SYSTEM VERIFICATION

ISTQB®

Certified Tester

Foundation Level

Syllabus 2011

Accredited by the
Swedish Software Testing Board

Version 1.05

TABLE OF CONTENTS

Hello! My name is

SYSTEM VERIFICATION

SYSTEM VERIFICATION

Hello! My name is

COURSE TIMELINE

DAY 1

ACTIVITY	K-LEVEL	TIME	DURATION
Course Introduction		9.00	45
1. Fundamentals of Testing		**9.45**	
Exercise		9.45	15
1.1 Why is Testing Necessary?	K2	10.00	20
1.2 What is Testing?	K2	10.20	30
Break		10.50	15
1.3 Seven Testing Principles	K2	11.05	35
1.4 Fundamental Test Process	K1	11.40	35
Lunch		12.15	50
1.5 The Psychology of Testing	K2	13.05	25
1.6 Code of Ethics	K1	13.30	10
Summary and Exercises		13.40	20
2. Testing Throughout the Software Life Cycle		**14.00**	
2.1 Software Development Models	K2	14.00	20
Break		14.20	15
2.2 Test Levels	K2	14.35	40
2.3 Test Types	K2	15.15	40
2.4 Maintenance Testing	K2	15.55	15
Summary and Exercises		16.10	20
Home Assignment • Review chapters 1 and 2 • Solve all knowledge check questions of chapters 1 and 2			

DAY 2

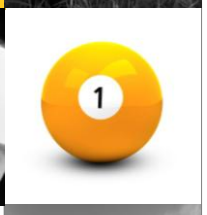
ACTIVITY	K-LEVEL	TIME	DURATION
Review of chapters 1 and 2 + knowledge check questions		9.00	30
3. Static Techniques		**9.30**	
3.1 Static Techniques	K2	9.30	15
3.2 Review Process	K2	9.45	25
3.3 Static Analysis by Tools	K2	10.10	20
Summary and Exercises		10.30	20
Break		10.50	15
4. Test Design Techniques		**11.05**	
4.1 The Test Development Process	K3	11.05	15
4.2 Categories of Test Design Techniques	K2	11.20	15
4.3 Specification-based or Black-box Techniques	K3	11.35	150
4.3.1 - 4.3.2 Equivalence Partitioning and Boundary Value Analysis	K3	11.35	20
Lunch		11.55	50
Exercise		12.45	35
4.3.3 Decision Table Testing	K3	13.20	20
Exercise		13.40	20
4.3.4 State Transition Testing	K3	14.00	20
Exercise		14.20	20
4.3.5 Use Case Testing	K2	14.40	20
Break		15.00	15
4.4 Structure-based or White-box Techniques	K4	15.15	30
Exercise		15.45	30
4.5 Experience-based Techniques	K2	16.15	30
4.6 Choosing Test Techniques	K2	16.45	15
Home Assignment • Review chapters 3 and 4 • Solve all knowledge check questions of chapters 3 and 4			

DAY 3

ACTIVITY	K-LEVEL	TIME	DURATION
Review of chapters 3 and 4 + knowledge check questions		9.00	30
5. Test Management		**9.30**	
5.1 Test Organisation	K2	9.30	30
5.2 Test Planning and Estimation	K3	10.00	40
Break		10.40	15
5.3 Test Progress Monitoring and Control	K2	10.55	20
5.4 Configuration Management	K2	11.15	10
5.5 Risk and Testing	K2	11.25	30
Lunch		11.55	50
5.6 Incident Management	K3	12.45	40
Summary and Exercises		13.25	20
6. Tool Support for Testing		**13.45**	
6.1 Types of Test Tools	K2	13.45	45
6.2 Effective Use of Tools: Potential Benefits and Risks	K2	14.30	20
6.3 Introducing a Tools into an Organization	K1	14.50	15
Break		15.05	15
Summary and Exercises		15.20	30
Course Assessment and Finish		15.50	10
ISTQB Foundation Certification		16.00	60/75

Please note that all times are approximate and may be subject to change.

- We create new opportunities



ISTQB® Certified Tester

Foundation Level

SYSTEM VERIFICATION

About System Verification

- Sweden's leading Quality Assurance company and experts in Software QA
- Consulting and Managed Services
- About 185 employees
- Founded 2002 in Malmö
- Offices in Malmö, Lund, Gothenburg, Stockholm, Copenhagen and Sarajevo

- References from finance, IT, telecom, life sciences, security, manufacturing, automotive and defense industries
- Professional consultants with high motivation, experience and social skills
- ISO 9001 certified
- All consultants are ISTQB® certified
- ISTQB® and IREB® accredited training owner

Årets Företagare i Sverige 2013

Di GASELL FÅR SVERIGE ATT VÄXA

SYSTEM VERIFICATION SWEDEN AB 556624-3852

Ernst & Young Entrepreneur Of The Year®

SYSTEM VERIFICATION

SYSTEM VERIFICATION

Some of our Clients

GAMBRO, IKANO BANK, Consafe Logistics, IKEA, AXIS COMMUNICATIONS, ALFA LAVAL, SCANIA, verisure, e·on, QlikView, CERDO BANKPARTNER, SECURITAS, SONY make.believe, Tetra Pak, H&M, Danske Bank, Aller media, TELE2, BorgWarner, CGI

Your Teacher: Magnus C. Ohlsson

- Ph.D. in Software Engineering at LTH
  - Controlling Fault-Prone Components for Software Evolution
- SAAB Combitech Systems
  - Everything between heaven and "test"
- Test Department Manager at Obigo (Teleca)
  - Test and quality assurance of mobile applications
- Technical Manager for NS L23 Test at ST-Ericsson
  - Test of signalling between cell phones and base stations
- Quality Assurance Specialist at System Verification
- Certified ISEB Foundation, ISTQB Advanced Test Manager and ISTQB Advanced Test Analyst

© COPYRIGHT SYSTEM VERIFICATION 2014

SYSTEM VERIFICATION

Your Teacher: Freddy Gustavsson

- B.Sc. In Software Engineering at Gotland University
  - Thesis about test process improvement
- Web Developer at SinnerSchrader, Hamburg
- QA Engineer at SinnerSchrader, Hamburg
- Lecturer at Gotland University, Visby
- Test Specialist at System Verification, Gothenburg, recent assignments include
  - Test Engineer at Ericsson NSPS
  - Test Coordinator at Ericsson SA Media
  - Agile Tester at TeliaSonera
- Certified ISEB + REQB Foundation and ISTQB Advanced Test Analyst

Participants Introduction

- Please tell us a little about yourself and your motives for attending this training course:
  - Name
  - Company
  - Job/role
  - Expectations and wishes

ISTQB® (www.istqb.org)

- International Software Testing Qualifications Board
- Founded in November 2002
- Not-for-profit association legally registered in Belgium
- Defines the ISTQB® Certified Tester scheme that has become the world-wide leader in the certification of competences in software testing
- Based on volunteer work by hundreds of international testing experts
- Has issued over 380.000 certifications in more than 70 countries world-wide (as of Dec, 2014)
- **The de facto standard in certification of software testing competencies**

SSTB (www.sstb.se)

- Swedish Software Testing Board
- Sweden's national institution of ISTQB®
- Aims to create a Swedish testing standard based on Swedish testing terminology
- Offers syllabi in Swedish based on the ISTQB® syllabi
- Offers and carries out certifications based on the testing standards of ISTQB®
- Accredits companies to give training courses according to the principles of ISTQB®

Certified Training Owner

- System Verification is a Certified Training Owner for ISTQB® Foundation Syllabus 2011
- Accredited by SSTB on behalf of ISTQB®
- All course material has been reviewed and approved
- Participant surveys and test results are regularly evaluated to ensure quality

Course Overview

Day 1

- Fundamentals of Testing
- Testing Throughout the Software Life Cycle

Day 2

- Static Techniques
- Test Design Techniques

Day 3

- Test Management
- Tool Support for Testing

The Course Binder

- Introduction and course timeline
- For each of the 6 course sections:
  - Course slides with additional text notes
  - Classroom exercises
  - Knowledge check (including answer keys)
  - New terms
- Syllabus
- Glossaries of terms
  - English only
  - English -> Swedish
  - English -> Swedish without definitions
- Practice exam (including answer keys)
- Study guide
- Reference materials

The Driver's License Analogy

- This training course alone will not make you a software tester – why is that?
- This course will give you an excellent theoretical base of methods, tools and terms
- **To fully master the profession, you will also need to collect practical experience; this happens when you apply what you have learned in your daily work**

Foundation Level Examination

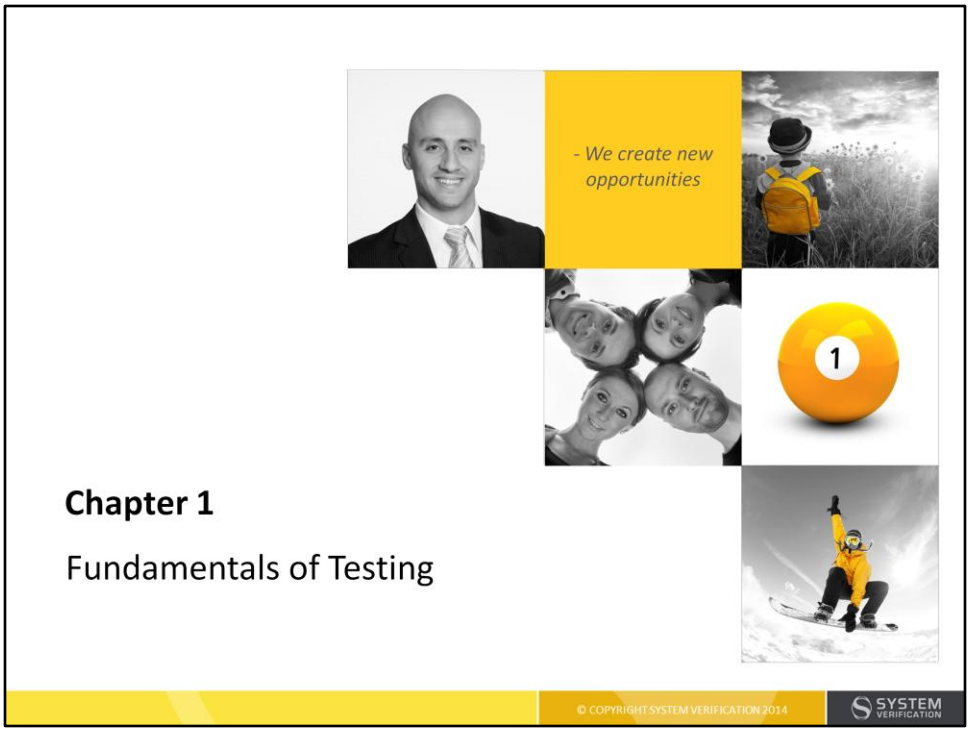
- The exam is based on the ISTQB® Foundation level syllabus
- All parts of the syllabus are examinable
- The exam comprises 40 multiple-choice questions on levels K1–K4
  - K1: remember, K2: understand, K3: apply, K4: analyze
- Questions will be asked on all main sections of the syllabus, and the weighting will be according to the amount of time spent on each section
- Each correctly answered question is allocated one point
- Pass mark is 65% (26 points or more)
- There are no penalties for incorrect answers
- Time allowed is 60 minutes (+ 15 minutes for non-native English speakers; please note that you must apply for the extended time before the exam)
- You may not bring any course material or notes the exam
- For exams held by SSTB you may bring
  - a standard English-Swedish dictionary
  - SSTB's English-Swedish list of test terms
- For exams held by other facilitators than SSTB no aids are allowed
- See the study guide for more information and tips

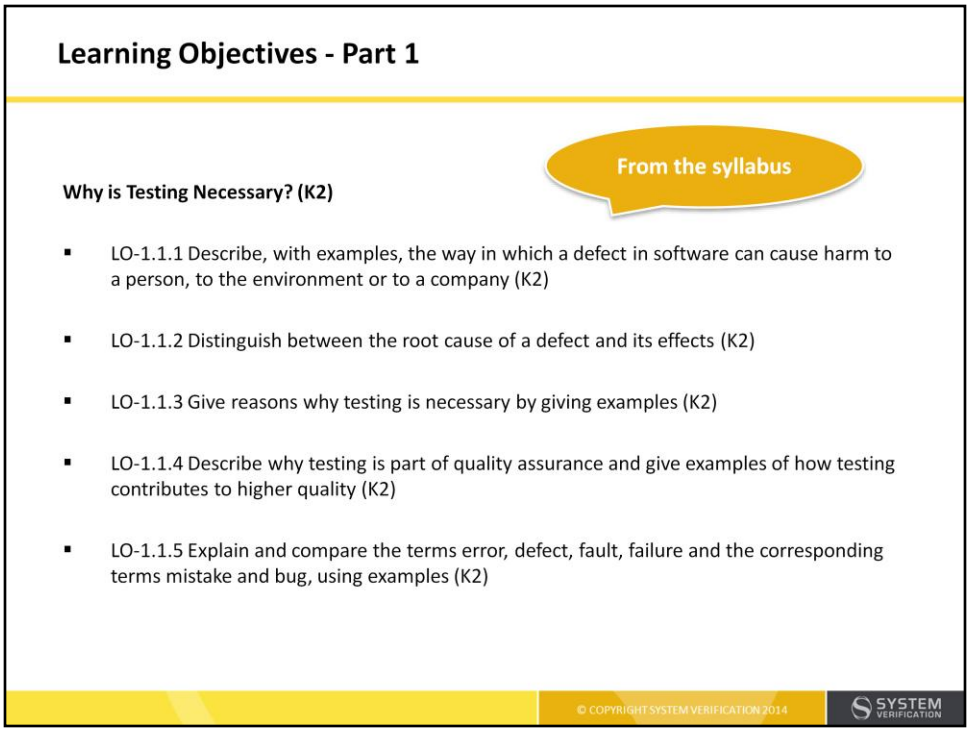
Practical Information

- This is a 3 day training course
- Course times (unless otherwise agreed) :
  - 9:30 – 17:00 (day 1)
  - 9:00 – 17:00 (day 2)
  - 9:00 – 16:00 (day 3)
  - Exam 16:00 – 17:15 (day 3)
- Coffee and lunch breaks (see course timeline)
- Facilities (reception, kitchen, restaurant)
- Safety (in case of emergency)
- Other practical questions

Contact

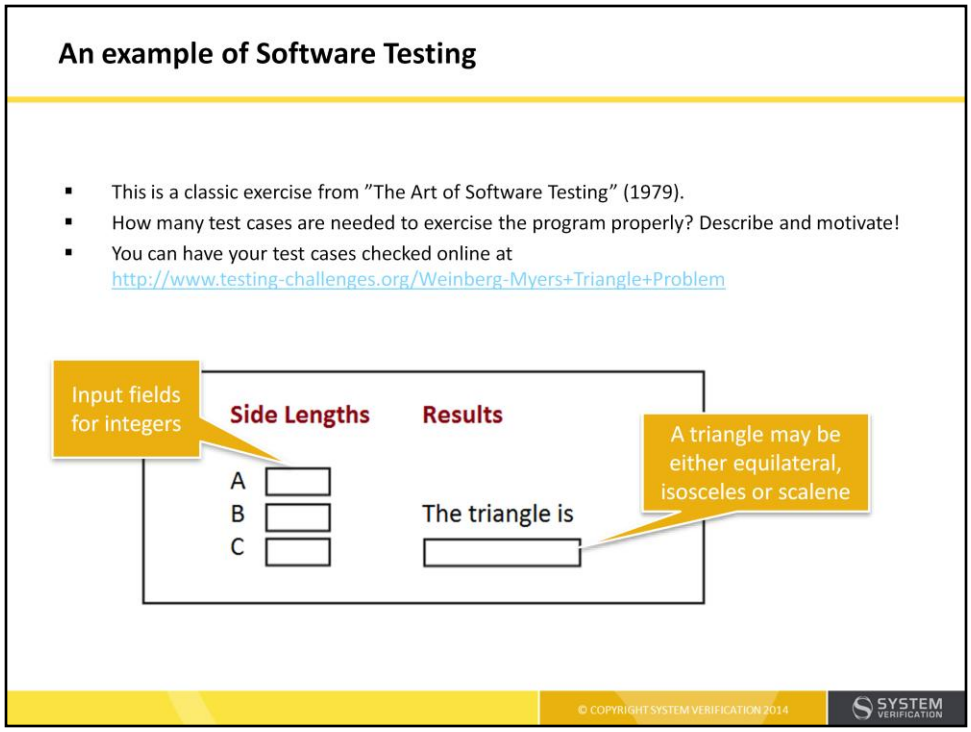
- Your teacher is available during the course to help you learn and prepare for the exam
- Stay in touch during and after the course
- Grab a business card
- Get in touch on LinkedIn
- Let us know if we can assist your organization with QA-related services





Time
Intended study time is 20 minutes

Terms
Bug, defect, error, failure, mistake, quality, risk



Testing involves a lot of different activities to find defects, ensure quality, investigate system behaviour and provide information for decision making.

In this example a software system has been developed to decide if a triangle is equilateral, isosceles or scalene. How many test cases are needed to test this program?

The first step could be to review the requirements to understand more in detail what the software shall do and to identify potential problems.

The next step could be to design and execute a number of test cases, i.e. based on a specific input, compare the actual result with the expected result.

Using 6, 6 and 6 as input should provide equilateral as result, but what happens if we use 6.0, 6.0 and 6.0 or letters instead of figures?

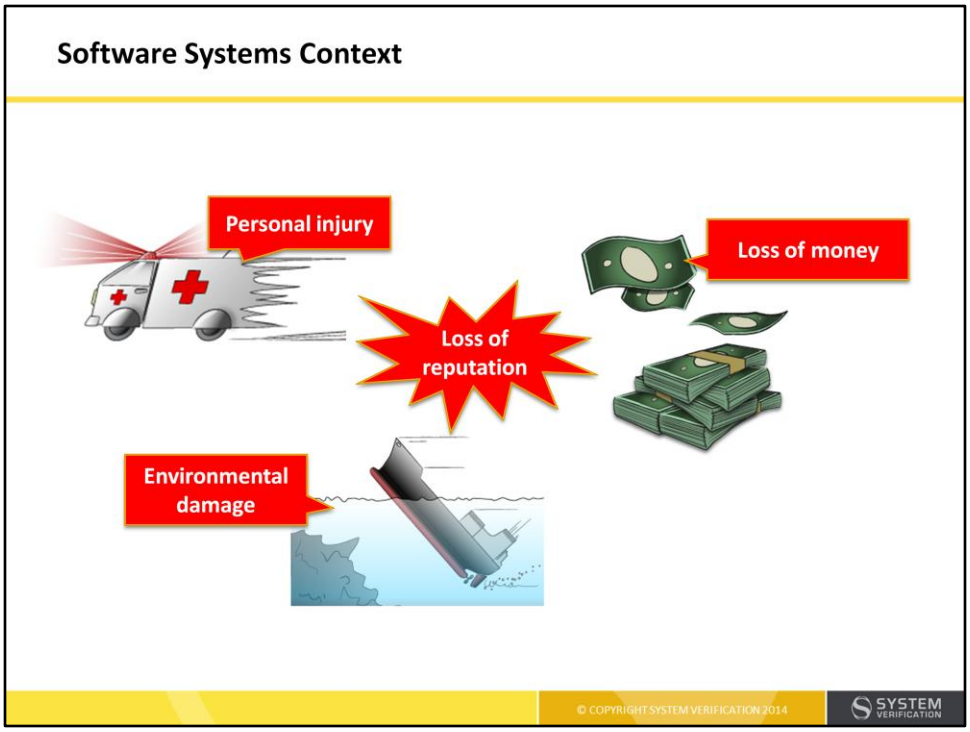
The Foundation of Testing

- Testing typically includes a number of activities:
  - Planning
  - Control
  - Choosing test conditions
  - Reporting results
- Testing will be different depending on your organization and project
- Testing needs a solid foundation for terms and processes

Normally there is more to test than design and execution of test cases, e.g. planning, control, choosing test conditions and reporting results.

Testing usually combines different kinds of testing methods and varies from organization to organization and from project to project. Testing needs to have a formal basis where terms and processes have a solid foundation.

This course aims to give you that foundation.



Software that does not work correctly can lead to many problems:
• Loss of money or reputation
• Personal injury
• Environmental damage

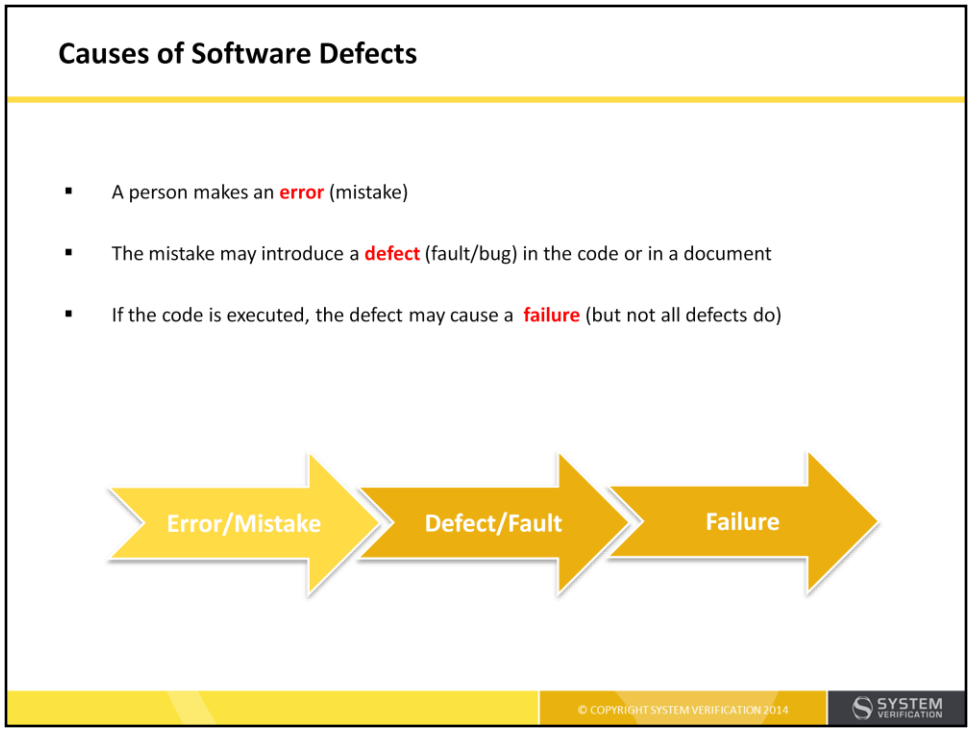
Rigorous testing with properly designed test cases can reduce the overall level of risk of these problems occurring in production by finding the problems before released in operational use. Software testing may also be required to meet contractual or legal requirements, or industry-specific standards.

Ariane 5 Flight 501
Flight 501 was the first test flight for the European Ariane 5 expandable launch system, unfortunately this was a very unsuccessful flight. A defect in the software caused the rocket to deviate from its planed flight path only 37 seconds after take-off. The defect triggered Ariane 5s self-destruct system and Ariane 5 was destroyed. Luckily this was an unmanned flight so there were no victims, but the loss of four Cluster mission spacecraft resulted in a loss of more than $370 million.

Personal Injury
GPS is used in emergency vehicles to locate and find the fastest way to an injured person. The human who made the emergency call had given the GPS coordinates where the injured man was. The GPS in the ambulance led the doctor to a place 1000m away from the correct location. Unfortunately the patient died due to late arrival of the doctor.

The reason for the wrong positioning was that the GPS client that the emergency caller used had a defect that caused the GPS to provide wrong coordinates under special circumstances.

Environmental Damage
Software systems are used to sort shipping containers evenly according to weight as they are loaded onto freighters. If an failure occurs, the ship will be unbalanced and its cargo may be lost in hard weather, damaging the environment.



The human makes an error (called a mistake) and it is done unintentionally. The mistake introduces a defect, sometimes called a fault or bug in the code or in a document.

If a defect in code is executed, the system may produce unintended results, causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so.

Error/Mistake

- A person makes an error/a mistake
- Possible reasons might include:
  - Time pressure
  - Complex code
  - Changing technologies
  - Many system interactions

An error is a human mistake which can lead to a chain of other negative effects.

The reason for the mistake could for example be:
• time pressure - tight deadlines in the project
• complex code - code constructions difficult to understand and maintain
• changing technologies - moving from Java to .NET development
• many system interactions - large scale systems with many dependencies to other systems

Defect/Fault/Bug

- A mistake may introduce a defect (fault and bug are other terms which are synonyms)
- A defect is likely to cause a failure, but not always (e.g. dead code)

A defect in software, systems or documents is a result of an error. Defects are also known as faults. Not all defects are detected because they can exist in code that never is executed.

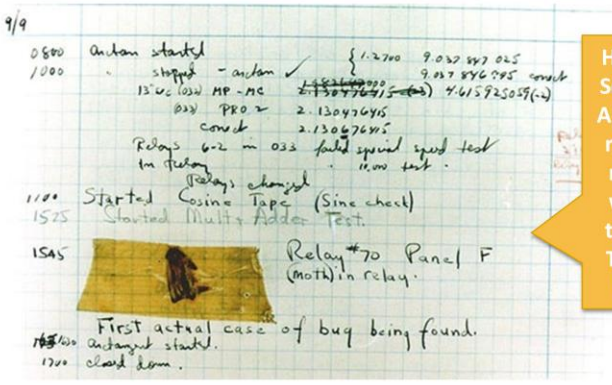
This is called "dead code" and could be any part of a program that can never be accessed because all calls to it have been removed, or because it is guarded by a control structure that always transfer control somewhere else.

Failure

- Possible causes of a failure:
  - Software defect
  - Hardware malfunction
  - Environmental conditions, e.g. radiation, magnetic interference
- **Testing should aim at detecting the failures that cause the highest risk**

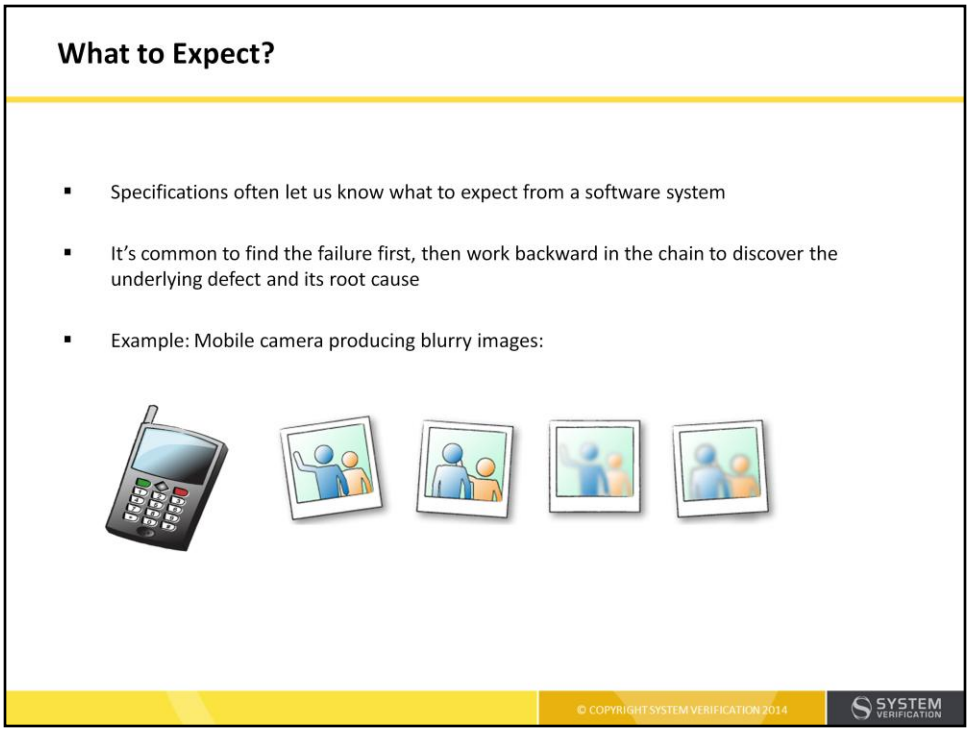
A failure is defined as a deviation of the component or system from its expected delivery, service or result. There are many possible causes of a failure, such as:
• Software defect
• Hardware malfunction
• Environmental conditions, for example radiation or magnetic interference

"First Actual Case of Bug Being Found"



Harvard University, September 9, 1947: A moth trapped in a relay had caused a runtime failure. It was removed and taped into the log. The computer had been debugged.

Photo: U.S. Naval Historical Center



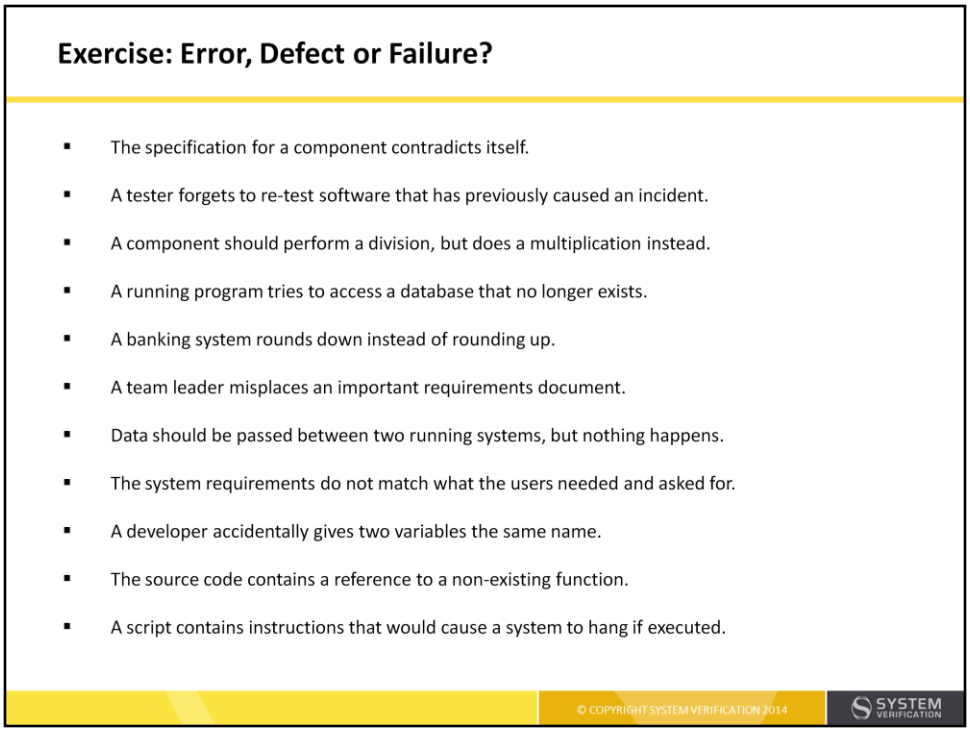
We know what to expect from a software system because we usually have specifications that outline the expected behaviour. We can use the specifications as a "test oracle" that predicts how systems will behave if we give them certain inputs.

In many cases, we find the failure first, then work backward in the chain of events to find the underlying defect and its root cause.

An example
In 2009, a major mobile handset maker released a new autofocus feature for its camera. Unfortunately an error was made and a defect introduced which caused some strange behaviour and failure.

The problem was that the camera would focus but then fail and blur out providing 'red corners'. For some reason the camera would function normally 24.5 days later, i.e. it would work for 24.5 days, then have poor performance for 24.5 days and then work again.

The reason for this failure was an error made when designing the code. A rounding-error had been introduced in the mobile's time and date module, i.e. the defect. The camera's autofocus routine used a time stamp value which caused the camera to fail in cycles.



• The specification for a component contradicts itself – Defect
• A tester forgets to re-test software that has previously caused an incident – Error
• A component should perform a division, but does a multiplication instead – Failure
• A running program tries to access a database that no longer exists – Failure
• A banking system rounds down instead of rounding up – Failure
• A team leader misplaces an important requirements document – Error
• Data should be passed between two running systems, but nothing happens – Failure
• The system requirements do not match what the users needed and asked for – Defect
• A developer accidentally gives two variables the same name – Error
• The source code contains a reference to a non-existing function – Defect
• A script contains instructions that would cause a system to hang if executed – Defect

SYSTEM VERIFICATION

The Role of Testing

- Extensive testing of system and documentation will contribute to the quality of the system and may reduce the risk of problems
- Testing may be required:
  - by contract
  - by industry-specific standards
  - by law

Extensive testing of system and documentation will contribute to the quality of the software system and may reduce the risk of problems when the system is operational. Other reasons for software testing could be that it is required by the contract, industry-specific standards or legal requirements.

Testing and Quality

- Testing makes it possible to measure the quality of software in terms of defects found
- If defects are found and corrected, the software will increase in quality
- By analysing the root causes of defects found, processes can be improved, preventing defects from reoccurring
- **Software testing is an important part of quality assurance**

Testing makes it is possible to measure the quality of software in terms of defects found for software requirements and characteristics. If defects are found and corrected, the software will increase in quality.

By learning from previous projects, through understanding the root causes of defects found, processes can be improved. This should prevent defects from reoccurring and thus improve the quality of future systems.

This means that software testing is an important part of quality assurance.

Confidence in the Quality

- Testing cannot prove the absence of defects.
- Testing can only prove that there are defects.
- No defects = good quality or bad testing.
- If you have confidence in your testing, your assessment of product quality will be better justified.

It is important to remember that testing **cannot** prove absence of defects, testing can only prove that there are defects.

If few or no defects are found when testing, this can mean that the quality is good but it can also mean that the testing have been done very poorly.

A properly designed test that passes reduces the overall level of risk in a system. If you have confidence in the testing, it is more justifiably to say that the quality is good.

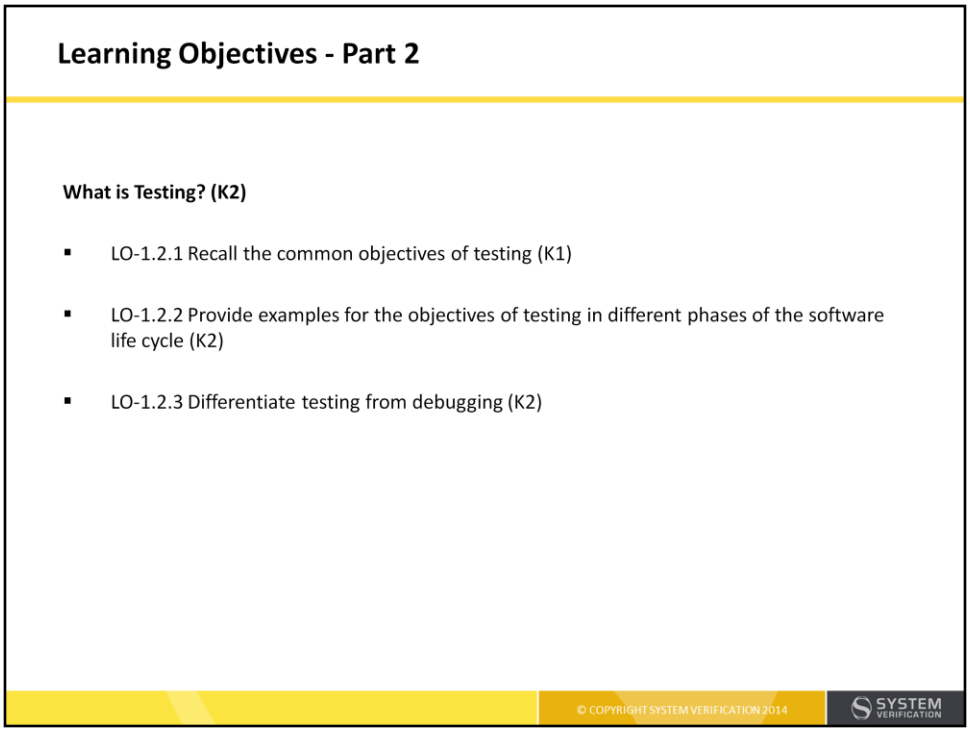
How Much Testing is Enough?

- Time is almost always insufficient, so the amount of testing needs to be appropriate.
- Prioritize testing based on risk (technical, safety and business risks) and project constraints (time, budget).
- Testing should provide stakeholders with proper and sufficient information to make decisions.

In the vast majority of projects, time is insufficient. It is important that enough testing is carried out.

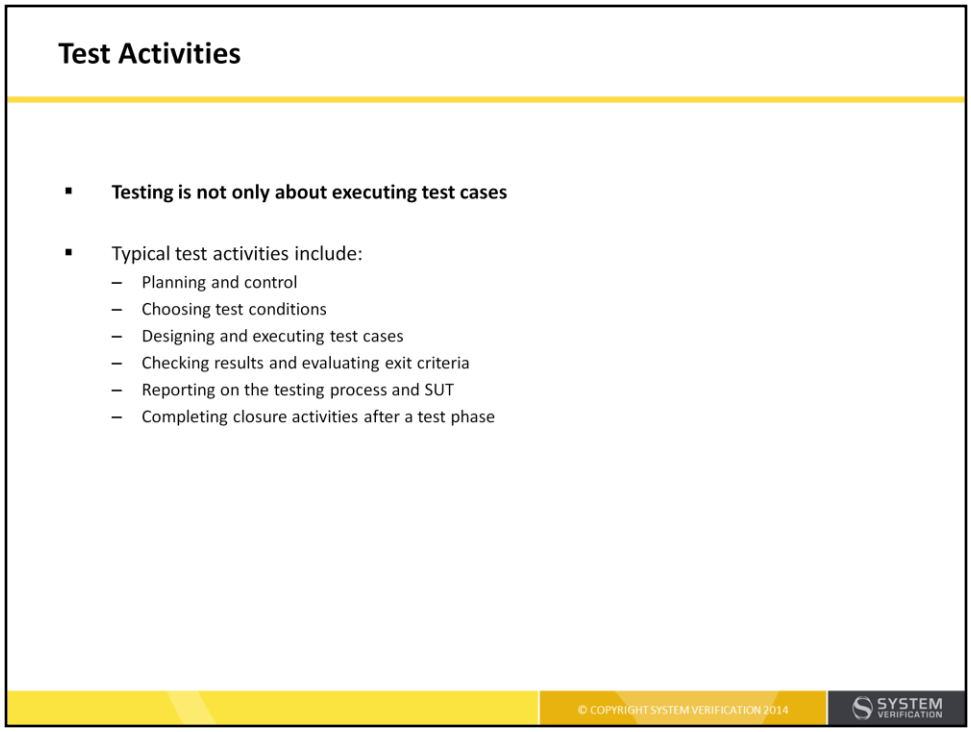
Deciding how much testing is enough should take account of the level of risk, including technical, safety, and business risks, and project constraints such as time and budget.

It is also important that the testing provides the stakeholders proper and enough information, enabling them to make correct decisions about when to proceed for the next development step or when to release the software or system to the customers.



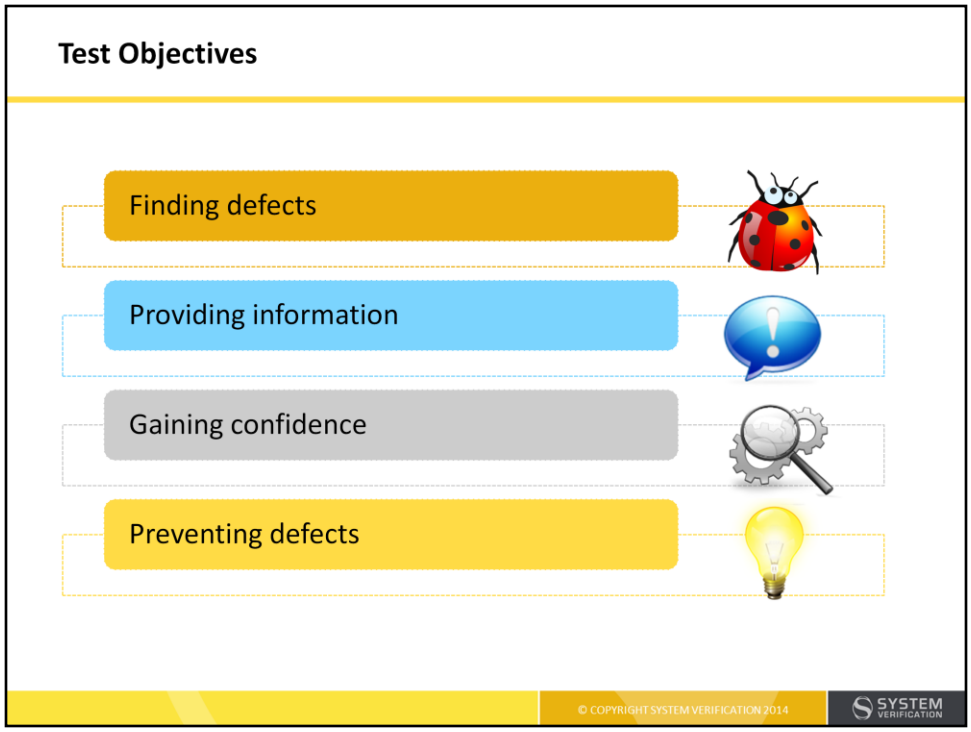
Time
Intended study time is 30 minutes

Terms
Debugging, requirement, review, test case, testing, test objective



Very often testing is seen as equal to running test when executing the software. This is one test activity but not all. Testing also includes reviewing documents and conducting static analysis. Other typical activities are:

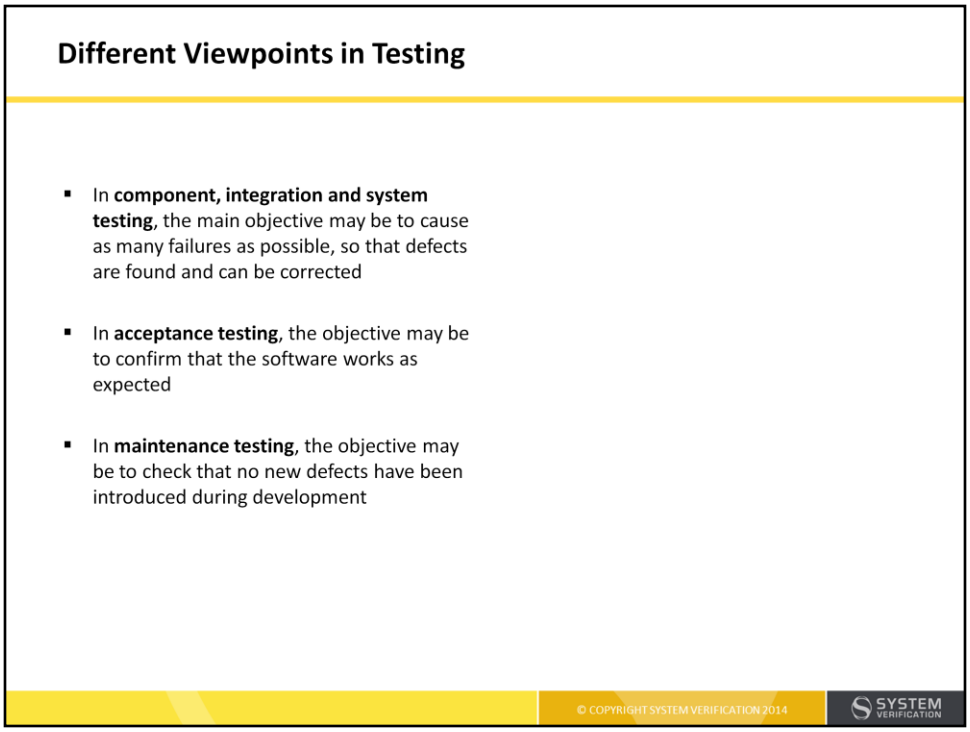
- Planning and control
- Choosing test conditions
- Designing and executing test cases
- Checking results and evaluating exit criteria
- Reporting on the testing process and system under test
- Completing closure activities after a test phase



Different types of testing can be used to achieve similar objectives and will provide information that can be used to improve both the system being tested and the processes. There can be different objectives:

- Finding defects
- Gaining confidence about the quality level
- Providing information
- Preventing defects

The thought process of designing tests early in the life cycle to verify the test basis, can help to prevent defects from being introduced into code. Reviews of documents, for example requirements, also help to prevent defects appearing in the code.



Different viewpoints in testing take different objectives into account. In development testing (e.g. component, integration and system testing), the main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed.

In acceptance testing, the main objective may be to confirm that the system works as expected, to gain confidence that it has met the requirements.

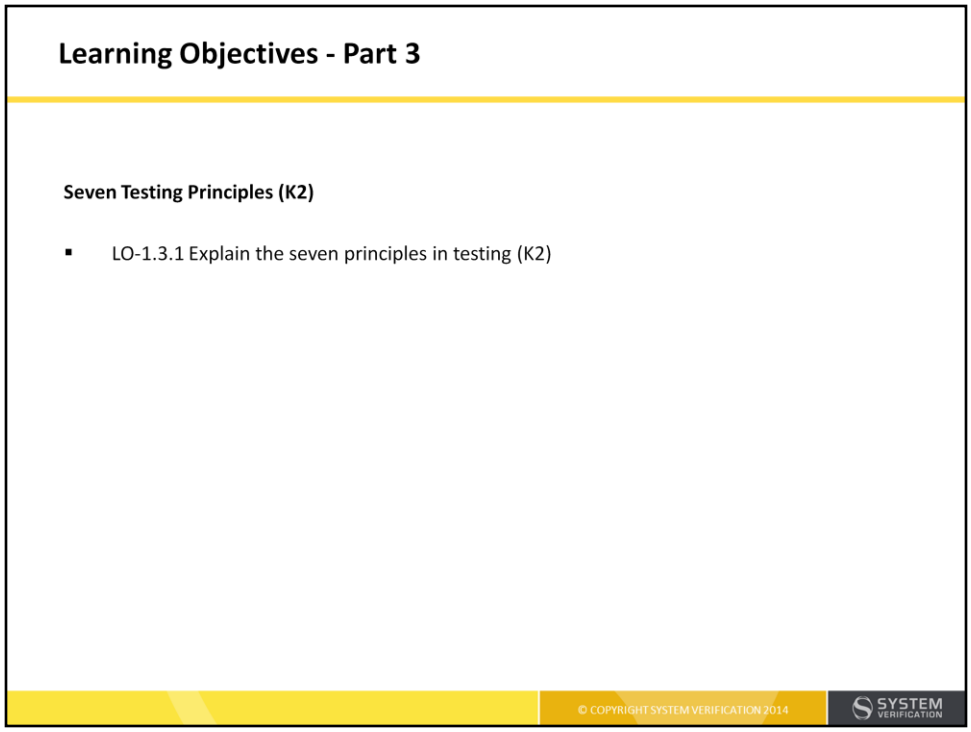
In some cases the main objective of testing may be to assess the quality of the software. This to be able to give, for example the stakeholders, the information of the risk releasing the system at a given time.

Maintenance testing often includes testing that no new defects have been introduced during development of the changes while the main objective for operational testing may be to assess system characteristics such as reliability or availability.

Debugging and Testing

- Debugging is a development activity that identifies the cause of a failure, repairs the defect and checks that it has been fixed correctly
- Debugging is normally done by developers, while testing is done by testers
- Subsequent re-testing should be done by a tester to verify that the fix has indeed resolved the failure

Debugging and testing are activities with different purposes. Testing is normally carried out by testers while debugging is done by developers. They can be differentiated as follows:
Dynamic testing can show failures that are caused by defects.
• Debugging is a development activity that identifies the cause of a failure, repairs the defect and checks that it has been fixed correctly.
• Subsequent re-testing by a tester ensures that the fix does indeed resolve the failure.



Time
Intended study time is 35 minutes

Terms
Exhaustive testing

Principle 1: Testing Shows Presence of Defects

- Testing cannot prove the absence of defects
- Testing can only prove that there are defects
- Because time and resources are limited, we must decide what to focus our test efforts on

Testing can never prove correctness. It can show that defects are present, but we **cannot** prove that we have found all of them.

This mean that you must think about what you would like to focus the testing on and how it should be carried because time and resources are limited (principle 2).

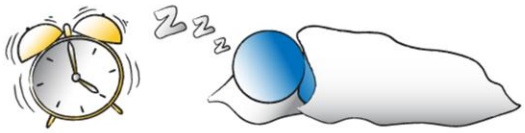
Principle 2: Exhaustive Testing is Impossible

- Testing all combinations of inputs and preconditions is not generally feasible in practise
- Instead, we should use risk analysis and priorities to focus our testing efforts according to objectives

Testing everything (all combinations of inputs and preconditions) is called exhaustive testing. In practice, it is not possible to do exhaustive testing. Most systems are too complex for this to be possible in any realistic time frame. Instead, we use risk analysis and priorities to focus testing efforts according to objectives.

Principle 3: Early Testing

- Begin testing as early as possible
- The sooner testing begins, the easier and cheaper it is to find and fix defects
- Testable requirements support early testing

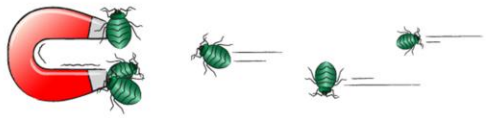


Testing activities should start as early as possible in the software or system development life cycle. The testing shall be focused on defined objectives. The sooner testing begins, the easier it is to find defects and cheaper to fix them.

Early testing means early involvement and put requirements on testability. A testable requirement shall be precisely defined and unambiguous to make it possible to write one or more test cases that would validate whether the requirement has or has not been implemented correctly. For every stage in the development the cost of fixing a defect approximately increase with a factor of 10.

Principle 4: Defect Clustering

- Often, a small number of modules contain most of the defects or account for the most failures
- Possible reasons for defect clustering:
  - Newly developed functionality
  - Complex modules or interfaces
  - Poor specifications
  - Time pressure



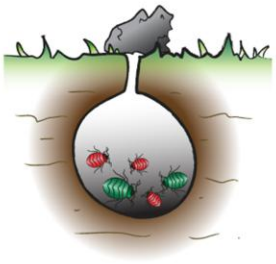
A small number of modules contain most of the defects discovered during pre-release testing, or are responsible for the most operational failures. This is known as defect clustering.

There are many different reasons for defect clustering. It could for example be:
• newly developed functionality
• complex modules
• poor specifications
• time pressure
• complex interfaces

By analysing the test results and gather information about changes it is possible to decide where more extensive testing might be necessary.

Principle 5: Pesticide Paradox

- Test cases run repeatedly will eventually stop finding new defects
- To overcome this, test cases need to be regularly revised
- This is especially important in automated functional regression testing
- Unless test suites are maintained, they will become outdated



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 paradox, the 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 potentially find more defects.

This is important to remember regarding regression test suites. The test suites have to be maintained and modified otherwise they will become outdated.

Principle 6: Testing is Context-Dependent

- Testing is executed differently in different contexts
- Compare the testing of
  - an iPad gaming app
  - an e-business web site
  - a flight control system

Testing is executed differently in different contexts. For example, safety-critical software is tested differently from software that is less critical.

Compare a business application, such as billing software, with the safety systems of a power station. It becomes apparent that the consequences of failure in the later case are much more serious. If it is a cold winter people could for example freeze to death due to the power cut.

Principle 7: Absence-of-Errors Fallacy

- A system must fulfil the users' needs and expectations
- Finding and fixing defects does not help if the system built is unusable
- Getting frequent feedback from stakeholders and potential users ensures that requirements and wishes are captured at an early stage

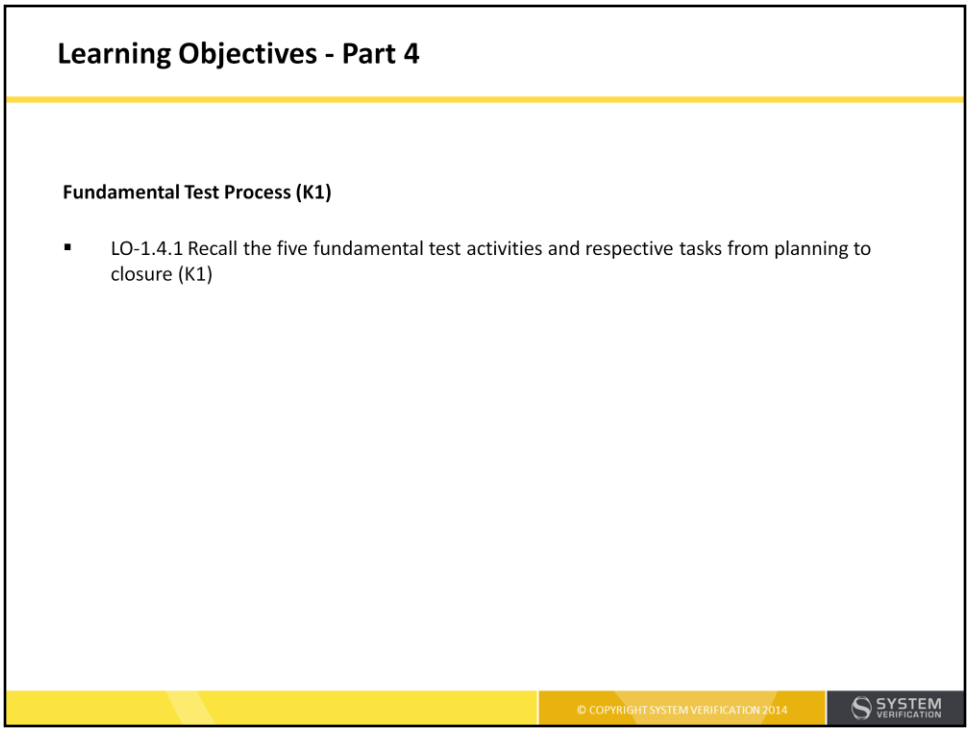
Finding and fixing defects does not help if the system built is unusable and does not fulfil the users needs and expectations.

By frequently demonstrating newly developed parts of a system, and asking for feedback from stakeholders and potential users, makes it possible to reveal implied requirements and wishes.

Exercise: Which Testing Principle?

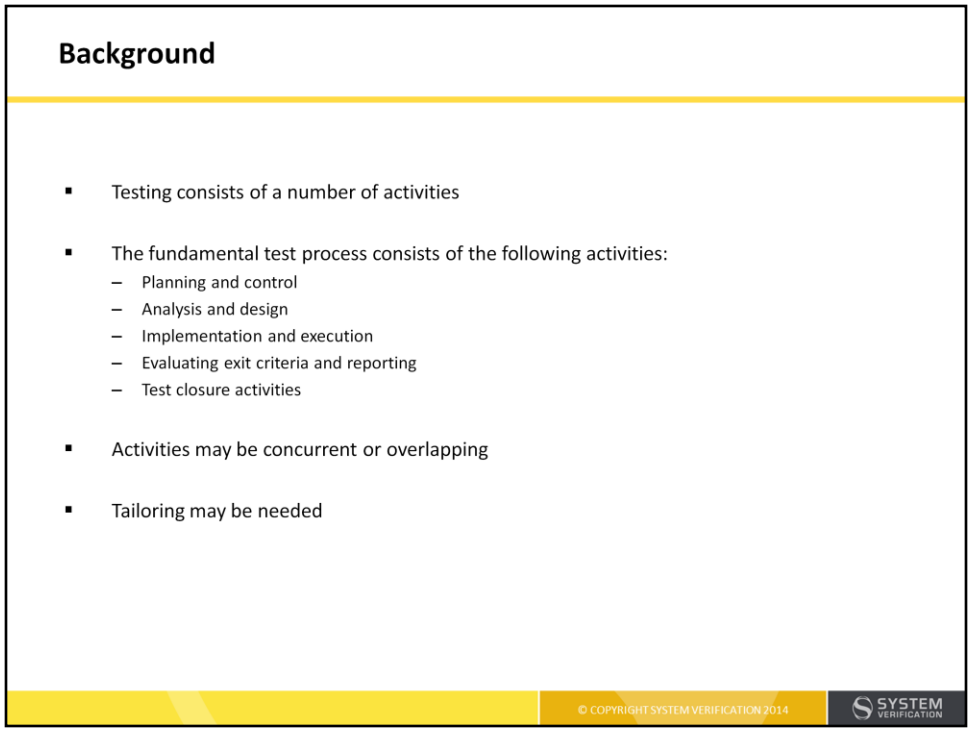
- Fixing defects is not useful if the system does not do what the user asked for.
- Most defects are usually found in the same component.
- Different projects require different testing procedures.
- Testing every possible input is not practical.
- Repeating the same test will result in no new defects found.
- Testing cannot prove there are no defects left in the code.
- Defects are cheaper to fix and easier to find the earlier testing starts.

• Fixing defects is not useful if the system does not do what the user asked for – Absence-of-errors fallacy
• Most defects are usually found in the same component – Defect clustering
• Different projects require different testing procedures – Testing is context-dependent
• Testing every possible input is not practical – Exhaustive testing is impossible
• Repeating the same test will result in no new defects found – Pesticide paradox
• Testing cannot prove there are no defects left in the code – Testing shows presence of defects
• Defects are cheaper to fix and easier to find the earlier testing starts – Early testing



Time
Intended study time is 35 minutes

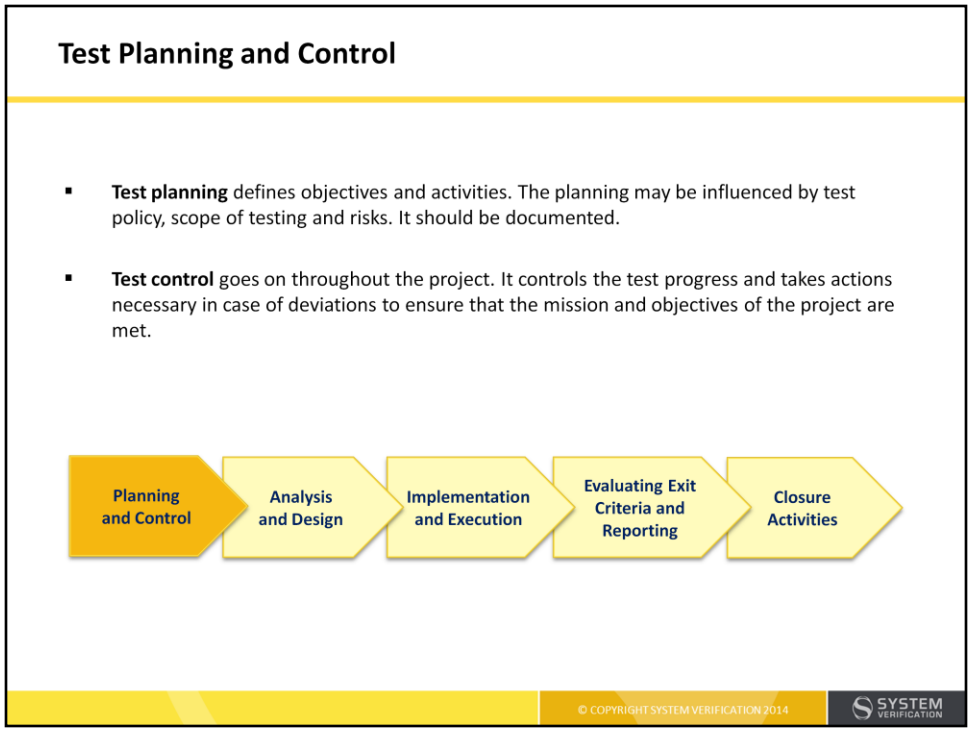
Terms
Confirmation testing, re-testing, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test procedure, test policy, test suite, test summary report, testware



The most visible part of testing is test execution but there are other activities as well. Test plans should also include time to be spent on planning the tests, designing test cases, preparing for execution and evaluating results to be as effective and efficient as possible.

The fundamental test process consist of the following main activities:
• planning and control
• analysis and design
• implementation and execution
• evaluating exit criteria and reporting
• test closure activities

The activities in the process may overlap and take places concurrently. Tailoring these activities within the context of the system and the project is usually required.



Test planning is when you define test objectives and decide the test activities in order to meet the objectives.

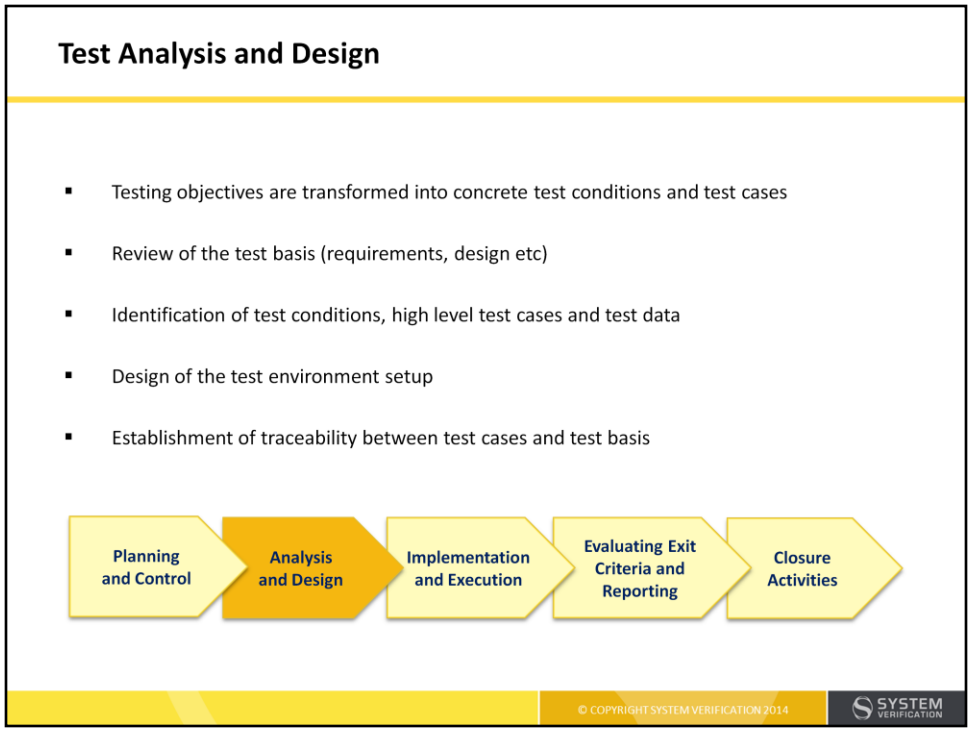
You also need to monitor and control the test progress, which starts during planning and goes on throughout the project.

Planning tasks
Things that influence the planning is, for example, test policy of the organization, scope of testing and risks. Based on this an overall approach must be decided, resources allocated, and test activities scheduled. This planning should be documented.

Test planning is a continuous activity and is performed in all life cycle processes and activities.

Control tasks
Test control is the ongoing activity of comparing actual progress against the plan and reporting the status, including deviations from the plan. It involves taking actions necessary to meet the mission and objectives of the project.

In order to control testing, it should be monitored throughout the project. Test planning takes into account the feedback from monitoring and control tasks.



During test analysis and design the general testing objectives are transformed into concrete test conditions and test cases.

The activity of identifying test conditions is test analysis while test design is the activity of creating the specific inputs that will be entered to the system and the exact results that will be checked.

Analysis tasks
Typical tasks are reviewing the test basis (such as requirements, architecture, design, interfaces) and evaluating the testability of the test basis and test objects. The evaluation can be done very early in the life cycle, i.e. as soon as the test basis documents are available.

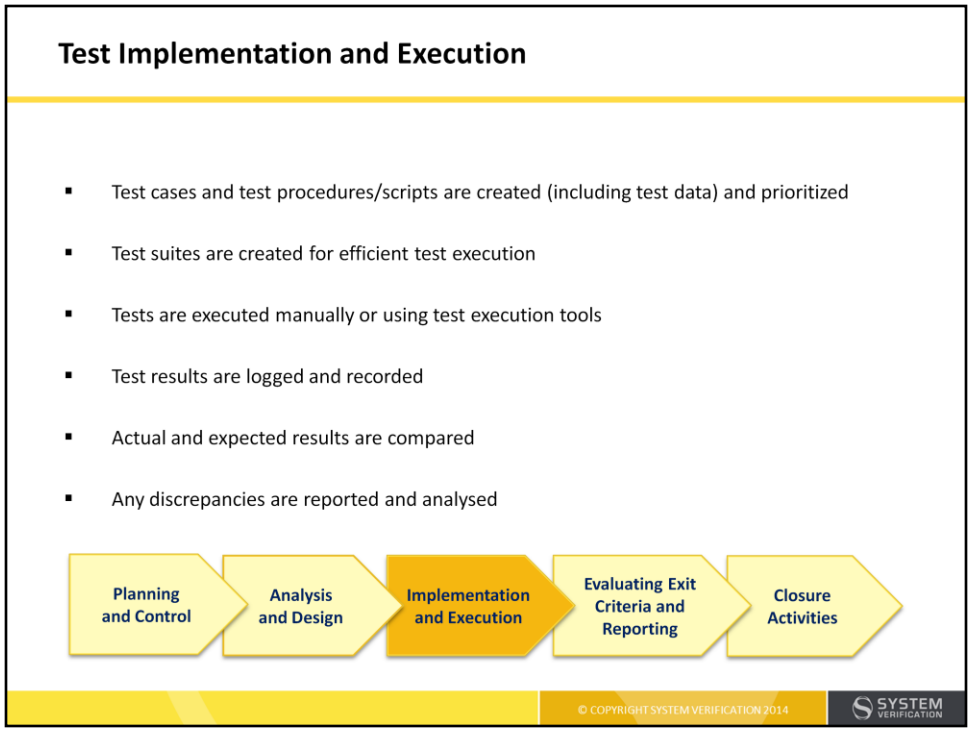
Part of the analysis are identifying and prioritizing test conditions based on analysis of the test items, the specifications, behaviour and structure of the software.

The analysis can be carried out by experienced persons or in joint sessions such as brainstorming meetings or workshops.

Design tasks
Typical design tasks include designing and prioritizing high level test cases and identifying necessary test data to support the test condition and test cases.

They also include designing the test environment set-up and identifying any required infrastructure and tools.

An important part is to have bi-directional traceability between the test basis and test cases, which should be established as a part of this activity.



Test implementation and execution is the activity where test procedures or scripts are specified, the environment is set up and the tests are run.

Implementation Tasks
Tasks that belong to this activity are finalizing, developing, implementing and prioritizing test cases and procedures. This includes identification and creation of test data. In some cases automated test scripts are also prepared. From the test procedures different test suites are created for efficient test execution.

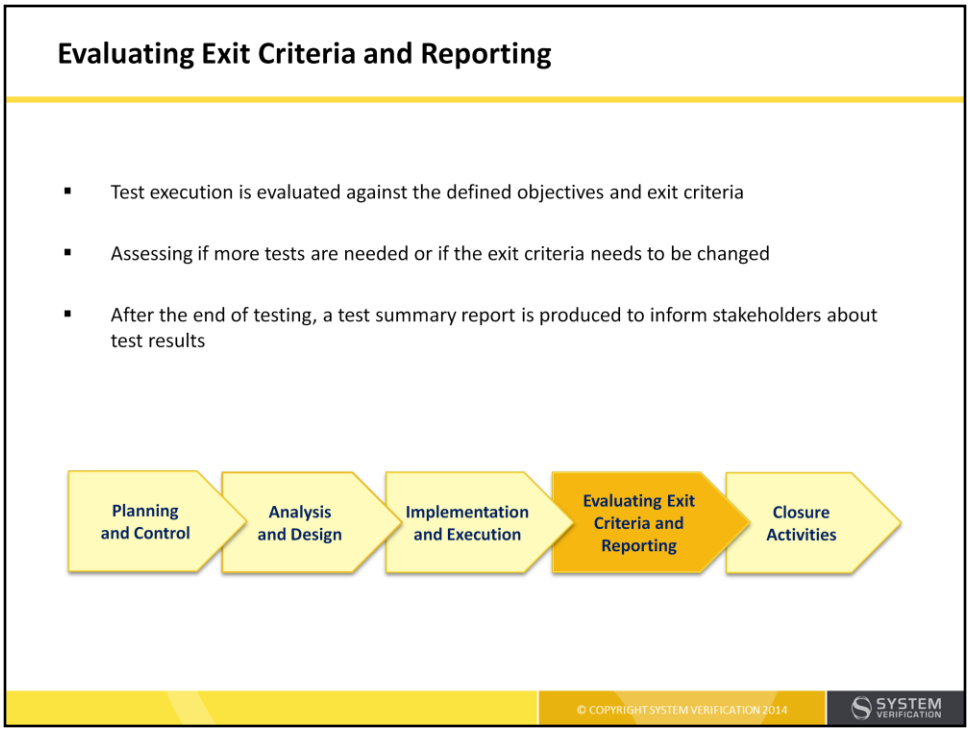
Before continuing, it should be verified that the test environment is correctly set up. The traceability between the test basis and the test cases should also be verified and updated if necessary.

Executions tasks
At this point, execution of test procedures begin according to the planned sequence, either manually or by using test execution tools.

The outcome of test execution is logged and recorded, including the identities and versions of the software and testware. Finally, actual results are compared with expected results.

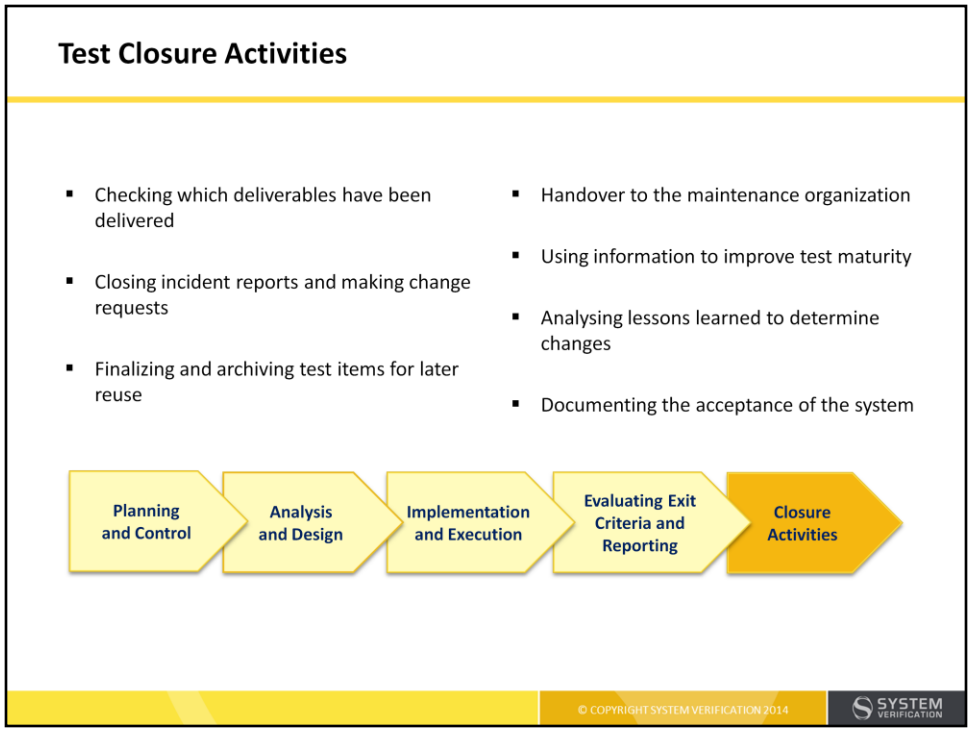
If discrepancies between actual and expected results are found, they are reported as incidents and are analyzed in order to establish their cause, for example, a defect in the code, in specified test data, in the test document, or a mistake in the way the test was executed.

It might also be necessary to re-execute a test that previously failed, in order to confirm that the defect was fixed, i.e. confirmation testing. Another repeating test activity is regression testing regression testing that is performed in order to ensure that defects have not been introduced in unchanged areas or that fixing defects have not uncovered other defects.



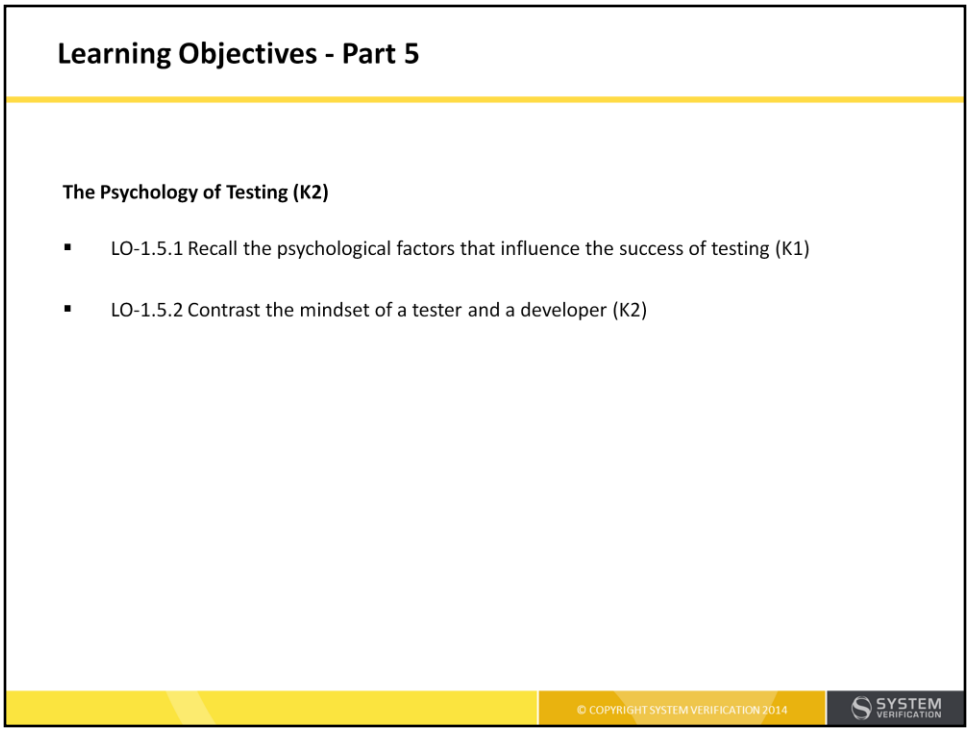
These activities aims at evaluating test execution progress against the defined objectives and reporting the results. This shall be done for each test level.

Evaluating exit criteria has the following major tasks:
• Checking test logs against the exit criteria specified in test planning.
• Assessing if more tests are needed or if the exit criteria specified should be changed.
• Writing a test summary report for stakeholders.



The closure activities occurs at project milestones, for example, when a software system is released, a test project is completed or cancelled, or a maintenance release has been completed. These activities collect data to consolidate experience, testware, facts and numbers.

Test closure activities include:
• Checking which planned deliverables have been delivered.
• Closing incident reports and raising change records for any that remain open.
• Finalizing and archiving testware, the test environment and the test infrastructure for later reuse.
• Handover of testware to the maintenance organization.
• Using the information gathered to improve test maturity.
• Analysing lessons learned to determine changes for future releases and projects.
• Documenting the acceptance of the system.



Time
Intended study time is 25 minutes

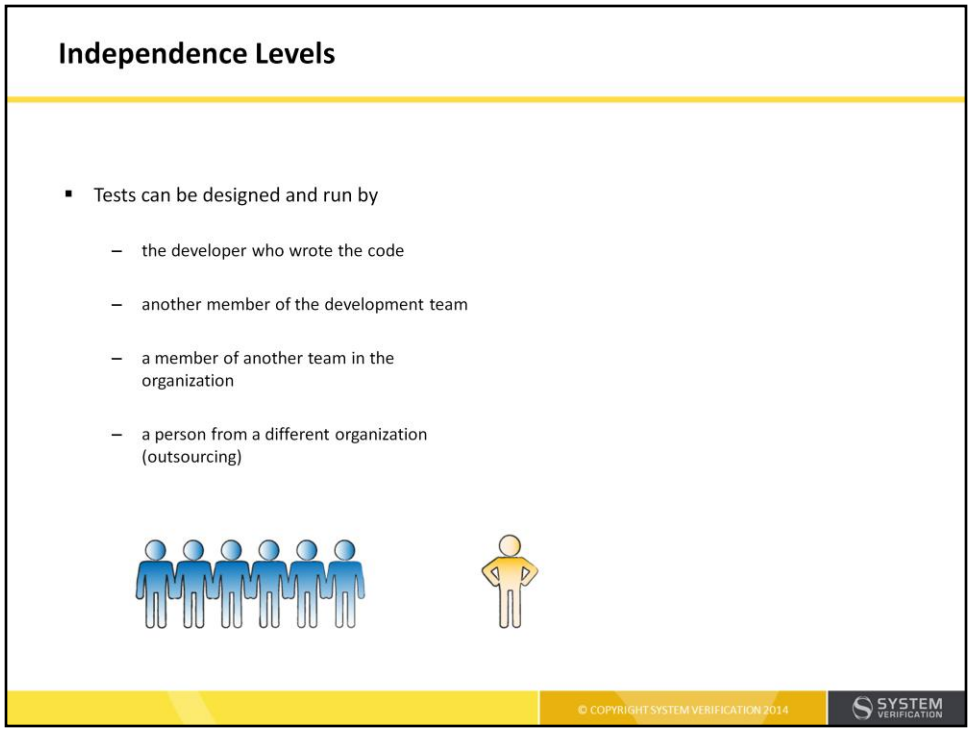
Terms
Error guessing, independence

Independence

- Working as developers and testers requires different mind sets (constructive vs. destructive)
- Developers can (and should) test their own code
- Professional testing resources can help provide a focused testing effort and an independent view

The right mindset is important while testing and reviewing software. It differs from that used while developing software, i.e. the developer tries to construct the correct software while the tester tries to find flaws in the construction.

With the right mindset developers are able to test their own code but it is normally a testers responsibility. This will help focus effort and provide an independent view by trained and professional testing resources. Independent testing can be carried out at any level of testing.



The tester is often more effective at finding defects and failures if he/she have a certain degree of independence and thus avoiding author bias. We have to keep in mind that independence is not a replacement for familiarity because developers can efficiently find many defects in their own code.

When talking about independence there is a number of levels from low to high:
Tests designed by the developer who wrote the code, i.e. low level of independence.
• Tests designed by another member of the development team, not the developer of the code.
• Tests designed by person from another team in the organization, for example an independent test team or test specialists.
• Tests designed by a person(s) from a different organization or company, for example a outsourced test site.

Objectives

- Ensure that objectives are clearly stated and understood
- Ensure that important objectives are not neglected
- Examples of objectives for testing:
  - Measuring quality
  - Finding defects
  - Building confidence

It is important that the objectives of testing is clearly stated because people and projects are driven by objectives, which could vary between projects. People tend to meet the objectives that apply, or at least that they think are applicable.

For example, in one project the objective could be finding as many defects as possible but in another project it could be to get confidence in a business process.

Focusing on only one objective could lead to that other product aspects are missed, e.g. only focusing on finding defects may result in a system that is not user friendly.

Communication

- Successful testers are typically experienced, curious, professional pessimists with a critical eye and attention to details, and good communicators
- Communication problems may easily occur
- Good practices to improve communication:
  - Collaboration rather than battles
  - Always stay neutral and fact-focused
  - Understand how the other person feels
  - Confirm that the other person understood and vice verse

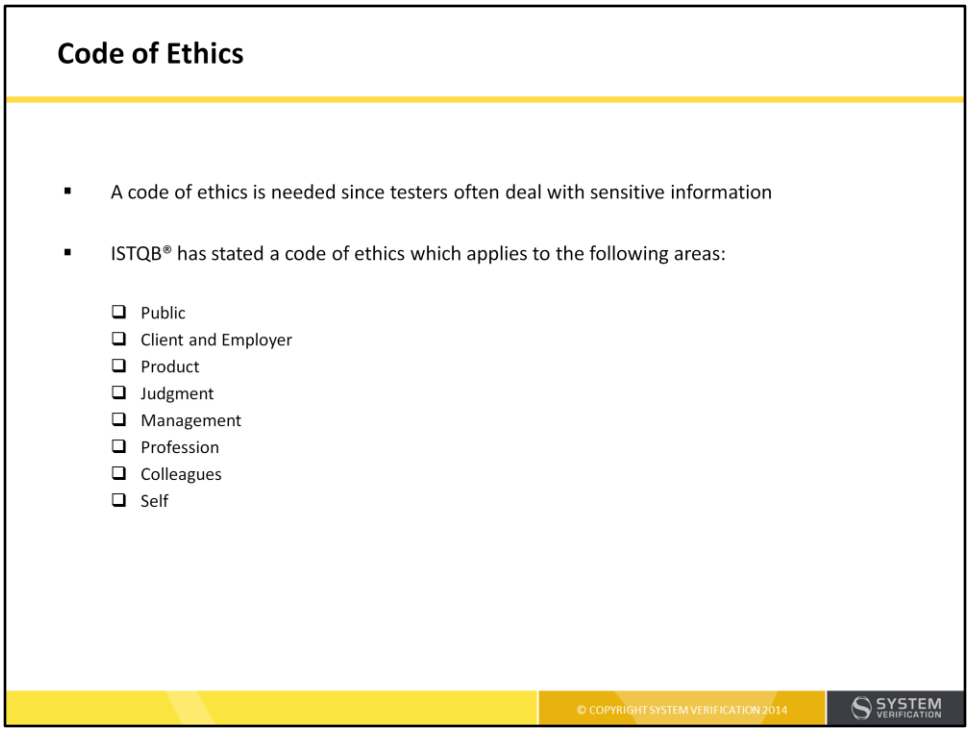
Identifying failures during testing may be perceived as criticism against the product and against the author. Therefore, testing is often seen as a destructive activity, even though it is very constructive in the management of product risks. To be successful looking for failures in a system requires:
• Curiosity · Professional pessimism · A critical eye · Attention to details
• Good communication with development peers · Experience on which to base error guessing

Bad feelings between the testers and colleagues, for example analysts, designers and developers, can be avoided if issues are communicated in a constructive way. This is true for testing as well as reviews. Testers and test leaders need good interpersonal skills to be able to communicate factual information in a constructive way about, for example, defects, progress and risks. This information can help the author of the software or document to improve their skills by learning from their mistakes.

Communication problems may occur if testers are seen only as messengers of unwanted news about defects, i.e. we shall not shoot the messenger. There are several ways to improve communication between testers and project members. Things to think about are for example:
• Start collaboration rather than battles
• Communicate findings on the product in a neutral and fact-focused way without criticizing the person who created it
• Try to understand how the other person feels and why they react as they do
• Confirm that the other person has understood what you have said and vice verse

Mutual understanding and collaboration between testers and developers is a powerful tool to increase product quality and project efficiency.

Always Be Neutral



Working with testing means that you and your colleagues will have access to confidential information about products and processes, e.g. requirements and test results. This information must be handled with care and not used in an inappropriate way. A code of ethics is necessary and ISTQB® states the following:

Public - Certified software testers shall act consistently with the public interest.

Client and employer - Certified software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.

Product - Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible.

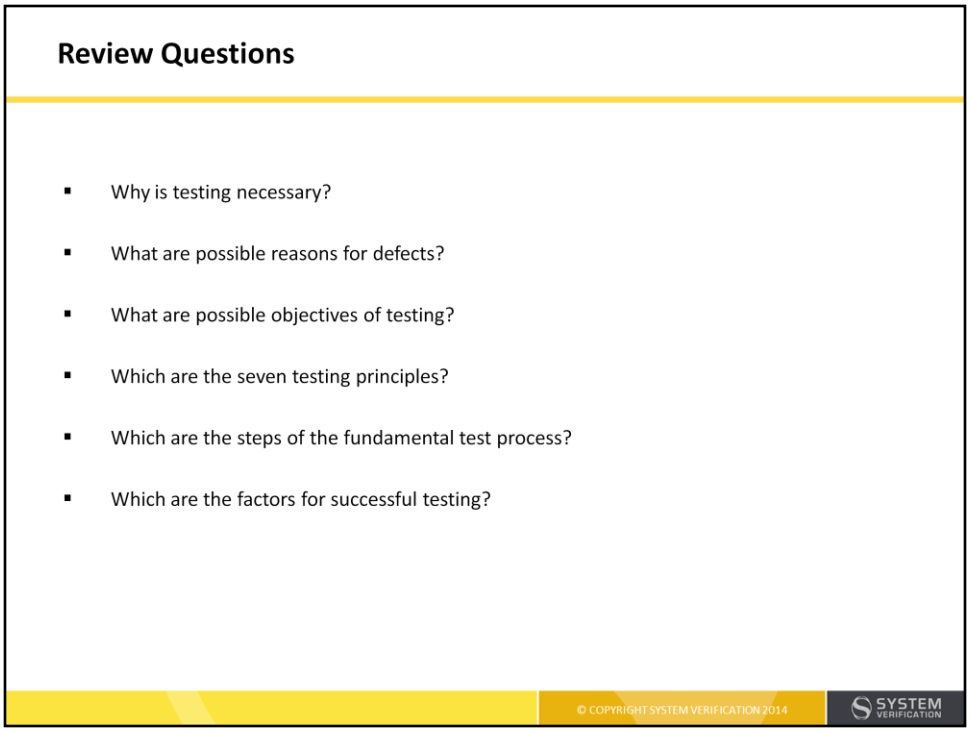
Judgment - Certified software testers shall maintain integrity and independence in their professional judgment.

Management - Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing.

Profession - Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest.

Colleagues - Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers.

Self - Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.



Summary

In this chapter you have learnt the basics for test, i.e. why testing is necessary, the purpose of it and that a non-tested product can cause problems or damage to a company or customer. Different types of testing used in the software industry today have been presented together with seven general testing principles that have been used for the past forty years as general guidelines common for all testing.

The test process and its main activities have been described to show that there are more activities than the actual testing. Finally, you learned about the human mindset, how humans react in different situations working in test projects and how the quality may be affected due to this.

Last words

• Testing is necessary because a failure can be very expensive.
• Reasons for defects can be human fallibility, time pressure and complexity.
• Objectives of testing could be to, for example, find defects, provide information, prevent defects and build confidence.
• The seven testing principles are presence of defects, exhaustive testing, early testing, defect clustering, pesticide paradox, context dependency and absence-of-errors fallacy.
• Fundamental test process consist of following main activities; planning & control, analysis & design, implementation & execution, evaluating exit criteria & reporting and test closure.
• Important factors for successful testing are clear objectives, independence and good communication.

SYSTEM VERIFICATION

TERMS FROM CHAPTER 1

1.1 WHY IS TESTING NECESSARY

Bug
Defect
Error
Failure
Fault
Mistake
Quality
Risk

1.2 WHAT IS TESTING

Debugging
Requirement
Review
Test case
Testing
Test objective

1.3 SEVEN TESTING PRINCIPLES

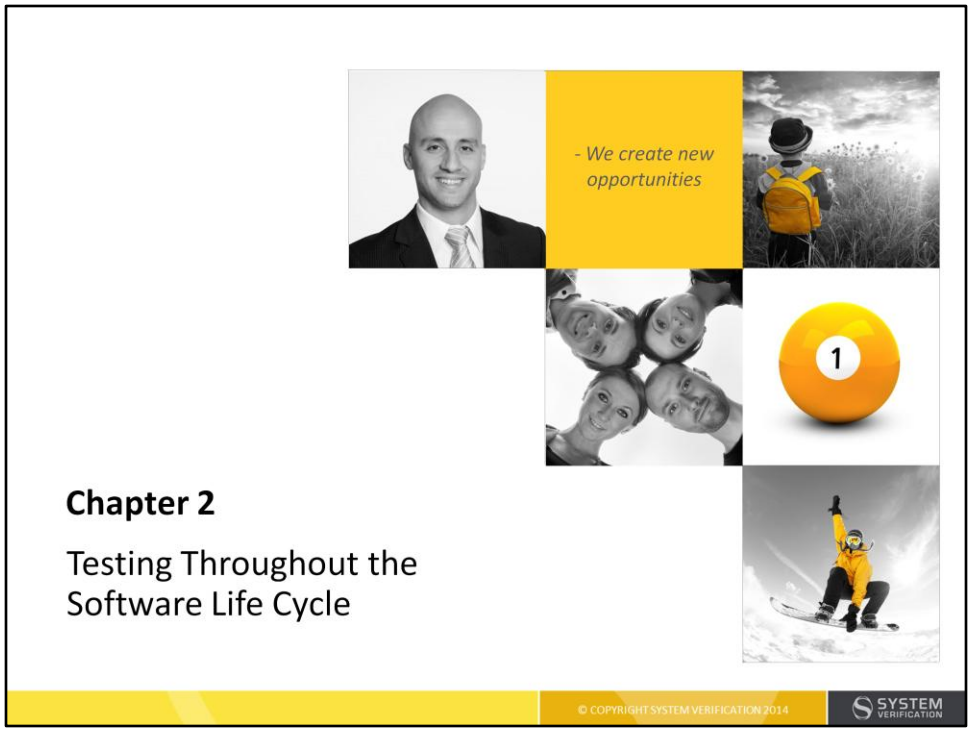
Exhaustive testing

1.4 FUNDAMENTAL TEST PROCESS

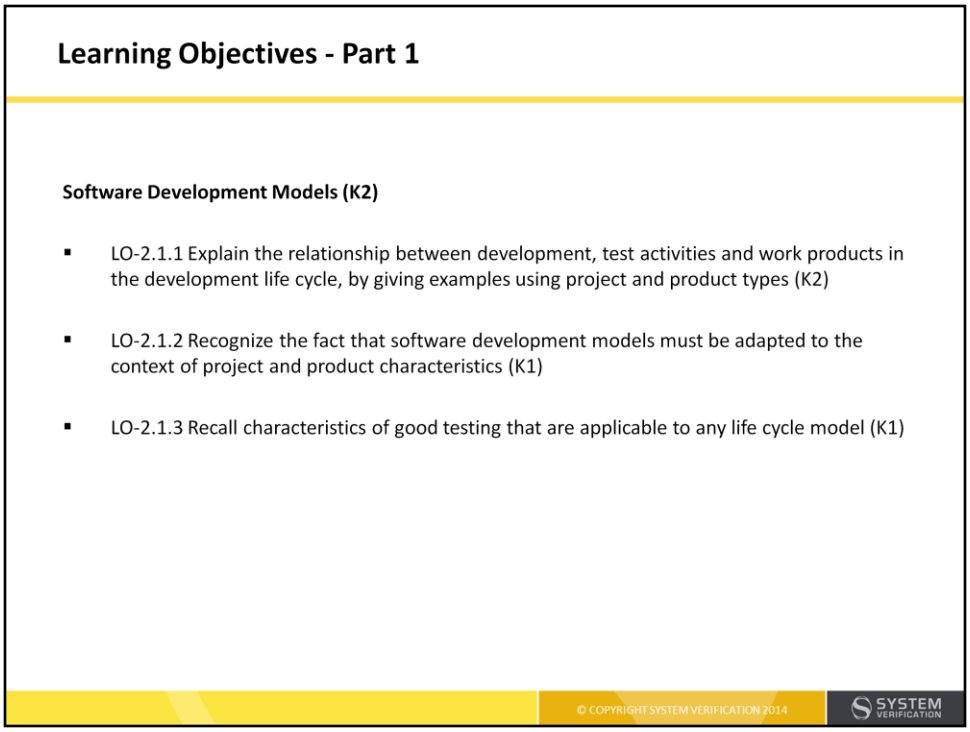
Confirmation testing
Re-testing
Exit criteria
Incident
Regression testing
Test basis
Test condition
Test coverage
Test data
Test execution
Test log
Test plan
Test procedure
Test policy
Test suite
Test summary report
Testware

1.5 THE PSYCHOLOGY OF TESTING

Error guessing
Independence



SYSTEM VERIFICATION

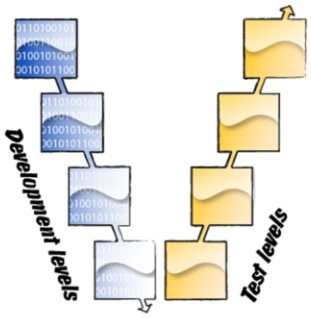


Time
Intended study time is 20 minutes

Terms
commercial off-the-shelf software(COTS), iterative-incremental development model, validation, verification, V-model

The V-Model

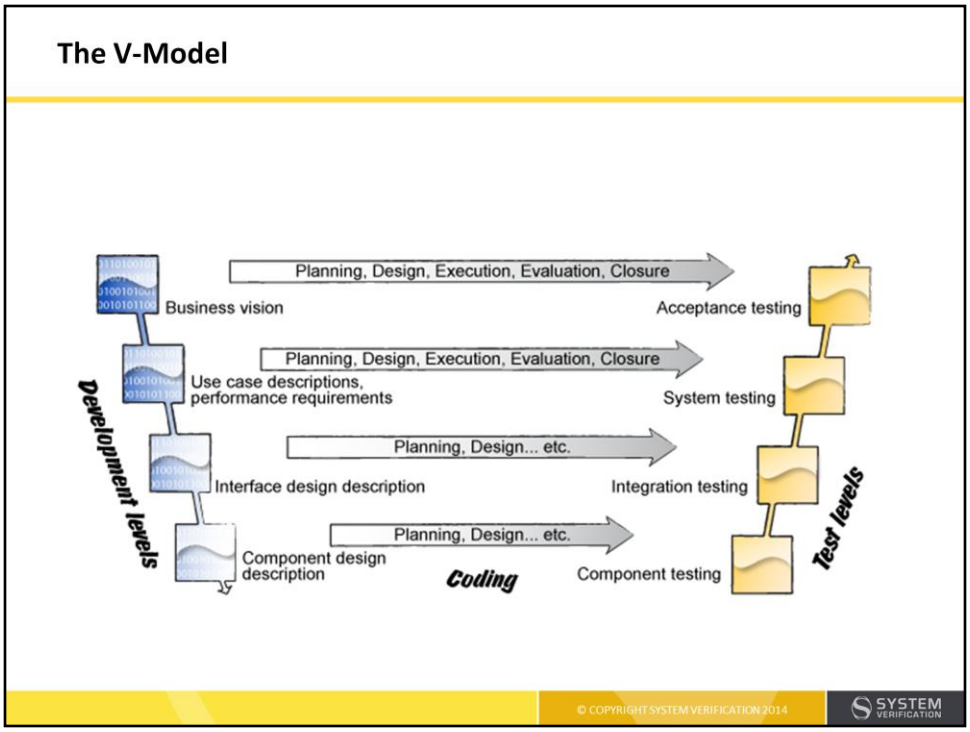
- Represents the development lifecycle
- Each development level has a matching test level
- Each development level produces work products used as basis for testing



The V-model is a graphical representation of the development life-cycle. The V-Model demonstrates the relationships between each level of the development life cycle and its associated level of testing.

The V-model normally consists of four development levels each with a matching test level. In practice the number of levels may vary depending on the project or software product.

Different work products produced during development are often the basis of testing in one or more test levels, e.g. requirement specifications, design documents and code.



Activities for a specific test level should begin as soon as the corresponding development level has resulted in test relevant documents and specifications. For example, the acceptance test planning should start as soon as the Business vision is available.

The Iterative-Incremental Model

- Examples:
  - Agile
  - Prototyping
  - Rapid Application Development
  - Rational Unified Process
- Short development cycles (iterations)
- Each iteration adds to the product, producing an increment
- Importance of regression testing

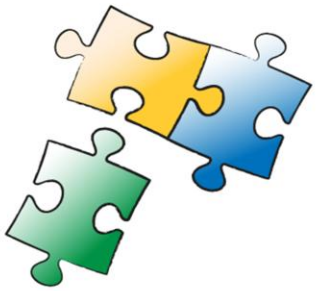
Examples of iterative-incremental models:
• Agile models
• Prototyping
• Rapid Application Development
• Rational Unified Process

These models are done as series of shorter development cycles, also known as iterations. Each iteration includes requirement, design, builds and test phases to deliver a new functional increment.

Verification and validation can be carried out on each of those increments. The resulting system delivered from a iteration may be tested on several test levels. Regression testing is increasingly important on all iterations after the first one.

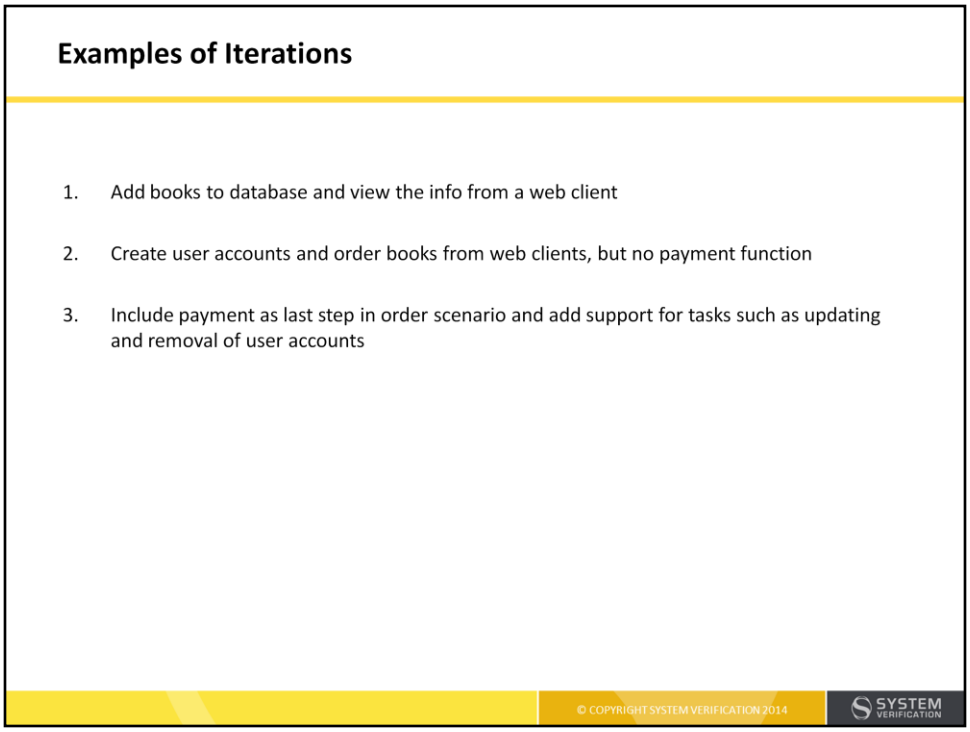
Example of Iterative-Incremental Testing

- Web application for selling books developed using the iterative development model
- What would possible iterations look like?



The following is an example of a web application from the development process that helps a publishing company reach worldwide consumers for sale of its books.

If we applied an iterative development model to this project we would want to divide the functionality into blocks that can be developed one-by-one. Each block would then be specified, designed, implemented and tested before the next one would start.



The goals for each iteration could look like this:

Iteration 1: Possibility to add books to database and view the info from a web client.

Iteration 2: Add possibility to create user accounts and order books from web clients. The number of books in stock are then decreased and a shipping the order created, but no payment needed.

Iteration 3: Include payment as last step in order scenario and add support for administrative tasks such as updating and removal of user accounts.

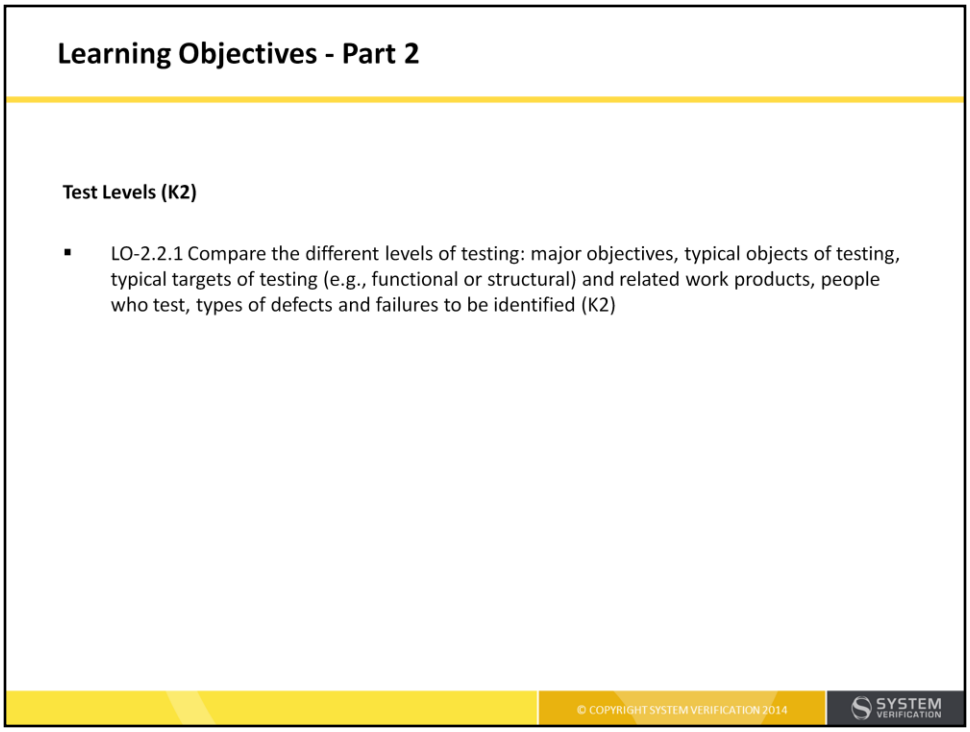
Testing within a Life Cycle Model

- All development activities have a corresponding testing activity
- Each test level has test objectives specific to that level
- The analysis and design of tests for a test level begins during the corresponding development level
- Testers should review development documents for each level as soon as drafts are available

Some general rules apply to good testing in any kind of software development model:

· All development activities have a corresponding testing activity
· Each test level has test objectives specific to that level
· The analysis and design of tests for a test level begins during the corresponding development level
· Testers should review development documents for each level as soon as drafts are available

Test levels can be combined or reorganized depending on the nature of the project or the system architecture.

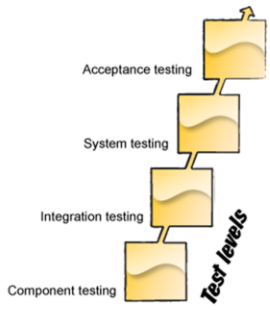


Time
Intended study time is 40 minutes

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

Component Testing

- Also module or unit testing
- Smallest separately testable part, e.g. object, class, page
- May include functional, non-functional and structural testing
- Based on component specification, design and data models



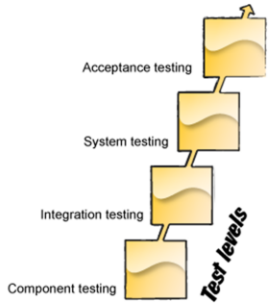
Component testing, also known as module or unit testing, is the process of searching for defects and verify functionality in the smallest separately testable part of a software system. A unit could for example be a software module, object or a class.

Component testing may include functional, non-functional (e.g. finding memory leaks) and structural testing (e.g. statement and decision coverage) and my be done separately from the system using stubs, simulators or drivers.

Component specification, design and data models are the base for the component test cases.

Component Testing

- Tester has access to code
- Using unit test framework, e.g. JUnit, or debugging tool
- Defects found and fixed early
- A special approach is Test-Driven Development (TDD) (development starts with creating test cases)

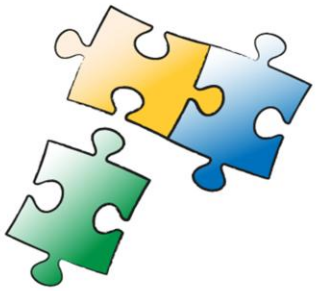


Component testing is usually performed with access to the code, using a unit test framework or a debugging tool. In practice it is the developer that performs unit test. As a result of this, defects are normally fixed directly when found.

One approach to component testing is This is called test-first approach or Test-Driven Development (TDD). It is based on cycles of developing component test cases, building and integrating pieces of code, executing the test cases and correcting any issues if necessary.

Example of Component Testing

- Test of database component
- Developer writes code to
  - insert data
  - verify output data

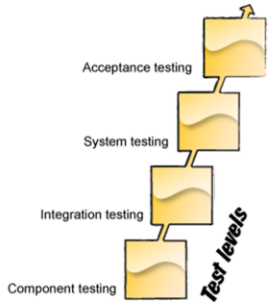


The following is an example from the development of a web application that helps a publishing company reach worldwide consumers for sale of its books.

The purpose is to test the database component which manages the storage of data. The Database developer writes code to insert data into the database and code for verifying the output data from the database. This helps the developer to automatically verify that input/output is correct.

Integration Testing

- Tests interfaces between components/ systems and interactions with different parts of a system
- Multiple levels of integration is possible
- Integration strategy should be based on system architecture



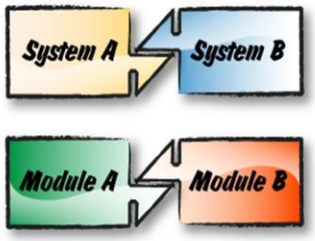
Integration testing aims at tests of interfaces between components and systems as well as interactions with different parts of a system, e.g. operating system, file system and hardware.

There may be more than one level of integration testing e.g. integration between components within a system or integration between two or more separate systems e.g. business process may involve a series of systems.

Integration strategies may be based on the system architecture (such as top-down and bottom-up), functional tasks or transaction processing sequences. If the integration scope gets to big it becomes difficult to isolate faults to a specific component or system and this may require additional time for troubleshooting.

Integration Testing

- Integration should rather be incremental than big bang
- Integration testing can be functional, non-functional and structural
- Knowledge of internal structure helps to achieve efficient testing



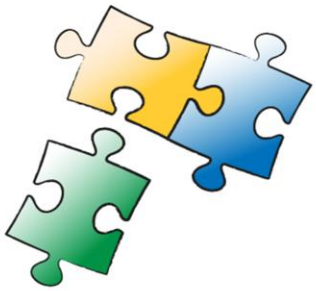
In order to reduce the risk of late defect discovery, integration should normally be incremental rather than "big bang". At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating module A with module B they are interested in testing the communication between the modules, not the functionality of either module.

Integration testing can be functional, non-functional and structural (white-box testing).

Ideally, for most efficient testing, testers should understand and influence the architecture to be able to plan integration test before components or systems are built.

Example of Integration Testing

- Test of multiple parts fitting together
- Skeleton code used to test database manager is replaced by actual code
- Inputs and outputs are verified through the web application

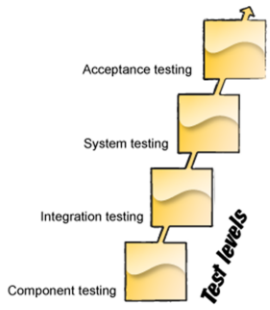


Integration testing may be carried out on test objects of varying size. Component integration testing tests the interaction between components and is done after component testing. System integration testing test the interaction between different systems or between hardware and software. This could mean that the development organisation may only control one side of the interface which might be considered as a risk.

In our web application example, we would like to test that parts of the system that have been developed independently fits together and have compatible interfaces. For example, the skeleton code used to test the database manager is replaced by the actual web server code and input/output is injected and verified through the web application instead.

System Testing

- Tests behaviour of entire system/product
- Realistic test environment important
- Often done by independent test team
- Can be tested using functional, non-functional and structural testing



System testing is concerned with the behaviour of a whole system or product. The scope of system testing shall be clearly documented in the test plan related to the system test level.

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

In System testing testers may also need to deal with incomplete or undocumented requirements to cover the system under test.

Often it is an independent test team that carries out testing on this level.

System Testing

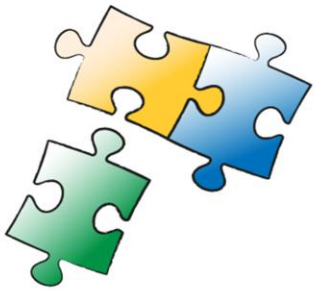
- Tests may be based on requirements specifications, business processes, use cases and product risks
- Its purpose is to investigate functional and non-functional behaviour. Structure-based techniques may also be used, e.g. for testing menu selections etc.

The tests may be based on requirement specifications, business processes, use cases or high level descriptions of system behaviour and interactions with e.g., the operating system or system resources. One other important input is the product risks.

The purpose of system testing is to investigate functional and non-functional behaviour. System testing of functional requirements start by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested Structure based techniques may also be used to evaluate the test coverage with respects to structural elements, e.g., drop-down menus.

Example of System Testing

- Test of complete system, i.e. web server, database, web clients, interface to shipping etc.
- System testing includes
  - functional tests (e.g. ordering a book)
  - non-functional tests (e.g. load, performance)



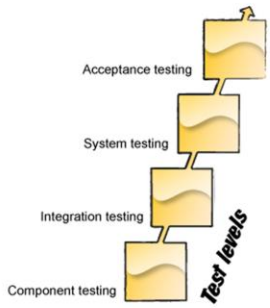
In our web application example, system test requires lots of preparations and planning to be efficient but it is also dependant on that testing on other test levels have been successfully. It takes time to set up a complete system with a web server, a database, web clients and interface to a shipping department if the preparations are not done properly it may be waste of time to continue with system test.

Functional tests are executed, such as ordering a book, including negative tests, for example, providing invalid login credentials.

Non-functional tests are also performed, such as load and performance tests with many simultaneous users.

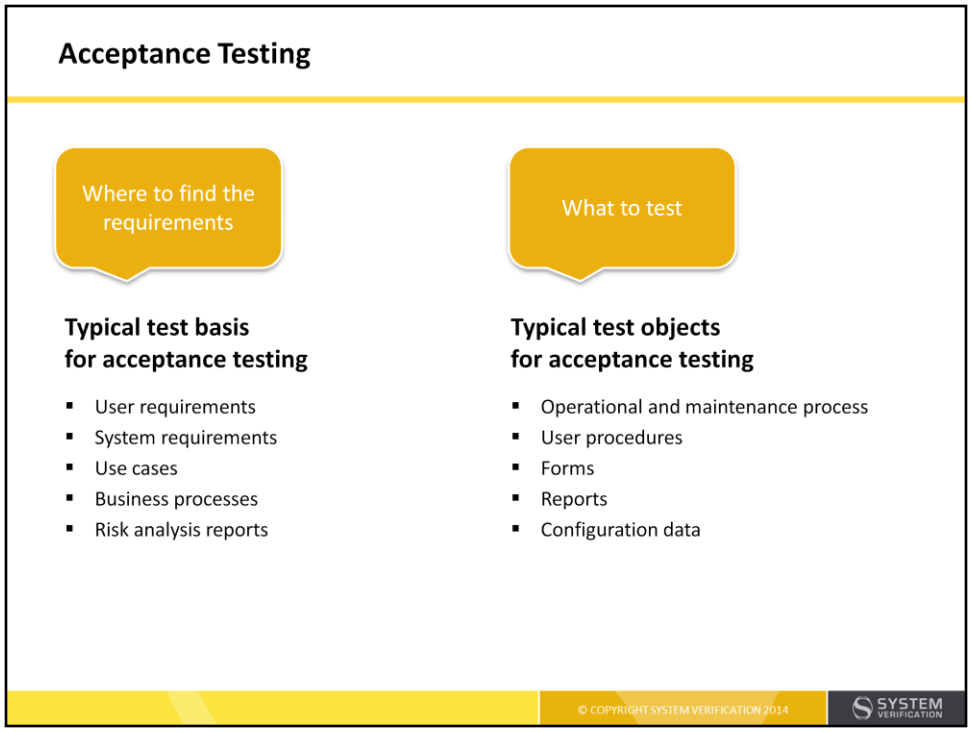
Acceptance Testing

- Goal is typically to establish confidence in the system rather than finding defects
- Should be carried out when most suitable in the life cycle
- Often the responsibility of customers or system users



The goal of acceptance testing is to establish confidence in the system, either the whole system or specific non-functional characteristics of the system. Finding defects is not the main focus. Acceptance testing may assess if the system is ready for deployment and use, although it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance test for a system.

Acceptance testing may be carried out at various times in the life cycle. Depending on purpose the testing can take place at all the different test levels. For example, acceptance testing of the usability of a component may be done during component testing. Acceptance testing is often the responsibility of the customers or system users.

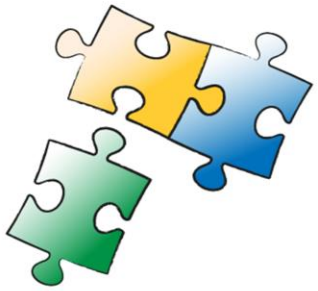


Typical test basis for acceptance testing are:
• User requirements
• System requirements
• Use cases
• Business processes
• Risk analysis reports

Test objects for acceptance testing are typically:
• Operational and maintenance process
• User procedures
• Forms
• Reports
• Configuration data

Example of Acceptance Testing

- Testing in which stakeholders are involved and execute typical scenarios and use cases.
- The goal is to verify fitness for use and ensure that stakeholders accept the delivery.
- User acceptance testing is a typical form of acceptance test. The fitness of the system is verified by real business users.



When the system is ready for delivery customers, users and other stakeholders are gathered to assure that they are all satisfied and have full confidence in the system.

An event is set up where typical scenarios and use cases are executed. All stakeholders participate and overview the testing to verify the fitness for use and hopefully accept the delivery.

User acceptance testing is a typical form of acceptance test. This to verify the fitness of the system by real business users.

Operational Acceptance Testing

- Done by system administrators
- Examples of areas to test:
  - User management
  - Back up and restore
  - Disaster recovery
  - Data load and migration tasks
  - Maintenance tasks
  - Periodic checks of security vulnerabilities

Operational acceptance testing is performed by system administrators. Things that could be tested are:
• User management
• Back up and restore
• Disaster recovery
• Data load and migration tasks
• Maintenance tasks
• Periodic checks of security vulnerabilities

Contract and Regulation Acceptance

- Contract acceptance testing
  is done against a contract's acceptance criteria for custom-developed software
- Regulation acceptance testing
  is done against any regulations that must be adhered to, e.g.
  - governmental regulations
  - legal regulations
  - safety regulations

Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software.

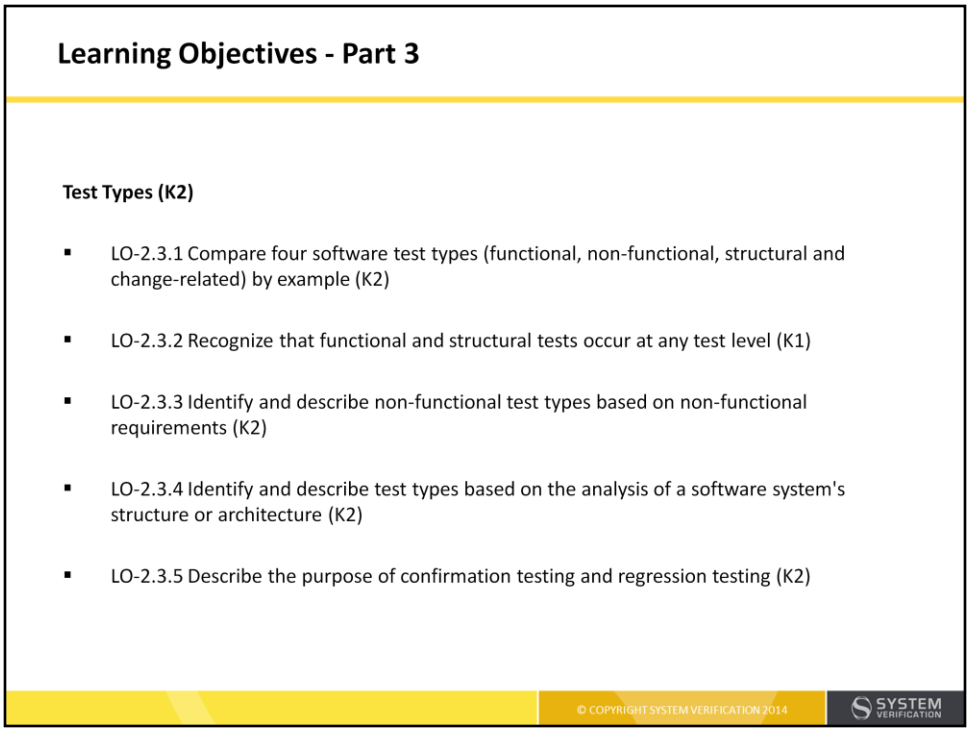
Regulation acceptance testing is performed against any regulations that must be adhered to, such as governmental, legal or safety regulations.

Alpha and Beta Testing

- Typically used by COTS (Commercial-off-the-Shelf) developers to get feedback from potential users.
- Alpha testing is performed at the developers' site.
- Beta testing is carried out by users at their own sites, e.g. by letting users download software and try out at home.

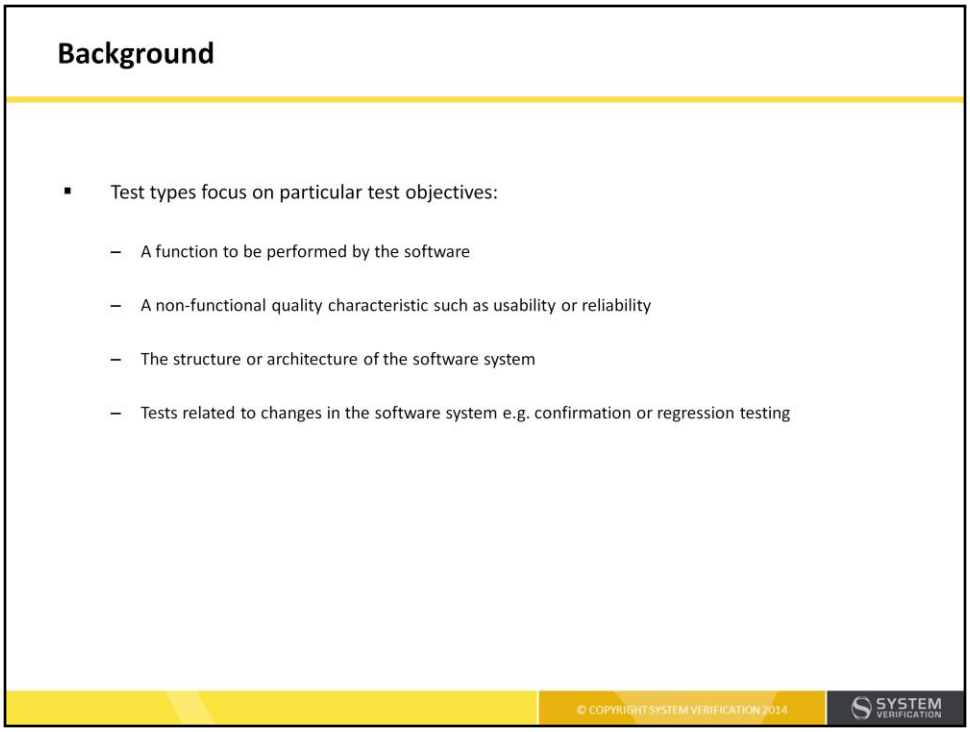
Developers of market, or commercial-off-the-shelf (COTS), software often want to get feedback from potential or existing customers in their market before the software product is put up for sale commercially.

Alpha testing is performed at the developing organization's site. Beta testing or field testing, is performed by people at their own locations. Both are tested by customers or potential customers, not the developers of the product.



Time
Intended study time is 40 minutes

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

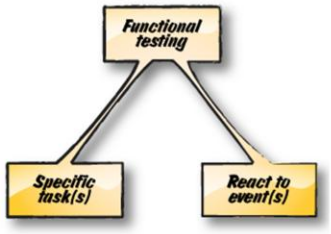


A set of activities aims to verify a software system or a part of the system based on a specific reason or target for testing. This differs from a test type that focus on a particular test objective. Typical test objectives could be:
• A function to be performed by the software
• A non-functional quality characteristic such as usability or reliability
• The structure or architecture of the software system
• Tests related to changes in the software system e.g. confirmation or regression testing.

A model of the software system could be developed and used in structural testing, non-functional testing and functional testing.

Testing of Function

- "What" the system does
- Functions and features and their interoperability with specific systems
- Looks at behaviour by a typical user (black box testing)
- Functions may be described in requirements, use cases, a functional specification, or not at all



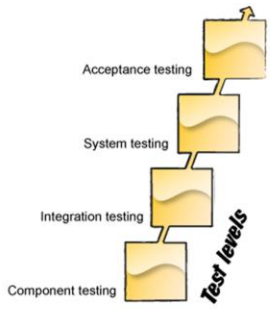
The functions are "what" the system does. Functional tests are based on functions and features and their interoperability with specific systems.

Functional testing considers how the software behaves during normal usage for a user i.e. black box testing.

The functions that a system, subsystem or a component are to perform may be described in a work product such as a requirement specification, use case, or a functional specification or even be undocumented.

Testing of Function

- Can be performed on all levels
- Specification-based techniques can also be used to derive test conditions and test cases



Functional tests can be performed on all test levels, for example, component tests are based on component specifications, while system tests are based on system specifications.

Specification-based techniques may be used to derive test conditions and test cases from the functionality of the software or system.

Examples of Software Characteristics for Functional Testing

- **Suitability** evaluates the capability to provide functions for specified tasks and user objectives
- **Accuracy** evaluates the capability to provide correct results or effects with the required degree of precision
- **Security** evaluates the capability to protect information and data from access by unauthorized persons or systems
- **Interoperability** evaluates the capability to interact with one or more specified components or systems

A type of functional testing, security testing, investigates the functions relating to detection of threats (e.g. a firewall), such as viruses from malicious outsiders.

Another type of functional testing, interoperability testing, evaluates the capability of the software product to interact with one or more specified components or systems.

Testing of Non-functional Software Characteristics

- ”How” the system works
- Can be done on all test levels
- Usually performed using black box testing
- Examples:
  - Reliability
  - Usability
  - Efficiency
  - Maintainability
  - Portability

Non-functional testing is when we test the software characteristics, i.e. "how" the system works. These test are broader and could include:
• Reliability
• Usability
• Efficiency
• Maintainability
• Portability

Non-functional testing may be performed at all test levels. It considers the external behaviour and the most common test design technique is black-box testing.

Example of Non-Functional Testing

- Inflicting heavy load or stress conditions in order to measure response times, availability, error handling, robustness etc.
- Various aspects of non-functional testing are described in the quality model ISO 9126.

These tests are also clearly quantifiable. For questions like: "How will the system perform as transactions increases?" we can e.g. measure the response times. Stress testing is an example of non-functional testing. It refers to tests that put a greater emphasis on robustness, availability, and error handling under a heavy load, rather than on what would be considered correct behaviour under normal circumstances.

A web server may be stress tested using scripts, bots and various denial of service tools to observe the performance of a web site during peak loads.

Test can be performed to a quality model such the one defined in "Software Engineering-Software Product Quality (ISO 9126).

Testing of Software Structure/ Architecture

- Referred to as white box testing
- Measures thoroughness of testing by investigating code coverage
- If coverage is less than 100 %, more testing might be needed to increase coverage

Structural testing is when we target the overall software architecture. It is referred to as white-box testing.

Structural testing helps measure the thoroughness of testing by investigating the coverage of a type of structure. It means studying the system and asking: "How much of this structure has been tested?"

If coverage is not 100% it may be desirable to design more test cases to increase coverage.

Testing of Software Structure/ Architecture

- Can be used on all levels
- Checking statements, decisions and paths
- Not always practical (or even possible) on a higher test level
- Tools can be used to measure coverage, e.g. statements and decisions

Structural testing can be used on all test levels. At the component level it means checking if all statements have been executed, or if all decision paths have been followed. On a higher test level it becomes more difficult to cover every statement and decision, and in some cases impossible.

At the integration level, it means checking how modules call other modules, and if all calls have been tested.

Tools can be used to measure, for example, statements and decisions.

Testing Related to Changes

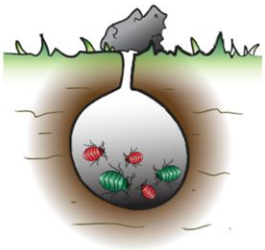
- Retesting (confirmation testing) secures that defects have been corrected properly
- Regression testing secures that old functionality still works after additions or changes have been done

Other kinds of test types are applied when change occurs. When a defect is uncovered and developers have fixed it, re-testing or confirmation testing shall be carried out to secure that it has been fixed properly.

When code has been changed in some way, for example, new functionality have been introduced or hardware has been updated, regression testing shall be carried out to secure that old functionality still works properly.

Re-Testing

- Testing uncovers a defect in the software. It's located and corrected (debugging), which is a development activity
- After fixing, re-testing should be done to verify that the defect has been removed
- Tests must be repeatable



When testing uncovers a defect in software, a developer is assigned to fix it. The action of locating and fixing the defect is called debugging. It is a development activity, not a testing activity.

When the developer has fixed the bug, the software should be retested exactly as before, to confirm that the defect has been successfully removed. This is called re-testing.

Tests must be repeatable or re-testing is not possible. If they are repeatable they could also assist regression testing.

An example of re-testing
A financial institution is building a system for electronic trade of financial instruments. By law, the system must complete each transaction under a given number of milliseconds. The first round of testing shows that the system would not pass these requirements.

After a fix has been applied, the tests are redone exactly as before. Speed is now within regulatory limits, but some transactions fail completely. Re-testing continues until the tests pass satisfactorily.

Regression Testing

- Modifications are done to a software system. They are followed by a regression test
- Its purpose is to verify that no new defects have been introduced in related (or unrelated) areas of the product
- Regression test suites are good candidates for test automation

Regression testing is performed after a change or modification to already tested system. The purpose of regression tests is to repeat testing of an already tested program to confirm that no new defects have been introduced or revealed in related or unrelated software areas.

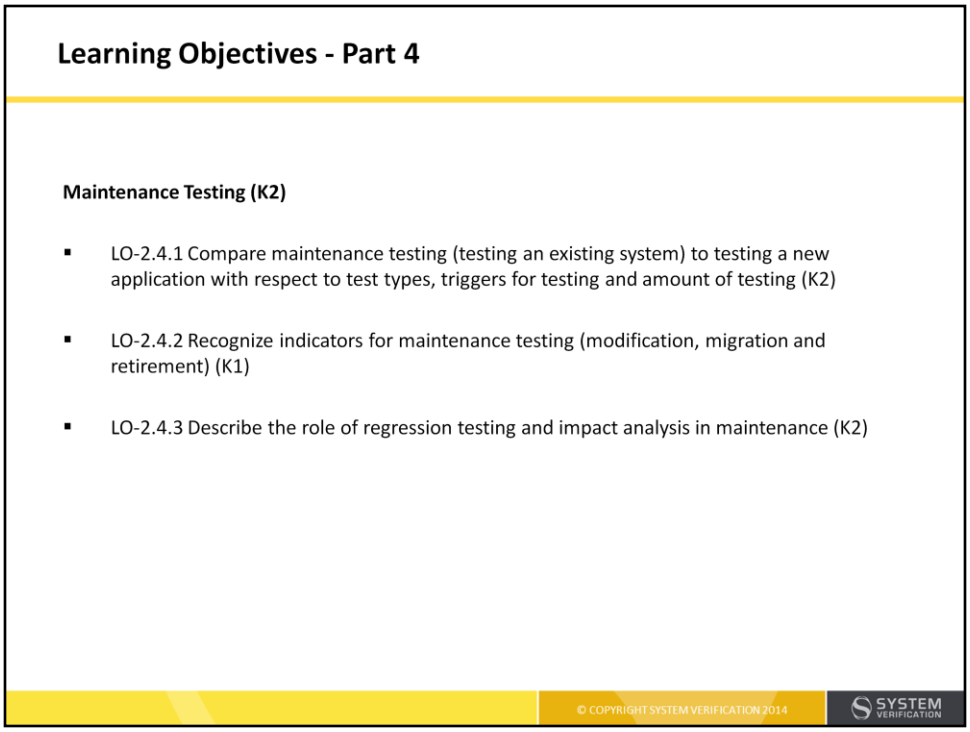
The extent of regression testing is based on the risk not finding any defects in software working previously. It can performed on all test levels and include all the test types mentioned earlier.

Regression test suites are run many times and generally evolve slowly. Regression testing is therefore a candidate for automation.

An example of regression testing
A development team is building an updated version of a widely used GPS navigation device. The new version includes new features, such as anti-theft measures and tracking of the module from a web page.

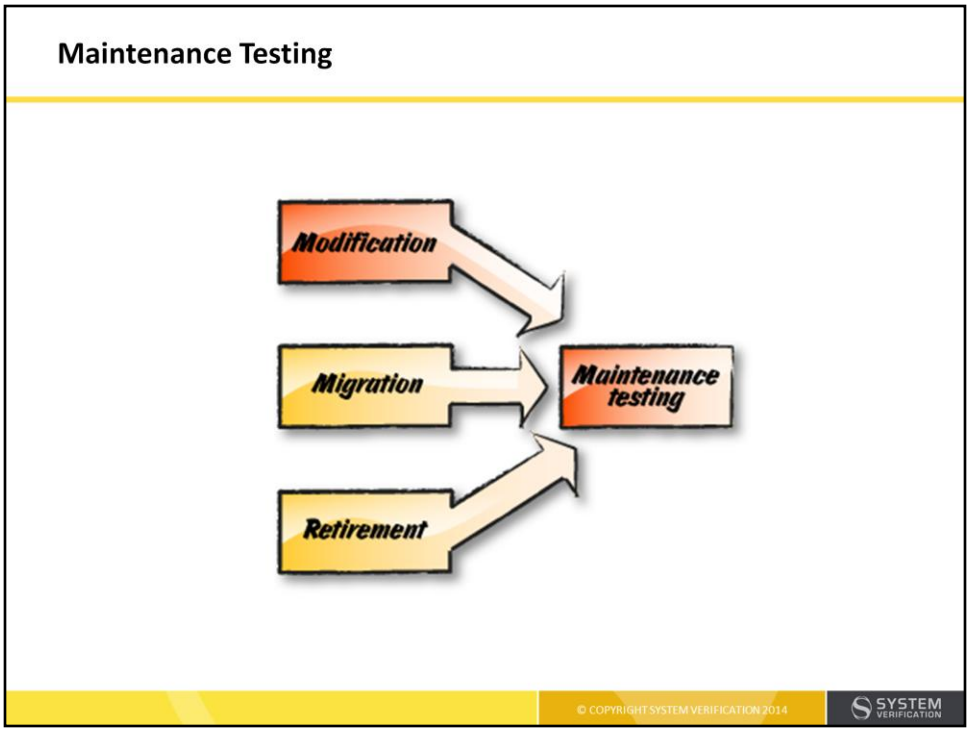
After new code has been implemented, the system is regression tested on the unit level by developers and on other levels by a quality assurance team to ensure that old functionality still functions properly.

What and how to perform regression test is decided by risk evaluation to avoid too much time spent in regression tests. Basic flows such as standard navigation, voice information and charging is selected to verify old important functionality.



Time
Intended study time is 15 minutes

Terms
impact analysis, maintenance testing



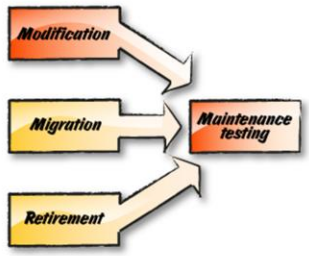
Software system are often in operation for a very long time. During its life cycle the software systems, its configuration data or its environment are often corrected, changed or extended. Maintenance testing is trigger by modifications, migrations or retirement of the software system.

It is necessary to distinguish between planned releases and hot fixes because planning of releases in advance is crucial for successful for maintenance testing.

Maintenance testing can be difficult if specifications are out of date or missing, or persons with domain knowledge are not available.

Modification

- Planned enhancements
- Corrective
- Emergency
- Environmental (e.g. database upgrades)
- Security patches

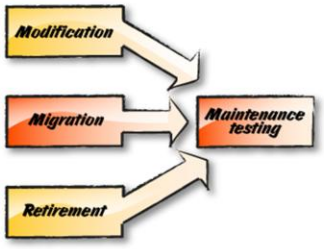


Modifications can include a number of different changes, for example:
• planned enhancements
• corrective
• emergency
• environmental (e.g. database upgrades)

It can also be patches to correct newly exposed or discovered vulnerabilities of the operating system.

Migration

- Moving software from one platform to another
- Moving data from one application into the system

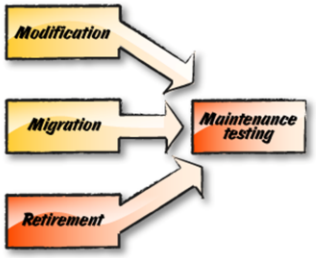


Migration could for example be moving software from one platform to another and should include operational tests of the new environment and the changed software.

Migration testing is also necessary when moving data from one application into the system being maintained.

Retirement

- Data migration
- Archiving
- Restoring



Retirement of a system may include testing of data migration or archiving, if long data-retention periods are required.

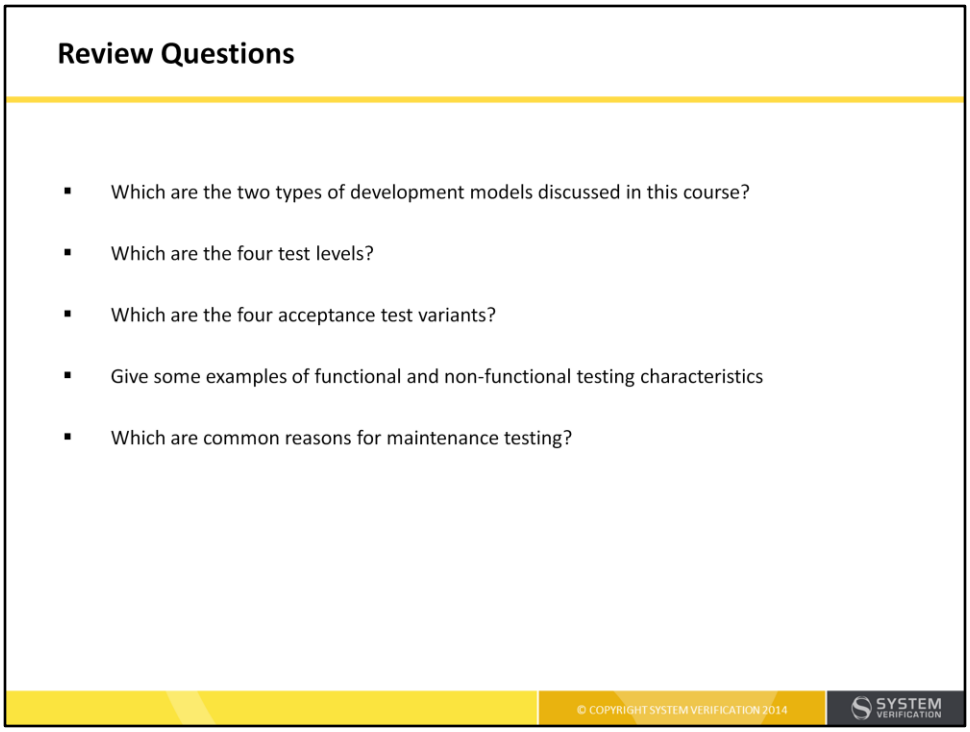
Impact Analysis

- Determining how the existing system may be affected by changes is called **impact analysis**
- Used to decide how much testing is needed
- The scope of testing is related to the
  - risk of the change
  - size of the change
  - size of the system

Determining how the existing system may be affected by changes is called impact analysis and is used to help decide how much testing is needed.

The scope of maintenance testing is related to the risk of the change, the size of the existing system and to the size of the change.

Depending on the changes, maintenance testing may be done at any or all test levels and for any or all test types.



Summary
In this chapter you have learnt about the testing process and when testing should be used within the development life cycle. Different test levels have been presented, e.g. component and system test levels, including the approaches used and responsibilities.

Groups of test activities, referred to as test types, that can be aimed of verifying the software based on a specific reason or target for testing have been presented. It includes functional, non-functional and structural tests. Other examples are re-test and regression test, which is performed when some change to a software has been carried out.

Changes that are done on an already deployed software system calls for testing as well. This is known as maintenance testing and different types have been presented.

Last words
• V-model and iterative model are two example of development models.
• Component, integration, system and acceptance are four different test levels
• User, operational, contract/regulation and alpha/beta testing are four different acceptance test types.
• Example of functional testing characteristics are suitability, accuracy, security and interoperability.
• Examples of non-functional testing characteristics are reliability, usability, efficiency, maintainability and portability.
• Modifications, migration and retirement are three reasons for maintenance testing.

SYSTEM VERIFICATION

TERMS FROM CHAPTER 2

2.1 SOFTWARE DEVELOPMENT MODELS

Validation
Verification
V-model
Commercial Off-The-Shelf (COTS)
Iterative-incremental development model

2.2 TEST LEVELS

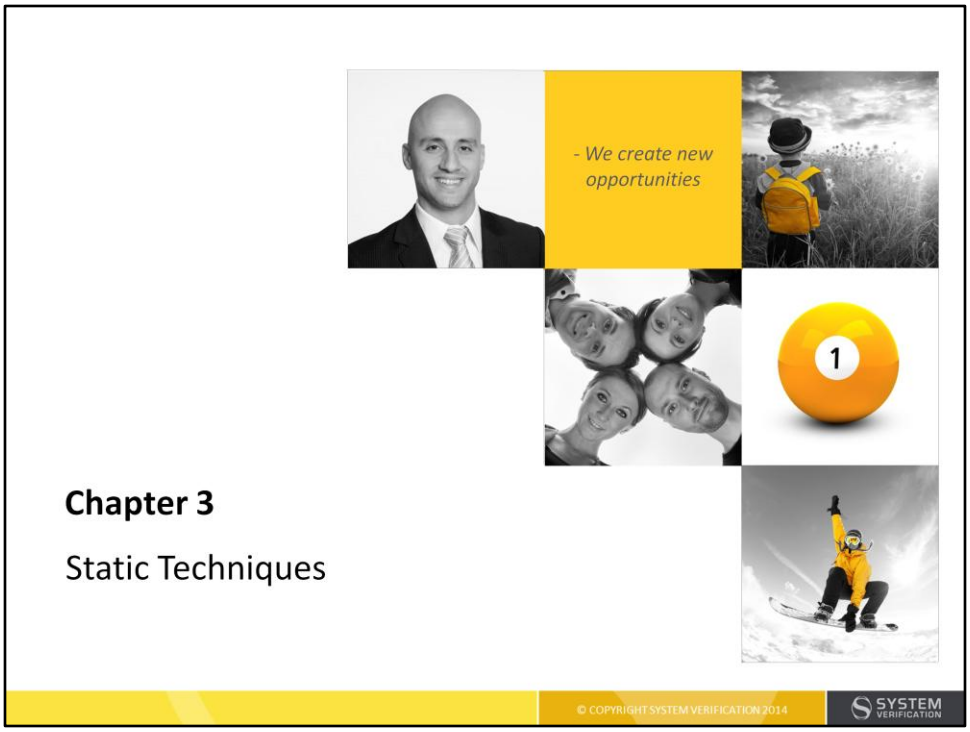
Alpha testing
Beta testing
Component testing
Driver
Field testing
Functional requirement
Integration
Integration testing
Non-functional requirement
Robustness testing
Stub
System testing
Test environment
Test level
Test-driven development
User acceptance testing

2.3 TEST TYPES

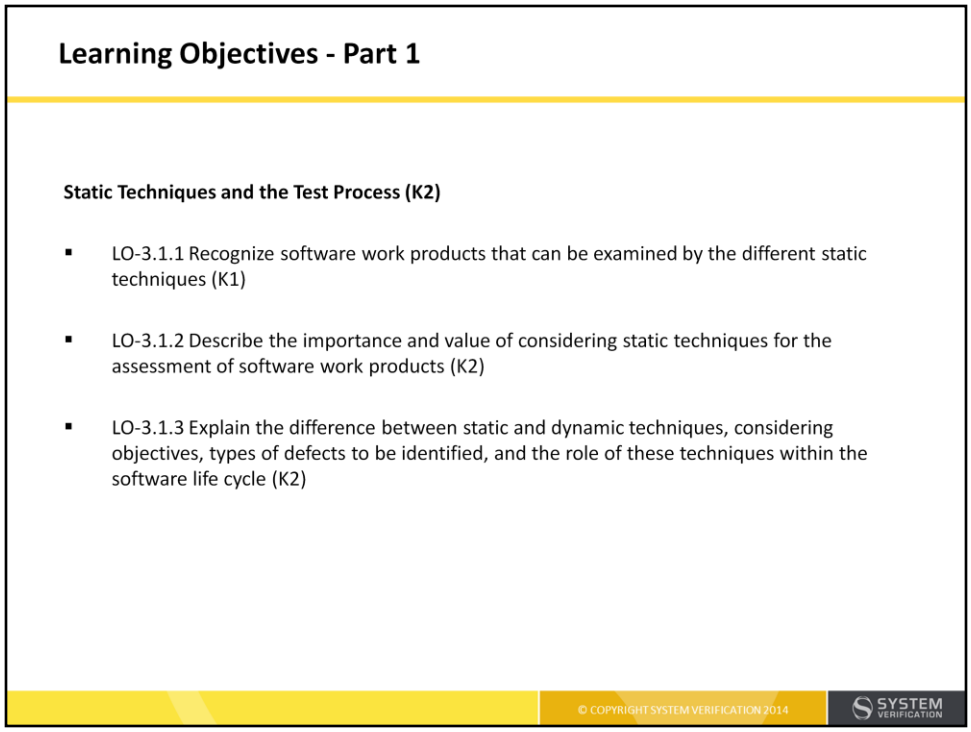
Black-box testing
Code coverage
Functional testing
Interoperability testing
Load testing
Maintainability testing
Performance testing
Portability testing
Reliability testing
Security testing
Stress testing
Structural testing
Usability testing
White-box testing

2.4 MAINTENANCE TESTING

Impact analysis
Maintenance testing



SYSTEM VERIFICATION

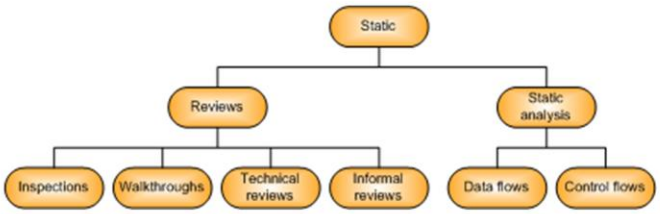


Time
Intended study time is 15 minutes

Terms
dynamic testing, static testing

Methods of Static Testing

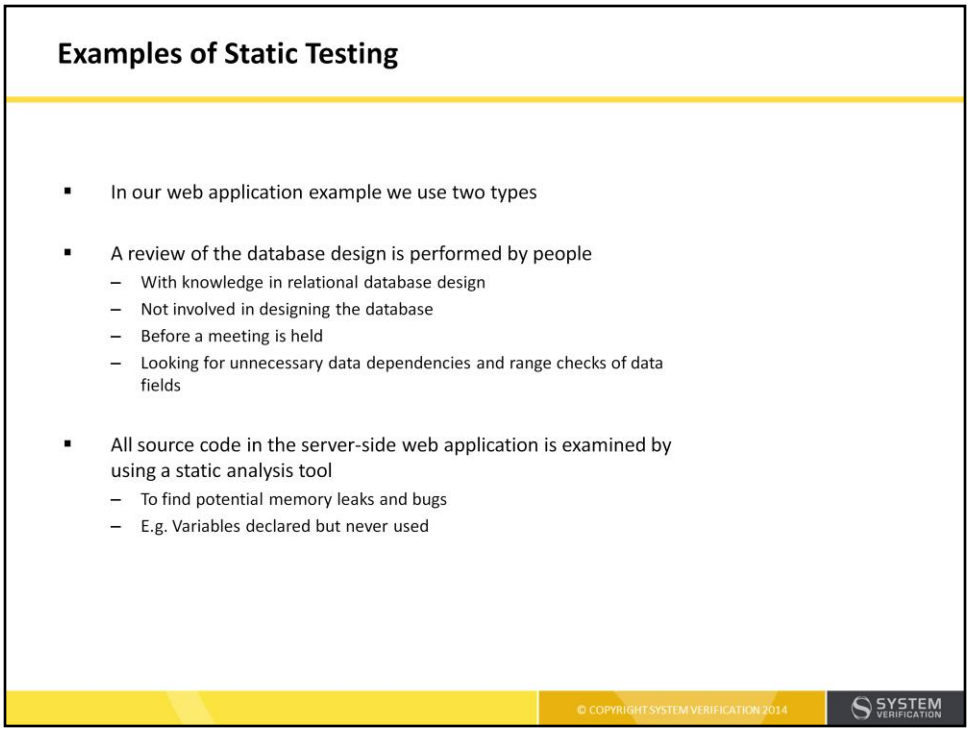
- Manual examination of documents and code
- No execution of the code
- Prevent causes of failures during software design
- Two different methods
  - Reviews, examination of software design documents
  - Static analysis, examination of source code



Static testing is about manual examination of documents and code. It does not require any execution of the code unlike dynamic testing.

It is performed to prevent causes of failures during software design. This technique can be divided into two different methods, i.e. reviews and static analysis.

Reviews focus at examination of software design documents while static analysis focus on examination of source code.



In our web application example we use two types of static testing:
• A review of the database design is performed. It is done by people with knowledge in relational database design, but not directly involved in designing the database. Reviewers are instructed to review the database before a meeting is held and look for unnecessary data dependencies and range checks of data fields.
• Al l source code in the server-side web application is examined by using a static analysis tool. The intention is to find potential memory leaks and bugs, for example variables that are declared but never used.

Review

- Reading and presentation technique
- Examine and evaluate work products to prevent defects
- Everything from project specifications to actual source code can be reviewed
- Start at an early stage before coding
- It is cheaper to solve defects found during review compared to defects found later in the development life cycle.

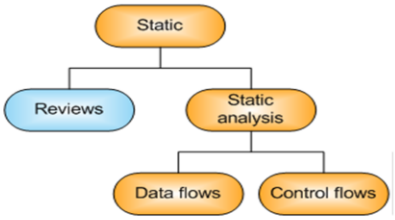
Reviewing is a reading and presentation technique where the main activity is to examine and evaluate work products to prevent defects.

Everything from project specifications to actual source code can be reviewed. This means that reviewing can start at an early stage, before coding.

Defects found during reviews may be cheaper to solve then defects found later in the development life cycle.

Static Analysis

- Checking the source code for defects
- Can be done with a tool, e.g. by using a program compiler
- Check the sequence of events (paths) during execution
  - This is called control flow
- It can also check the state of variables and objects in various states
  - This is called data flow



Static analysis means checking the source code for defects. This can be done with a tool, for example by using a program compiler.

Static analysis can check the sequence of events (paths) that would occur during execution. This is called control flow. It can also check the state of variables and objects in various states. This is called data flow.

Static vs. Dynamic Testing

- Reviews, static analysis and dynamic testing have the same objective
  - Identifying defects
- They are complementary
  - They find different types of defects effectively and efficiently
- It is generally better than dynamic testing at finding the causes of failures
  - E.g. in requirements, unlikely to be found in dynamic testing

Reviews, static analysis and dynamic testing have the same objective, i.e. identifying defects. They are complementary, different techniques can find different types of defects effectively and efficiently.

Compared to dynamic testing, static testing is generally better at finding the causes of failures. Reviews can find omissions in, for example requirements, which are unlikely to be found in dynamic testing

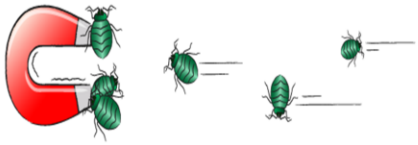
Benefits of Static Testing

- Early defect detection and correction
- Development productivity improvements
- Reduced testing time and cost
- Fewer defects and improved communication

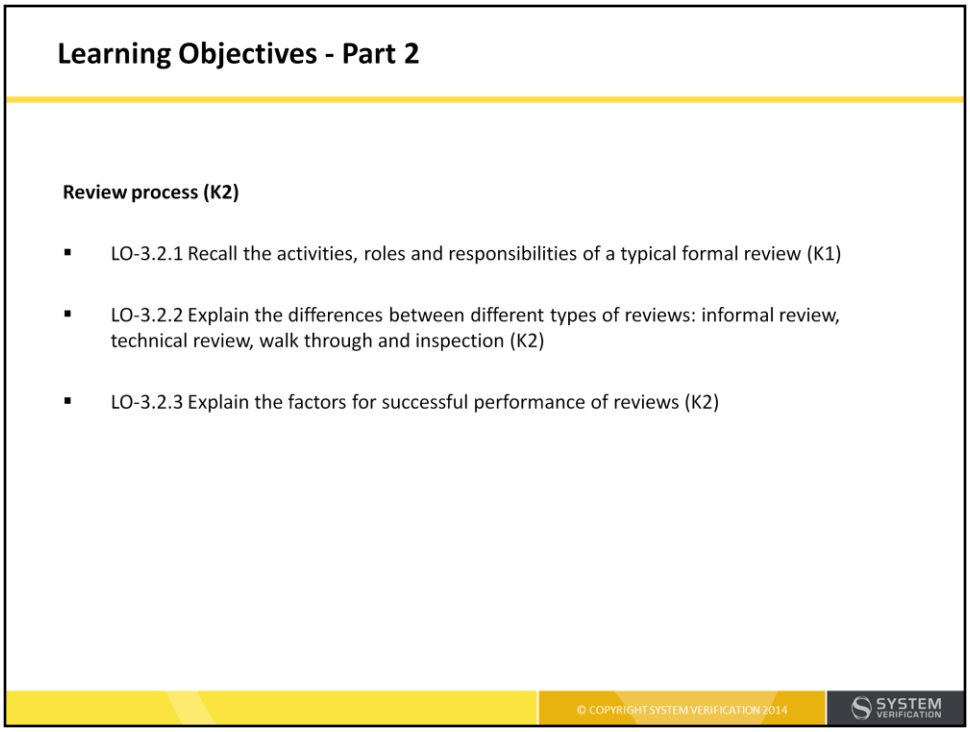
Benefits of static testing include, for example, early defect detection and correction, development productivity improvements, reduced testing time and cost, fewer defects and improved communication.

Typical Defects found in Static Testing

- Deviations from standards
- Requirement defects
- Design defects
- Insufficient maintainability
- Incorrect interface specifications



Typical defects that are easier to find in static testing than in dynamic testing are, for example, deviations from standards, requirement defects, design defects, insufficient maintainability and incorrect interface specifications.



Time
Intended study time is 25 minutes

Terms
entry criteria, formal review, informal review, inspection, metric, moderator, peer review, reviewer, scribe, technical review, walkthrough

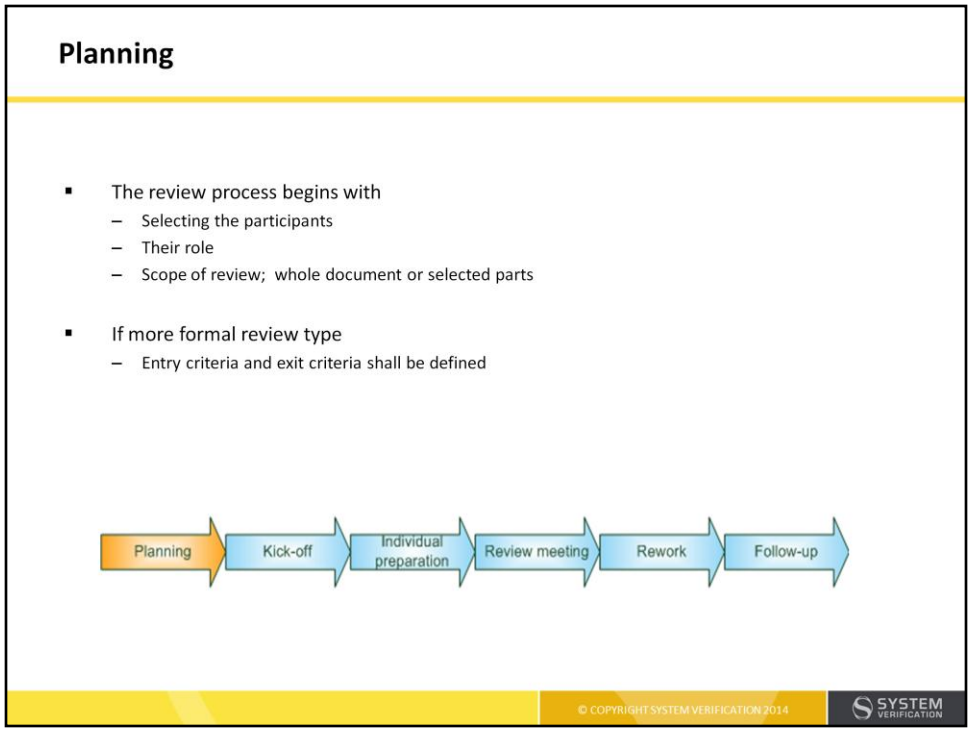
Review

- All artefacts for a product can be reviewed, for example, requirement specifications, design specifications, test plans and code
- Vary from informal to systematic based on factors such as
  - Maturity of the development process
  - Legal or regulatory requirements
  - The need for an audit trail
- Objectives
  - To find defects
  - Gain understanding
  - Educate team members
  - Discussion and decision by consensus

All artefacts for a product can be reviewed, for example, requirement specifications, design specifications, test plans and code.

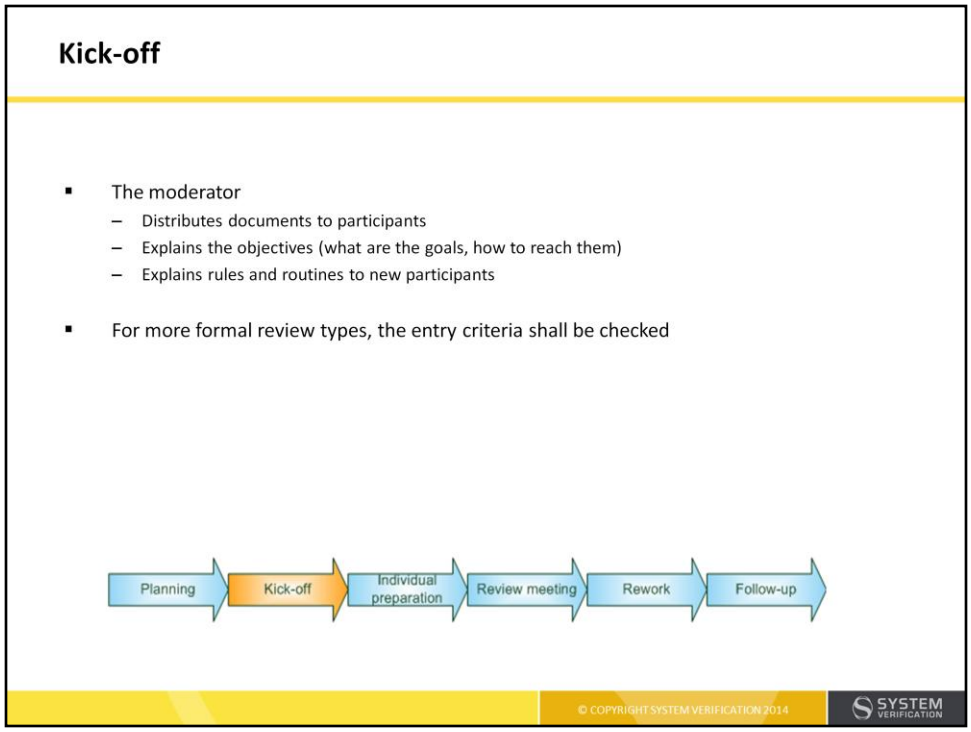
Reviews can vary from informal to systematic based on factors such as the maturity of the development process, legal or regulatory requirements or the need for an audit trail.

Often the objective of a review would be to find defects but it could also be gain understanding, educate team members, or discussion and decision by consensus.



The review process begins with selecting the participants, deciding their role and if they should review the whole document or selected parts.

If more formal review types will be performed, entry criteria and exit criteria shall be defined.



During kick-off, the moderator distributes documents to participants and explains the objectives. It may be necessary to educate inexperienced reviewers in rules, routines and in the review process.

Decisions about the goals of the inspection should be clarified. Strategies on how to reach these goals should also be decided.

For more formal review types, the entry criteria shall be checked.

Individual Preparation

- Inspects code and documents
- Writes down potential defects in a log
- Questions and comments are also written down
- Submitted to the inspection moderator
  - Before the review session
  - Copies are also sent to the author.
- Important that the participants get enough time to review and prepare before the review meeting

During preparation, each participant inspects code and documents, and writes down potential defects in a log. Questions and comments are also written down. All defects, questions and comments is then submitted to the inspection moderator before the review session. Copies are also sent to the author.

It is very important that the participants get enough time to review and prepare before the review meeting.

Example of items from a requirements review checklist:
• Have appropriate requirements documentation standards been followed?
• Does the specification include all known customer or system needs?
• Have all dependencies on other systems been identified?
• Are all security requirements properly specified?
• Is each requirement written in a clear, consistent language?
• Is each requirement testable?

Review Meeting

- Discuss defects, comments and questions
- The author answers questions and makes clarifications
  - New defects may be revealed
  - Recommendation on how to handle the defects
  - Decisions about the defects

Defects, comments and questions are discussed during a review meeting. The author answers questions from reviewers and makes necessary clarifications.

During the meeting new defects may be revealed, which should be documented as well. Except from noting the defects, making recommendation on how to handle the defects and making decisions about the defects is also done.

Rework

- Fixing the defects found during the review meeting
  - Performed by the author
- Fixed defects status is submitted to the moderator for follow-up

This activity entails fixing the defects found during the review meeting, typically performed by the author. When defects have been fixed the updated defect status is submitted to the moderator for follow-up.

Follow-up

- Documents from the rework activity are received by the moderator
- The moderator checks that the defects have been addressed
- Moderator decides if a new review is needed
- For more formal review types exit criteria shall be evaluated
- Metrics could also be gathered
  - Number of defects found
  - Preparation time

Documents from the rework activity are received by the moderator. The moderator checks that the defects from the review meeting have been addressed.

Then moderator decides if a new review is needed after the changes have been made. For more formal review types exit criteria shall be evaluated.

Different metrics could also be gathered, for example, number of defects found and preparation time.

Roles and Responsibilities

- A typical formal review include these roles:
  - Manager: Responsible for the review
  - Moderator: Leads the review
  - Author: Creator of the review documents
  - Reviewers: Reviews the documents
  - Scribe: Documents issues and problems during review meeting

SYSTEM VERIFICATION

Manager Role

- Makes decision to the execute reviews
- For formal review process, determines if the review has met its objectives
- Allocates time and resources to the review process
  - According to project schedules

The person who has final say on the execution of reviews. In a formal review process, this person will determine if the review has met its objectives.

The manager also allocates time and resources to the review process, according to project schedules.

Moderator Role

- Tasks include planning the review, running the meeting and follow-up
- Usually experienced, impartial team members
  - Cannot be the author
- Mediator between the various points of view
  - Often the person upon whom the success of the review rests

A moderator's tasks include planning the review, running the meeting and follow-up after the meeting.

Moderators are usually experienced, impartial team members i.e. they cannot be the author of documents under review.

The moderator can act as a mediator between the various points of view and is often the person upon whom the success of the review rests.

Author Role

- The writer or person with chief responsibility for the documents to be reviewed

Reviewer Role

- Specific technical or business background
  - Identify and describe findings (e.g. defects)
- Represent different perspectives and roles
- Have time to prepare for each meeting
- Also called checkers or inspectors

People with a specific technical or business background who identify and describe findings (e.g. defects) in the product under review.

Reviewers should be chosen to represent different perspectives and roles in the review process, and should have time to prepare for each meeting. Reviewers are also called checkers or inspectors.

Scribe Role

- Documents all the issues, problems and open points that were identified during the meeting

Types of Review

- Inspection is the most formal type of review
  - Requires preparation
  - Led by a trained moderator
- Technical review are also moderator-led
  - Aim to discuss, evaluate options and solve problems
- A walkthrough is led by the author
  - Aims to gain understanding of the documents
- An informal review is the least formal review
  - No formal process
  - Inexpensive way to get some benefit

Inspection is the most formal type of review. It requires preparation and is led by a trained moderator

Technical review are also moderator-led and aim to discuss, evaluate options and solve problems

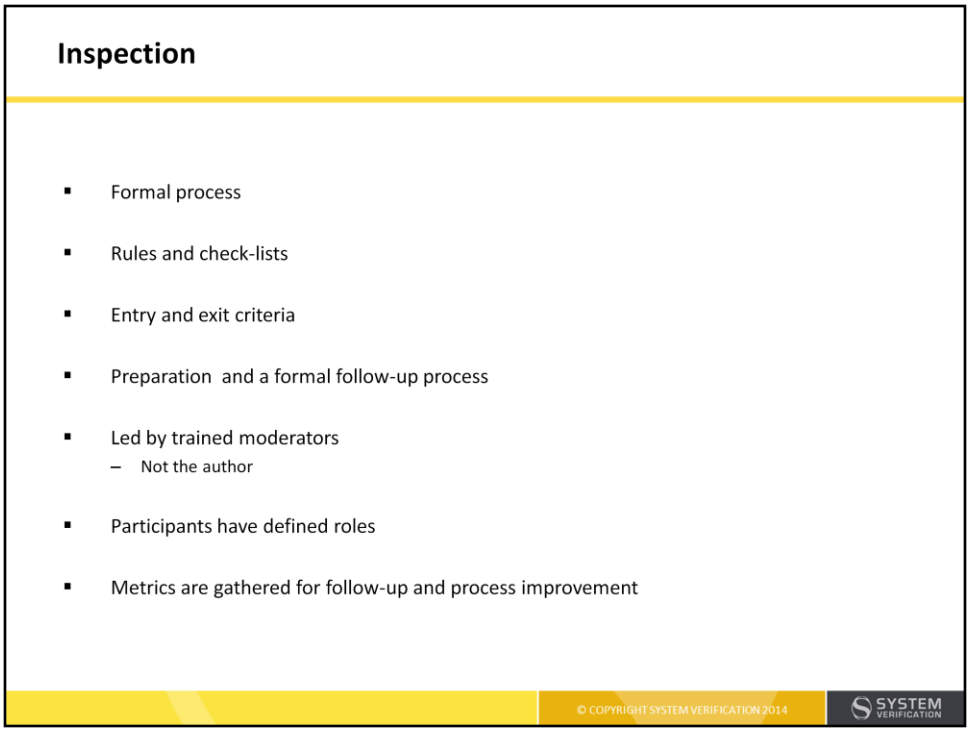
A walkthrough is led by the author and aims to gain understanding of the documents

An informal review is the least formal review. It has no formal process but it is an inexpensive way to get some benefit

Peer Review

- Walkthroughs, technical reviews and inspections can be performed within a peer group
  - Colleagues at the same organizational level
  - This type of review is called a "peer review"

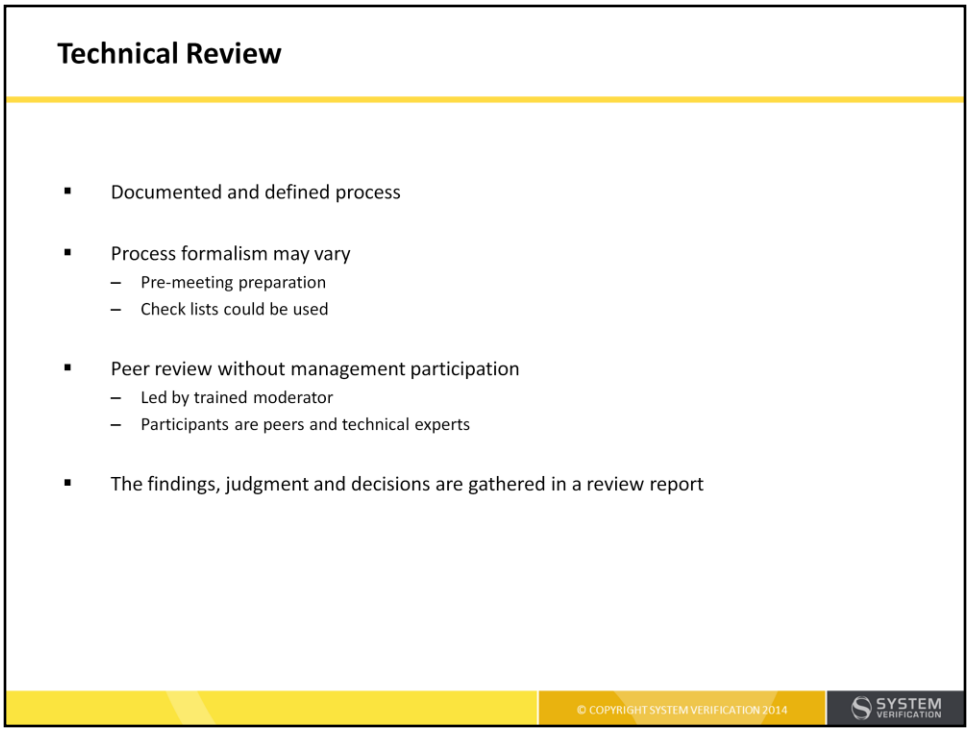
SYSTEM VERIFICATION



Inspections use a formal process based on rules and check-lists with entry and exit criteria and usually include pre-meeting preparation as well as a formal follow-up process. Entry and exit criteria shall be defined for acceptance of the software product.

They are led by trained moderators who is not the author of the documents under review. All members on the review have defined roles.

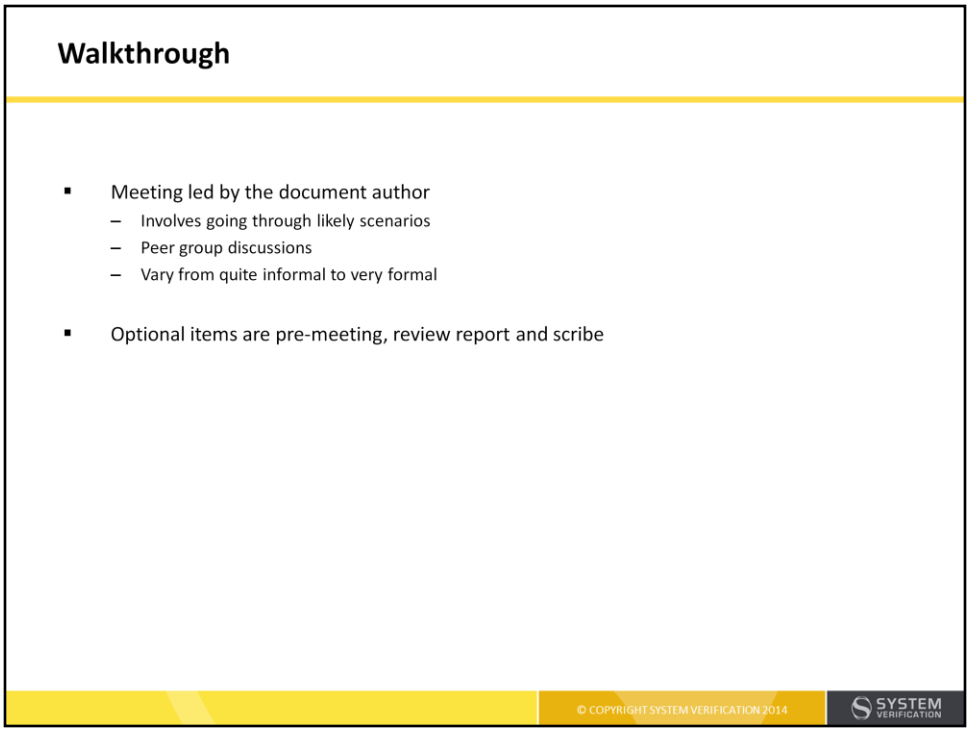
Metrics are gathered for follow-up and process improvement.



The technical review is a documented and defined defect-detection process. Process formalism may vary but include pre-meeting preparation by reviewers. Check lists could be used to support the review.

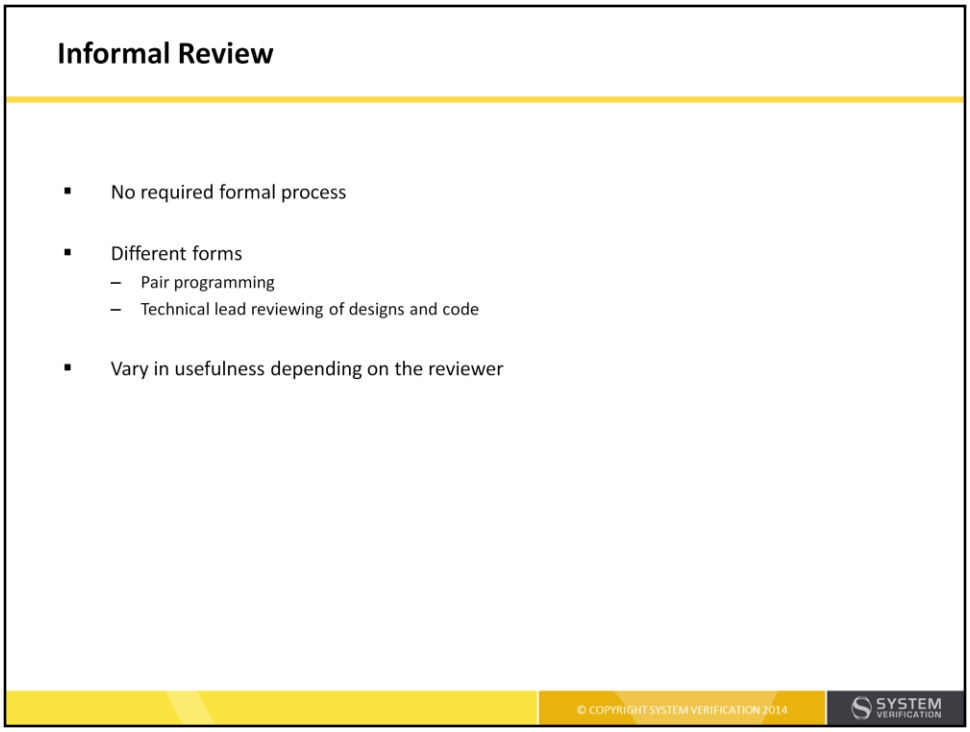
It may be performed as a peer review without management participation. It is ideally led by trained moderator who is not the document or software author. The participants are peers and technical experts.

The findings together with the verdict whether the software product meets its requirements is gathered in a review report.



A walkthrough is a meeting led by the document author. It involves going through likely scenarios and is an impartial peer group discussion. It may vary from quite informal to very formal.

Optional items are pre-meeting, review report and scribe (who is not the author).



Informal reviews have, as the name implies, no required formal process. It may take the form of pair programming or a technical lead reviewing of designs and code with documentation of the results as an option. Informal reviews vary in usefulness depending on the reviewer.

Success Factors for Reviews

- Clear objectives
- The right people
- Testers are valued reviewers
  - Early test preparations
  - Comments can be collected in advance
- Defects found are welcomed and expressed objectively
- People issues and psychological aspects are dealt with
  - Positive experience for the author
- Atmosphere of trust
  - Outcome not used for evaluation of the participants

SYSTEM VERIFICATION

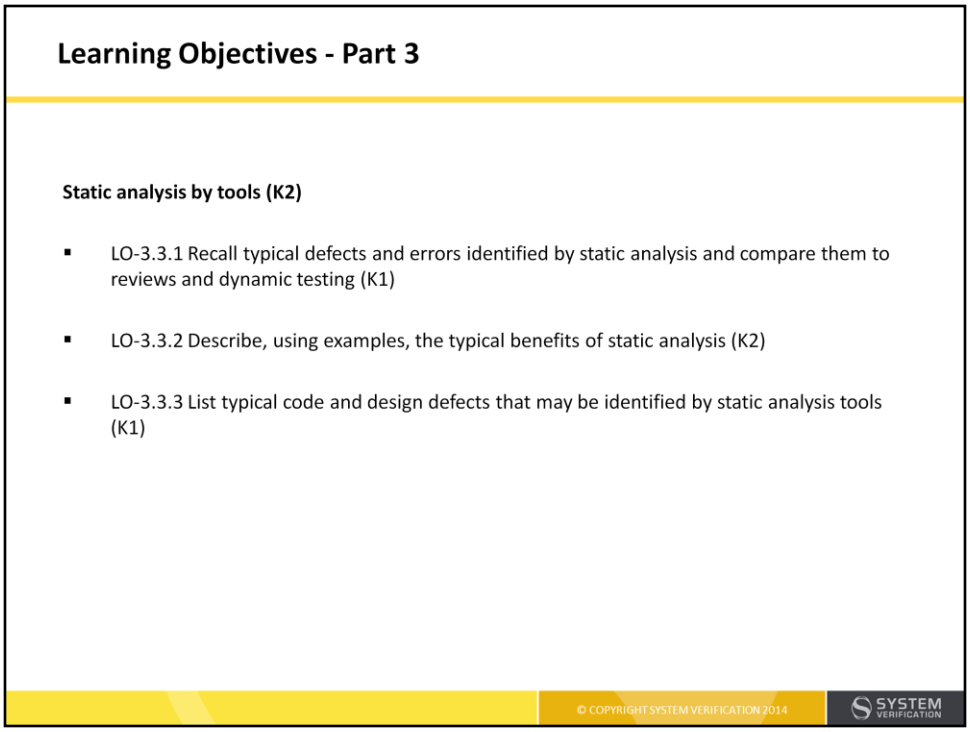
• Clearly predefined objectives
• The right people for the review are involved
• Testers are valued reviewers that also learn about the product which enables early test preparations
• Defects found are welcomed and expressed objectively
• People issues and psychological aspects are dealt with (making it a positive experience for the author)
• The review is conducted in an atmosphere of trust, i.e. the outcome will not be used for evaluation of the participants

Success Factors for Reviews (continued)

- Techniques are applied that are suitable to achieve the objectives
- Check-lists or roles to increase effectiveness of defect identification
  - Assign responsibilities for rework and follow- up
- Training is given in formal review techniques, e.g. inspection
- Management support
- Emphasis on learning and process improvement

SYSTEM VERIFICATION

• Review techniques are applied that are suitable to achieve the objectives
• Check-lists or roles are used if appropriate to increase effectiveness of defect identification
• Training is given in review techniques, especially the more formal techniques, such as inspection
• Management support (e.g. by incorporating adequate time for review activities in project schedules)
• Emphasis on learning and process improvement

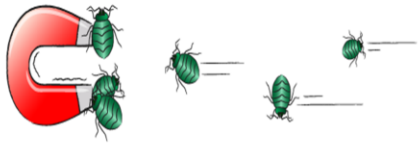


Time
Intended study time is 20 minutes

Terms
compiler, complexity, control flow, data flow, static analysis

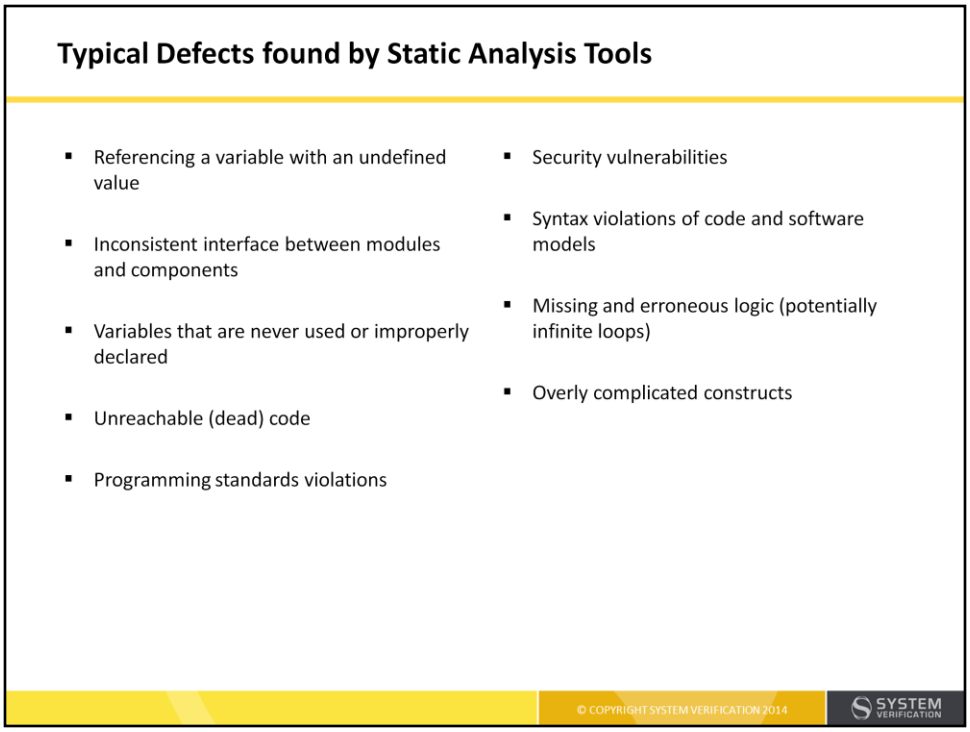
The Objective of Static Analysis

- The main objective to find defects in software source code and models
- Performed without executing the software
  - Finds defects rather than failures
- Find defects that are hard to find in dynamic testing
- Static analysis tools analyse
  - Program code, e.g. control flow and data flow
  - Generated output e.g. HTML and XML



The main objective of static analysis is to find defects in software source code and models. It is performed without executing the software being examined compared to dynamic testing that executes the software code, i.e. static analysis finds defects rather than failures.

The advantage of static analysis is that it can locate defects that are hard to find in dynamic testing. Static analysis tools analyse program code (control flow and data flow) as well as generated output (HTML and XML).

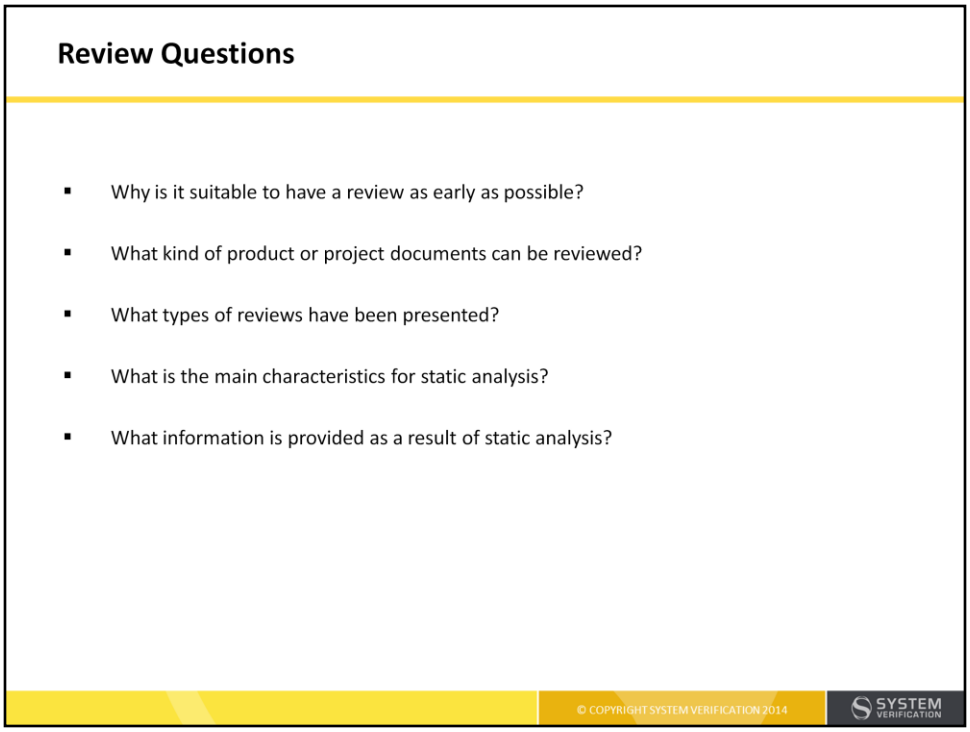


Typical defects discovered by static analysis tools include:
• Referencing a variable with an undefined value
• Inconsistent interface between modules and components
• Variables that are never used or improperly declared
• Unreachable (dead) code
• Programming standards violations
• Security vulnerabilities
• Syntax violations of code and software models
• Missing and erroneous logic (potentially infinite loops)
• Overly complicated constructs

Benefits of Static Analysis Tools

- Enables early detection of defects prior to execution
- Early warning about suspicious aspects of the code or design
  - Calculation of metrics
  - High complexity measure
- Dependencies and inconsistencies in software models
- Improve maintainability of code and design

SYSTEM VERIFICATION



Summary

In this chapter you have learnt more about static testing, e.g. reviews and static analysis, and what differs them from dynamic testing. Different review types have been presented, the process of conducting them and the benefits. Roles and responsibilities in the review process have been covered as well. Tool support for static analysis have been discussed and a summary of the typical types of defects that can be found.

Last words

- A review shall be done as early as possible to save as much money as possible.
- Any product or project related document could be reviewed e.g. development and test documents.
- Informal, walk-through, technical review and inspections are all different types of reviews.
- Main characteristics for static analysis is that the code is not executed.
- Static analysis could provide information on defects, quality of the code and code metrics.

TERMS FROM CHAPTER 3

3.1 STATIC TECHNIQUES AND THE TEST PROCESS

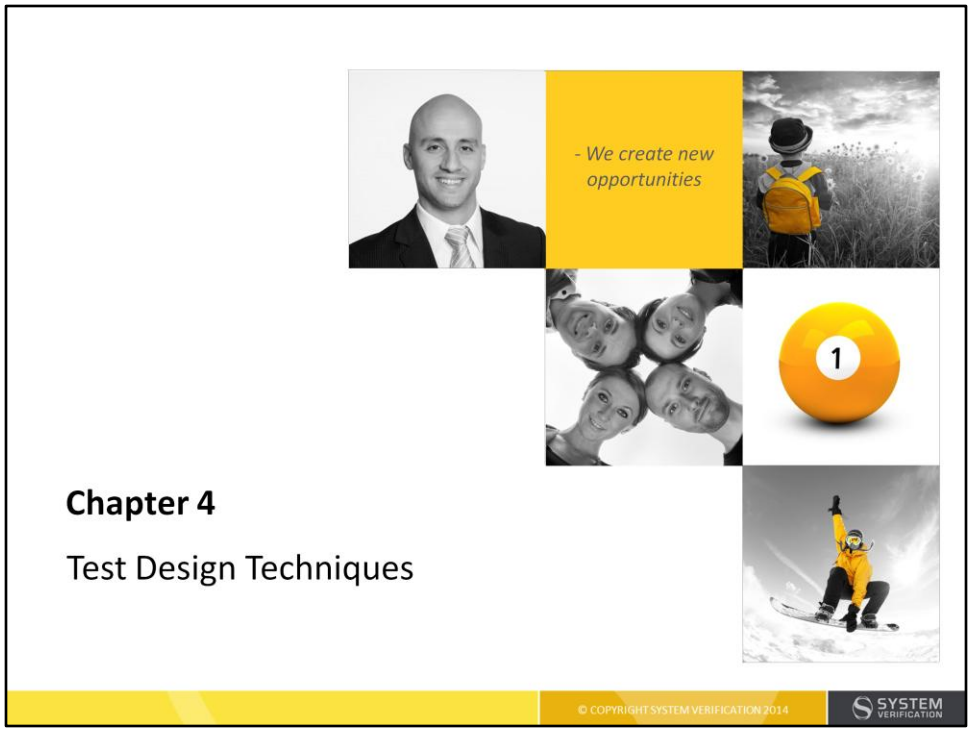
Dynamic testing
Static testing

3.2 REVIEW PROCESS

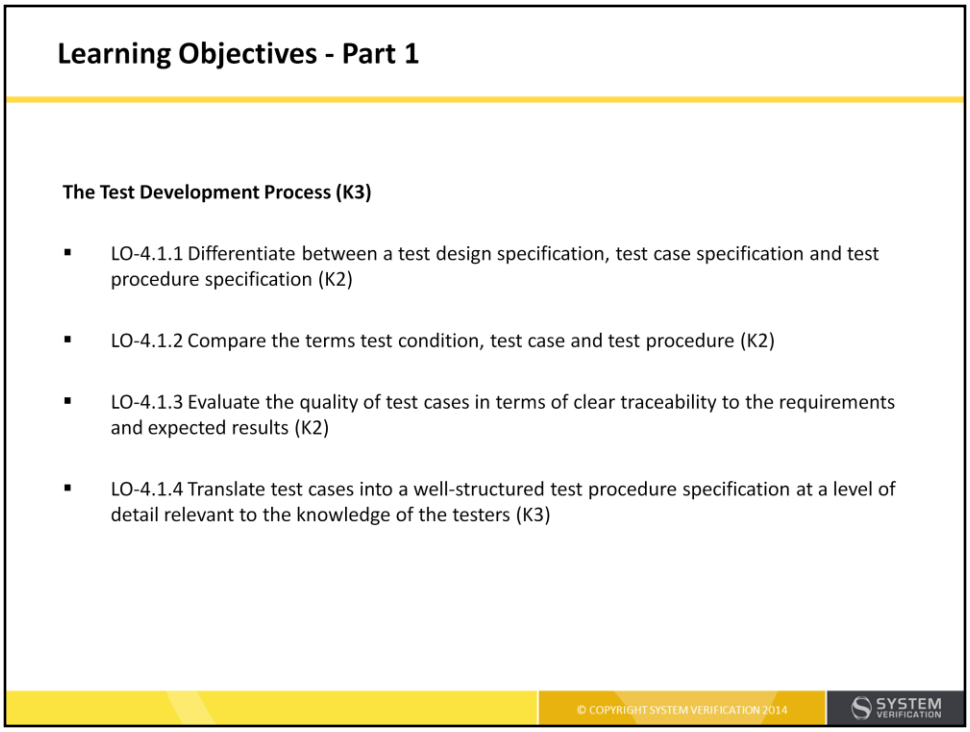
Entry criteria
Formal review
Informal review
Inspection
Metric
Moderator
Peer review
Reviewer
Scribe
Technical review
Walkthrough

3.3 STATIC ANALYSIS BY TOOLS

Compiler
Complexity
Control flow
Data flow
Static analysis



SYSTEM VERIFICATION

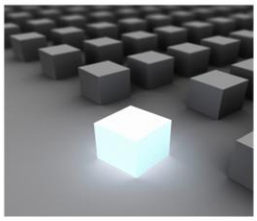


Time
Intended study time is 15 minutes

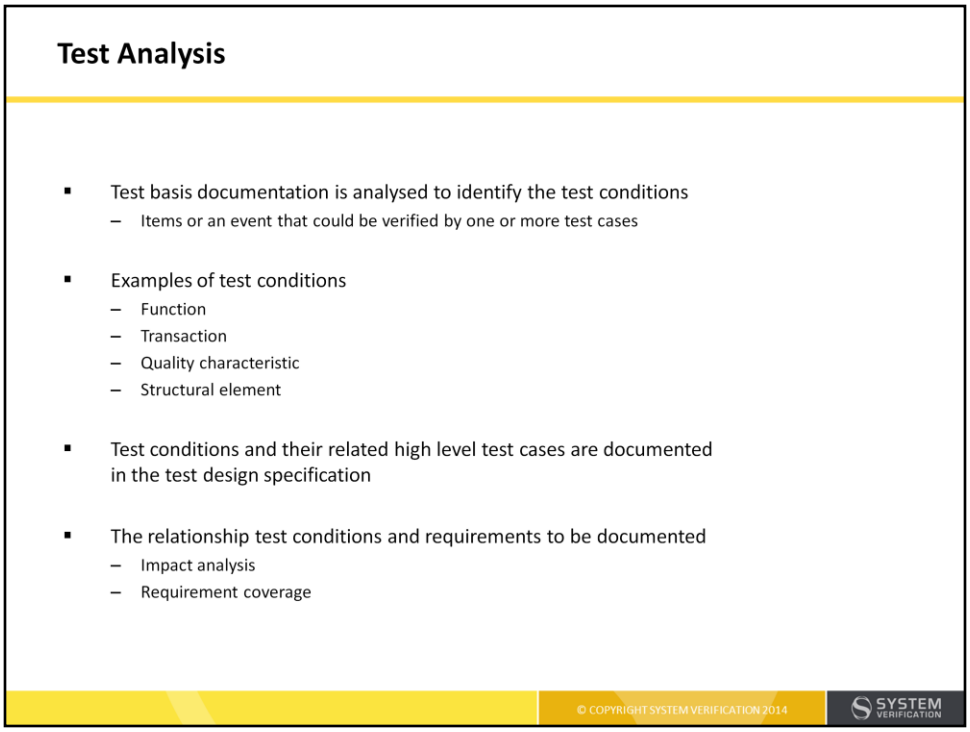
Terms
test case specification, test design, test execution schedule, test procedure specification, test script, traceability

The Test Development Process

- Varies from informal with little or no documentation, to very formal depending on
  - Context of the testing
  - The maturity of testing and development process
  - Time constraints
  - Safety or regulatory requirements
  - People involved



The test development process can vary, from very informal with little or no documentation, to very formal. It depends on the context of the testing, the maturity of testing and development process, time constraints, safety or regulatory requirements and people involved.



During test analysis, the test basis documentation is analysed in order to determine what to test, i.e. to identify the test conditions. A test condition is an item or an event that could be verified by one or more test cases, for example, function, transaction, quality characteristic or structural element.

Test conditions and their related high level test cases are documented in the test design specification. The relationship between test conditions and requirements should also be documented. This enables effective impact analysis when requirements change and to determine requirement coverage.

An Example: Background

- A bank is building a web application to be used by customers to apply for a home loan. The system shall work in the following way:
- The customers enter their annual income, the cost of the house/apartment and the size of their down payment
  - The loan will be granted if
    the annual income is more than 20% of the loan
    and
    the down payment is at least 10% of the house/apartment cost
  - A payback plan is required if the down payment is less than 30% of house/apartment cost

SYSTEM VERIFICATION

An Example: System Requirements

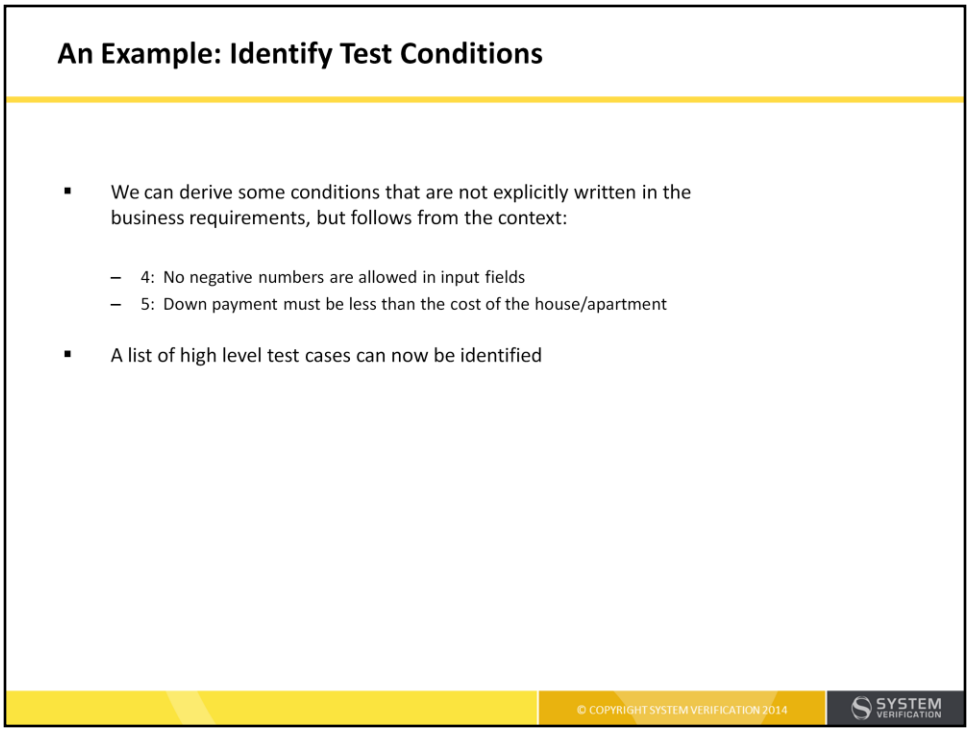
- The bank's web-based application has three input fields:
  - Input 1: Cost of house or apartment
  - Input 2: Down payment
  - Input 3: Annual income

SYSTEM VERIFICATION

An Example: System Requirements

- The bank has two conditions before the loan can be granted:
  - 1: Annual income must exceed 20% of sum of loan (input 3 > 0.2 * input 1 - input 2)
  - 2: Down payment must be more than 10% of the cost (input 2 > 0.1 * input 1).
- If condition 1 and 2 are fulfilled the loan is granted
- The bank has an extra condition where a payback plan is necessary if down payment does not exceed 30% of the value:
  - 3: Down payment exceeds 30% of the cost (input 2 > 0.3 * input 1)

SYSTEM VERIFICATION



We can derive some conditions that is not explicitly written in the business requirements, but follows from the context:
• Condition 4: No negative numbers are allowed in input fields
• Condition 5: Down payment must be less than the cost of the house/apartment

A list of test cases can now be identified based on the mentioned conditions above.

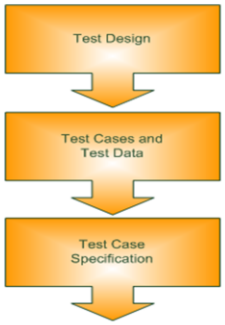
An Example: Identify Test Cases

- Based on the test conditions a list of high level test cases is derived.

ID	Test case name	Test conditions
TC-1	Yearly income too low	1
TC-2	Cash payment too low	1,2
TC-3	Loan granted but with payback plan required	1,2,3
TC-4	Loan granted without need of payback plan	1,2,3
TC-5	Negative input	4
TC-6	Invalid cash payment	5

Create Test Cases

- During test design, test cases and test data are created and specified in the test case specification
- A test case consists of:
  - Pre-conditions
  - A set of input values
  - Expected results
  - Post-conditions



During test design, test cases and test data are created and specified in the test case specification. A test case consists of:
• Pre-conditions
• A set of input values
• Expected results
• Post-conditions

These are specified in detail to make sure they cover the given test condition or test objective.

Specify Expected Results

- Part of the specification of a test case:
  - Outputs
  - Changes to data and states
  - Other consequences of the test
- Defined prior to test execution
  - Clearly so it cannot be misinterpreted

Expected results should be produced as part of the specification of a test case and include outputs, changes to data and states, and any other consequences of the test.

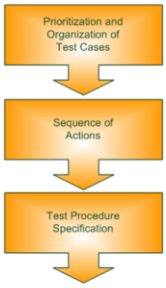
If expected results have not been defined, a plausible but erroneous result may be interpreted as the correct one. Expected results should ideally be defined prior to test execution. The expected results must be defined clearly so it cannot be misinterpreted.

Test Case Specification

- **Pre-conditions:** Web-application started, user logged in
- **Inputs:** Cost 2.500.000 SEK, cash payment 250.000 SEK and yearly income 400.000 SEK
- **Expected result:** No loan granted
- **Post-conditions:** Web-application running, user logged in, expected result match

Develop Test Procedures

- During test implementation, the test cases are developed, implemented, prioritized and organized in the test procedure specification.
- Manual test procedure specifies the execution sequence
- If using a test automation tool, the sequence of actions is specified in a test script (i.e. an automated test procedure)



During test implementation, the test cases are developed, implemented, prioritized and organized in the test procedure specification. The test procedure (or manual test script) specifies the sequence of action for the execution of a test.

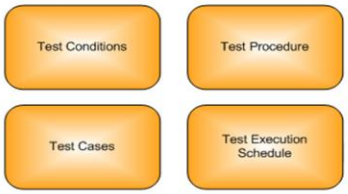
If tests are run using a test execution tool, the sequence of action is specified in a test script (which is an automated test procedure).

An Example: Test Procedure

Step number	Action	Expected result
1	Restart web application	The application is shown with input field of cost, cash payment and yearly income
2	Enter cost of 2.500.000 SEK	-
3	Enter cash payment of 250.000 SEK	-
4	Enter yearly income of 400.000 SEK	-
5	Press "Apply"-button	A dialog is displayed saying that loan cannot be granted since yearly income is less than 20% of loan

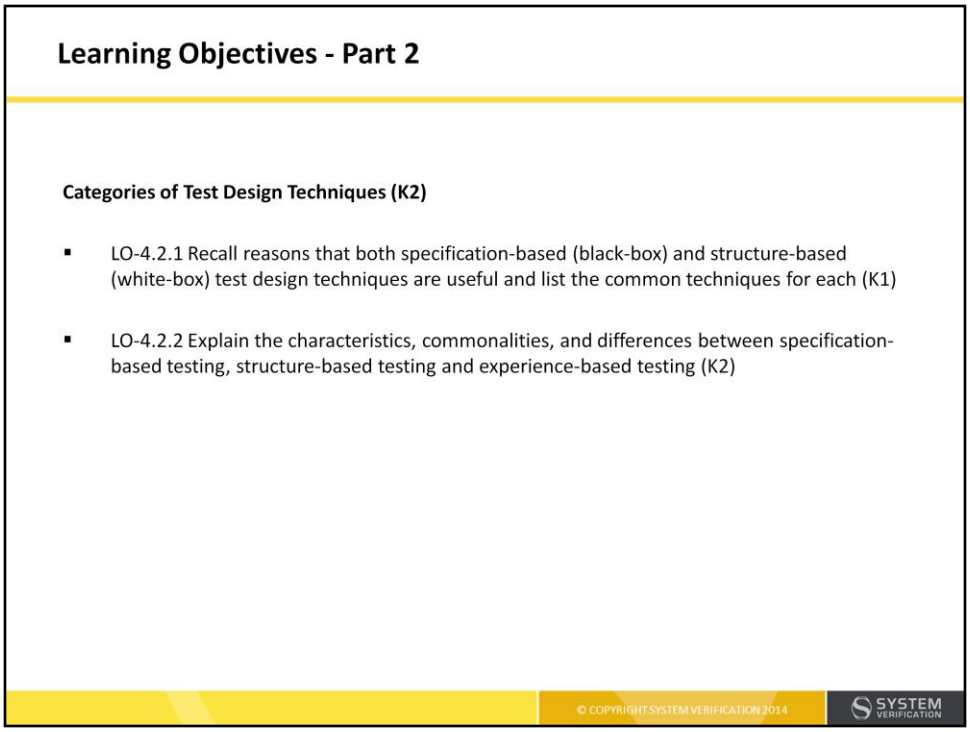
Schedule Test Execution

- Procedures and automated test scripts are subsequently formed into a test execution schedule
- Defines the order of execution, when and who
- Things to take into account are
  - Regression tests
  - Prioritization
  - Technical/logical dependencies



The various test procedures and automated test scripts are subsequently formed into a test execution schedule that defines the order in which the various test procedures, and possibly automated test scripts, are executed, when they are to be carried out and by whom.

Things to take into account are factors such as regression tests, prioritization and technical/logical dependencies.

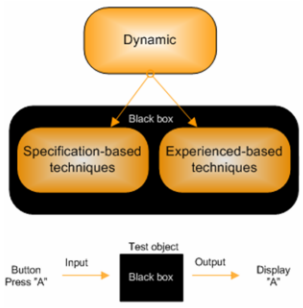


Time
Intended study time is 15 minutes

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

Black-box Techniques

- Black-box techniques also called specification-based techniques
  - No information about the internal structure
  - Analyse the behaviour or output based on a certain input
- Method to derive and select test conditions, test cases or test data based on
  - Test basis documentation
  - Experience of developers, testers and users
- Black-box tests include
  - Functional testing
  - Non-functional testing



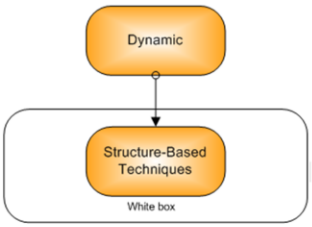
Black-box techniques also called specification-based techniques. The test object is represented as a black-box, i.e. we do not use any information about the internal structure, but we can analyse the behaviour or output based on a certain input.

Black-box techniques are a way to derive and select test conditions, test cases or test data based on an analysis of the test basis documentation describing the software or expected behaviour of its components.
The experience of developers, testers and users could also be used.

Black-box tests include both functional and non-functional testing.

White-box Techniques

- Also called structural or structure-based techniques
- Based on analysis of the structure of the component or system
- Test cases are derived from the code and detailed design
- Coverage can be measured for existing test cases
- Further test cases can be derived systematically to increase coverage



White-box techniques (also called structural or structure-based techniques) are based on an analysis of the structure of the component or system.

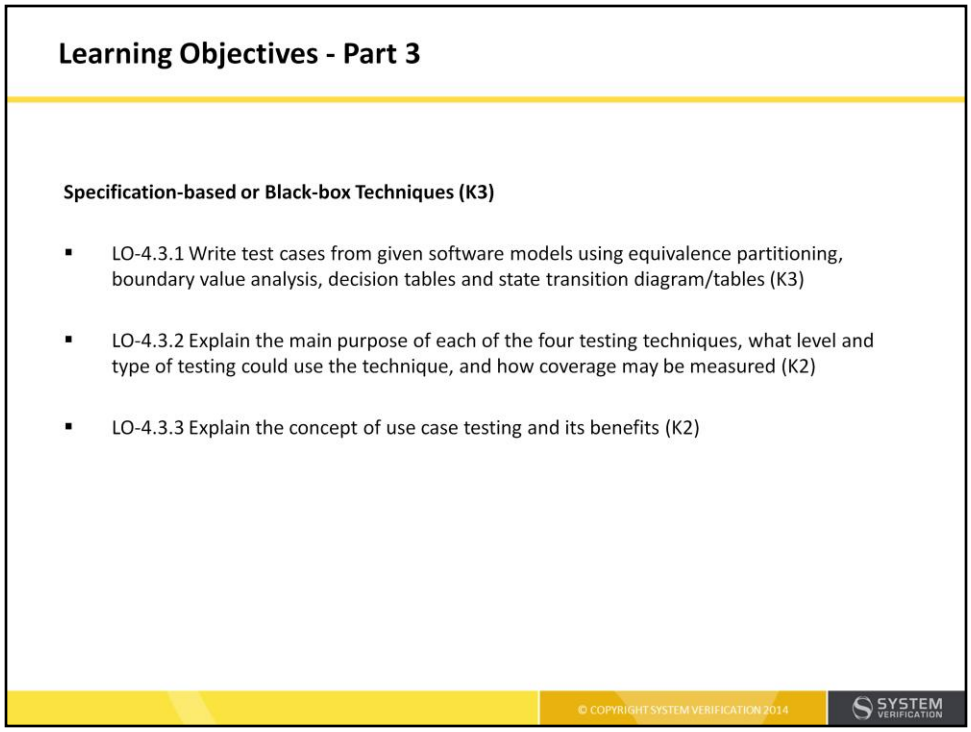
Information about how the software is constructed is used to derive the test cases. The information could, for example, be derived from the code and detailed design.

The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage.

Experience-based Techniques

- The knowledge and experience of people are used to derive the test cases
- Knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment
- Knowledge about likely defects and their distribution

SYSTEM VERIFICATION

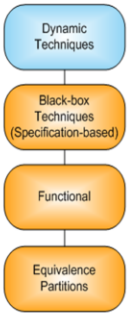


Time
Intended study time is 150 minutes

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

Equivalence Partitioning

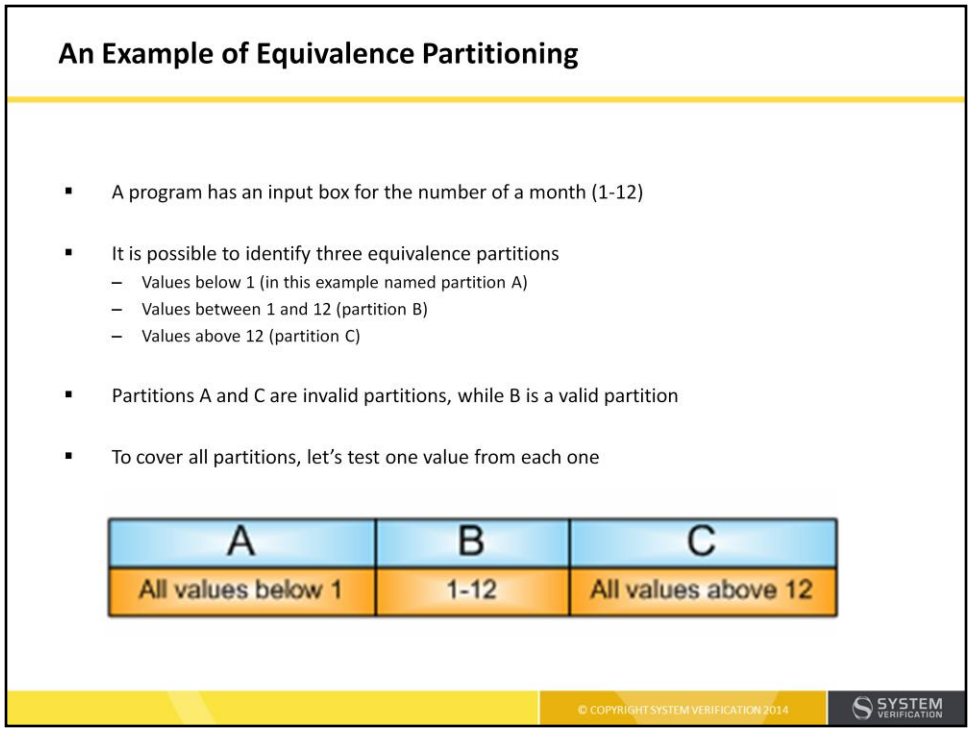
- Input values are divided into groups
- Expected to exhibit similar behaviour
  - Valid and invalid data
- Partitions can also be identified for
  - Outputs
  - Internal values
  - Time related values
  - Interface parameters
- Tests can be designed to achieve input and output coverage
- Equivalence partitioning is applicable at all levels of testing



Input boxes are divided into groups and consists of input values that are expected to exhibit similar behaviour. Equivalence partitions or classes can be found for both valid and invalid data, i.e. values that should be rejected.

Partitions can also be identified for outputs, internal values, time related values and for interface parameters.

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



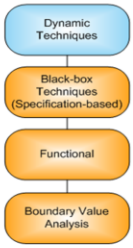
A program has an input box for the number related to a specific month, e.g. number 4 for the month of April. Based on this it is possible to identify three equivalence partitions.

Partitions A and C are invalid partitions, while B is a valid partition. We need (at least) one test for each partition.

For partition A we could use number -6, for partition B number 5 and for partition C number 22.

Boundary Value Analysis

- Valid and invalid partitions have boundary values
  - At the edges of equivalence partitions
  - Maximum and minimum values
- Behaviour at the boundaries are more likely to be incorrect
  - "Off-by-one" defects caused by coding mistakes
  - Testing is likely to yield defects
  - Detail specifications determines the interesting boundaries
  - A test case for each boundary value
- Can be applied at all test levels



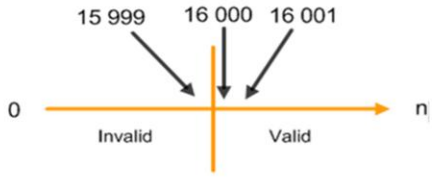
Boundary values are values at the edge of an equivalence partition, the maximum and minimum value. Both valid and invalid partitions have boundary values.

Behaviour at the edge of each equivalence partition is more likely to be incorrect, so boundaries are an area where testing is likely to yield defects. Detail specifications are helpful to in determining the interesting boundaries.

When using this technique, a test case for each boundary value is used. Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect finding capability is high

An Example of a Boundary Value Analysis

- In the loan web application have three different input fields:
  - Cost
  - Down payment
  - Income
- For the income field, valid inputs are $16 000 and above
  - Valid and invalid partitions are tested
- The following two or three input values are needed:
  - $15 999 (invalid)
  - $16 000 (valid)
  - Optional: $16 001 (valid, but may be considered redundant)



In the loan web application, there are three different input fields where the user shall enter cost of house or apartment, down payment, and annual income.

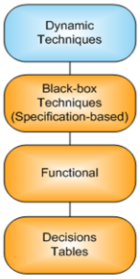
Focusing on the income field, valid inputs are $16 000 and above. This field will be tested with the boundary value analysis technique for valid and invalid partitions.

The following two or three input values are needed:
• $15 999 (invalid)
• $16 000 (valid)
• Optional: $16 001 (valid)

In this case we have identified three boundary values. However, because $16 000 is in the valid partition, the value $16 001 may be considered redundant. The test designer may choose to omit it. Whether two or three boundary values are used is a matter of risk and what level of test coverage is needed.

Decision Table Testing

- Suitable to capture system requirements that contain logical conditions
  - Specification is analysed, and conditions and actions are identified
  - Conditions are true or false
- Each column corresponds to a business rule
  - Combination of conditions
  - Actions associated with that rule
- Useful to ensure that important combinations are not overlooked



Using decision tables is a good way to capture system requirements that contain logical conditions.

When creating a decision table the specification is analysed, and conditions and actions are identified. They are stated in such a way that they can either be true or false.

Each column of the table corresponds to a business rule that defines a unique combination of conditions, which result in the execution of the actions associated with that rule.

Decision tables are especially useful to ensure that important combinations are not overlooked.

Decision Table Testing

- Create at least one test per column
  - Covers all combinations of trigger conditions
- Strength
  - It creates combinations of conditions that might not otherwise have been exercised during testing
- Applied to all situations when the action depends on several logical decisions

Input	New customer?	TRUE	TRUE	FALSE	FALSE
Input	Order over $100?	TRUE	FALSE	TRUE	FALSE
Action	Give discount	FALSE	FALSE	TRUE	FALSE

The coverage standard commonly used with decision table testing is to have at least one test per column, which typically involves covering all combinations of triggering conditions.

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

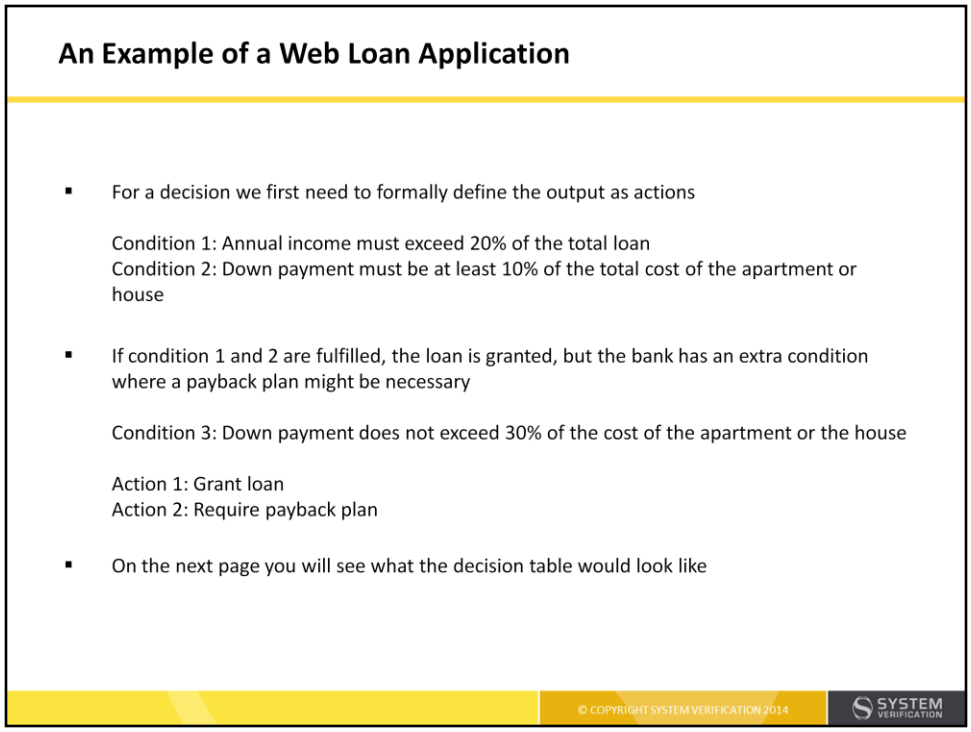
An Example of a Decision Table

- This example is about giving customers in a web store discount, or not
  - New or returning customer?
  - Order value above a certain limit?
- A discount shall only be given to **returning customers** who place an **order over $100** (represented by the 3rd case in the decision table below)

		Case 1	Case 2	Case 3	Case 4
Input	New Customer?	TRUE	TRUE	FALSE	FALSE
Input	Order over $100?	TRUE	FALSE	TRUE	FALSE
Action	Give discount	FALSE	FALSE	TRUE	FALSE

This example is about giving customers in a web store discount, or not. Depending on if the customer is new or returning and the order value is above a certain limit, discount should be provided in certain cases.

For example, looking at business rule 3, discount shall be provided when it is a returning customer and the order value is over $100.



If a decision table was used to derive the test cases of our home web loan application we first need to formally define the output as actions.

Condition 1: Annual income must exceed 20% of the total loan for the apartment or house

Condition 2: Down payment must be at least 10% of the total cost of the apartment or house

If condition 1 and 2 are fulfilled, the loan is granted, but the bank has an extra condition where a payback plan might be necessary.

Condition 3: Down payment does not exceed 30% of the cost of the apartment or the house

Action 1: Grant loan

Action 2: Require payback plan

On the next page you will see what the decision table would look like.

An Example of a Web Loan Application

- Each column is a real world scenario
  - One test case for each column
- If a customer does not fulfil conditions 1-3 the loan will not be granted (action 1) and no payback plan is needed (action 2). Since the first condition makes the other two conditions irrelevant, the first four columns (C1-C4) can merge as well as column 5 and 6. This results in four test cases.

Test case 1: column 1-4
Test case 2: column 5-6
Test case 3: column 7
Test case 4: column 8

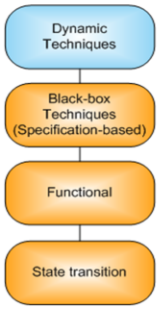
	C1	C2	C3	C4	C5	C6	C7	C8
Cond 1	F	F	F	F	T	T	T	T
Cond 2	F	F	T	T	F	F	T	T
Cond 3	F	T	F	T	F	T	F	T
Action 1	F	F	F	F	F	F	T	T
Action 2	-	-	-	-	-	-	T	F

Each column in the table can be implied as a real world scenario. There would normally be one test case for each column above. If a customer does not fulfil conditions 1-3 the loan will not be granted (action 1) and no payback plan is needed (action 2). Since the first condition makes the other two conditions irrelevant, the first four columns (C1-C4) can merge as well as column 5 and 6.

This results in four test cases:
• Test case 1: column 1-4
• Test case 2: column 5-6
• Test case 3: column 7
• Test case 4: column 8

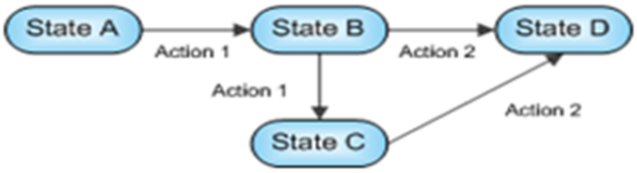
State Transition Testing

- An objects state is depending on current conditions or previous state
  - It can be shown as a state transition diagram
- A state transition diagram provides
  - Overview of its states
  - Transitions between states
  - Inputs or events that trigger transitions
  - Actions as a result
- The goal is to exercise all the states and transitions of the application



State Transition Testing

- Tests can be designed to cover every state or a typical sequence of states.
- It can also be designed to exercise every transition, specific sequences of transitions or to test invalid transitions.
- Test cases are added until all states and transitions are covered
- Some transitions start and end in same state and are easily missed



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

According to this technique, the basic strategy when creating test cases is to keep adding test cases until all states and transitions are covered.

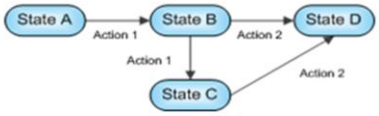
It is important to note that some transitions start and end in same state. Such transitions are easily missed.

State Tables and State Diagrams

- A state table shows the relationship between the states and inputs
  - Invalid transitions can be highlighted
- This technique is often used within
  - The embedded software industry and technical automation.
  - Testing screen-dialogue flows and modelling business objects having specific states

	State A	State B	State C	State D
Action 1	B	C	-	-
Action 2	-	D	D	-

=

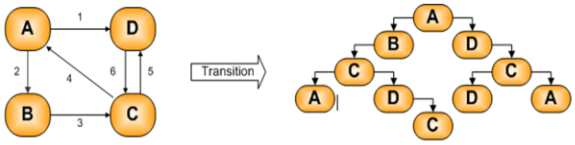


State transitions can also be represented as a table. A state table shows the relationship between the states and inputs, and can highlight possible invalid transitions.

This technique is often used within the embedded software industry and technical automation. It is also suitable for testing screen-dialogue flows and modelling business objects having specific states.

Method for Designing Tests

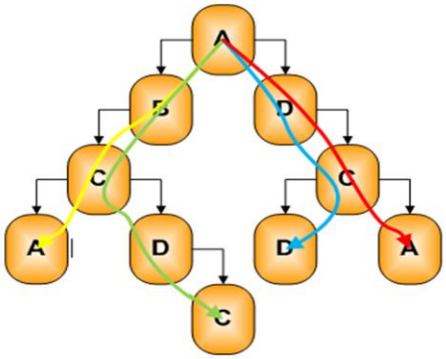
- Choose a start state in the state diagram
- For all transitions from the start state, draw a branch in the tree to the next state
- Repeat recursively for every state
  - Not drawn before
  - Stop at the end state
- Choose tests for all root-to-leaf paths in the tree



SYSTEM VERIFICATION

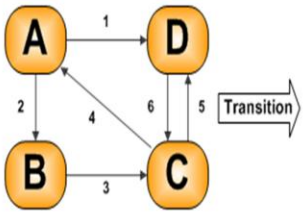
Example of Test Design for State Transitions

- The following tests have been identified:
  - A-B-C-A
  - A-B-C-D-C
  - A-D-C-D
  - A-D-C-A



Example of State Transition Decision Table

- Only tested the valid state transitions – no invalid transitions!



	State A	State B	State C	State D
Action 1	D	-	-	-
Action 2	B	-	-	-
Action 3	-	C	-	-
Action 4	-	-	A	-
Action 5	-	-	D	-
Action 6	-	-	-	C

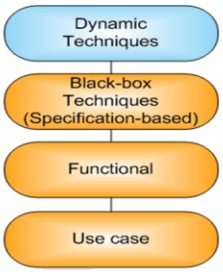
Experience from Testing State Transitions

- Test at least all transitions
- Test all invalid and ”strange” transitions
- Test the longest sequence through the state diagram
- Repeat all tests twice without resetting the system in between

SYSTEM VERIFICATION

Use Case Testing

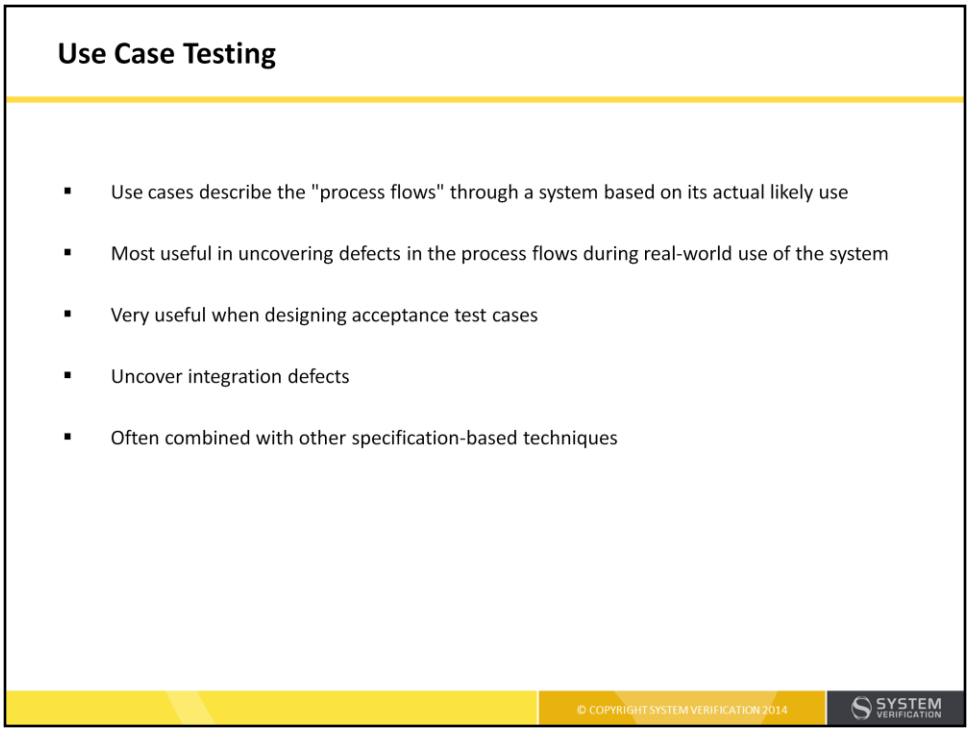
- Tests can be specified from use cases or business scenarios
  - Interactions between actors (users or system)
  - Produce a result of value to a system user
- Each use case has
  - Pre-conditions
  - Post-conditions
  - Mainstream scenario
  - Alternative paths



Tests can be specified from use cases or business scenarios. A use case describes interactions between actors, including users and the system, which produce a result of value to a system user.

Each use case has preconditions that need to be met to work successfully and postconditions that are the observable results and the final state.

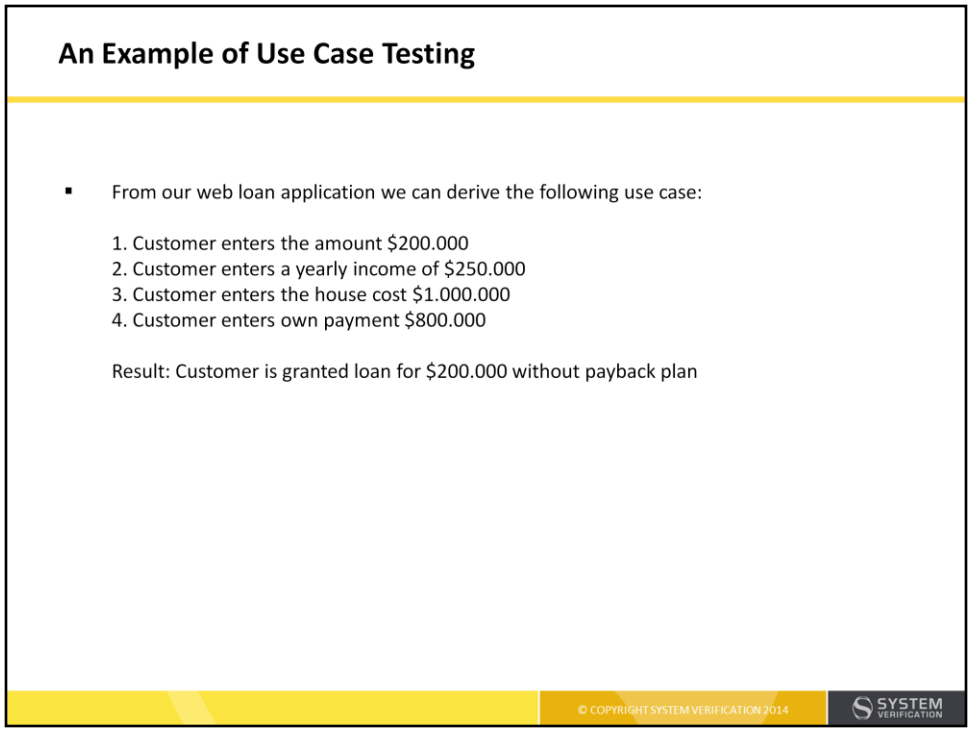
A use case usually has a mainstream or a most likely scenario, and sometimes alternative paths.



Use cases describe the "process flows" through a system based on its actual likely use. Test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system.

Use cases, often referred to as scenarios, are very useful when designing acceptance tests with customer/user participation. They also help uncover integration defects caused by the interaction and interference of different components, which individual component testing would not detect.

Use case testing may be combined with other specification-based techniques designing test cases.

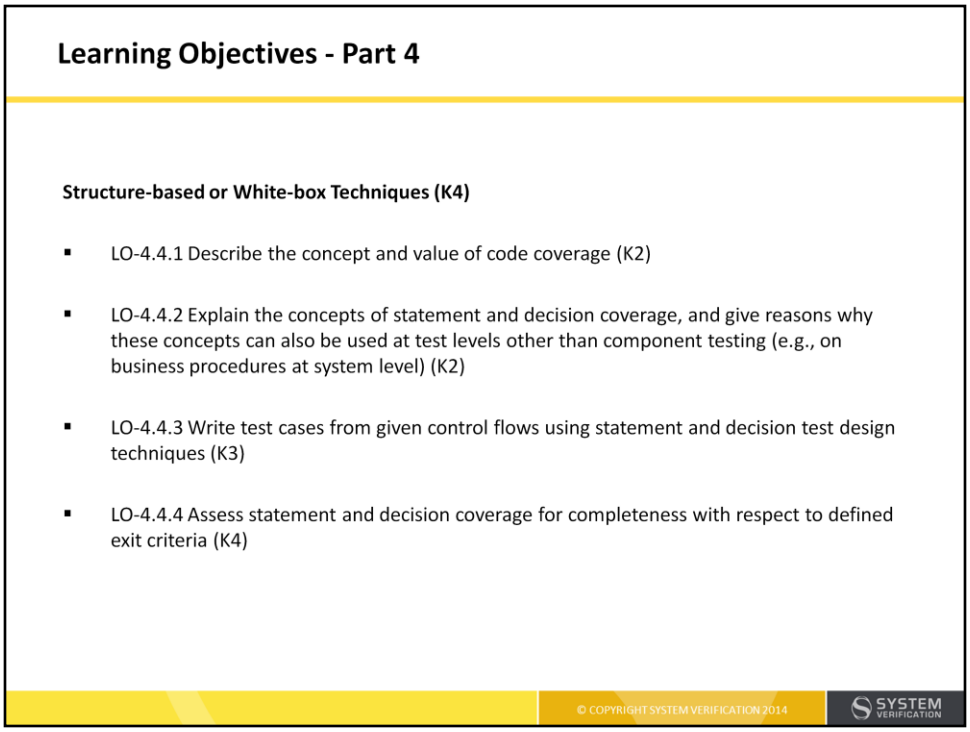


From our web loan application we can derive the following use case.

1. Customer enters the amount $200.000
2. Customer enters a yearly income of $250.000
3. Customer enters the house cost $1.000.000
4. Customer enters own payment $800.000

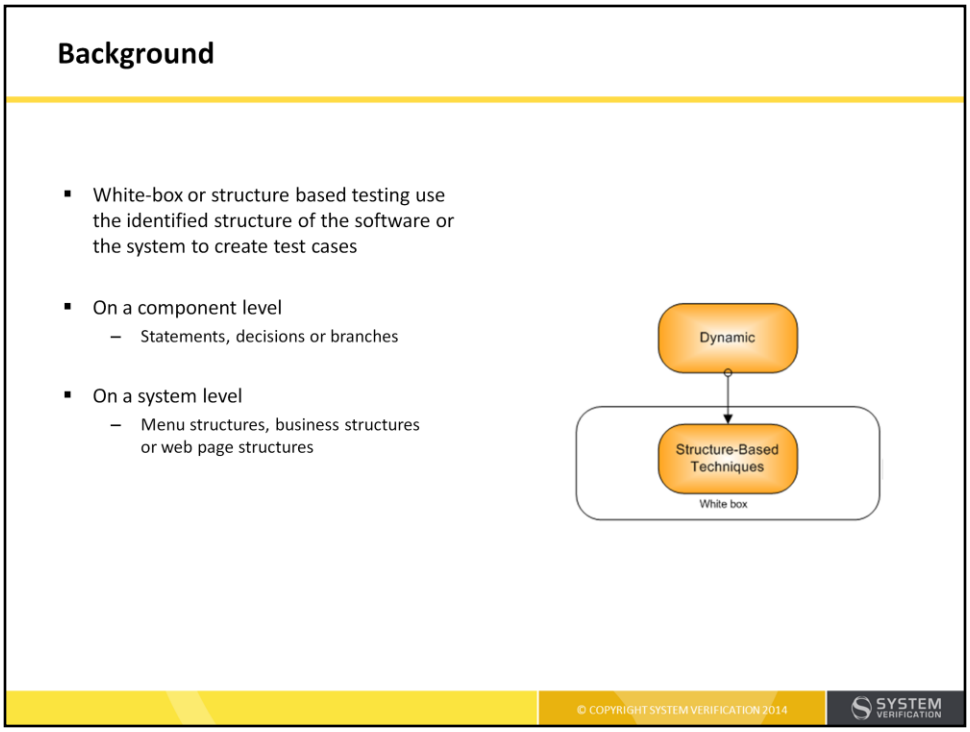
Result: Customer is granted loan for $200.000 without payback plan

SYSTEM VERIFICATION



Time
Intended study time is 60 minutes

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



White-box or structure based testing use the identified structure of the software or the system to create test cases. On a component level it could be statements, decisions or branches. On a system level it may be menu structures, business structures or web page structures.

Three different test design techniques for code coverage, based on statements, branches and decisions, are discussed.

Statement Testing and Coverage

- Statement coverage is a type of structural testing
  - Measure the percentage of all code statements executed
  - Intention is to increase statement coverage by designing more test cases
- In this example, we want to cover the statements using as few test cases as possible

```
if (age >= 18)
   print "Hello adult"
```

- We can use a value equal to or larger than 18 for testing, in which case both statements will be executed (100 % statement coverage)
- If we use a value less than 18, the print-statement will not be executed (50 % statement coverage)

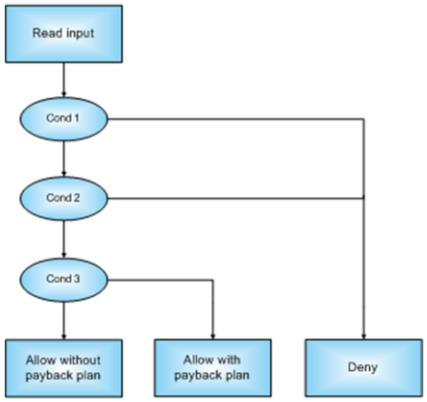
Statement coverage is a type of structural testing that looks at the percentage of all code statements that have been executed by a test case suite. The intention is to increase statement coverage by designing more test cases until we reach defined goals.

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

In this example, we want to cover 100 % of the statements. However, if we choose a value of age less than 18, we will only reach 50 % coverage, because the command print “Hello adult” will not be executed.

An Example of Statement Coverage

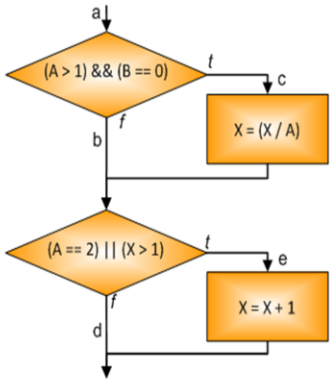
- Based on our web loan application we have three conditions and three possible actions
- Conditions:
  1. Annual income exceed 20% of loan
  2. Down payment at least 10% of cost
  3. Down payment doesn't exceed 30% of cost
- Actions:
  1. Grant loan
  2. Require payback plan
  3. Deny loan
- It is easy to see from the corresponding flowchart diagram that three test cases are enough to reach 100% statement coverage.



Another Example of Statement Coverage

```
if (A > 1) && (B == 0)
   X = (X / A)
if (A == 2) || (X > 1)
   X = X + 1
```

- In this example there are different paths marked a-e, that can be either true (t) or false (f)
- How many tests do we need to cover all statements?

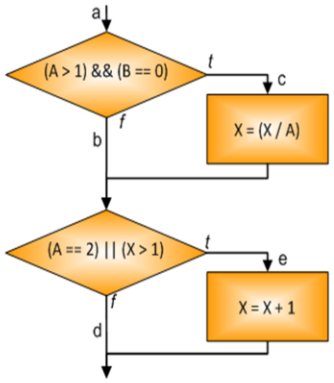


Another Example of Statement Coverage (continued)

- 100 % statement coverage can be achieved through path (a c e) and by choosing the following input:

  A = 2, B = 0, X = (an arbitrary value)

- We need only one test case
- Note that we do not cover path (a b d)



In this case we shall cover all statements. It can be achieved through path (a c e) and by choosing the following input:

A = 2, B = 0, X = (an arbitrary value)

We need only one test case but this does not cover (a b d).

Decision Coverage

- Decision coverage looks at the percentage of decision outcomes that have been reached
- See the example: to reach 100% decision coverage the input value for "x" needs to be
  1. x < 0
  2. x = 0
  3. x > 0
- 100 % decision coverage guarantees 100 % statement coverage, but not vice versa
  - Decision coverage is stronger criteria than statement coverage

```
if (x < 0)
   y = 3
else if (x == 0)
   y = 4
else if (x > 0)
   y = 5
```


Decision coverage is a technique that looks at the percentage of decision outcomes that have been reached. Take a look at the following example:

The input value for "x" needs to be different in each case so all decision paths can be covered. In this case we need three different test cases to reach 100 % decision coverage. One where x is less than zero, one where x equals zero and one where it exceeds zero.

Decision coverage has higher percentage coverage than statement coverage i.e. 100 % decision coverage guarantees 100 % statement coverage, but not vice versa.

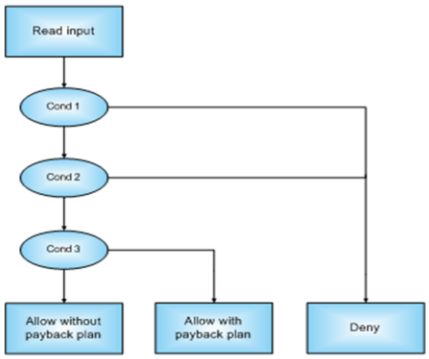
An Example of Decision Coverage

- The decision to allow or deny a bank loan is illustrated in this picture

```
if (income > (0.2 * (cost - cash)))
   if (cash > (0.1 * cost))
      if (cash > (0.3 * cost))
         AllowWithoutPlan()
      else
         AllowWithPlan()
Deny()
```

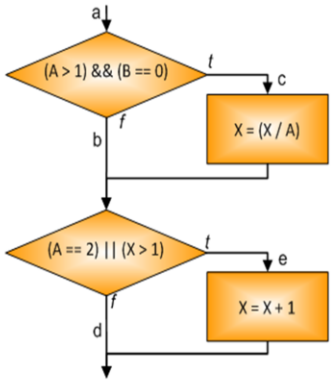
An Example of Decision Coverage (continued)

- To reach a 100 % decision coverage, we need 4 test cases (since all outcomes of decision blocks shall be tested)



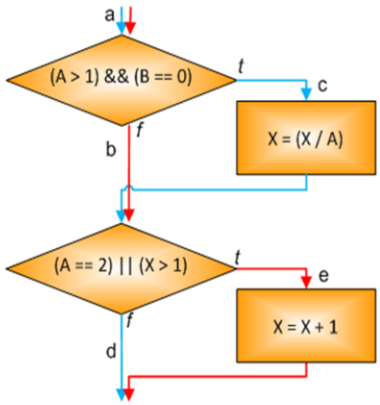
Another Example of Decision Coverage

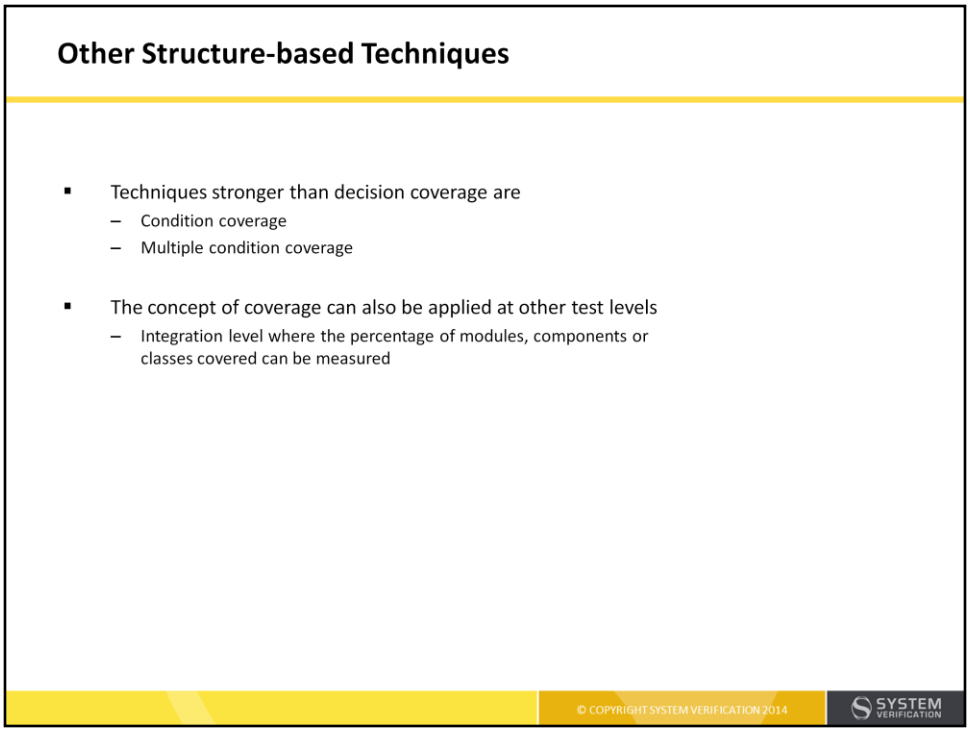
- In this case the two decisions shall be both true and false once
  - ((A > 1) && (B == 0)) = true,
    ((A > 1) && (B == 0)) = false
  - ((A == 2) || (X > 1)) = true,
    ((A == 2) || (X > 1)) = false



Another Example of Decision Coverage (continued)

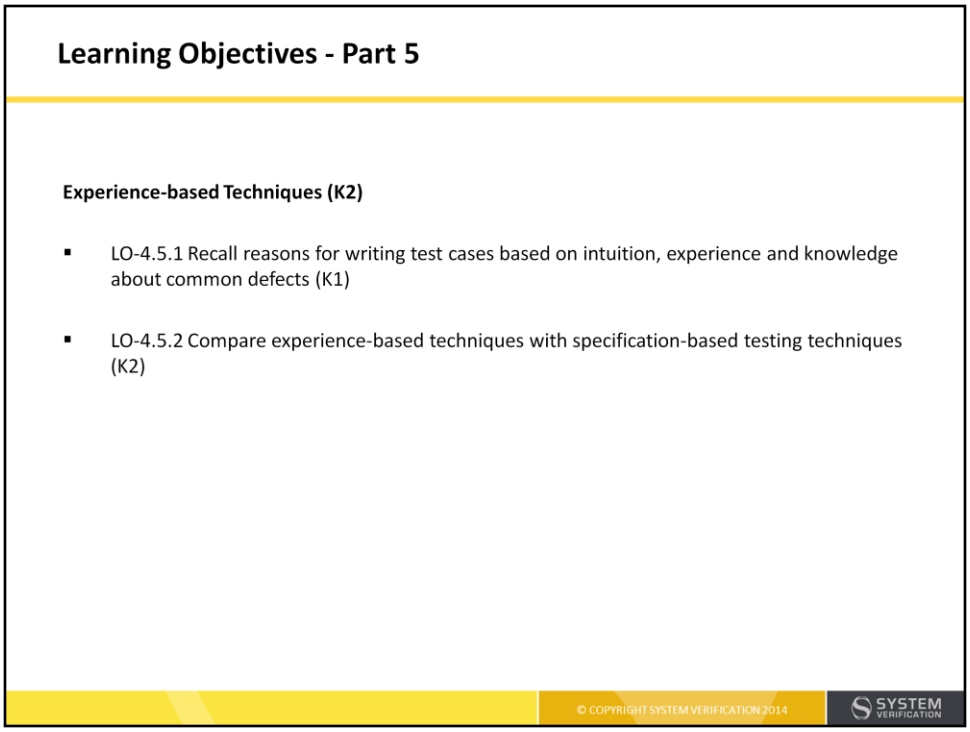
- It can be achieved through the following paths and related input:
  - (a c e), (a b d) or (a c d), (a b e)
  - (a c d): A = 3, B = 0, X = 3
  - (a b e): A = 2, B = 1, X = 1





There are stronger levels of structural coverage beyond decision coverage, for example, condition coverage and **multiple condition coverage**.

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



Time
Intended study time is 30 minutes

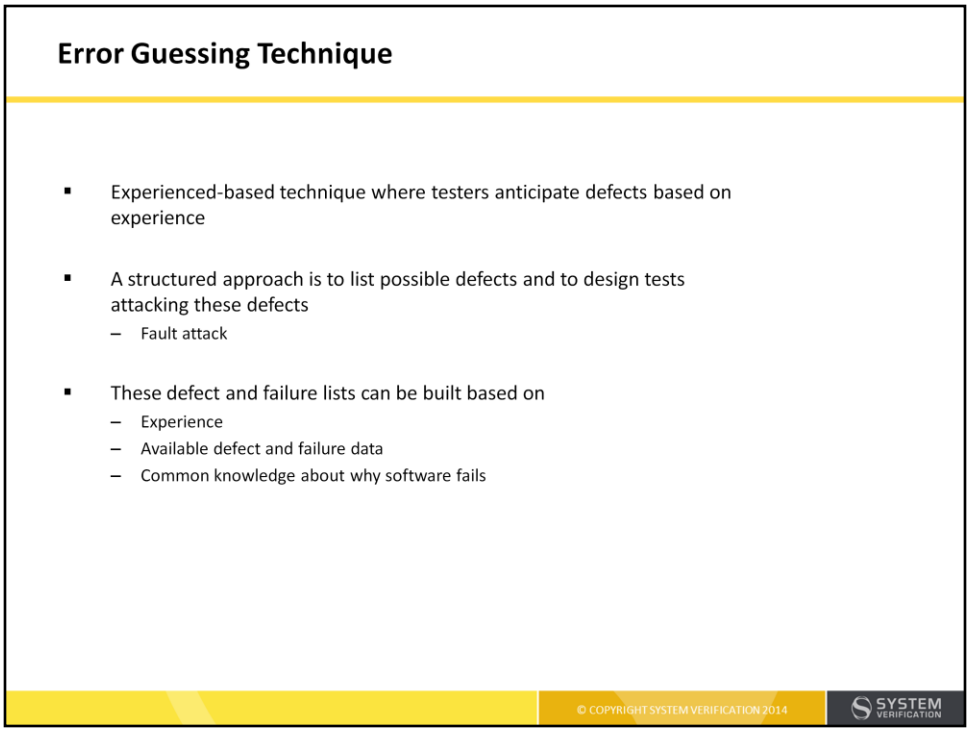
Terms
exploratory testing, (fault) attack

Experience-based Techniques

- Tests are derived from the tester's skill, intuition and experience with similar applications and technologies
- Useful in identifying special tests not easily captured by formal techniques
- May yield varying degrees of effectiveness depending of the tester's experience

Experienced-based testing is when tests are derived from the tester's skill and intuition as well as their experience with similar applications and technologies.

These techniques can be useful in identifying special tests not easily captured by formal techniques. An experienced-based technique may yield varying degrees of effectiveness depending of the tester's experience.



A commonly used experienced-based technique is error guessing, i.e. testers anticipate defects based on experience.

A structured approach is to enumerate a list of possible defects and to design tests attacking these defects. This systematic approach is called fault attack.

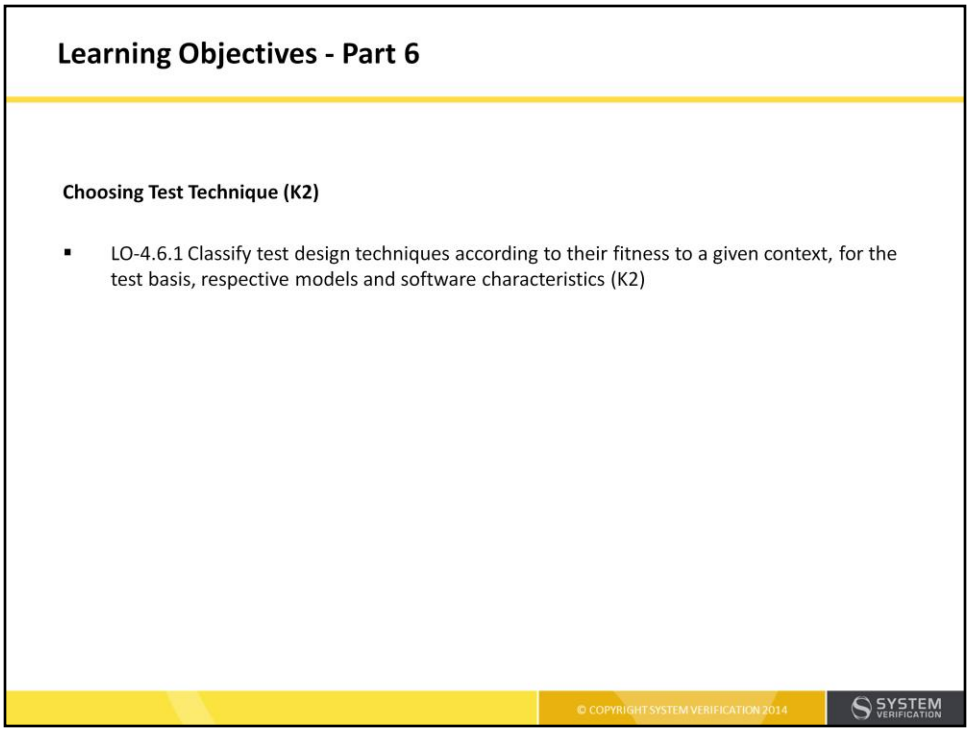
These defect and failure lists can be built based on experience, available defect and failure data, and from common knowledge about why software fails.

Exploratory Testing

- Exploratory testing is about frequent time-boxed cycles of
  - Test design
  - Test execution
  - Test logging
  - Learning
- Based on a test charter containing test objectives
- Can be useful in many situations:
  - Few or inadequate specifications available
  - Severe time pressure
  - To complement other, more formal testing
- It can serve as a check to ensure that the most serious defects are found

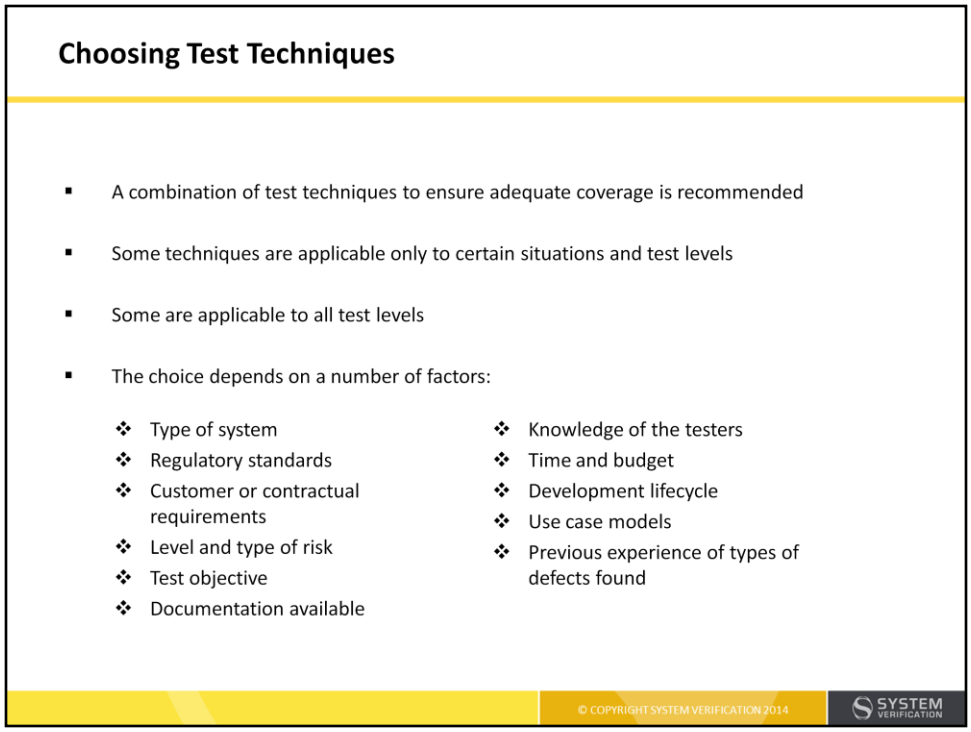
In exploratory testing, test design, test execution, test logging and learning, based on a test charter containing test objectives and performed within time-boxes.

It is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing. It can serve as a check to help ensure that the most serious defects are found.



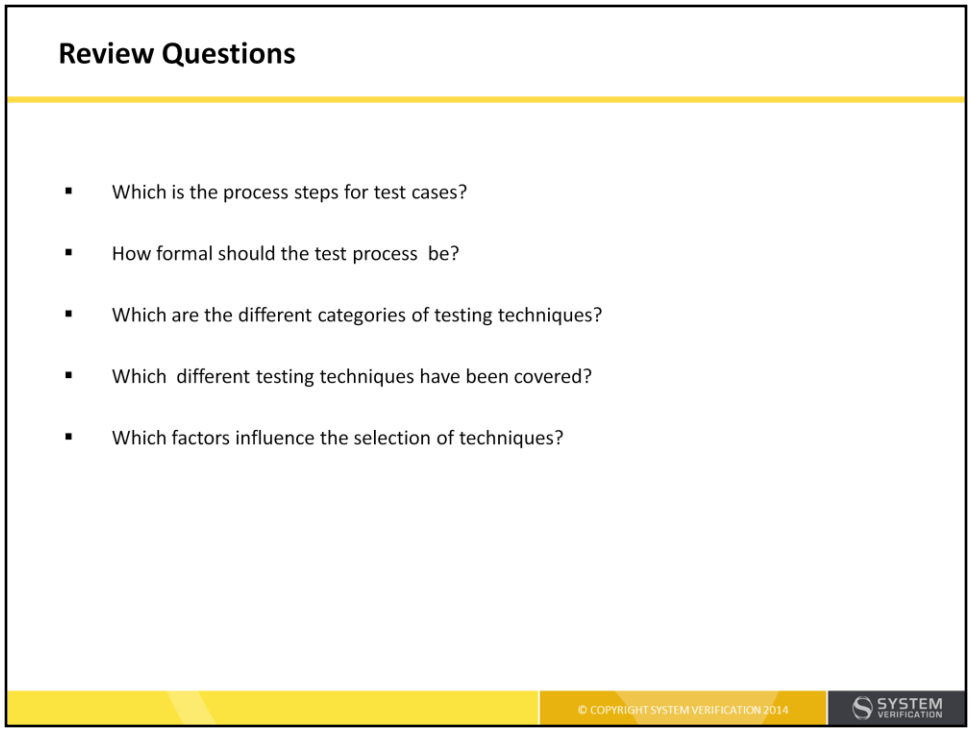
Time
Intended study time is 15 minutes

Terms
No specific terms



The choice of which test techniques to use depends on a number of factors. It could for example be:
• type of system
• regulatory standards
• customer or contractual requirements
• level and type of risk
• test objective
• documentation available
• knowledge of the testers
• time and budget
• development lifecycle
• use case models
• previous experience of types of defects found

Some techniques are more applicable to certain situations and test levels, while others are applicable to all test levels. Testers generally use a combination of test techniques to ensure adequate coverage of the object under test.



Summary

In this chapter you have learnt about a number of different techniques to ease the process identifying test conditions, test cases and test data. We have looked on both specification-based or black-box techniques and structure-based or white-box techniques. Black-box techniques are used to design functional and non-functional tests. Boundary value analysis and equivalence classes are two examples of techniques described. The white-box techniques use the internal structure of the software to design tests and is often used to improve code coverage of the tests. Experience-based testing has also been covered and techniques such as error guessing and the more exploratory approach.

Last words

Testing process is normally done in following steps: Analyse test basis, identify test conditions, design & specify test conditions and develop test procedure. For each step the outcome shall be prioritized. The formality of the test process depends on the context.

Specification-based (black box), structure-based (white box) and experienced based (black box) are all different category of testing techniques. Testing techniques covered are equivalence partitioning, boundary value analysis, decision tables, state transitions, use cases, statement coverage, decision coverage, error guessing and exploratory testing.

Which techniques to use depends on number of factors e.g. risk, system type, requirements, models or knowledge.

TERMS FROM CHAPTER 4

4.1 THE TEST DEVELOPMENT PROCESS

Test case specification
Test design
Test execution schedule
Test procedure specification
Test script
Traceability

4.2 CATEGORIES OF TEST DESIGN TECHNIQUES

Black-box test design technique
White-box test design technique
Experience-based test design technique
Test design technique

4.3 SPECIFICATION-BASED OR BLACK-BOX TECHNIQUES

Boundary value analysis
Decision table testing
Equivalence partitioning
State transition testing
Use case testing

4.4 STRUCTURE-BASED OR WHITE-BOX TECHNIQUES

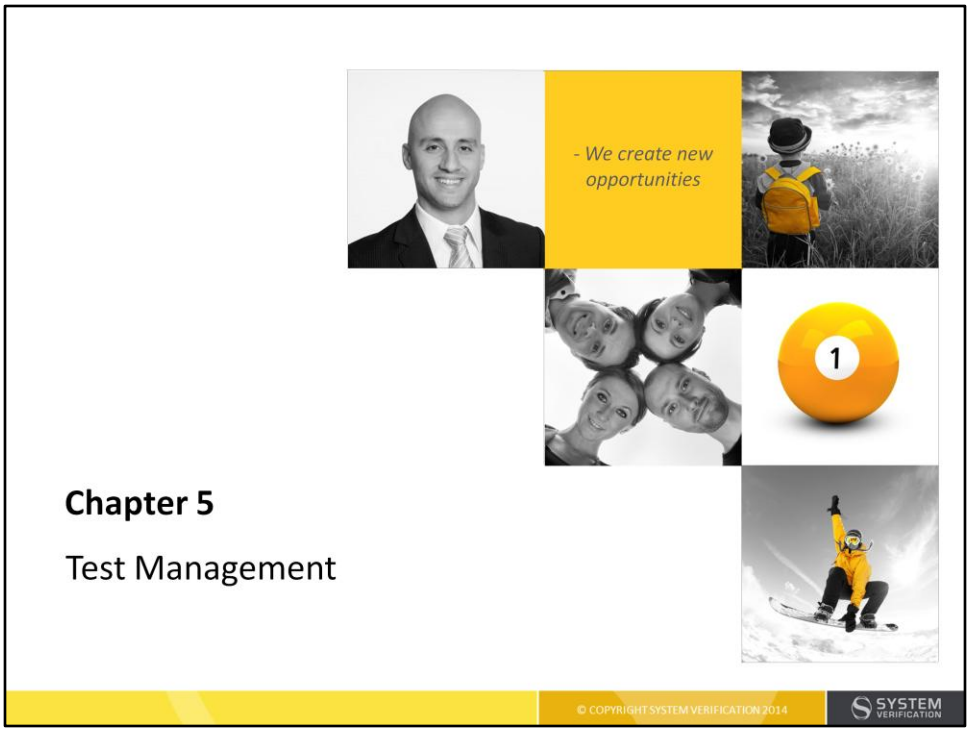
Code coverage
Decision coverage
Statement coverage
Structure-based testing

4.5 EXPERIENCE-BASED TECHNIQUES

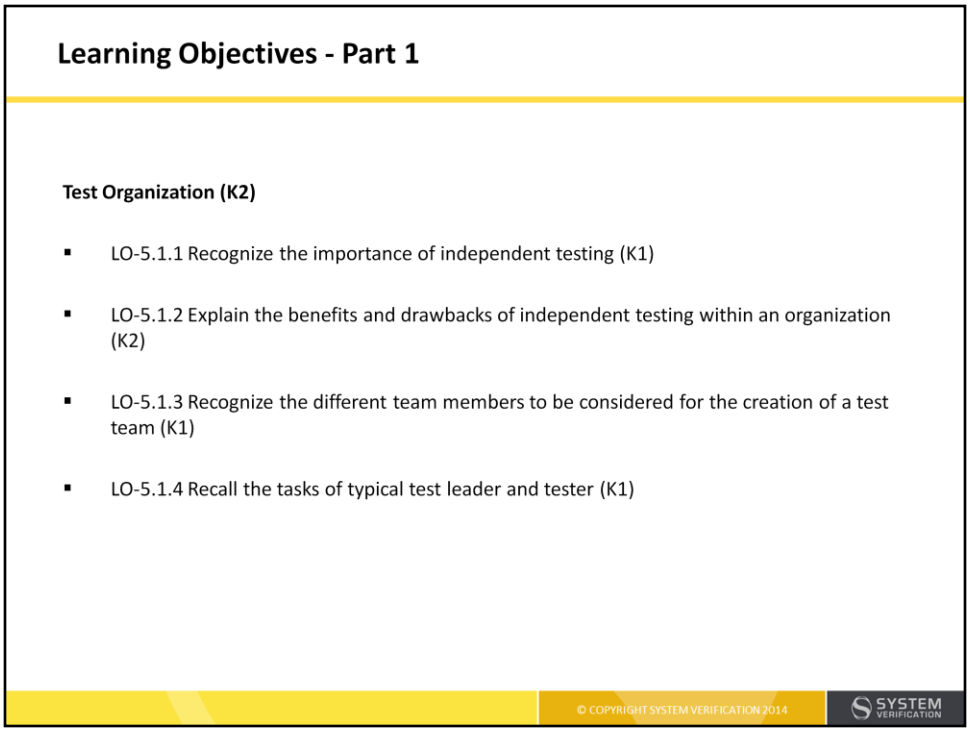
Exploratory testing
(Fault) attack

4.6 CHOOSING TEST TECHNIQUES

-

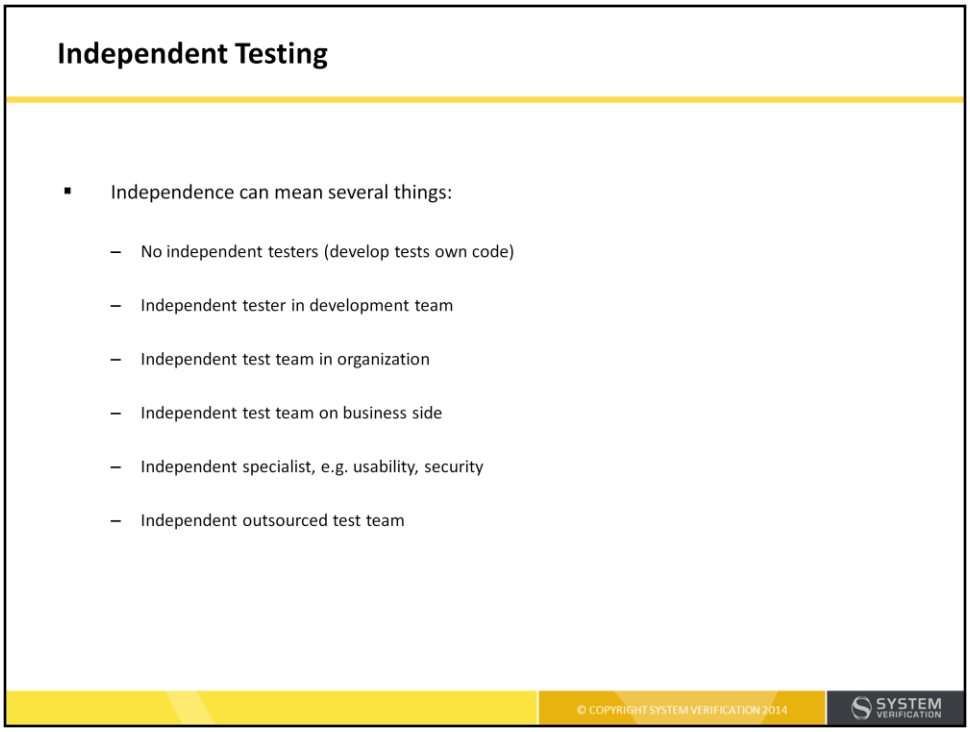


SYSTEM VERIFICATION



Time
Intended study time is 30 minutes

Terms
tester, test leader, test manager



When dealing with large, complex or safety critical projects it is normally recommended to apply multiple levels of testing with some or all of the levels done by independent testers. Development staff may be involved at the lower levels of testing, but their lack of objectivity limits their effectiveness.

Independent testers may have the authority to require and define test process and rules. This kind of process related roles should be taken on only when their is a clear management mandate to do so.

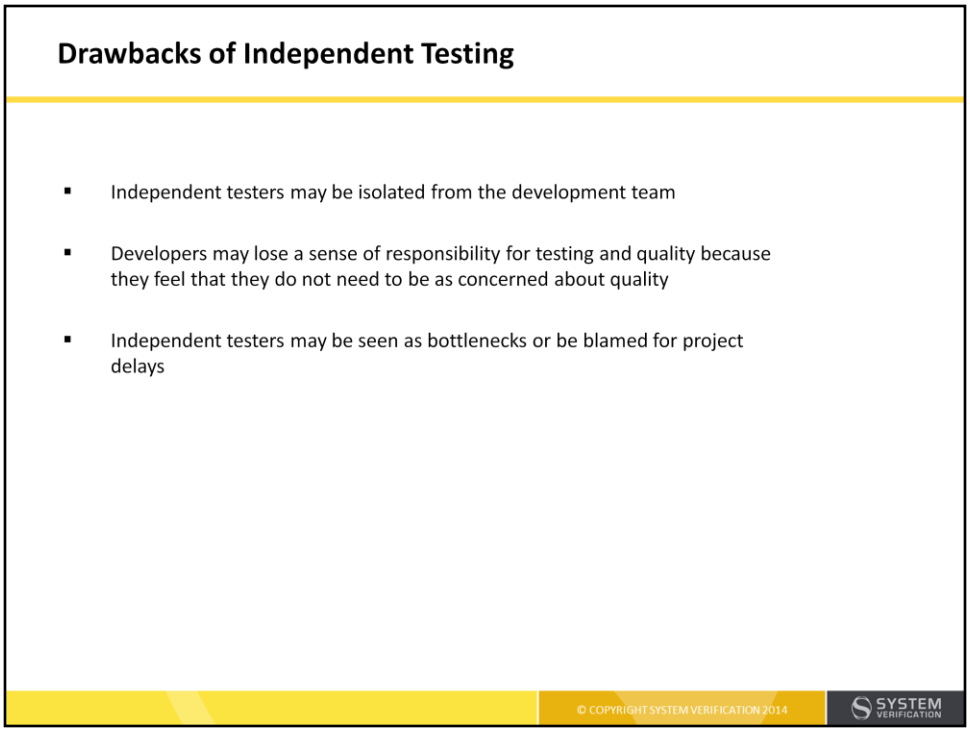
Testing tasks may be done by people in a specific testing role or by someone in another role, e.g. project manager, quality manager, developer, business and domain expert, infrastructure or IT operations.

Independence options
The effectiveness to finding defects and failures by reviews and testing can be improved by using independent testers. When talking about independence there is a number of options from low to high:
• No independent testers, i.e. developers test their own code
• Independent tester with in the development teams
• Independent test team or a group within the organization, reporting to project management or executive management
• Independent testers from the business organization or user community
• Independent test specialist for specific test types, e.g. usability testers, security testers or certification testers
• Independent testers outsourced or external to the organization

Benefits of Independent Testing

- Independent testers find other defects
- Independent testers can verify assumptions made during specification and implementation of the project
- Independent testers are unbiased

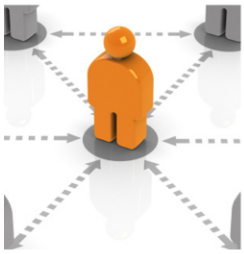
Example of benefits are:
• Independent testers see other and different defects
• An independent tester can verify assumptions people made during specification and implementation of the system
• Independent testers are unbiased



Example of drawbacks are:
• Isolation from the development team if treated as totally independent
• Developers may lose a sense of responsibility for quality because they could feel that quality is not their concern
• Independent testers may be seen as bottleneck or blamed for delays in release

Tasks of the Test Leader (1/2)

- Write test policy, test strategy and test plan
- Coordinate test strategy and test planning
- Plan tests based on context and risk selecting test approaches, estimating time, costs, acquiring resources and test environment
- Initiate the specification, preparation, implementation and execution of tests, monitor test results and check exit criteria
- Adapt planning based on test results and progress, and taking necessary actions
- Set up configuration management of testware



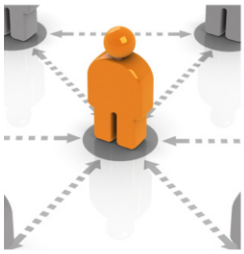
In this course two test roles are covered, test leader and tester. The activities and tasks related to do these two roles depend on the product and project context, the people in the roles and the organization.

Typically the test leader plans, monitors and controls the testing activities, while the tester is responsible for carrying out test base analysis, design and implementation of tests.

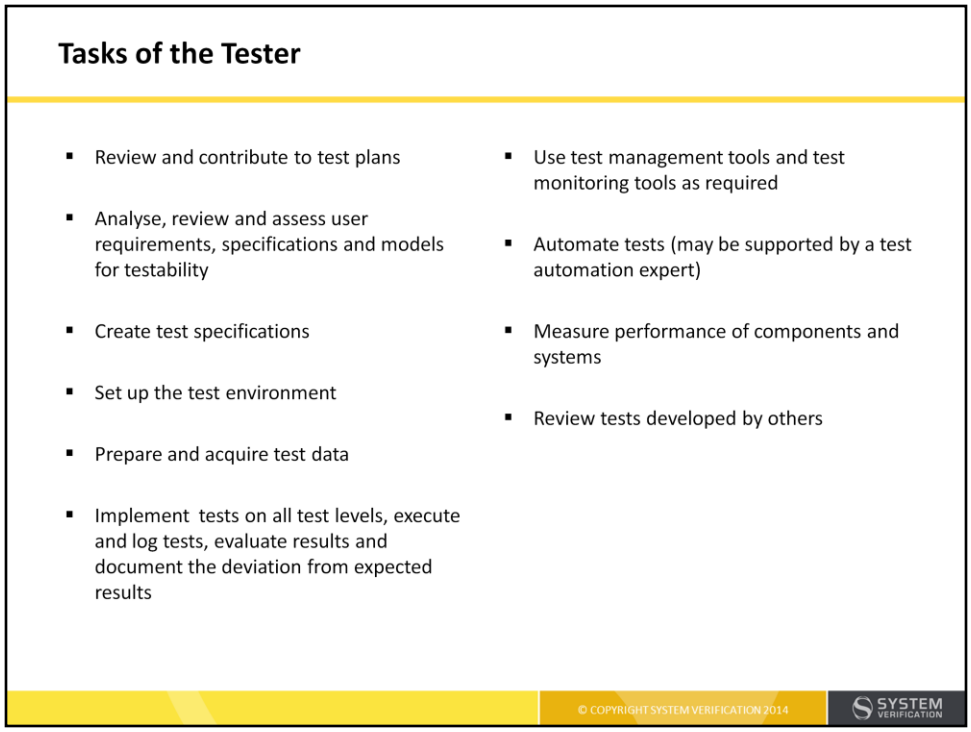
Typical test leader tasks may include:
• Coordinate the test strategy and plan with project managers and others
• Write or review a test strategy for the project and a test policy for the organization
• Planning of the tests considering the context and understanding the test objectives and risk including selecting test approaches, estimating the time, effort cost of testing, acquiring resources, defining test levels, cycles and planning incident management
• Initiate the specification, preparation, implementation and execution of test, monitor the test results and check the exit criteria
• Adapt planning based on test results and progress, and take any action necessary to compensate for problems

Tasks of the Test Leader (2/2)

- Introduce metrics for measuring the quality of the testing and the product
- Decide what should be automated, to what degree and how it should be done
- Select tools to support testing
- Decide about the implementation of the test environment
- Write test summary reports based on the information gathered during testing
- Organize training for testers



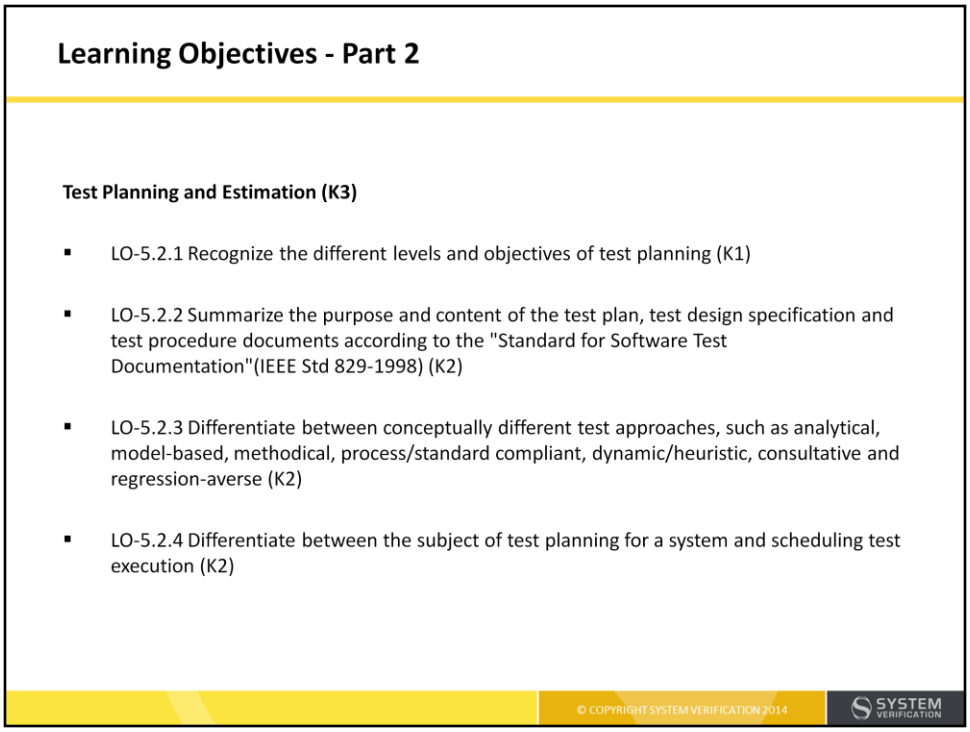
Cont.
• Set-up adequate configuration management of testware for traceability
• Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
• Decide what should be automated, to what degree and how it should be done
• Select tools to support testing and organize any training in tool use for testers
• Decide about the implementation of the test environment
• Write test summary reports based on the information gathered during testing



Typical tasks for a tester may include:
• Review and contribute to test plans
• Analyse, review and assess user requirements, specifications and models for testability
• Create test specifications
• Set up the test environment
• Prepare and acquire test data
• Implement tests on all test levels, execute and log the tests, evaluate the results and document the deviations from expected results
• Use test administration or management tools and test monitoring tools as required
• Automate tests (may be supported by a developer or a test automation expert)
• Measure performance of components and systems
• Review tests developed by others

Tasks of the Test Leader and Tester
People who work with test activities such as analysis, design, specific test types or automation may be specialist in these roles. Different people may take the role as a tester, keeping some degree of independence, depending on the test level and the risks related to the product and/or the project.

At the component and integration level typical testers could be developers, at the acceptance test level it would be business experts and users, and for operational acceptance testing the operators would be the testers.



Time
Intended study time is 40 minutes

Terms
test approach, test strategy

Learning Objectives - Part 2

Test Planning and Estimation (K3)

- LO-5.2.5 Write a test execution schedule for a given set of test cases, considering prioritization, and technical and logical dependencies (K3)
- LO-5.2.6 List test preparation and execution activities that should be considered during test planning (K1)
- LO-5.2.7 Recall typical factors that influence the effort related to testing (K1)
- LO-5.2.8 Differentiate between two conceptually different estimation approaches: the metrics-based approach and the expert-based approach (K2)
- LO-5.2.9 Recognize/justify adequate entry and exit criteria for specific test levels and groups of test cases (e.g. for integration testing, acceptance testing or test cases for usability testing) (K2)

Test Planning

- Planning is influenced by a number of things:
  - The test policy of the organization
  - The scope of testing
  - Objectives
  - Risks
  - Constraints
  - Criticality
  - Testability
  - Availability of resources
- Planning is not magic
  - Based on estimates and improved or enhanced over time

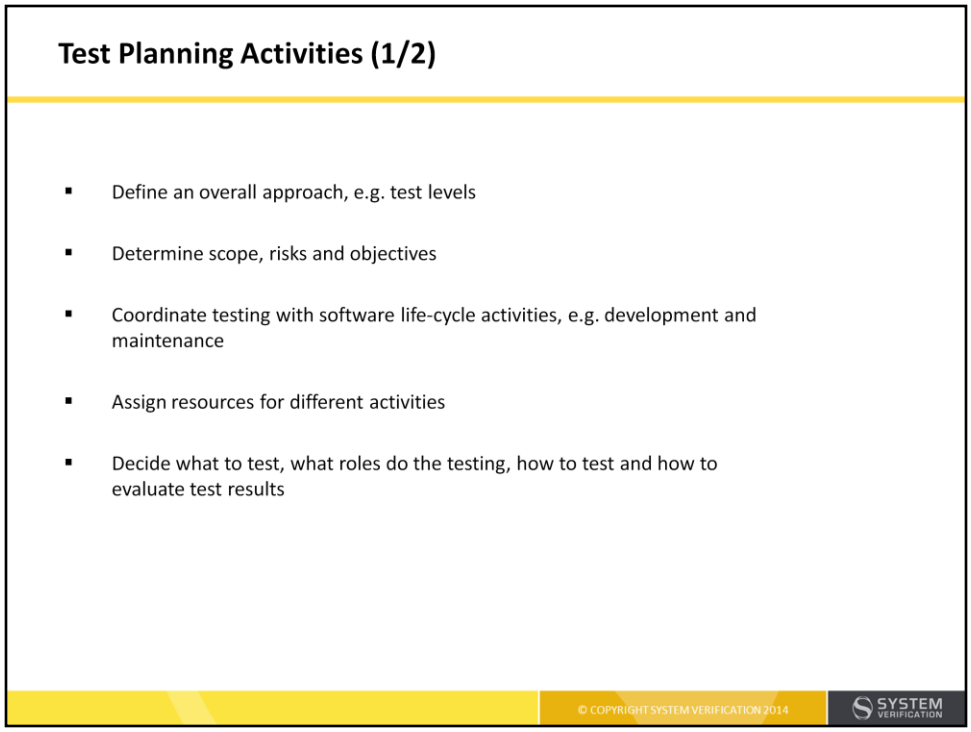
Planning is influenced by the test policy of the organization, the scope of testing, objectives, risks, constraints, criticality, testability and the availability of resources.

As the project and planning progresses and the more information is available, the more detail can be included in the plan.

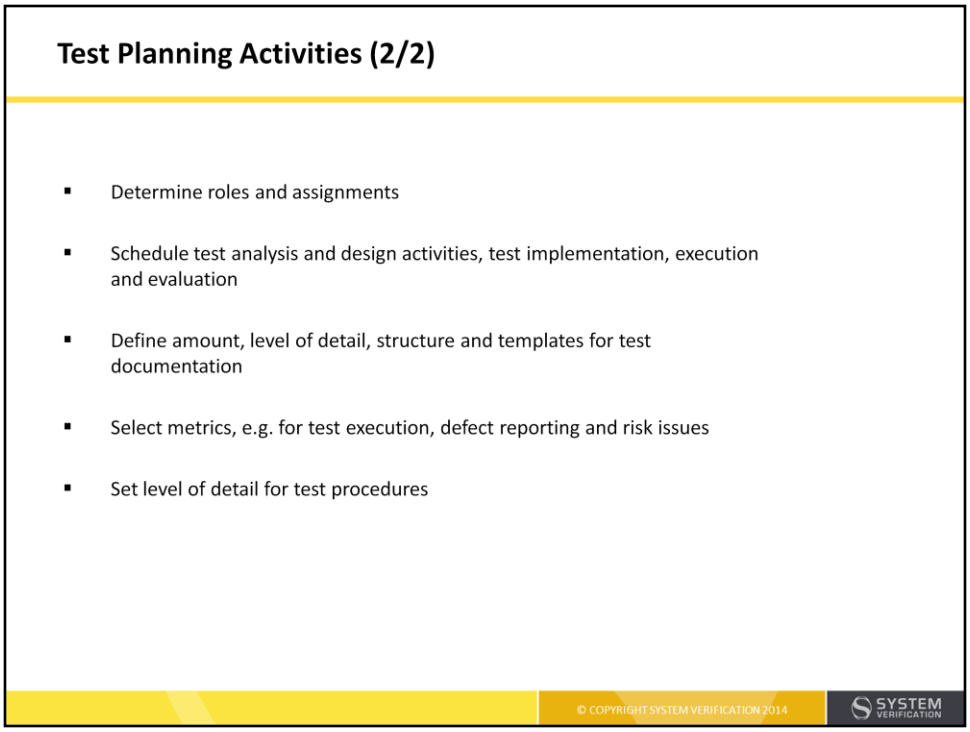
Test planning is a continuous activity and is performed in all life cycle processes and activities. Feedback from test activities is used to identify changing risk so that planning can be adjusted.

Test Planning

- Planning may be documented in a master test plan, and in separate test plans for different test levels, such as system testing and acceptance testing
- Templates are defined in 'Standard for Software Test Documentation' (IEEE 829)



Test planning activities may include:
• Defining a overall approach for the project, including the definition of test levels and entry and exit criteria
• Determining the scope, risks and objectives of testing
• Integrating and coordinating testing activities with software lifecycle activities, e.g. acquisition, supply, development, operation and maintenance
• Assigning resources for the different activities defined
• Making decisions about what to test, what roles will perform the test activities, how the activities should be done, and how the test results will be evaluated



Test planning activities may include:
• Determine roles and assignments to the project
• Scheduling test analysis and design activities, test implementation, execution and evaluation
• Defining the amount, level of detail, structure and templates for the test documentation
• Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues
• Setting the level of detail for test procedures in order to provide enough information to support reproducible test preparation and execution

Set Entry Criteria

- Entry criteria define when to start testing, for example:
  - Test environment availability and readiness
  - Test tool readiness in the test environment
  - Availability of testable code and test data

Entry criteria define when to start testing such as the beginning of the test level or when a set of test is ready for execution. Typically entry criteria may cover the following:

- Test environment availability and readiness
- Test tool readiness in the test environment
- Availability of testable code and test data

Set Exit Criteria

- Exit criteria define when to stop testing, for example:
  - Code/functionality/risk coverage
  - Defect density and reliability
  - Cost (budget)
  - Time schedule
  - Risks

It is not possible to find all bugs and defects in a product, because you cannot test everything. This is one reason why an exit criteria is needed. The exit criteria define when to stop testing such as the end of the test level or when set of tests has achieved a specific goal.

Typical exit criteria may cover the following:
• Thoroughness measures, for example, coverage of code, functionality or risk
• Estimates of defect density or reliability measures
• Cost
• Schedules such as those based on time to market
• Residual risks, for example, defects not fixed or lack of test coverage in certain areas

Test Estimation

- Planning includes estimating how much time and resources are needed
- Metrics-based
  - Former/similar projects or typical values
- Expert-based
  - Expert estimates
- Factors affecting planning
  - Characteristics of the product
  - Characteristics of the development process
  - Outcome of the testing

Because time and resources are limited, planning involves estimating how much test effort will be needed. Two approaches for estimation of test effort are:
• The metrics-based approach - Estimating the testing effort based on metrics of former or similar projects or based on typical values
• The expert-based approach - Estimating the tasks based on estimates made by the owner of the tasks or by experts

Once the test effort is estimated, resources can be identified and a schedule can be created.

The testing effort may depend on a number of factors that can be organized into three main groups.

Characteristics of the product
This includes the quality of the specification and other information used for test models (i.e. the test basis), the size of the product, the complexity of the problem domain, the requirements for reliability and security and the requirements for documentation

Characteristics of the development process
This includes the stability of the organization, tools used, test process, skills of the people involved and time pressure

The outcome of testing
This includes the number of defects and the amount of rework required. Used on ongoing projects which requires re-scheduling

Test Strategy and Test Approach

- The test approach is the implementation of the test strategy in the project
- It is the starting point for:
  - Planning test process
  - Selecting test design techniques and test types
  - Defining entry/exit criteria

The test approach is the implementation of the test strategy for a specific project. The approach is defined in the test plans and test design.

Typically includes the decisions made based on the project's goal and risk assessment. It is the starting point for:
• Planning the test process
• Selecting test design techniques and test types to be applied
• Defining entry and exit criteria

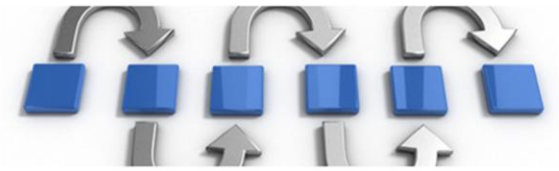
Test Strategy and Test Approach

- The selected approach is context-dependent and may consider:
  - Risks, hazards, safety
  - Available resources
  - Team member skills
  - Technology
  - Nature of the system
  - Regulations

The selected approach depends on the context and may consider risks, hazards and safety, available resources and skills, the technology, the nature of the system, test objectives and regulations. It should be noted that it is possible to combine different approaches.

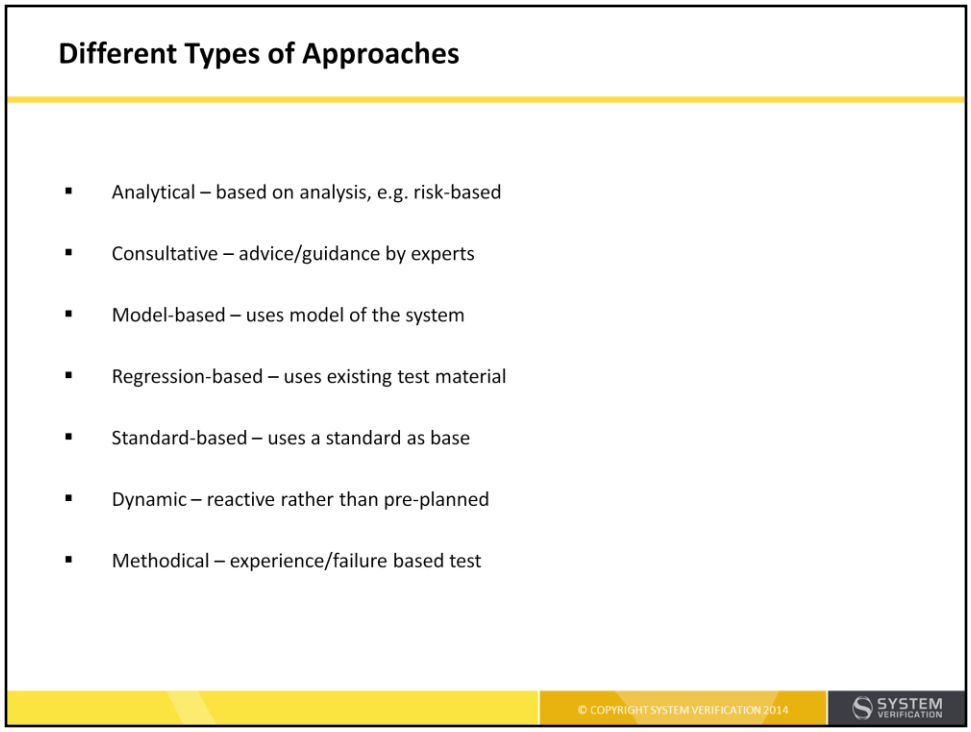
Classification of Test Approaches

- Test approaches and strategies can be classified depending on the point in time where the test design work is done
- Preventive approaches
  - Tests are designed as early as possible
- Reactive approaches
  - Test design starts after the software or system has been produced



Classifying test approaches or strategies is based on the point in time at which the bulk of the test design work is begun:
• Preventative approaches, where tests are designed as early as possible
• Reactive approaches, where test design comes after the software or system has been produced

We can base our testing strategy on a variety of factors - risks, standards and methodologies.



Analytical
These approaches are based on some type of analysis. Risk-based testing is an example of an analytical approach where testing is directed to areas of greatest risk.

Consultative
Consultative approaches, such as those where test coverage is driven primarily by the advice and guidance of technology and/or business domain experts outside the test team.

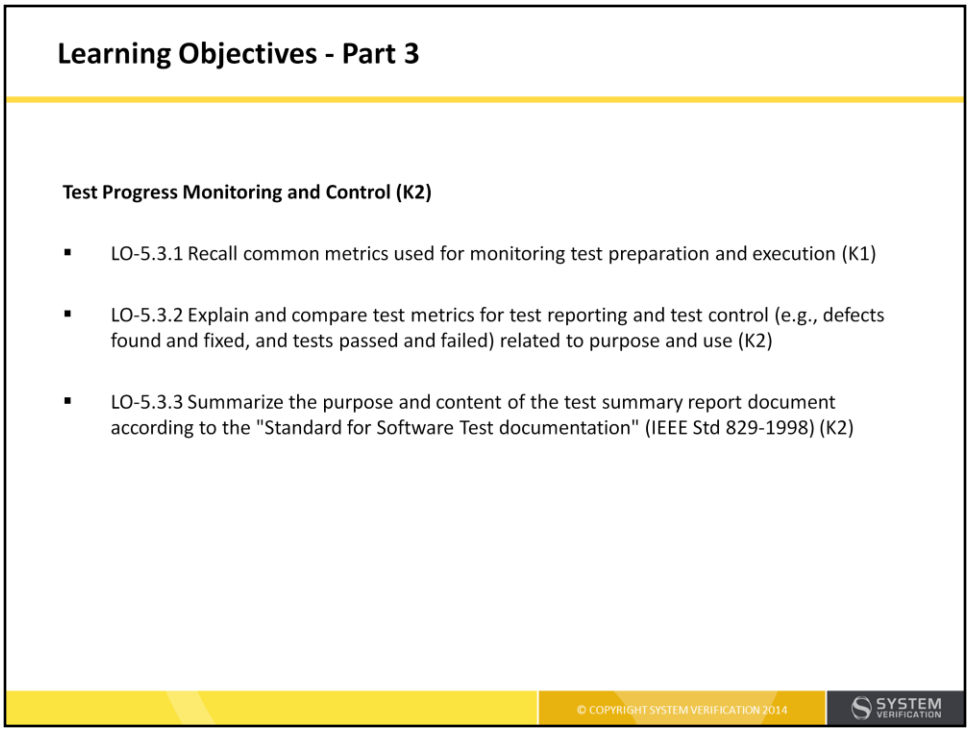
Model-based
Use a model of a system to determine test strategy. The model can be a reliability growth model, or an operational profile.
An example of model-based testing is using statistical information about failure rates to generate test cases to test reliability.

Regression-based
Those that include reuse of existing test material, extensive automation of functional regression tests, and standard test suites.

Process- or standard-compliant
Base their strategy on the framework of standards used by a specific industry. They can also use a process or methodology, such as various agile development methods.

Dynamic and heuristic
Such as exploratory testing, is more reactive to events than pre-planned and therefore execution and evaluation are concurrent tasks.

Methodical
Include failure-based approaches, such as error guessing and fault-attacks. They also include experienced-based testing, check-list based testing, and testing based on quality characteristics.

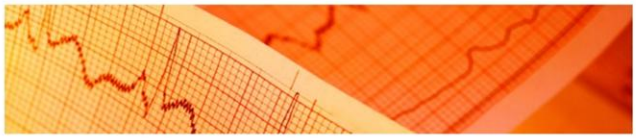


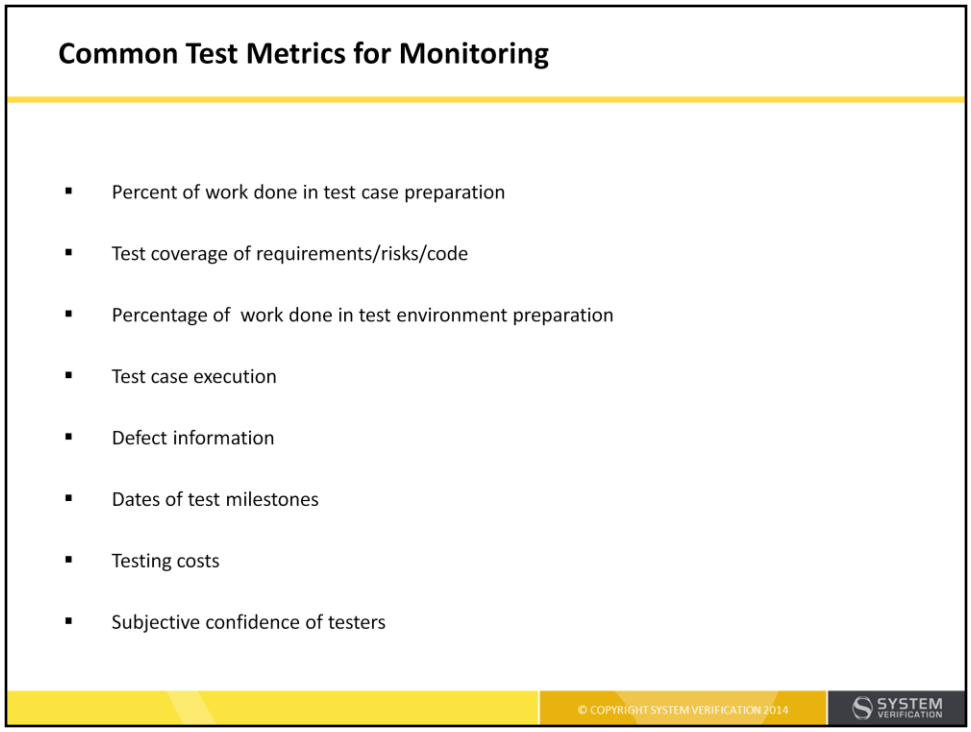
Time
Intended study time is 20 minutes

Terms
defect density, failure rate, test control, test monitoring, test summary report

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





Common test metrics for progress monitoring include:
• Percentage of work done in test case preparation (or percentage of planned test cases prepared)
• Test coverage of requirements, risks or code
• Percentage of work done in test environment preparation
• Test case execution (e.g. number of test cases run/not run, and test cases passed/failed)
• Defect information (e.g. defect density, defects found and fixed, failure rate, and re-test results)
• Dates of test milestones
• Testing costs, including the cost compared to the benefit of finding the next defect or to run the next test
• Subjective confidence of testers in the product

SYSTEM VERIFICATION

Test Control

- Test control describes guiding or corrective actions taken after information or metrics have been gathered, for example:
  - Deciding after test monitoring
  - Reprioritizing when an identified risk occurs, e.g. late delivery
  - Changing schedule due to availability of test environments

Test control describes any guiding or corrective action taken as a result of information and metrics gathered and reported. Actions may cover any test activity and may affect any other software life cycle activity or task.

Examples of test control actions are:
• Making decisions based on information from test monitoring
• Reprioritizing test when an identified risk occurs, e.g. late delivery of a software build
• Changing the test schedule due to availability on unavailability of a test environment
• Setting an entry criterion requiring fixes to have been re-tested (conformation tested) by a developer before accepting them into a build

Test Reporting

- At the end of a test level a test summary report is written containing
  - What happened during the test period
  - Information and metrics to support recommendations and decisions for future actions, e.g. defects found/remaining, outstanding risks, level of confidence in the software

At the end of a test level, a test summary report is created with information about the testing endeavour. This report includes:
• What happened during a period of testing, such as dates when exit criteria were met
• Analysed information and metrics to support recommendations and decisions about future actions, for example, assessment of defects remaining, the economic benefit of continued testing, outstanding risks, and the level of confidence in tested software

A test summary report should contain metrics collected during and 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 taken
• The effectiveness of the testing with respect to its objectives

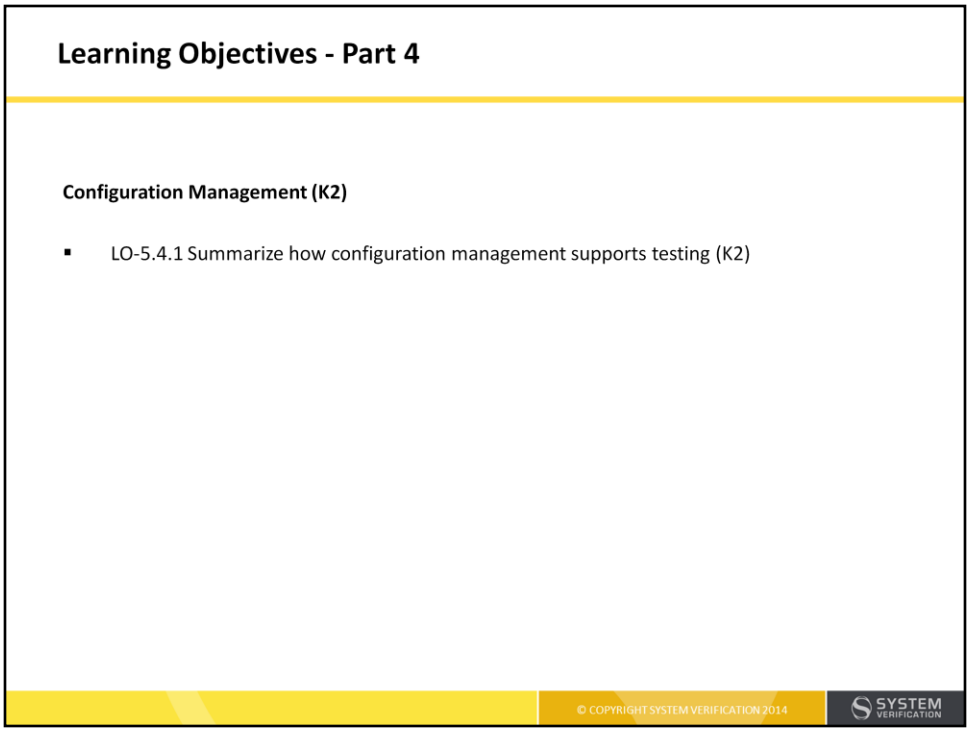
Test Summary Report

- The outline of a test summary report is given in 'Standard for Software Test Documentation' (IEEE Std 829)
- Contains important information uncovered by the accomplished
  - Assessment of the quality of the testing effort
  - Quality of the software system under test
  - Statistics derived from incident reports

The outline of a test summary report is given in 'Standard for Software Test Documentation' (IEEE Std 829).

It contains important information uncovered by the tests accomplished, and including assessments of the quality of the testing effort, the quality of the software system under test, and statistics derived from incident reports.

The report also records what testing was done and how long it took, in order to improve any future test planning.

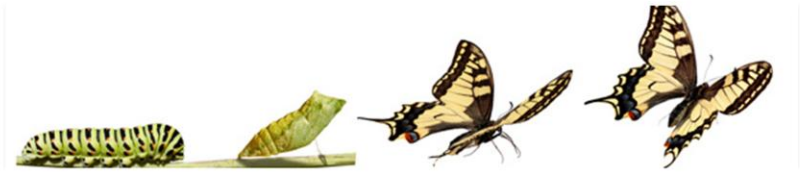


Time
Intended study time is 10 minutes

Terms
configuration management, version control

Configuration Management

- The purpose of configuration management is to maintain the integrity and relation of the products (data, documents, components, etc.) throughout the project and product life cycle
- All items need to be version controlled, checked for traceability and unambiguously referenced in test documentation



The purpose of configuration management is to establish and maintain the integrity and relation of the products, such as components, data and documentation, of the software or system through the project and product life cycle.

For testing, configuration management could involve ensuring that all items of testware are identified, version controlled, tracked for changes, related to each other and related to development items so that traceability can be maintained throughout the test process. It could also involve to ensure that all identified documents and software items are referenced unambiguously in test documentation.

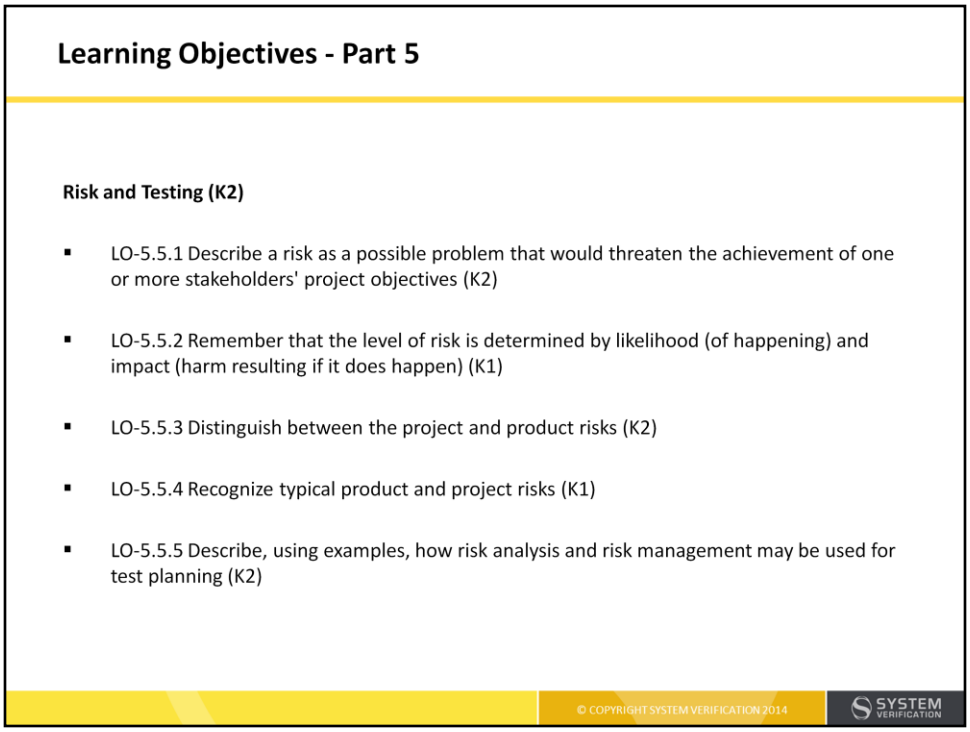
Configuration Management

- Examples of artefacts that could be handled by configuration management are:
  - Test plans
  - Test cases
  - Test data
  - Test results
  - Test tools
  - Test environments

Configuration management helps the tester to uniquely identify and to reproduce the tested items, test documents, the tests and test harnesses.

During the test planning the configuration management procedures and infrastructure, including tools, should be chosen, documented and implemented. Example of artefacts that could be handled by configuration management are:

- Test plans
- Test cases
- Test data
- Results
- Tools
- Test environments



Time
Intended study time is 30 minutes

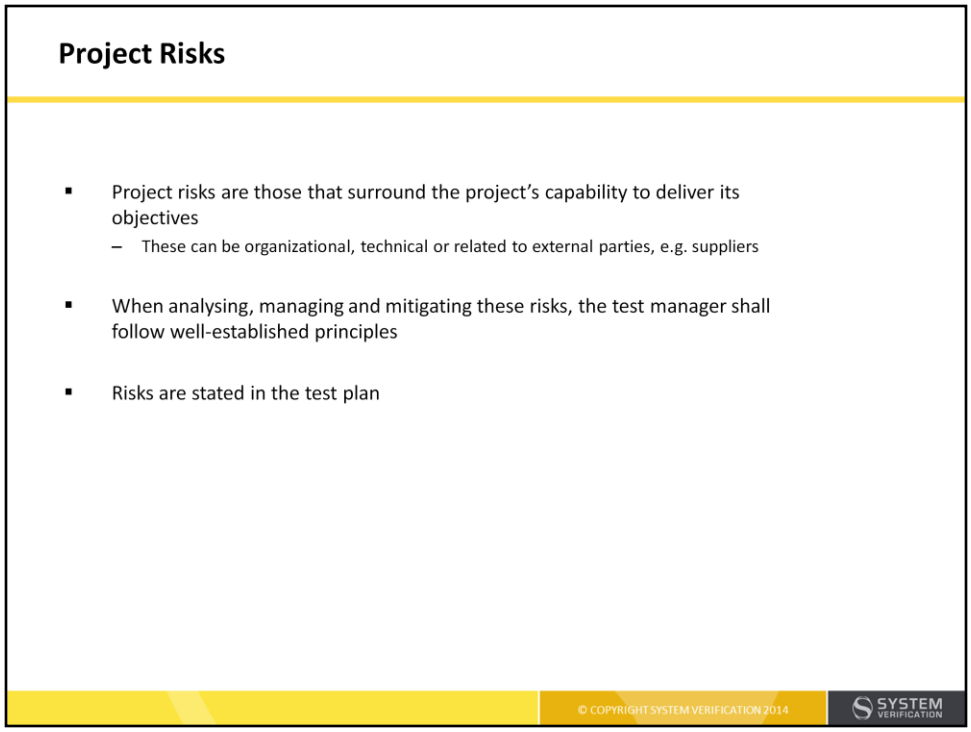
Terms
product risk, project risk, risk, risk-based testing

What is Risk?

- Risk can be defined as the chance of an event, hazard, threat or situation occurring and resulting in undesirable consequences or a potential problem
- The level of risk will be determined by the likelihood of an adverse event happening and the impact, i.e. the harm resulting from that event

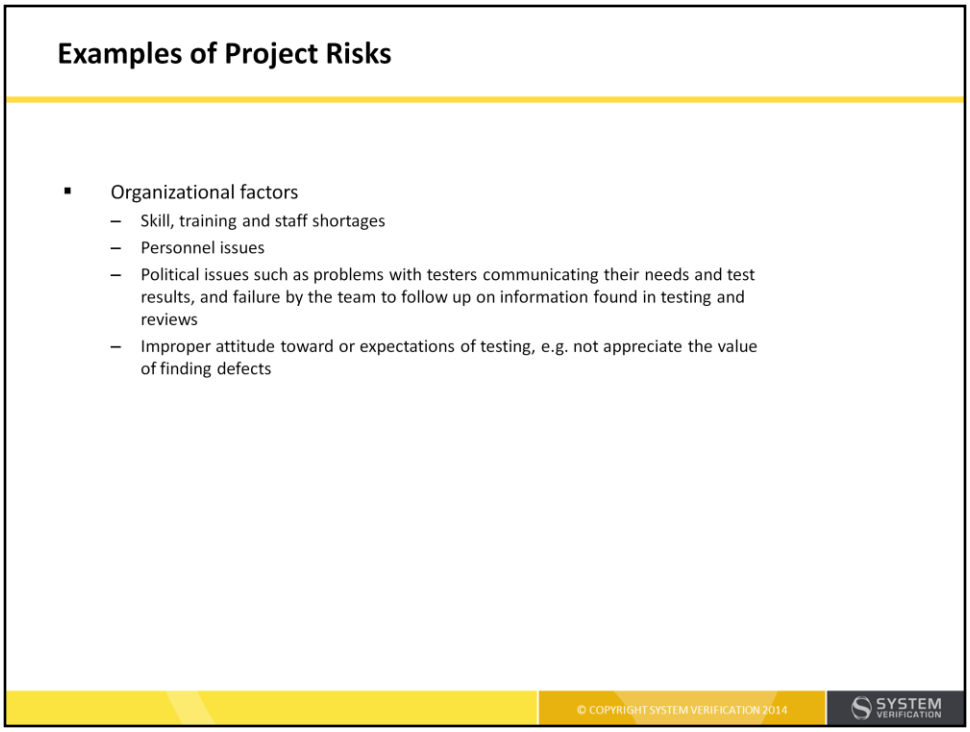
Risk can be defined as the chance of an event, hazard, threat or situation occurring and resulting in undesirable consequences or a potential problem.

The level of risk will be determined by the likelihood of an adverse event happening and the impact, i.e. the harm resulting from that event.

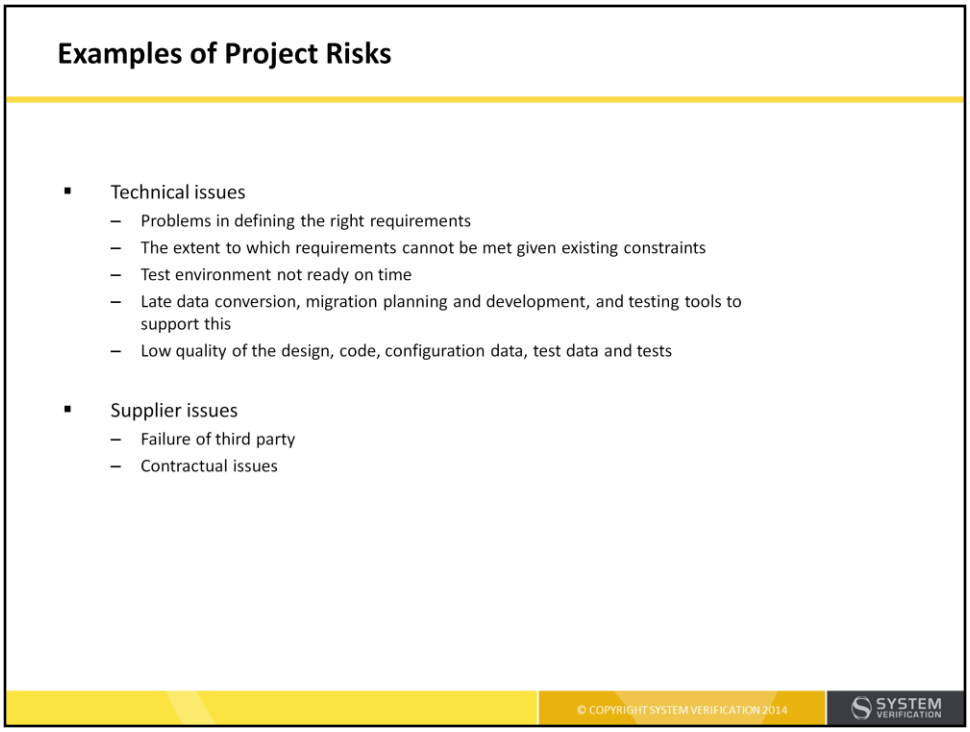


Project risks are the those that surround the project's capability to deliver its objectives. These can be organizational, technical or related to external parties, such as suppliers.

When analyzing, managing and mitigating these risks, the test manager shall follow well-established project management principles. In IEEE Std 829 the test plan outline requires risks and contingencies to be stated.



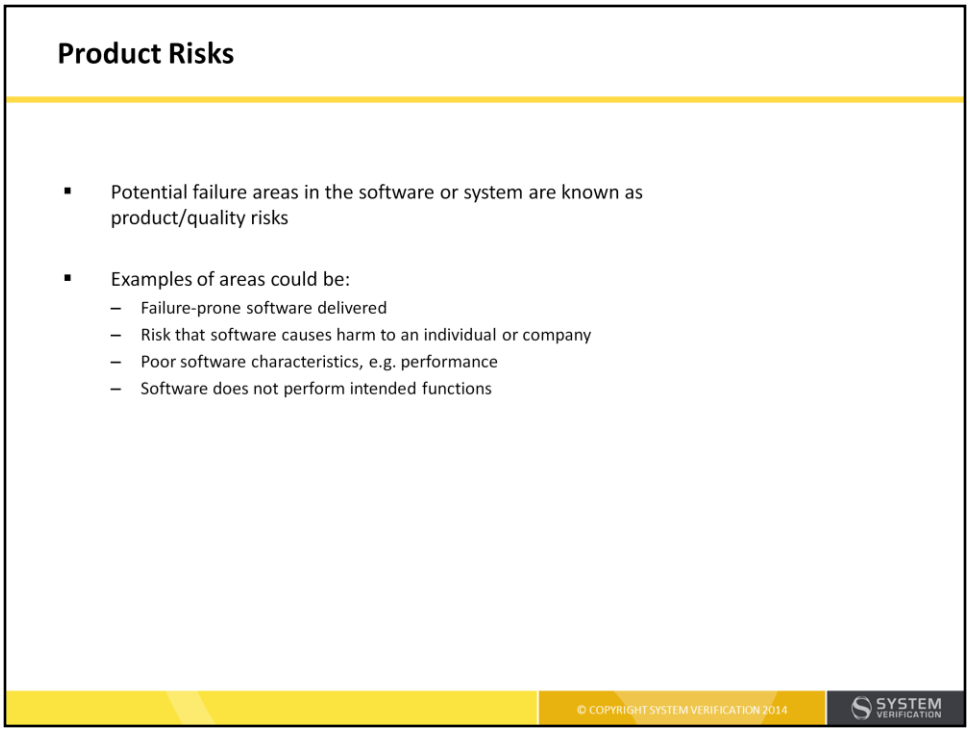
Organizational factors
• Skill, training and staff shortages
• Personnel issues
• Political issues such as problems with testers communicating their needs and test results, and failure by the team to follow up on information found in testing and reviews
• Improper attitude toward or expectations of testing, e.g. not appreciate the value of finding defects



Technical issues
• Problems in defining the right requirements
• The extent to which requirements cannot be met given existing constraints
• Test environment not ready on time
• Late data conversion, migration planning and development, and testing tools to support this
• Low quality of the design, code, configuration data, test data and tests

Supplier issues
• Failure of third party
• Contractual issues (due to existing project constraints such as time plan and budget, product requirements cannot always be met)

Responsible person, test managers or test leaders should state these risks and contingencies when planning for testing.



Potential failure areas in the software or system are known as product risks, as they are a risk to the quality of the product. Products risks are a special type of risk to the success of a project.

Testing as a risk-control activity provides feedback about residual risk by measuring the effectiveness of critical defect removal and contingency plans.

Risks are used to decide where to start testing and where to test more, i.e. test is used to reduce the risk of an adverse effect occurring or to reduce the impact of the adverse effect.

Example of potential failure areas could be:
• Failure-prone software delivered
• The potential that the software/hardware could cause harm to an individual or company
• Poor software characteristics, .e.g. reliability, usability and performance
• Poor data integrity and quality, e.g. migration issues, conversion problems and violation of data standards
• Software that does not perform its intended functions

Using a Risk-Based Approach

- Identified risks may be used to:
  - Determine what techniques to use
  - 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 non-testing activities could be employed to reduce risk, e.g. providing training to inexperienced testers

A risk-based approach to testing provides proactive opportunities to reduce the levels of product risk, starting in the initial stages of a project.

It involves the identification of product risks and their use in guiding test planning and control, specification, preparation and execution of tests.

In a risk-based approach the risks identified may be used to:
• Determine what techniques to use
• 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 non-testing activities could be employed to reduce risk, e.g. providing training to inexperienced testers

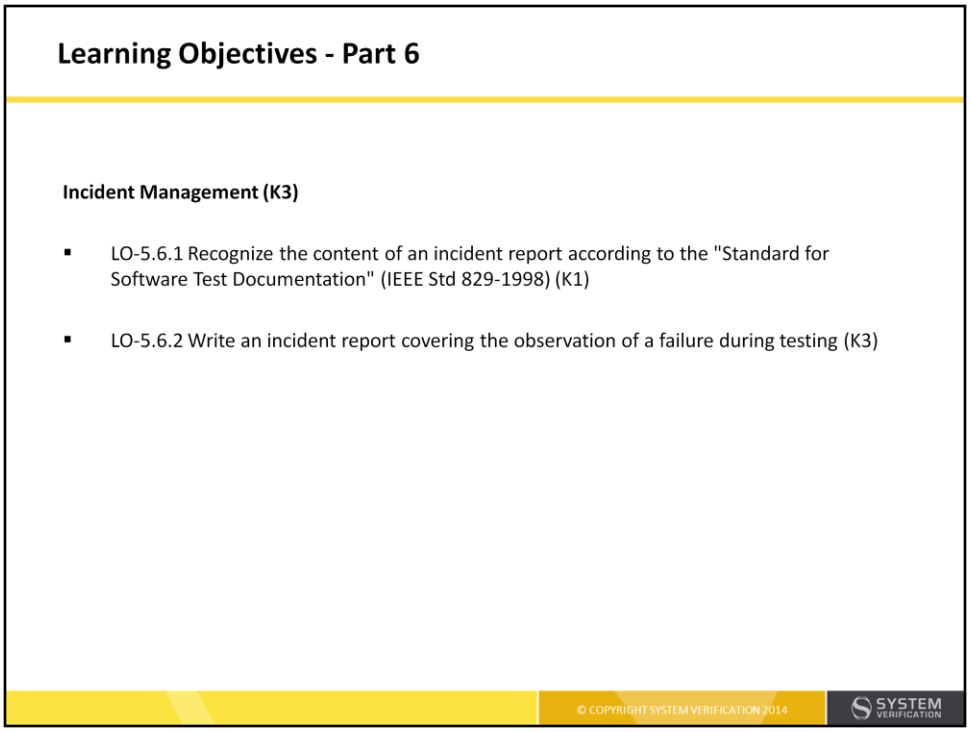
Using a Risk-Based Approach

- To ensure that the chance of a product failure is minimized, risk management activities provide a disciplined approach to:
  - Assess and re-assess on a regular basis what can go wrong
  - Determine what risks are important to deal with
  - Implement actions to deal with those risks

Risk-based testing draws on the collective knowledge and insight of the project stakeholders to determine the risks and the levels of testing required to address those risks.

To ensure that the chance of a product failure is minimized, risk management activities provide a disciplined approach to:
• Assess and re-assess on a regular basis what can go wrong
• Determine what risks are important to deal with
• Implement actions to deal with those risks

In addition, testing may support the identification of new risks, may help to determine what risks should be reduced, and may lower uncertainty about risks.



Time
Intended study time is 40 minutes

Terms
incident logging, incident management, incident report

Incident Management

- Any discrepancies between actual and expected results need to be logged as incidents
- An incident is investigated and may be a defect
- Incidents need to be tracked
- They may be raised at any time during the life-cycle, for both code and documentation

Any discrepancies between actual and expected outcomes need to be logged as incidents. An incident shall be investigated and may turn out to be a defect. Appropriate actions to dispose incidents and defects shall be defined.

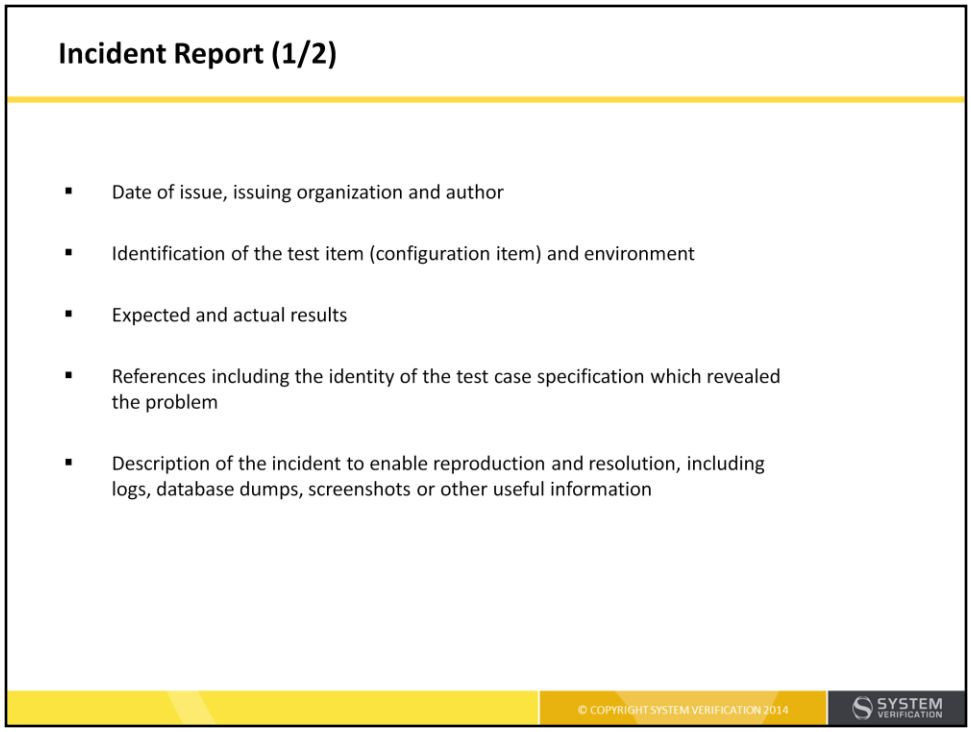
Managing incidents means to track them from discovery and classification to correction and conformation of the solution. In order to manage all incidents, an incident management process and rules for classification should be established.

Incidents may be raised during development, review, testing or use of a software product. They may be raised for issues in code or the working system, or in any type of documentation, e.g. requirements, development documents, test documents and user information.

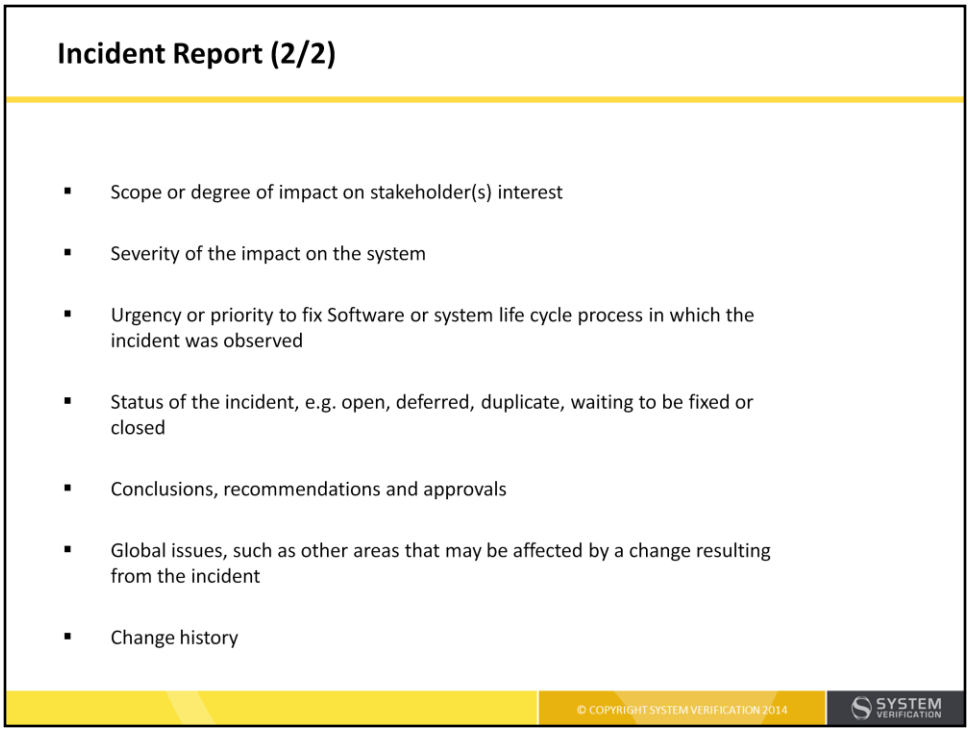
Objectives of Incident Management

- Provide developers and other parties with feedback about the problem to enable identification, isolation and correction as necessary
- Provide test leaders with a means of tracking the quality of the system under test and the progress of the testing
- Provide ideas for test process improvement

Incident reports have the following objectives:
• Provide developers and other parties with feedback about the problem to enable identification, isolation and correction as necessary
• Provide test leaders with a means of tracking the quality of the system under test and the progress of the testing
• Provide ideas for test process improvement

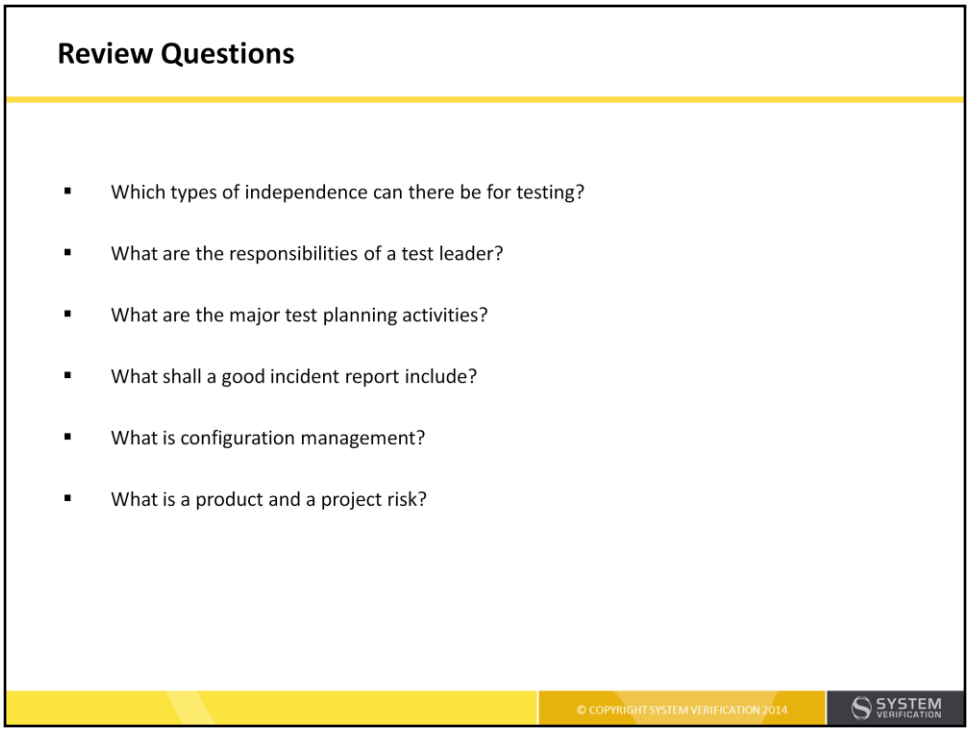


Details of the incident report may include:
• Date of issue, issuing organization and author
• Expected and actual results
• Identification of the test item (configuration item) and environment
• Software or system life cycle process in which the incident was observed
• Description of the incident to enable reproduction and resolution, including logs, database dumps, screenshots or other useful information
• Scope or degree of impact on stakeholder(s) interest
• Severity of the impact on the system



Cont.
• Urgency or priority to fix
• Status of the incident, e.g. open, deferred, duplicate, waiting to be fixed or closed
• Conclusions, recommendations and approvals
• Global issues, such as other areas that may be affected by a change resulting from the incident
• Change history
• References including the identity of the test case specification the revealed the problem

The incident report is also covered in the 'Standard for Software Test Documentation' IEEE Std 829-1998.



Summary

In this chapter we have looked into test organization and the members of the test team(s). Planning and estimation are important parts of the test work that has been covered and how to determine entry and exit criterion. It is also necessary to follow up on the progress and summarize the progress to be able to act upon deviations from initial plans and unexpected events.

During the testing we identify defects and failures. The importance of writing incident reports for these defects and failures have been covered as well as what kind of information is needed. You have also learnt about the importance to keep track of documents, code and other important information, their versions and relationships, i.e. configurations management. Finally, different risks that you may encounter during a project have been presented and how to deal with project and product risks.

TERMS FROM CHAPTER 5

5.1 TEST ORGANIZATION

Tester
Test leader
Test manager

5.2 TEST PLANNING AND ESTIMATION

Test approach
Test strategy

5.3 TEST PROGRESS MONITORING AND CONTROL

Defect density
Failure rate
Test control
Test monitoring
Test summary report

5.4 CONFIGURATION MANAGEMENT

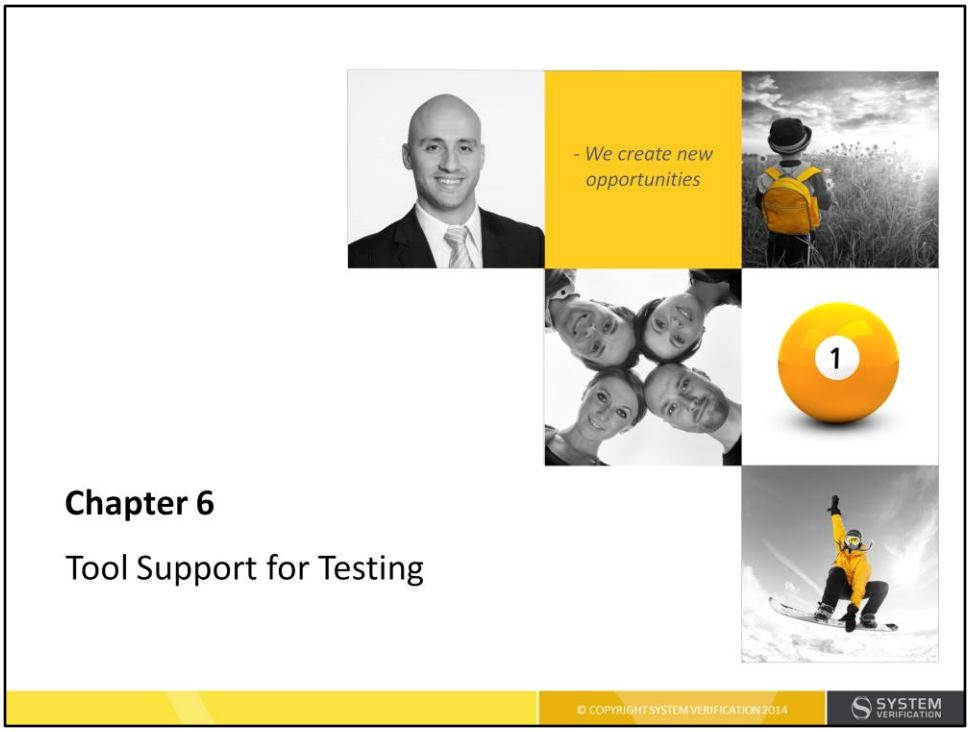
Configuration management
Version control

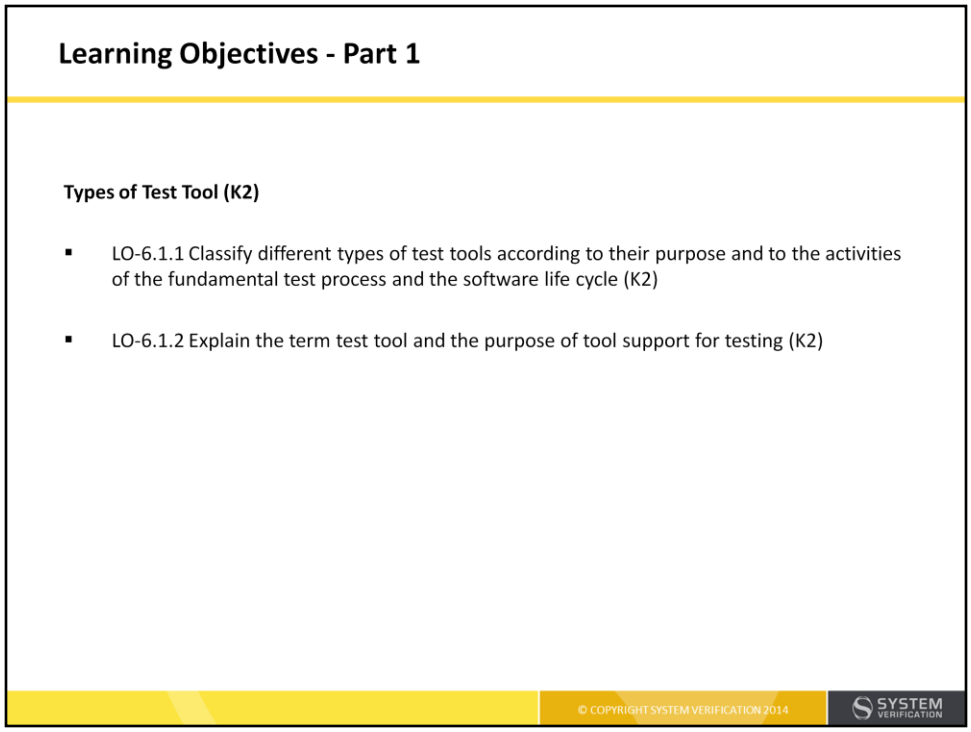
5.5 RISK AND TESTING

Product risk
Project risk
Risk
Risk-based testing

5.6 INCIDENT MANAGEMENT

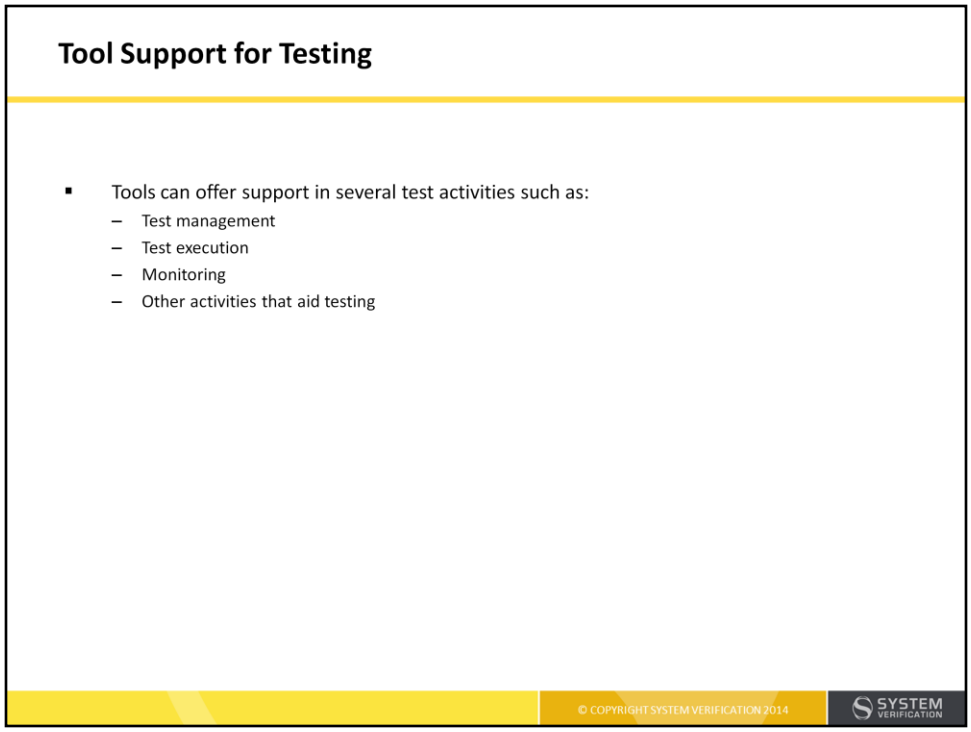
Incident logging
Incident management
Incident report



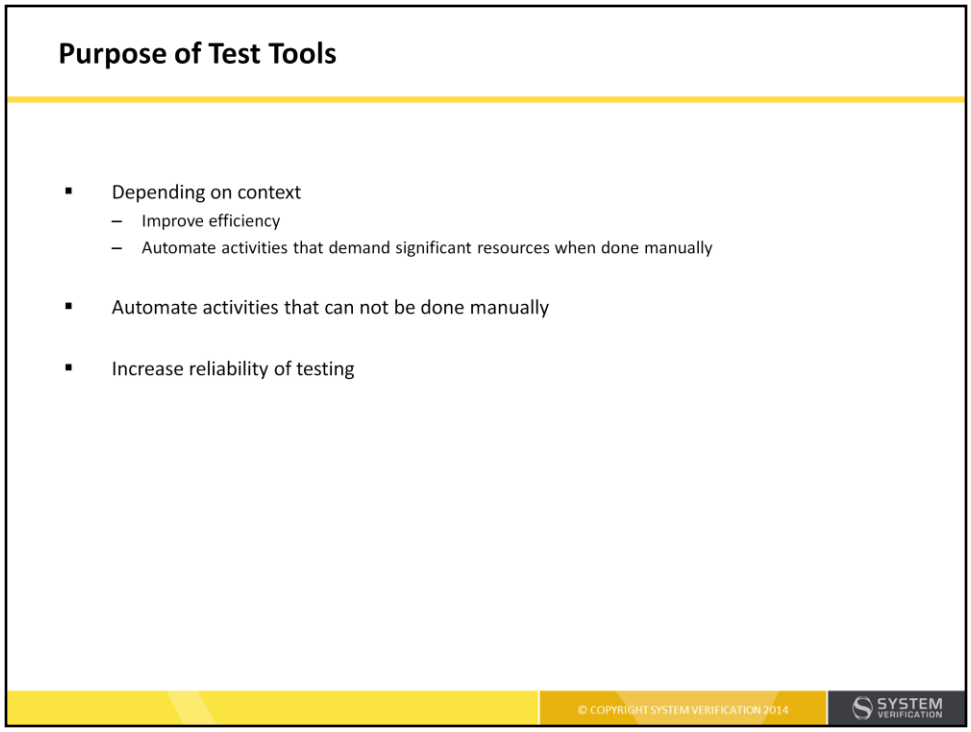


Time
Intended study time is 45 minutes

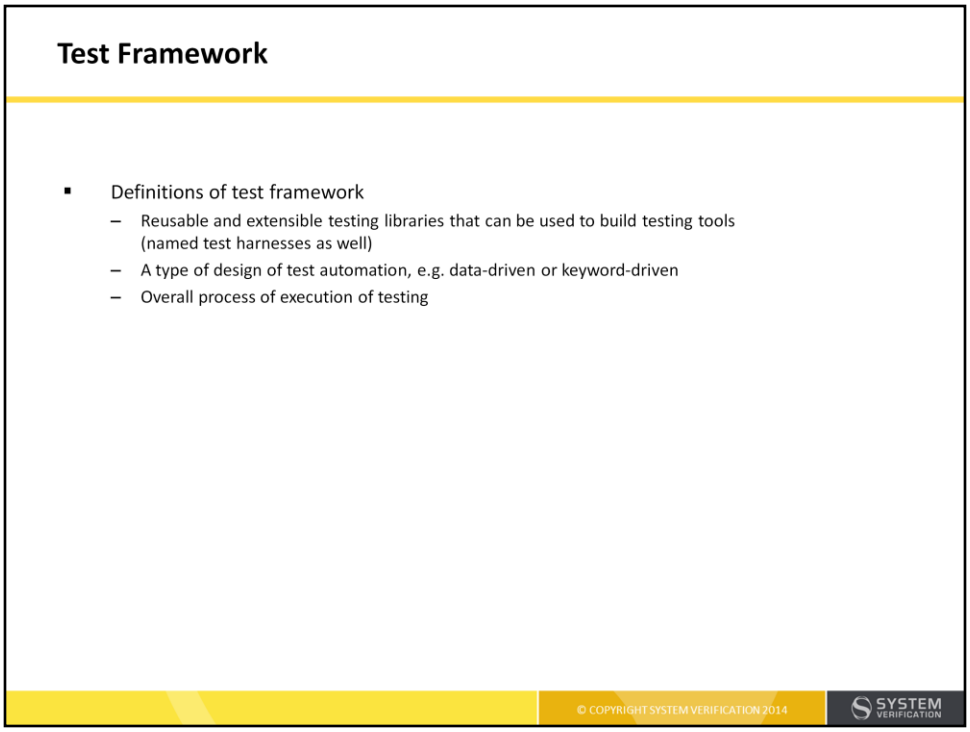
Terms
configuration management tool, coverage tool, debugging tool, dynamic analysis tool, incident management tool, load testing tool, modelling tool, monitoring tool, performance testing tool, probe effect, requirements management tool, review tool, security tool, static analysis tool, stress testing tool, test comparator, test data preparation tool, test design tool, test harness, test execution tool, test management tool, unit test framework tool



Tool support for testing can have one or more of the following purposes depending on the context:
• Improve efficiency of test activities by automating repetitive tasks or supporting manual test activities, e.g. planning, design, reporting and monitoring
• Automate activities that require significant resources when carried out manually, e.g. static testing
• Automate activities that cannot be executed manually, e.g. large scale performance testing
• Increase reliability of testing, e.g. automating large data comparisons or simulating behaviour



Test tools can be used for several different activities that supports daily work for a tester. Example of activities are:
• Execution tools, test data generation tools and result comparison tools are examples of tools that are used directly for testing.
• Test management tools used to manage tests, results, data, requirements, incidents, defects etc. They could also support reporting and monitoring of test execution.
• Reconnaissance tools used for exploration, for example, monitor the file activity for an application.
• Common tools that aids the testing, e.g. a spreadsheet that contains information about the tests or testing.



The term Test Frameworks is used frequently in the industry and has at least three meanings (in this course two first meanings are used):

- Reusable and extensible testing libraries that can be used to build testing tools (called test harnesses as well)
- A type of design of test automation, e.g. data-driven or keyword-driven
- Overall process of execution of testing

Classification of Test Tools

- Test tools can be classified depending on purpose, technology, manufacturer, licensing etc.
- Some of them support only one activity, others offer support for multiple activities. Tools can also be designed to work together
- Test tools can also be classified as intrusive or non-intrusive
- Intrusive tools demand modifications to the test object, e.g. inserting extra code (instrumentation)
- Be aware of the probe effect

There are a number of tools that support different aspects of testing. Tools can be classified based on several criteria such as purpose, licensing, technology used and so forth. Testing tools can improve the efficiency of testing activities by automating repetitive tasks. Testing tools can also improve the reliability of testing.

Some tools clearly support one activity, others may support more than one activity, but are classified under the activity which they are most closely associated with. Tools designed to work together, from a single provider, may be bundled into one package.

There are also types of test tools which can be intrusive in that the tool itself can affect the actual outcome of the test. For example, the actual timing may be different due to the extra instructions that are executed by the tool. The measure for code coverage could also differ. This is called the probe effect.

Some tools offer support more appropriate for developers. This could be tools used during component and component integration testing.

Category 1: Test Management Tools

- Types of Test management tools
  - Test management
  - Requirements management
  - Incident management
  - Configuration management

Management tools apply to all test activities over the entire software life cycle. They can be grouped into four categories:
• Test management tools
• Requirement management tools
• Incident management tools (Defect tracking tools)
• Configuration management tools

Test Management Tools

- Test management
  - Managing test plans
  - Storing test cases and test procedures
  - Executing test cases
  - Tracking test results and incidents
  - Managing requirements
- Requirements management
  - Storing requirements
  - Providing identifiers to requirements
  - Warning for inconsistencies and untested requirements
  - Providing traceability

Test management tools
These tools provide interfaces for managing test plans, storing test cases and test procedures, executing test cases, tracking test results and defects, and managing requirements along with support for quantitative analysis and reporting of the test objects.

They also support tracing the test objects to requirement specifications and might have an independent version control capability or an interface to an external one.

Requirement management tools
Requirement management tools store requirement statements, related attributes (e.g. priority) and provide unique identifiers to the requirements. They can also give warning if there are inconsistencies or undefined (missing) requirements.

Requirement management tools enable individual tests to be traceable to requirements, functions and/or features.

Test Management Tools

- Incident management
  - Storing incident reports
  - Supporting life-cycle of incident reports
  - Help prioritizing reports
  - Help assigning actions
  - Tracking status
  - Extracting reports and statistics
- Configuration management
  - Storing testware and related software
  - Enabling traceability between testware and work products
  - Particularly useful when developing on multiple configurations

Incident management tools
Incident management tools store and manage incident reports, i.e. defects, failures, change requests or perceived problems and anomalies, and help in managing the life cycle of incidents. Example of supported functionality are:
• Prioritization of incidents
• Assigning actions to people
• Tracking incident status (e.g. rejected, ready to be tested or deferred to next release)

These tools often provide support for statistical analysis.

Configuration management tools
Configuration management tools are not strictly testing tools but are necessary for storage and version management of testware and related software. These tools enable traceability between testware, software work products and product variants.

They are particularly useful when developing on more than one configuration of the hardware/software environment (e.g. for different operating system versions, libraries, compilers or browsers).

Category 2: Static Testing Tools

- Types of static testing tools
  - Review
  - Static analysis
  - Modelling

Static testing tools provide a cost effective way of finding defects at an early stage in the development process. They can be grouped into three categories:
• Review tools
• Static analysis tools
• Modelling tools

Static Testing Tools

- Review tools
  - Assisting in review process
  - Providing checklists
  - Checking metrics
  - Possibly also assisting in doing online reviews
- Static analysis tools
  - Enforcing of code standards
  - Analysis of structures and dependencies in code
  - Providing help for risk analysis through metrics, e.g. cyclomatic complexity

Review tools
Review tools assist the review processes, checklists and review guidelines. They are used to store and communicate review comments, reports on defects and efforts.

They may also support online reviews, which is useful if the team is geographically dispersed.

Static analysis tools
Static Analysis Tools support developers and testers in finding defects prior to dynamic testing.

The static analysis tools may support:
• Enforcement of coding standards
• Analysis of structures and dependencies in the code
• Help in planning and risk analysis through metrics for the code, e.g. complexity

Static Testing Tools

- Modelling tools
  - Validating models of the software, for example a database model
  - Creating test cases
  - Finding defects early (even before the software has been implemented)

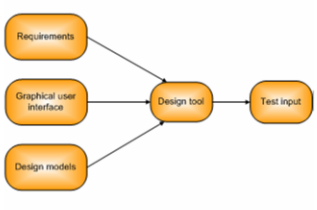
Modelling tools
Modelling tools are used to validate models of the software. For example, it may find defects and inconsistencies in a physical data model (PDM) for a relational database.

Other modelling tools may find defects in a state model or an object model. These tools can often aid in generating some test cases based on the model.

The major benefit of static analysis tools and modelling tools is the cost effectiveness of finding defects earlier in the development process.

Category 3: Test Specification Tools

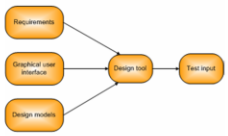
- Types of test specification tools
  - Test design
  - Test data preparation



Test specification tools are used for generating inputs and tests as well as preparing data for manipulating databases. They can be grouped into two categories:
• Test design tools
• Test data preparation tools

Test Design Tools

- Test design tools
  - Generating test inputs from
    - Requirements
    - Models
    - Code
    - GUI
  - Generating expected results (if an oracle exists)



- Test data preparation tools
  - Extracting existing data from files/databases
  - Modifying data records to avoid using real data (for data protection)
  - Generating new data for testing
  - Generating large volumes of data, e.g. for volume testing

Test design tools
Test design tools generate test inputs or executable tests from requirements, graphical user interface, design models (state, data or object) or code. This type of tool may generate expected outcomes as well.

The generated tests from a state or object model are useful but rarely sufficient for verifying all aspects of the software or system.

Test data preparation tools
Test data preparation tools manipulate databases, files or data transmissions to set up test data to be used during the execution of tests to ensure security through data anonymity.

Category 4: Test Execution/Logging Tools

- Types of test execution/logging tools
  - Test execution
  - Test harness/unit test framework
  - Test comparator
  - Coverage measurement
  - Security testing

Tools that support test execution and logging can provide a number of different features. Some aim at automation of the execution while others are specialized in evaluating security characteristics of software. They can be grouped into five categories:

- Test execution tools
- Test harness/unit test framework
- Test comparators
- Coverage measurement tools
- Security testing tools

Test Execution/Logging Tools

- Test execution tools
  - Executing automated tests (or semi-automated)
  - Recording tests (capture and playback)
  - Using a scripting language
  - Logging test results

- Unit test framework tools
  - Simulating the environment for running the software
  - Usually component level
  - Stubs and drivers
  - Providing a predictable environment in which failures can be detected

Test execution tools
Test execution tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language. This makes it possible to repeat the test with different data. Normally they also provide a test log for each run.

Test execution tools can also be used to record tests, they may be referred to as capture-playback tools. These tools usually support scripting language or GUI-based configuration for parameterization of data and other customization in the tests.

Unit test framework tools
A test harness, or unit testing framework, is used to simulate the environment in which a test object will run. It could either be a component or a part of a system.

This may be done either because other components of that environment are not yet available and are therefore replaced by stubs and/or drivers, or simply to provide a predictable and controllable environment in which any faults can be localized to the object under test.

Test Execution/Logging Tools

- Test comparator tools
  - Determining differences between files, databases or test results
  - Can be done during runtime or post-execution
  - May use an oracle, especially if automated
- Coverage measurements tools
  - Measuring the percentage of code structures tested, e.g. statements or decisions
  - Can be either intrusive or non-intrusive

Test comparators
Test execution tools can include a comparator. Comparators determine differences between files, databases or test results.

Test execution tools typically include dynamic comparators, but post-execution comparison may be done by a separate comparison tool.

A test comparator may use a test oracle, especially if it is automated.

Coverage measurements tools
Code coverage tools measure the percentage of specific types of code structures that have been exercised (e.g. statements, branches or decisions, and module or function calls). These tools show how thoroughly the measured type of structures has been exercised by a set of tests.

Coverage measurement tools can be either intrusive or non-intrusive depending on the measurement techniques used, what is measured and the coding language.

Test Execution/Logging Tools

- Security testing tools
  - Evaluating security characteristics
  - Identifying threats, viruses, malware etc.
  - Detecting intrusions
  - Simulating attacks
  - Doing security checks

Security testing tools
Security Testing Tools are used to evaluate the security characteristics of software. This includes evaluating the ability of the software to protect data confidentiality, integrity, authentication, authorization, availability and non-repudiation.

Security tools are mostly focused on a particular technology, platform and a specific purpose.

Category 5: Performance and Monitoring Tools

- Types of test performance /monitoring tools
  - Dynamic analysis
  - Performance, load and stress testing
  - Monitoring

Tool support for performance and monitoring
Tools that support performance testing and monitoring can be grouped into three categories:
• Dynamic analysis tools
• Performance testing/load testing/stress testing tools
• Monitoring tools

Performance and Monitoring Tools

- Dynamic analysis tools
  - Detecting memory leaks
  - Detecting pointer problems (null pointers)
  - Detecting time dependencies
- Performance, Load and Stress Testing Tools
  - Simulate load of real users through load generators
  - Load testing determines what load the system can handle
  - Stress testing tests beyond the expected load

Dynamic analysis tools
Dynamic analysis tools find defects that are evident only when software is executing, such as time dependencies or memory leaks.

Dynamic analysis tools are typically used in component and component integration testing, and when testing middleware.

Performance/Load/Stress Testing tools
Performance testing tools monitor and report how a system behaves under a variety of simulated usage conditions. It could be the number of concurrent users, ramp-up patterns, frequency or relative percentage of transactions.

They simulate a load on an application, a database, or a system, e.g. a server. Performance testing tools are often based on automated repetitive execution of tests, controlled by certain parameters.

The simulation load is achieved by means of creating virtual users carrying out a selected set of transactions, spread across various test machines commonly known as load generators.

Performance and Monitoring Tools

- Monitoring tools
  - Measuring number of users, connections, transactions etc. in the system
  - Reporting and analyzing statistics and trends
  - Notifying administrator when limits are reached

Monitoring tools
Monitoring tools continuously analyze, verify and report on usage of specific system resources, and give warnings of possible service problems.

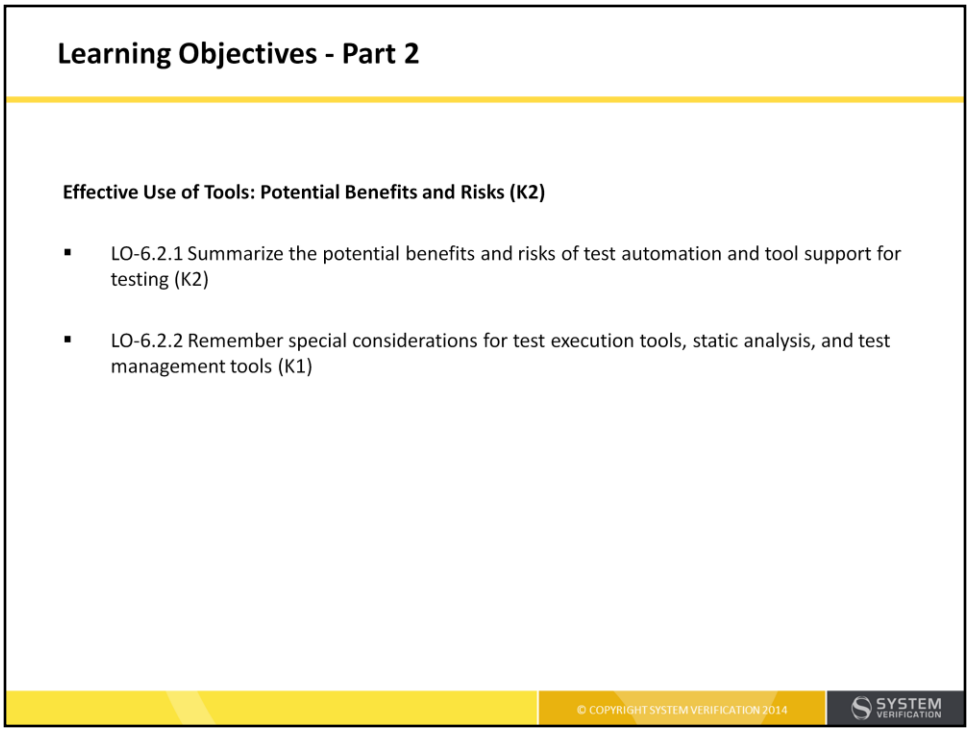
Category 6: Tools for specific Testing Needs

- Types of tools in this category
  - Data quality assessment
  - Usability testing
  - Word processors
  - Spreadsheets

Tool support for specific testing needs
Data is the centre of some projects such as data conversion/migration projects and applications like data warehouses and its attributes can vary in terms of criticality and volume.

In such contexts, tools for **Data Quality Assessment** need to be employed to review and verify the data conversion and migration rules to ensure that the processed data is correct, complete and complies to a pre-defined context-specific standard.

Another specific testing need is for example **Usability Testing**. For this type of testing there are specific tools as well.



Time
Intended study time is 20 minutes

Terms
data-driven testing, keyword-driven testing, scripting language

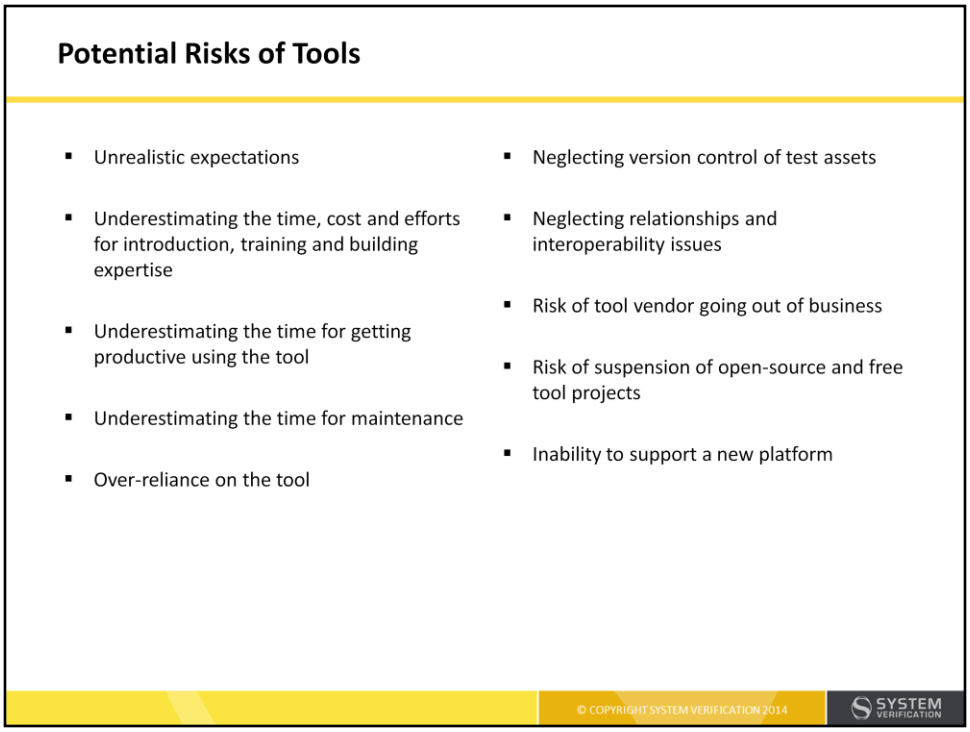
Potential Benefits of Tools

- Reducing repetitive work
- Greater consistency and repeatability
- Objective assessment
- Ease of access to information, e.g. statistics and graphs about test progress, incident rates and performance

Simply purchasing or leasing a tool does not guarantee success with that tool. Each type of tool may require additional 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.

Potential benefits of using tools include:
• Repetitive work is reduced, e.g. running regression tests, re-entering the same test data and checking against coding standards
• Greater consistency and repeatability, e.g. test executed by a tool in the same order with the same frequency and tests derived from requirements
• Objective assessment, e.g. static measures and coverage measures
• Ease of access to information about tests or testing, e.g. statistics and graphs about test progress, incident rates and performance



Potential risks of using tools include:
• Unrealistic expectations for the tool, including functionality and ease of use
• Underestimating the time, cost and effort for the initial introduction of a tool, including training and external expertise
• Underestimating the time and effort needed to achieve significant and continuing benefits from the tool, including the need for changes in the testing process and continuous improvement of the way the tool is used
• Underestimating the effort required to maintain the test assets generated by the tool
• Over-reliance on the tool, e.g. replacement for test design or use of automated testing where manual testing would be better
• Neglecting version control of test assets within the tool
• Neglecting relationships and interoperability issues between critical tools, such as requirements management tools, version control tools, incident management tools, defect tracking tools and tools from multiple vendors
• Risk of tool vendor going out of business, retiring the tool or selling the tool to a different vendor
• Poor response from vendor for support, upgrades and defect fixes
• Risk of suspension of open-source and free tool projects
• Unforeseen things such as the inability to support a new platform

Special Considerations

- Test management tools
  - Test management tools need to interface with other tool for importing and exporting information
  - Investigation of interoperability essential
- Static analysis tools
  - Static analysis tools can generate a lot of warning messages, especially initially
  - A gradual implementation using filters to sort out warnings would be a good approach

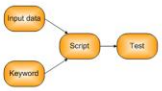
Test management tools
Test management tools need to interface with other tools or spreadsheets in order to produce information in the best format for the current needs of the organization. It is therefore necessary to carefully investigate interoperability between tools as well as how the reports should be designed and monitored so that they provide benefit.

Static analysis tools
Static analysis tools applied to source code can enforce coding standards, but if applied to existing code may generate a large amount of warning messages. These messages do not stop the code being translated into an executable program, but should ideally be addressed so that maintenance of the code is easier in the future. A gradual implementation with initial filters to exclude some messages would be an effective approach.

Special Considerations

- Test execution tools
  - Automating test execution using capture and replay is usually too unstable
  - A good approach would be to use data-driven and keyword-driven testing
- Two approaches:
  - Data-driven approach
    - Separates out test input and uses more generic scripts
    - Reads data and perform same test with different data
    - Easy to use for testers not familiar with the scripts
    - Algorithms to generate data can also be used

  - Keyword-driven approach
    - Contains keywords describing common actions
    - Can be tailored for the application being tested



Test execution tools
A **data-driven approach** separates out the test inputs (the data), usually into a spreadsheet, and uses a more generic script that can read the test data and perform the same test with different data.

Testers who are not familiar with the scripting language can enter test data for these predefined scripts.

There are other techniques employed in data-driven techniques, where instead of hard-coded data combinations placed in a spreadsheet, data is generated using algorithms based on configurable parameters at run time and supplied to the applications.

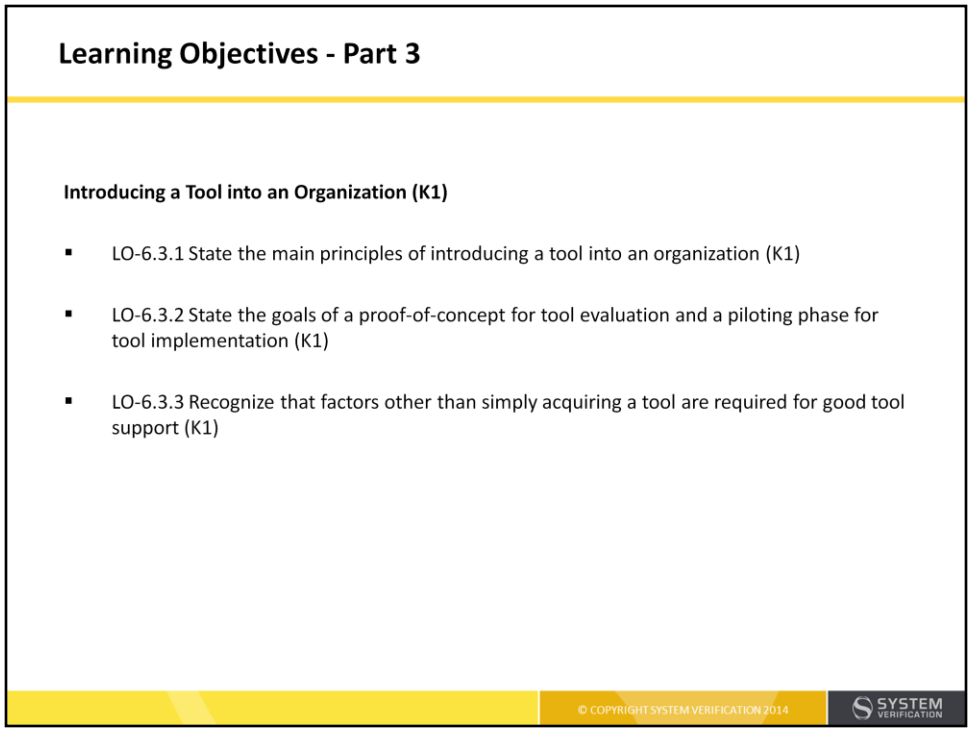
For example, a tool may use an algorithm, which generates a random user-ID, and for repeatability in pattern, a seed is employed for controlling randomness.

In a **keyword-driven approach**, the spreadsheet contains keywords describing the actions to be taken (also called action words) and test data.

Even if the testers are not familiar with the scripting language they can then define tests using the keywords, which can be tailored to the application being tested.

Whichever scripting technique is used, the expected results for each test need to be defined and stored for later comparison.

Technical expertise in the scripting language is needed for all approaches, either by testers or by specialists in test automation.



Time
Intended study time is 15 minutes

Terms
No specific terms

Introducing a Tool

- Many aspects need to be considered, for example:
  - Organizational maturity (what processes can be improved by using a tool)
  - Evaluation against clear requirements/criteria
  - Doing a proof of concept
  - Evaluation of the vendor (including training, support, service agreements etc.)
  - Evaluation of training needs
  - Estimation of return on investment (using a concrete business case)

Before a tool can be introduced, many aspects needs to be considered. The main considerations in selecting a tool for an organization include:
• Assessment of organizational maturity, strengths and weaknesses and identification of opportunities for an improved test process supported by tools
• Evaluation against clear requirements and objective criteria
• A proof-of-concept by using a test tool during the evaluation phase to establish whether it performs effectively with the software under test and within the current infrastructure or to identify changes needed to that infrastructure to effectively use the tool

More considerations in selecting a tool for an organization include:
• Evaluation of the vendor, including training, support and commercial aspects, or service support suppliers in case of non-commercial tools
• Identification of internal requirements for coaching and mentoring in the use of the tool
• Evaluation of training needs considering the current team's test automation skills
• Estimation of cost-benefit ration based on a concrete business case

Pilot Project

- Aims to learn more details about the tool and evaluate how it fits with the organisation's process
- Determine standard ways of working
- Involves very few users

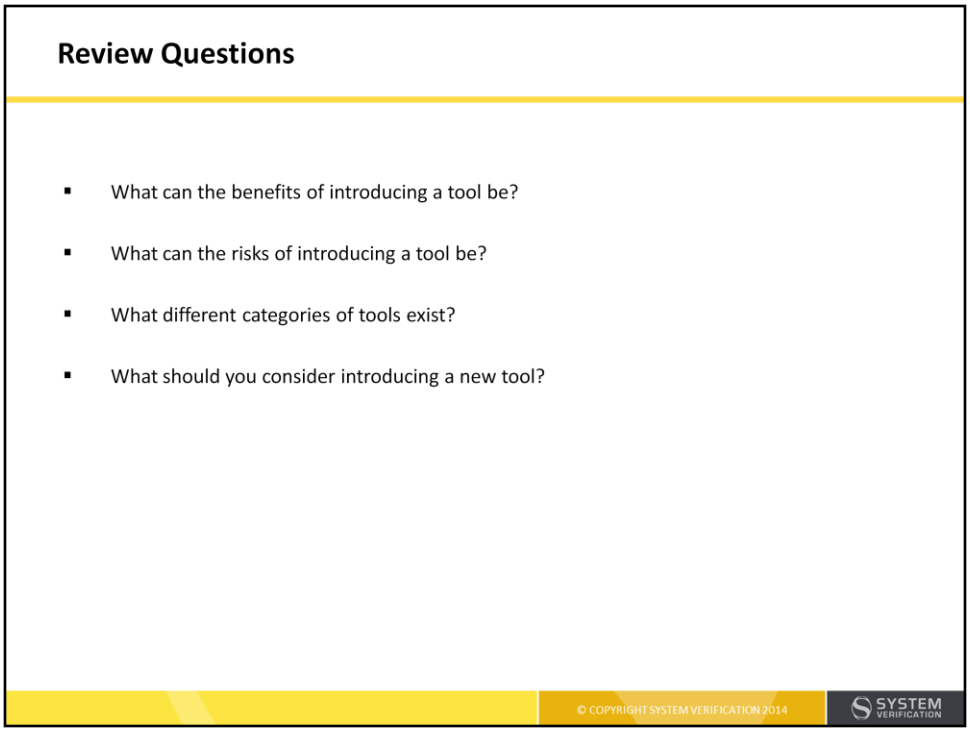
The evaluation of a tool usually starts with a pilot project. This project aims to learn more details about the tool, evaluate how the tool fits with existing processes and practices, and determine what need to be changed.

It helps an organization decide on standard ways of using, managing, storing and maintaining the tool and the test assets, e.g. deciding on naming conventions for files and tests, creating libraries and defining test modularity of test suites. The pilot project should also let us assess whether the benefits will be achieved at a reasonable cost.

Success Factors for new Tools

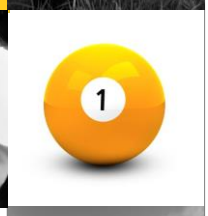
- Rolling out the tool incrementally
- Adapting processes to fit the use of the tool
- Providing training and coaching/mentoring for new users
- Defining usage guidelines
- Monitoring tool use
- Gathering lessons learned from users

There are general recommendations for introduction, such as:
• 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 and coaching/mentoring for new users
• Defining usage guidelines
• Implementing a way to gather usage information from the actual use
• Monitoring tool use and benefits
• Providing support for the test team for a given tool
• Gathering lessons learned from all teams



Summary
This chapter have covered the usefulness of different tools that can improve and speed up the test process. You have learnt about the benefits and risks introducing a new tool in your organization and how common pit-falls can be avoided.

- We create new opportunities



Study Guide

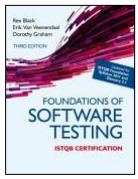
How to Prepare Yourself for the ISTQB Foundation Level Exam

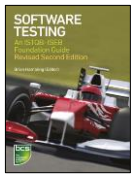
SYSTEM VERIFICATION

Step 1: Study the Course Material

- The course folder contains everything you will need to pass the exam.
- Start by carefully studying the lesson slides and comments within each chapter.
- You may also find the official ISTQB documentation useful:
  - Syllabus
  - Glossaries
- If you want to get an additional text book, we recommend one of the following:
  - Foundations of Software Testing ISTQB Certification (Black/Veenendaal/Graham)
  - Software Testing: An ISTQB-ISEB Foundation Guide (Morgan/Samaroo/Hambling)
- Keep in mind that exam questions are distributed in proportion to the size of each chapter (as described in the syllabus).
  - Chapter 4: 12 questions
  - Chapter 5: 8 questions

  } 50% of the ISTQB FL exam
- The exam is based on the ISTQB syllabus only.
- Questions are asked according to the syllabus, so make sure to answer accordingly. Don't try to argue with the syllabus in the exam!





SYSTEM VERIFICATION

SYSTEM VERIFICATION

Step 2: Terms

- ISTQB FL contains a large number of terms that you're expected to know.
- Each chapter contains a summary of all new terms.
- You can find explanations in the glossaries.
- Here is one way to memorize terms by using paper cards:
  1. Get a number of blank index cards.
  2. Prepare each card by writing the term on one side and the definition on the other side.
  3. Go through the cards while studying, either on your own or with a colleague.
  You may want to do this as a game or contest to have some fun while practicing.

Step 3: Review Questions

- The course material contains a large set of review questions (answer keys included).
  - Answer each question.
  - Check that you got the right answer.
  - If you got it wrong, check that you understand why the other answer is correct.
  - If you don't understand why a particular answer is correct, discuss with your colleagues or course instructor.
- Check the learning objectives of each chapter in the syllabus.
  - Make sure you can give the answer of each learning objective.
  - These points are essentially what the exam is all about.

Step 4: Sample Exams

- Use sample exams to check your knowledge.
- You may want to practice under realistic circumstances, i.e. using the same time limit as in the real exam (1 h / 1.15 h depending on exam language and your native language).
- If you get a pass result on several sample exams, you will most likely pass the real exam.
- Below are links to a few useful sample exams (both official and non-official):
  - http://www.istqb.org/downloads/finish/41/87.html (official ISTQB sample exam)
  - http://www.testingexcellence.com/istqb-quiz/ (two exams + standalone questions)
  - http://www.softwaretestinghelp.com/istqb-testing-certification-sample-question-papers-with-answers/ (three exams)
  - http://www.softwaretesting.no/istqb/istqb_CTFL_trialexam_eng.pdf (one exam)
- There are many sample exams on the Internet, but some of them are rather low quality. Use your own judgment to decide whether you can trust the information or not.
- Observe how exam questions are constructed.
- Make sure you are familiar with the way questions are asked.
- There are always four options and one correct answer.
- On the ISTQB FL exam you will need 26 correct answers out of 40 in total.

Step 5: Before the Exam

- Review the main parts once more.
- Take a sample exam to check your level of knowledge.
- If possible, avoid studying the entire night before the exam.
- Try to get a good night's sleep before the exam.
  - Your brain will thank you.

Step 6: During the Exam

- Make sure you show up early, at least 15 minutes before the exam begins.
- The ISTQB Foundation exam is closed-book, so you can't bring course materials or notes.
  - In some cases you can bring a general dictionary. Please check with the exam facilitator.
- Most people feel a bit nervous before the exam. This is perfectly normal. Try to relax.
- When the exam begins, take a few minutes to read through the exam questions. This will give you a good overview, so you know better what to expect.
- Read each question carefully! Make sure you really know what they ask about.
- Each correct answer will give you 1 point regardless of difficulty level.
- You can answer the questions in any order you like.
- Always answer the easy questions first. This way you will collect lots of points quickly. It will also increase your confidence level.
- Next, move on to the more complex questions. You will have pencils and papers at hand. Use them to make notes, draw sketches etc. This is especially important for questions where you are asked to combine multiple pieces of information.

Step 6: During the Exam (continued)

- You can also make marks on the questionnaire itself. For example, to cross out invalid options. This will help you narrow down the number of options.
- There are no penalties for incorrect answers.
- Don't get stuck on any single question! If you feel unsure about a particular question, it might be a good idea to skip it and get back to it later.
- Watch out for absolute words such as *always, never, all, none* and similar. Those often indicate incorrect options. Testing is context dependent, remember?
- Be aware of modus variations. There is a clear difference between the statements "independent testing **will** lead to more bugs being found" and "independent testing **may** lead to more bugs being found". The second statement is more correct.
- Be attentive! You might actually find the answer you're looking for in another question.
- Take an occasional look at the clock. Most people find the time allocated for the exam fully sufficient. You will need to keep a steady pace, though. The exam facilitator will usually let you know when time is about to run up, e.g. 10 minutes before the end.
- Before you hand in your exam, check that you've marked the correct boxes on the answer sheet. You don't want to miss points because you ticked the wrong boxes.

Step 7: After the Exam

- If you take a computer-based exam, you will get your result immediately.
- If you take a paper-based exam, you will receive your result by mail within 1 – 2 weeks.
- Hopefully, you've passed the exam. In this case you will receive your certificate along with the result. Congrats! ☺
- If you did not pass the exam on the first try, there is no cause for alarm.
  - You can make a new attempt as soon as you feel ready (another exam fee will be charged).
  - Your course instructor can help you decide which chapters you need to study more.

Summary

- Get familiar with the course material.
- Get familiar with the official documents (syllabus, glossaries).
- Learn the terms.
- Practice using review questions.
- Practice using sample exams.
- Check your knowledge and fill any gaps.
- Remember the exam-related tips.
- Relax and feel confident in your abilities.
- Receive your certificate, frame it and put it up on the wall. ☺

1 TEST DOCUMENTATION ACCORDING TO IEEE 829-2008

This material is included for reference only.

1.1 MASTER TEST PLAN

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References
1.4. System overview and key features
1.5. Test overview
  1.5.1 Organization
  1.5.2 Master test schedule
  1.5.3 Integrity level schema
  1.5.4 Resources summary
  1.5.5 Responsibilities
  1.5.6 Tools, techniques, methods, and metrics

2. Details of the Master Test Plan

2.1. Test processes including definition of test levels
  2.1.1 Process: Management
    2.1.1.1 Activity: Management of test effort
  2.1.2 Process: Acquisition
    2.1.2.1: Activity: Acquisition support test
  2.1.3 Process: Supply
    2.1.3.1 Activity: Planning test
  2.1.4 Process: Development
    2.1.4.1 Activity: Concept
    2.1.4.2 Activity: Requirements
    2.1.4.3 Activity: Design
    2.1.4.4 Activity: Implementation
    2.1.4.5 Activity: Test
    2.1.4.6 Activity: Installation/checkout
  2.1.5 Process: Operation
    2.1.5.1 Activity: Operational test
  2.1.6 Process: Maintenance
    2.1.6.1 Activity: Maintenance test
2.2. Test documentation requirements
2.3. Test administration requirements
2.4. Test reporting requirements

3. General

3.1. Glossary
3.2. Document change procedures and history

1.2 LEVEL TEST PLAN

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References
1.4. Level in the overall sequence
1.5. Test classes and overall test conditions

2. Details of the Level Test Plan

2.1. Test items and their identifiers
2.2. Test Traceability Matrix
2.3. Features to be tested
2.4. Features not to be tested
2.5. Approach
2.6. Item pass/fail criteria
2.7. Suspension criteria and resumption requirements
2.8. Test deliverables

3. Test Management

3.1. Planned activities and tasks; test progression
3.2. Environment/infrastructure
3.3. Responsibilities and authority
3.4. Interfaces among the parties involved
3.5. Resources and their allocation
3.6. Training
3.7. Schedules, estimates, and costs
3.8. Risk(s) and contingency(s)

4. General

4.1. Quality assurance procedures
4.2. Metrics
4.3. Test coverage
4.4. Glossary
4.5. Document change procedures and history

1.3 LEVEL TEST DESIGN

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References

2. Details of the Level Test Design

2.1. Features to be tested
2.2. Approach refinements
2.3. Test identification
2.4. Feature pass/fail criteria
2.5. Test deliverables

3. General

3.1. Glossary
3.2. Document change procedures and history

1.4 LEVEL TEST CASE

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References
1.4. Context
1.5. Notation for description

2. Details *(repeat for each test case)*

2.1. Test case identifier
2.2. Objective
2.3. Inputs
2.4. Outcome(s)
2.5. Environmental needs
2.6. Special procedural requirements
2.7. Intercase dependencies

3. Global

3.1. Glossary
3.2. Document change procedures and history

1.5 LEVEL TEST PROCEDURE

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References
1.4. Relationship to other procedures

2. Details

2.1. Inputs, outputs, and special requirements
2.2. Ordered description of the steps to be taken to execute the test cases

3. General

3.1. Glossary
3.2. Document change procedures and history

1.6 LEVEL TEST LOG

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References

2. Details

2.1. Description
2.2. Activity and event entries

3. General

3.1. Glossary

1.7 ANOMALY REPORT

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References

2. Details

2.1. Summary
2.2. Date anomaly discovered
2.3. Context
2.4. Description of anomaly
2.5. Impact
2.6. Originator's assessment of urgency (see IEEE 1044-1993 [B13])
2.7. Description of the corrective action
2.8. Status of the anomaly
2.9. Conclusions and recommendations

3. General

3.1. Document change procedures and history

1.8 LEVEL INTERIM TEST STATUS REPORT

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References

2. Details

2.1. Test status summary
2.2. Changes from plans
2.3. Test status metrics

3. General

3.1. Document change procedures and history

1.9 LEVEL TEST REPORT

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References

2. Details

2.1. Overview of test results
2.2. Detailed test results
2.3. Rationale for decisions
2.4. Conclusions and recommendations

3. General

3.1. Glossary
3.2. Document change procedures and history

1.10 MASTER TEST REPORT

1. Introduction

1.1. Document identifier
1.2. Scope
1.3. References

2. Details of the Master Test Report

2.1. Overview of all aggregate test results
2.2. Rationale for decisions
2.3. Conclusions and recommendations

3. General

3.1. Glossary
3.2. Document change procedures and history