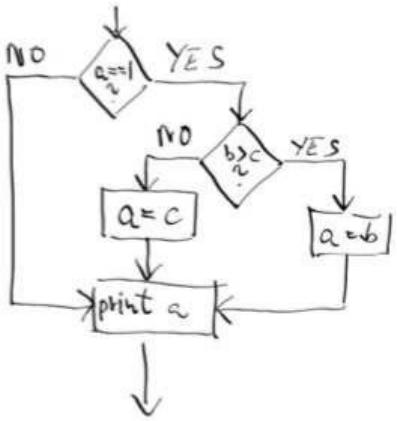
- Comments frequency
- Depth of nesting
- Cyclomatic complexity /complexity metrics

  The number of independent paths through a program. Cyclomatic complexity is defined as: L - N + 2P, where

  - L = the number of edges/links in a graph
  - N = the number of nodes in a graph
  - P = the number of disconnected parts of the graph (e.g. a called graph or subroutine).

Can be measured in different ways, e.g. based on the number of decisions in the program (the number of binary decisions)

```
if( a == 1 )
{   if( b > c )
      a = b;
    else
      a = c;
}
System.out.println( a );
```



Code structure

- Control flow structure

  The sequence in which the instructions are executed
- Data flow structure

  follows the trail of a data item as it is accessed and modified by the code
- Data structure

  The organization of the data itself , independent of the program (Array, list, stack, queue, tree, graph, ...)

The value of static analysis is especially for

- Early detection of defect prior to test execution
- Early warning about suspicious aspects of the code, design and requirements, ref. P4: defect clustering

• Identifying defects not easily found in dynamic testing
• Improve maintainability of code and design
• Prevention of future defects

Chapter 4

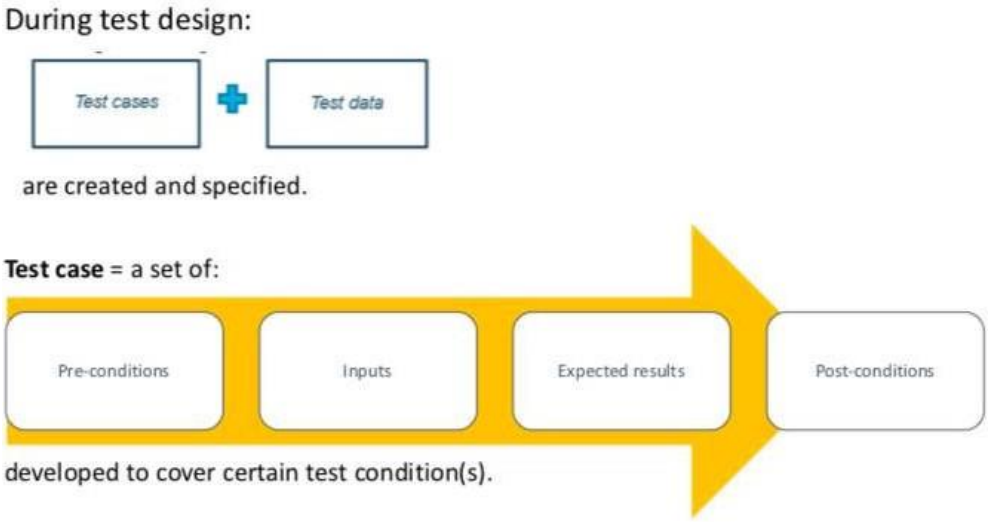
The test design process can be done in different ways, from very informal (little or no documentation), to very formal.
The level of formality depends on the context of the testing, including:

Test development:

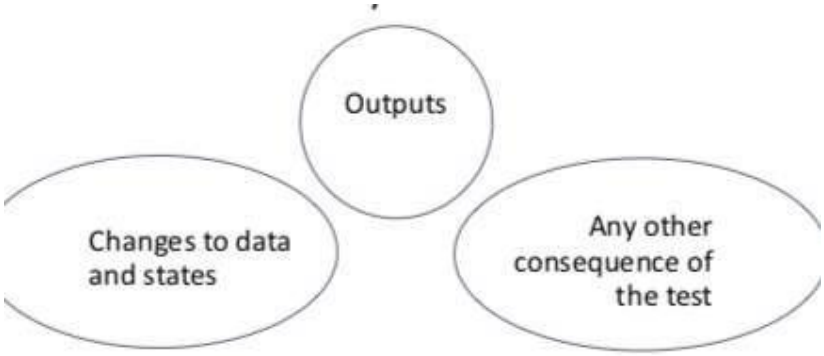
1. Test analysis

The test basis documentation is analyzed in order to determine what to test, i.e. to identify the test conditions.
Test condition (Def.) = an item or event that could be verified by one or more test cases
Examples
• A function
• Atransaction
• Aqualitycharacteristic
• Other structural elements (menus in web pages, etc.)
Test possibilities:
"Throw a wide net!"
First; identify as many test conditions as possible Second; select which one to develop in more detail
We can't test everything (P2). We have to select a subset of all possible tests, but this subset must have a high probability of finding most of the defects in the system.
We need a suitable test design technique to guide our selection and to prioritize the test conditions.

2. Test design



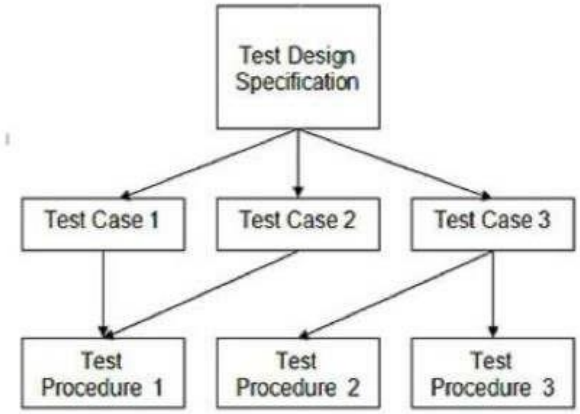
Test oracle:
In order to know what the system should do, we need to have a source of information about the correct behavior of the system – an oracle.
Expected results include:



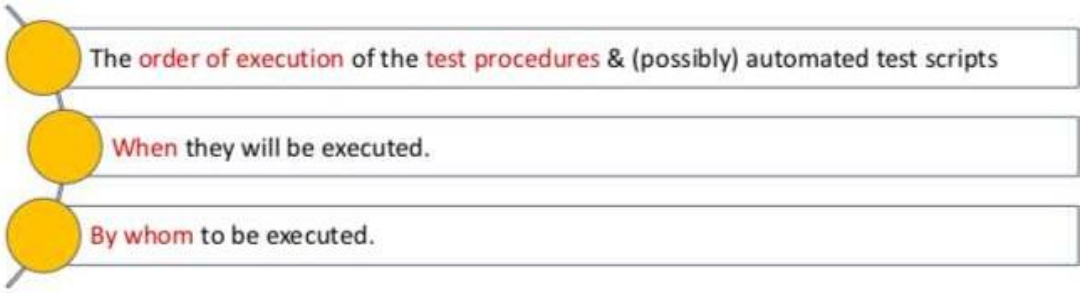
If expected results have not been defined, then a plausible but erroneous result may be interpreted as the correct one.
Expected results should ideally be defined prior to test execution.

3. Test implementation
During test implementation the test cases are organized
in the test procedures:

A manual test procedure
specifies the sequence of action to be taken for executing of a test.
An automated test procedure (test script)
If tests are run using a test execution tool, the sequence of action is specified in a test script.



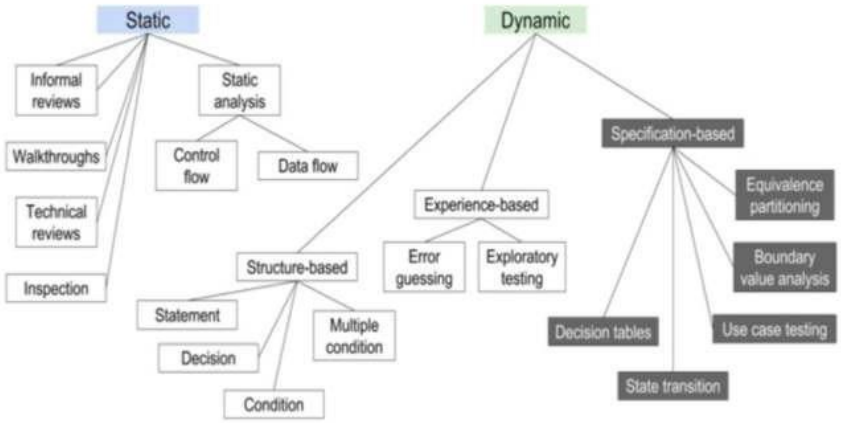
The test execution schedule defines:



The test execution schedule will take into account such factors as:
- risks
- regression tests
- prioritization
- technical and logical dependencies

Writing the test procedure is another opportunity to prioritize the tests, to ensure that the best testing is done in the time available.
A good rule of thumb is ‘Find the scary stuff first’. However the definition of what is ‘scary’ depends on the business, system or project and on the risks of the project.

Categories of test design technicues:



Black-box testing:

- We use models (formal or informal) to specify the problem to be solved, as well as the software or its components.
- We derive systematically the test cases from these models.
- The test cases are derived from information about how the software is constructed, e.g. code and design.
- For the existing test cases, we can measure the test coverage of the software.
- Further test cases can be derived systematically to increase the test coverage.
- The test cases are derived from the knowledge and experience of people:
    - Knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment.
    - Knowledge about likely defects and their distribution.

Equivalence partitioning

• The basic idea is to divide a set of test conditions into partitions (subsets) where the elements in each partition that can be considered the same.
• It is important that the different partitions do not have common elements.
• We need only to test one condition from each partition, because all the conditions in the same partition will be treated in the same way by the software.

Eksempel: se side 81-83 i boka

Technique

Inputs/outputs/internal values of the software are divided into groups that are expected to exhibit similar behavior.
Equivalence partitions can be found for both valid data and invalid data, i.e. values that should be rejected.

Notes

• Tests can be designed to cover more than one partitions.
• Equivalence partitioning is applicable at all levels of testing.
• Equivalence partitioning as a technique can be used to achieve input and output coverage.

Boundry value analysis

Behavior at the edge of each equivalence partition is more likely to be incorrect than behavior within the partition.
Boundary are an area where testing is likely to yield defects.

Boundary analysis

Analysis at the edge of each equivalence partition.Why? Because there, the results are more likely to be incorrect.
The maximum and minimum values of a partition are its boundary values.

Valid and invalid boundary

- A boundary value for a valid partition is a valid boundary value.
- A boundary value for an invalid partition is an invalid boundary value.
- Tests can be designed to cover both valid and invalid boundary values.

Notes

- Boundary value analysis can be applied at all test levels.
- Its relatively easy to apply and its defect finding capability is high.
- Detailed specifications are helpful.
- This technique is often considered as an extension of equivalence partitioning.
- Boundary values are used for test data selection.

Why do both equivalence partitioning and boundary value analysis?

Boundary values are usually extreme values. To gain confidence to the system we also want to test it under normal circumstances.

Decision table testing

Decision tables are a good way

- to capture system requirements that contain logical conditions
- to document internal system design
- to record complex business rules that a system is to implement

When creating decision tables, the specification is analyzed, and actions of the system are identified.

If the input conditions and actions are stated in a way where they are either be true or false (Boolean), decision tables can be useful.

The decision table contains the triggering conditions, i.e. all combinations of true and false for all input conditions, and the resulting actions for each combination of conditions.

State transition testing

A system can be in a finite number of different states. This aspects of the system can be described as a 'finite state machine' ; a state diagram.

Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.

The transition from one state to another are determined by the rules of the 'machine'.

State transition testing

System can be in a finite number of different states

- Elements of state transition models

States → The SW may occupy

E.g. open / closed, active / inactive

Transitions → From one state to another

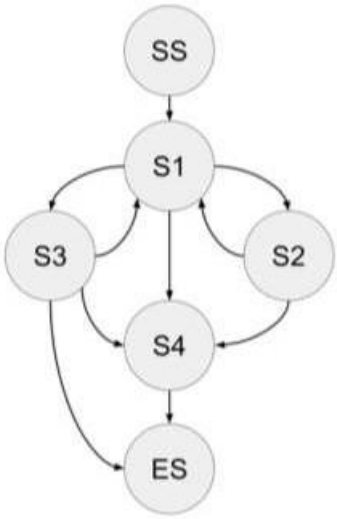
Not all transitions are allowed

Events → Causing state transitions

E.g. closing a file, withdrawing money

Actions → Actions resulting from transitions

E.g. error message



Why state transition testing?

Because a system may exhibit a different response depending on current conditions or previous history.

State transition testing allows the tester to view:

- the software in terms of its states
- transitions between states
- the inputs or events that trigger state changes (transitions)
- the actions which may result from those transitions

Tests can be designed

- to cover a typical sequence of states
- to exercise specific sequences of transitions
- to cover every state
- to exercise every transition
- to test invalid transitions

State transition testing is much used within the software industry and technical automation in general.

Use case testing

Use case - describes interactions between actors (users and the system), which produce a result of value to a system user.

Use case testing

- Identify test cases that exercise the whole system
  - Transaction by transaction basis
  - From start to finish
- Describes interactions between actor and system
  - Achieve a specific task
  - Produce something of value to the user
- Defined in terms of the actor, not the system
  - Describes process flows through a system
    - Based on its actual use
  - Can uncover integration defects

Use case name	<name>	
Actor(s)	<actor1>, ...	
Pre-conditions	<cond1>, ...	
Post-conditions	<cond1>, ...	
Main Success Scenario	Step	Description
	1	A: <action>
	2	S: <response>
	3	A: <action>
	4	S: <response>
	...	...
Extensions	Step	Description
	S.X	<cause> S: <response>
	S.Y	<cause> S: <response>

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

Structure-based testing.
White-box techniques.
Structure-based techniques serve two purposes:
Test coverage measurement
We can assess the amount of testing performed by tests derived from e.g. specification-based technique to asses coverage.
Structural test case design
We can generate additional test cases with the aim of increasing the test coverage.

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

A coverage item is whatever we have been able to count and see whether a test has exercised or used this item.
NB! 100% coverage does not mean that 100% tested!

Structure-based testing (white-box) is based on an identified structure of the software.
Component level - the structure is that of the code itself: - statements, decisions/branches.
Integration level - the structure may be a call tree (a diagram in which modules call other modules).

System level - the structure may be a menu structure, business process or web page structure.

How to measure coverage?

The steps typically taken to measuring coverage is useful in understanding the relative merits of each technique:

1. Decide on the structural element used, i.e. the coverage items to be counted.
2. Count the structural elements or items.
3. Instrument the code.
4. Run the tests for which the coverage measurement is required.
5. Using the output from the instrumentation, determining the percentage of elements or items exercised.

In component testing

Statement coverage is the percentage of executable statements that have been exercised by a test case suite.

The **statement testing** technique derives test cases to execute specific statements, normally to increase statement coverage.

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\ \%$$

Decision coverage - is the assessment of the percentage of decision outcomes (e.g. the True and False options of an IF statement) that have been exercised by a test case suite.

$$\frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\ \%$$

Decision coverage is stronger than statement coverage. 100% decision coverage guarantees 100% statement coverage

Transition coverage is stronger than state coverage

100% transition coverage guarantees 100% state coverage Not the other way around!

Experience-based testing

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

Error guessing = a commonly used experienced-based technique. Generally testers anticipate defects based on experience.

A structured approach to the error guessing technique is to enumerate a list of possible errors and to design tests that attack these errors.

This systematic approach is called fault attack.

Exploratory testing = concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, and carried out within time-boxes.

It is most useful ...

- where there are few or inadequate specifications
- under severe time pressure
- to complement other, more formal testing
- It can serve to help ensure that the most serious defects are found.

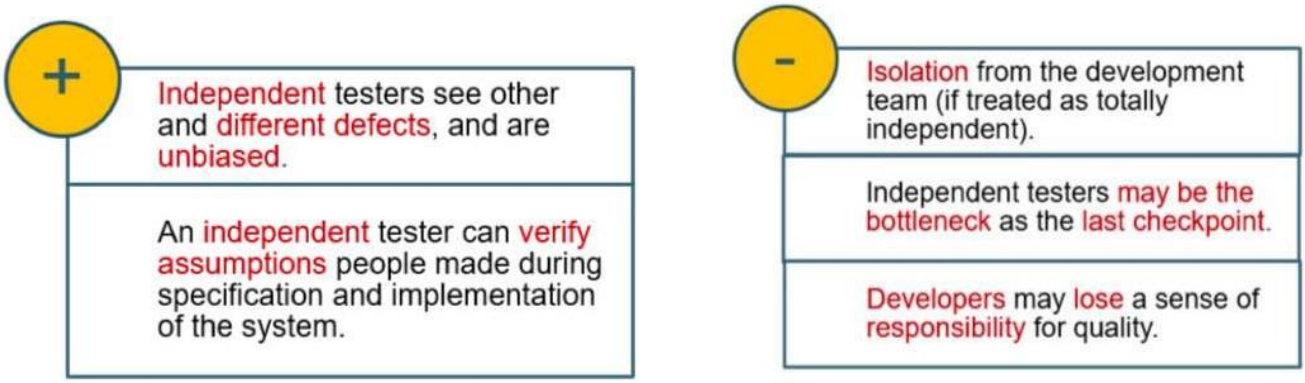
Chapter 5

- Testing is an assasment of quality
- Separate the testers from the developers

- The effectiveness of finding defects by testing and reviews can be improved by using independent testers.

- Options for independence are:

- For large, complex or safety critical projects, it is usually best to have multiple levels of testing, with some or all of the levels done by independent testers.
- Development staff can participate in testing, especially at the lower levels



Note

Testing tasks may be done by people in a specific testing role, or may be done by someone in another role, such as:

- project manager
- quality manager
- developer
- business and domain expert
- infrastructure or IT operations

(this can be both good and bad)

Tasks of the test leader and tester:

There is wide variation in the roles that people within the test team play. The two most common roles are

- *test leader*
- *tester*

Test leader = test manager / test coordinator.

The test leader plans, monitors and controls the testing activities and tasks.

The tester

The tester reviews and contributes to test plan, analyses, designs, prepares, implements and executes tests.

Tasks of the test leader	
Coordination	of the test strategy and plan with project managers
Plan the tests	Understanding the test objectives and risks – including: • selecting test approaches • estimating the time, effort and cost of testing • acquiring resources • defining test levels, cycles • planning incident management
Test specifications, preparation and execution	Initiate the specification, preparation, implementation and execution of tests • monitor the test results • check the exit criteria.
Adapt planning	based on test results and progress and take any action to compensate for problems.
Manage test configuration	Set up adequate configuration management of testware for traceability.
Introduce metrics	for measuring test progress and evaluating the quality of testing & product.
Automation of tests	Decide what should be automated, to what degree, and how.
Select test tools	Select tools to support testing and organize trainings for tool users.
Test environment	Decide about the implementation of the test environment.
Test summary reports	Write test summary reports based on the information gathered during testing.

Tasks of the tester	
Test plans	Review and contribute to test plans
Requirements and specifications	Analyze, review and assess user requirements, specifications and models for testability.
Test specifications	Create test specifications
Test environment	Set up the test environment (often coordinating with system administration and network management).
Test data	Prepare and acquire test data.
Testing process	Implement tests on all test levels, execute and log the tests, evaluate the results and document the deviations from expected results.
Test tools	Use test tools (for administration, management or monitoring) as required.
Test automation	Automate tests (may be supported by a developer or a test automation expert).
Other metrics	Measure performance of components and systems (if applicable).
Help the others	Review tests developed by others

People involved in testing need

- Application or business domain: A tester must understand the intended behavior and the problem the system will solve in order to spot improper.
- Technology: A tester must be aware of issues, limitations and capabilities of the chosen implementation technology, in order to locate problems and the 'likely to fail' functions and features.
- Testing: A tester must know the testing topics discussed in this book - in order to carry out the test tasks assigned.
- Basic professional and social qualifications such as • literacy
  - the ability to prepare and deliver written and verbal reports
  - the ability to communicate effectively
  - ...

Test planning and estimation

A test plan is the project plan for the testing work to be done.

- Writing a test plan forces us to confront the challenges that await us and focus our thinking on important topics.
- The test planning process and the plan itself serve as vehicles for communicating with other members of the project team, testers, peers, managers and other stakeholders.
- The test plan also helps us manage change. As we gather more information, we revise our plans.

Planning may be documented in:

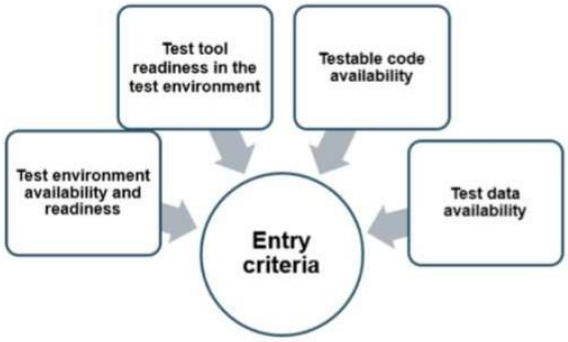
- a project or master test plan
- and in separate test plans for test levels, such as integration testing, system testing and acceptance testing.

• Outlines of test planning documents are covered by the 'Standard for

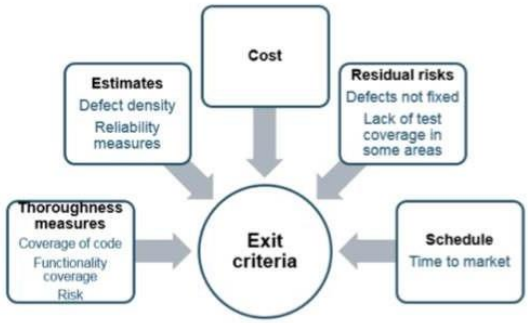
Test planning is a continuous activity and is performed in all life cycle processes and activities.
Feedback from test activities is used to recognize changing risks so that planning can be adjusted.

Test planning activities	
Scope and risk	Determining the scope and risks of testing
Objectives	Identifying the objectives of testing.
Overall approach	Defining the overall approach of testing, including: • the definition of the test levels • entry and exit criteria.
Test activities	Integrating and coordinating the testing activities into the software life cycle activities: • acquisition and supply • development • operation • maintenance
Strategy	Making decisions about: • what to test • what roles will perform the test activities • how the test activities should be done • and how the test results will be evaluated.
Schedule	Scheduling test analysis and design activities. Scheduling test implementation, execution and evaluation
Resources	Assigning resources for the different activities defined.
Metrics	Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues.

Entry criteria defines when to start testing.



Exit criteria is to define when to stop testing, such as at the end of a test level, end of project or when a set of tests has a specific goal.



Test estimation

The testing work is usually a subproject within the larger project. Fundamental techniques of estimation can be adapt for testing

We can break down a testing project into phases • planning and control

- analysis and design
- implementation and execution
- evaluating exit criteria and reporting
- test closure

Within each phase we identify activities and within each activity we identify tasks and perhaps subtasks.

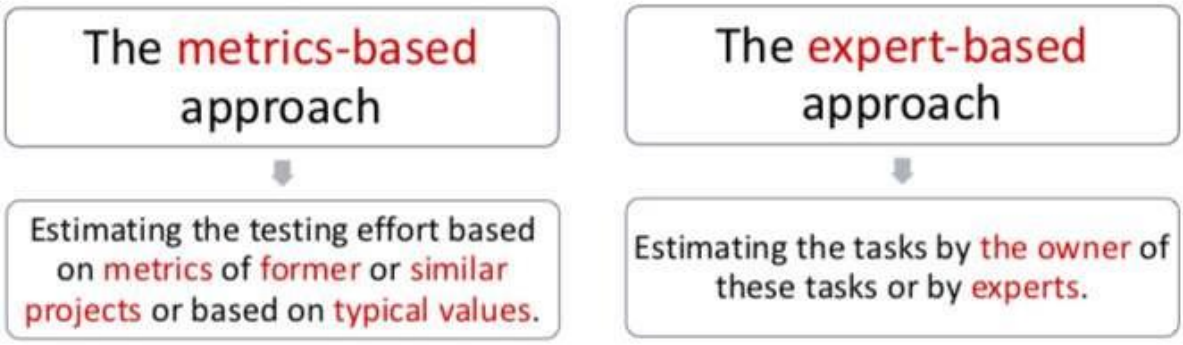
To identify the activities and tasks, we work both

- forward
- backward

To ensure accuracy of the estimate, make sure that you subdivide the work
into tasks that are short in duration, say one to three days.

If they are much longer - say two weeks – there will be a risk that long and complex sub-tasks are 'hidden' within the larger task.

Two approaches for the estimation of test effort are covered in this syllabus



A good solution is to combine the two strategies:

- First create the work-breakdown structure and a detailed bottom-up estimate.
- Second; We then apply models and rules of thumb to check and adjust the estimate bottom-up and top-down using past history.

This approach tends to create an estimate that is both more accurate and more defensible than either technique by itself.

The testing effort may depend on a number of factors, including:

Product factors	• the quality of the specification (the test basis) • the size of the product • the complexity of the problem domain • the importance of non-functional quality e.g. usability, performance, security etc.
Process factors	• the development model • availability of test tools (e.g. test executing tools) • skills of the people involved • time pressure
The outcome of testing	• the number of defects • the amount of rework required

Test approaches and strategies:

The test approach is the implementation of the test strategy for a specific project.

Since the test approach is specific to a project, the approach should be documented in the test plan. One way to classify test approaches or strategies is based on the point in time at which the bulk of the test design work is begun:

Preventative approaches

- Tests are designed as early as possible

Reactive approaches

- Test design comes after the software or system has been produced

Analytical approaches	e.g. risk-based testing - testing is directed to areas of greatest risk, requirement-based testing
Model-based approaches	e.g. testing using statistical information about failure rates (such as reliability growth models)
Methodical approaches	e.g. failure-based (including error guessing and fault-attacks), experienced-based, check-list based, and quality characteristic based.
Process- or standard-compliant approaches	e.g. specified by industry-specific standards or the various agile methodologies.
Dynamic and heuristic approaches	e.g. exploratory testing, execution & evaluation are concurrent tasks.
Consultative approaches	e.g. test coverage is evaluated by domain experts outside the test team.
Regression-averse approaches	include reuse of existing test material, extensive automation of functional regression tests.

The choice of test approaches and strategies is one powerful factor in the success of the effort and the accuracy of the test plans and estimates. When we choose test strategies there are many factors to consider:

- Risks
- Skills
- Objectives
- Regulations
- Product
- Business

Test progress monitoring

- The purpose of test monitoring is to give feedback and visibility about test activities.
- Information to be monitored may be collected manually or automatically and may be used to measure exit criteria, such as test coverage.
- Metrics may also be used to assess progress against the planned schedule and budget.

IEEE 829
829 Standard for Software and System Test Documentation Test log - template

Test reporting

Test reporting is concerned with summarizing information about the testing endeavor, including:

- What happened during the period of testing? (ex: dates when exit criteria were met)

- Analyzed metrics to support decisions about future actions (e.g. the economic benefit of continued testing)

Metrics are collected at the end of a test level in order to assess:

- The adequacy of the test objectives for that test level
- The adequacy of the test approaches with respect to their objectives
- The effectiveness of the testing with respect to its objectives

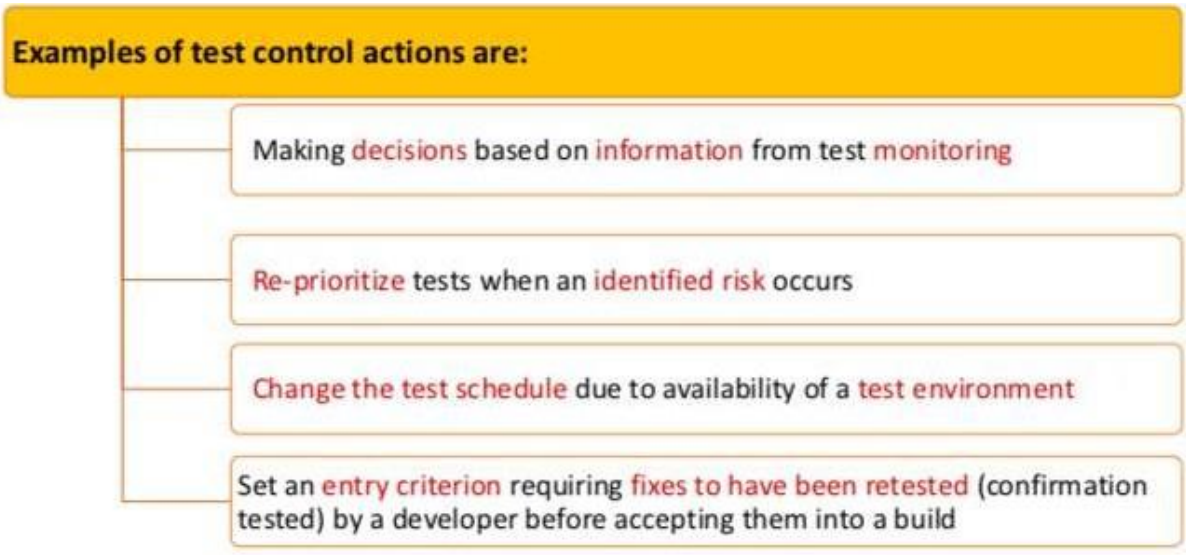
IEEE 829
829 Standard for Software and System Test Documentation Test summary report – template

Test control

Projects do not always go according to plan. Different factors can lead to deviations:

- New risks
- New needs
- Test findings
- External events
- Test environment problems

Test control describes any guiding or corrective actions taken as a result of information and metrics gathered and reported.



Configuration management

The purpose of configuration management is to establish and maintain the integrity of the software and related products through the project and product life cycle.
The configuration management shall ensure that all items of testware are

- identified
- version controlled
- tracked for changes

so that traceability can be maintained throughout the test process.

All identified documents and software items should be referenced unambiguously in test documentation. Configuration management helps to uniquely identify (and to reproduce)

the tested item □test documents □the tests □the test harness
Configuration management procedures and tools should be selected during the project planning stage.

Risk and testing
• Risk is the possibility of a negative or undesirable outcome, the possible problems that might endanger the objectives of the project stakeholders.
• Risks are related to the product/project
• Risk analysis and risk management can help us plot a course for solid testing.
The level of risk is determined by:

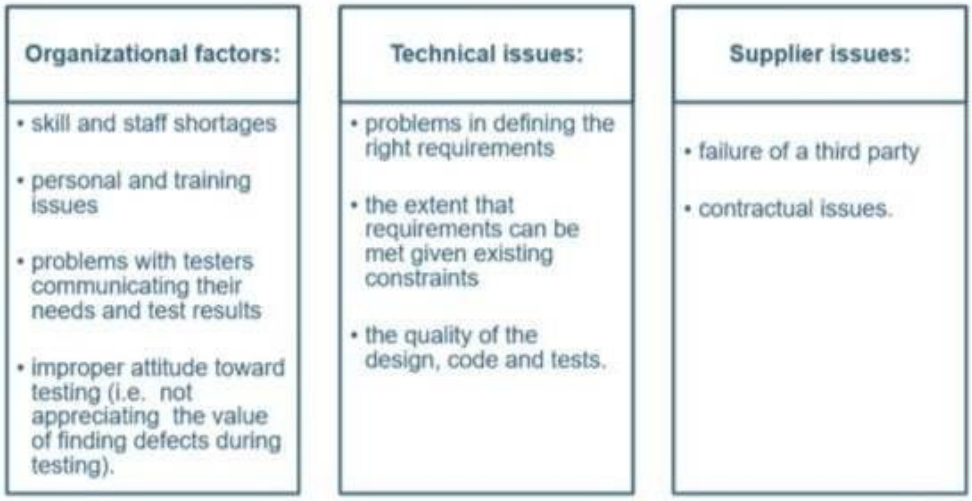
- The likelihood of an adverse event happening
- The impact (the harm resulting from that event)

For any risks you have four possibilities: • Mitigate • Contingency • Transfer
• Ignore

Project risks

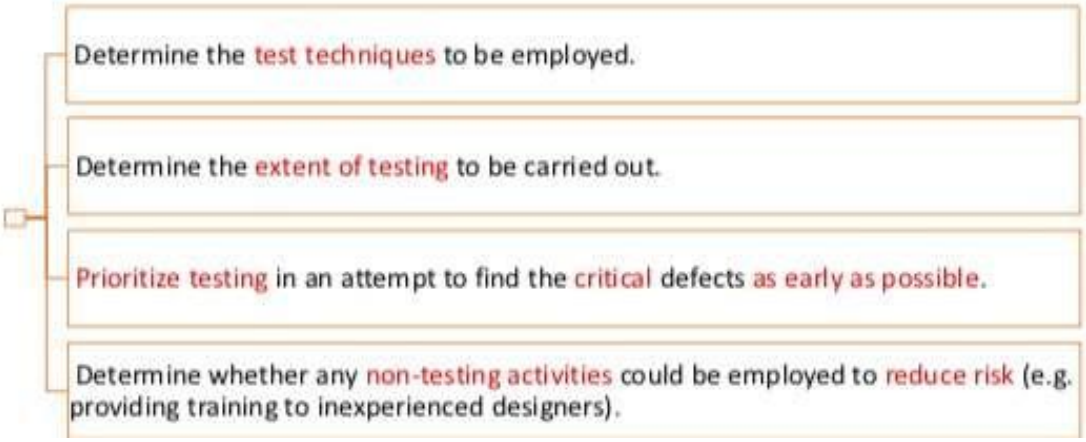
- Logistics or product quality problems that block tests
- Test items that won't install in the test environment
- Excessive change to the product that invalidates test results or requires updates to test cases, expected results and environments
- Insufficient or unrealistic test environments that yield misleading results

Project risks = the risks that surround the project's capability to deliver its objectives, such as:



Product risk = is a risk directly related to the test object
Product risk is the possibility that the system or software might fail to satisfy some reasonable customer, user, or stakeholder expectation.
Product risks = Potential failure areas in the software. Risks are used to decide where to start testing and where to test more.

Risk-based approach
Testing is used to reduce the risk of an adverse effect occurring, reduce the impact of an adverse effectas early as possible. In a risk-based approach the risks identified may be used to:



Risk analysis should start as early as possible:

- Identifying the risk items
- Determine the likelihood and impact for each item
- Use a rating scale (1 – 10) classify the level of risk for each item
- Priority the risk items according to their rating values

1. Analyze risks early in the project.
2. You should manage risks appropriately, based on likelihood and impact, but do not confuse impact with likelihood or vice versa.
3. The goal of risk-based testing should not be - cannot practically be - a risk-free project.
4. Best practices in risk management to achieve a project outcome that balances risks with quality, features, budget and schedule.

Incident management

Incident - Discrepancies between actual and expected test outcomes.

Incident management - The process of recognizing, investigating, taken action and disposing of incidents.

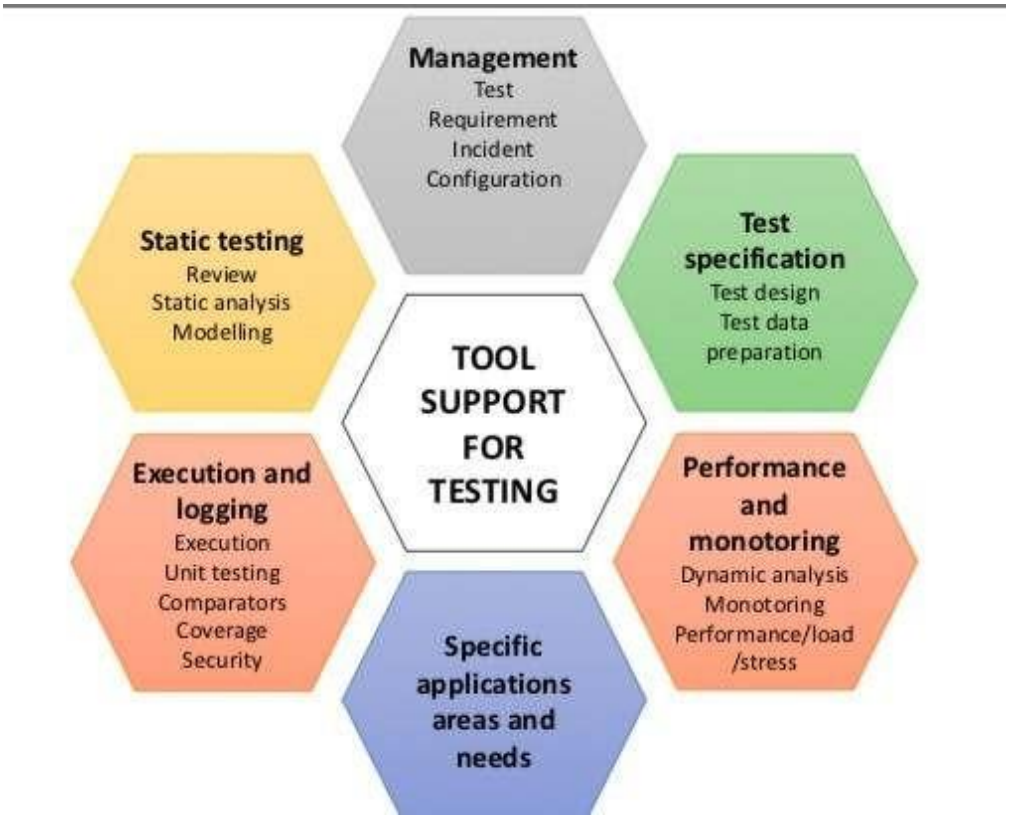
Incident rapport - A rapport document reporting on any event that occurred, e.g. during testing, which requires investigation.

What goes in an incident rapport?

- A description of some situation, behavior or event that occurred.
- One or two screens – with information gathered by a defect-tracking tool.
- A description of the steps done to reproduce and isolate the incident.
- The impact of the problem.
- Classification information (i.e. the scope, severity and priority of the defect).
  A level of priority, assigned by the test managers
- The risks, costs, opportunities and benefits associated with fixing or not fixing the defect assigned by the project team or a committee.
- The root cause, captured by the programmer, - the phase of introduction
  - the phase of removal
- Conclusions and recommendations captured by the managers, programmers or others
- Throughout the life cycle of the incident report, the defect-tracking system should allow each person who works on the incident report to enter status and history information.

Chapter 6

Types of test tools



Types: • Tools that are directly used in testing (e.g. test execution tools, test data generation tools, result comparison tools)

• Tools that help in managing the testing process (e.g. test results, requirements, incidents, defects) and for monitoring and reporting the test execution

• Tools that are used in exploration (e.g. tools that monitor the file activity for an application)

• Any tool that aids in testing

Purposes: • improve the efficiency of the test activities (e.g.: by automating repetitive tasks)

• automate activities that require significant resources when done manually (e.g. static testing)

• automate activities that cannot be done manually (e.g. large-scale performance testing of client-server applications)

• increase reliability of testing (by automating large data comparisons or simulating complex behavior)

Notes

• Some types of test tool can be intrusive - the tool itself can affect the outcome of the test. (e.g. timing measurements may be different depending on how you measure it with different performance tools).

• The consequence of intrusive tools is called the probe effect.

• Some tools offer support more appropriate for developers. Such tools are marked with "(D)" in this chapter.

Tolls support for management

Characteristics

- Support for the management of tests and the testing activities.
- Support for traceability of tests, test results and incidents to source documents, such as requirements specifications.
- Generation of progress reports.
- Logging test results. Monotoring
- Offer info on metrics related to the tests.

Incident management tools	store and manage incident reports
	support management of incident reports

Requirements management tools	store requirements
	check for consistency and undefined (missing) requirements
	allow prioritization
	enable individual tests to be traceable to requirements

Configuration management tools	are necessary to keep track of different versions and builds of the SW and tests
	are particularly useful when developing on more than one configuration of the HW/SW environment

Tools support for static testing

Tools for static testing

Tools that aid in improving the code / work product, without executing it

Categories

Review tools

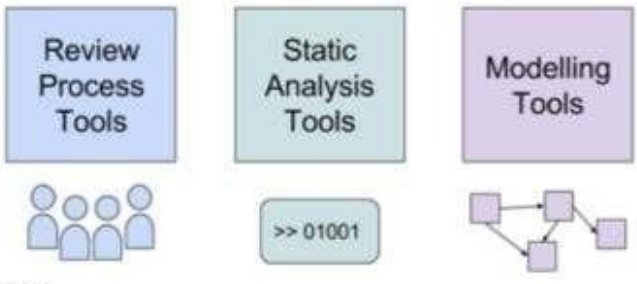
Supports the review process

Static analysis tools

Supports code examination

Modelling tools

Validate models of system / software



Review process tools

Common reference for the review processes conducted

Keep track of all the information from the review process

Store and communicate review comments, report on defects and effort

Monitoring review status → Passed, passed with corrections, requires re-review

When to use?

Suitable for more formal review processes

Geographically dispersed teams

Static analysis tools (D)	Major purpose: - The enforcement of coding standards. - The analysis of structures and dependencies (e.g. linked webpages) - Aiding in understanding the code.
	support developers, testers and quality assurers in finding defects before dynamic testing.
	Static analysis tools can calculate metrics from the code (e.g. complexity), which can give valuable information for planning or risk analysis.