BCS/ISTQB® Certificate in Software Testing Foundation

STF-2 v6.3

Contents

How to use this workbook

Activity

Alongside this icon you will find details of the group/ individual activity or a point for everyone to discuss.

Definition

Where a word with a very specific definition (or one that could be described as jargon) is introduced this will highlight that a definition is provided.

Expansion materials

This manual contains examinable materials. The QA++ symbol contains further information. Skip over these during class. **They are not needed for the examination.**

Glossary

Definition of a term.

Helpful hint

This icon guides you to tips or hints that will help you avoid the standard pitfalls that await the unwary practitioner or to show you how you might increase your effectiveness or efficiency in practising what you have learnt.

Important idea or concept

Generally this icon is used to draw your attention to ideas that you need to understand by this point in the course. Let your trainer know if you do not understand or see the relevance of this idea or concept.

Key point

This icon is used to indicate something that practitioners in this field should know. It is likely to be one of the major things to remember from the course, so check you do understand these key points.

Reference material

When we have only touched briefly on a topic this icon highlights where to look for additional information on the subject. It may also be used to draw your attention to International or National Standards or Web addresses that have interesting collections of information.

Reinforcement

From time to time, there will be places within the course where it is useful for you to reinforce your understanding. This might be in the form of a question to ponder or a short end-of-module test.

Useful tool

This icon indicates a technique that will help you put what you have learnt into action.

Warning

This icon is used to point out important information that may affect you and your use of the product or service in question.

Course Introduction

Introduction

Welcome to QA’s Software Testing Foundation course! During the next few days, you will learn the skills, techniques and knowledge required to pass the BCS examination in this subject.

Full details of the syllabus are at http://certifications.bcs.org/upload/pdf/swt-foundation-syllabus.pdf

Course Administration

Before we begin the course, your instructor needs to take you through a number of administrative points as shown below.

Safety

Timings

Breaks/ Meals

Rooms

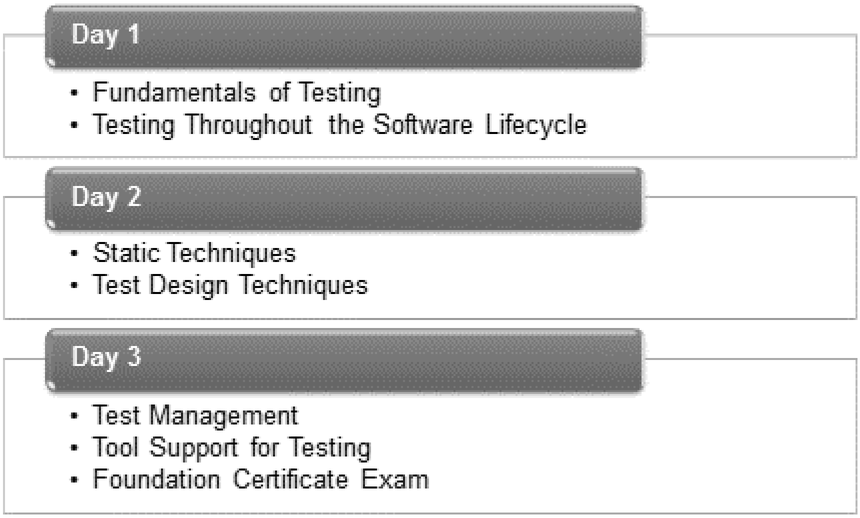
Security

BCS Course Objectives

Foundation Level professionals should be able to:

- Use a common language for efficient and effective communication with other testers and project stakeholders
- Understand established testing concepts, the fundamental test process, test approaches, and principles to support test objectives
- Design and prioritise tests by using established techniques; analyse both functional and non-functional specifications at all test levels for systems with a low to medium level of complexity
- Execute tests according to agreed test plans, and analyse and report on the results of tests
- Write clear and understandable incident reports
- Effectively participate in reviews of small to medium-sized projects
- Be familiar with different types of testing tools and their uses; assist in the selection and implementation process

Course Timetable



Day 1

- Fundamentals of Testing
- Testing Throughout the Software Lifecycle

Day 2

- Static Techniques
- Test Design Techniques

Day 3

- Test Management
- Tool Support for Testing
- Foundation Certificate Exam

Learning Objectives

Learning objectives are indicated for each section in this syllabus and classified as follows:

- K1: Remember – recognise, remember and recall a term or concept.
- K2: Understand – be able to select the reasons or explanations for statements related to the topic and summarise, differentiate, classify, and give examples
- K3: Apply – select the correct application of a concept or technique and apply it to a given context.
- K4: Analyse – separate information related to a procedure or technique into its constituent parts for better understanding and distinguish between facts and inferences.

The Exam

All candidates must provide suitable photographic identification before taking the examination. This will be verified by the trainer. Failure to comply with this requirement means that BCS will withhold the candidate's result until satisfactory evidence has been provided and authenticated.

An invigilator from the BCS will conduct the exam. The BCS Candidate Identification form (supplied by the trainer) must be completed in advance and handed to the trainer to check.

The examination is a one-hour, closed book multiple choice exam, with 40 questions, and 4 options per question. You need 26/40 to pass (65%). It is administered by the BCS to the ISTQB® syllabus.

If English is not your first language or your business language then you are entitled to 25% extra time (15 minutes). If you have a condition or disability that requires extra time then prior approval of the BCS is required. For information on eligibility criteria, please refer to the Reasonable Adjustments Policy on the BCS website.

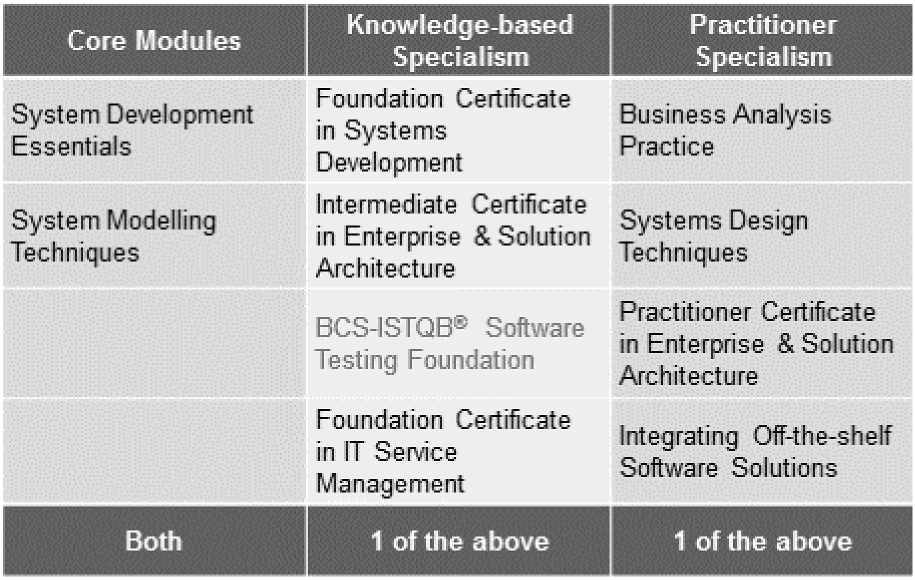
To help you prepare for the exam, there are a number of review questions at the end of each section, with the answers at the back of this book. The BCS has supplied two sample exam papers together with the answers, which you will also be given.

Course Materials

In addition to this course workbook, access to ISTQB® Glossary (http://istqb.org/downloads/glossary.html) as well as the ISTQB® Syllabus (http://istqb.org/downloads/syllabi.html) are highly recommended.

BCS Solution Development Diploma

The Foundation certificate is one of the options for working towards the BCS International Diploma in Solution Development.



Core Modules	Knowledge-based Specialism	Practitioner Specialism
System Development Essentials	Foundation Certificate in Systems Development	Business Analysis Practice
System Modelling Techniques	Intermediate Certificate in Enterprise & Solution Architecture	Systems Design Techniques
	BCS-ISTQB® Software Testing Foundation	Practitioner Certificate in Enterprise & Solution Architecture
	Foundation Certificate in IT Service Management	Integrating Off-the-shelf Software Solutions
Both	1 of the above	1 of the above

Next Course?

Thinking about your next course? Please find below the list of testing courses that naturally follow from this Foundation course.

- **QAATF – ISTQB Agile Tester Foundation Level Extension**
  - 2-day course
- **STI – BCS Software Testing Intermediate**
  - 3-day course
- **QASTATM – ISTQB Advanced Level Test Manager**
  - 5-day course
- **QASTATA – ISTQB Advanced Level Test Analyst**
  - 4-day course
- **QACAT – iSQI Certified Agile Tester**
  - 5-day course

Module 1 – Fundamentals of Testing

Topics

1. **Why is Testing Necessary?**
2. **What is Testing?**
3. **Seven Testing Principles**
4. **Fundamental Test Process**
5. **The Psychology of Testing**
6. **Code of Ethics**

1.1 Why is Testing Necessary?

Learning Objectives

- Describe, with examples, the way in which a defect in software can cause harm to a person, to the environment or to a company (K2)
- Distinguish between the root cause of a defect and its effects (K2)
- Give reasons why testing is necessary by giving examples (K2)
- Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality (K2)
- Explain and compare the terms error, defect, fault, failure and the corresponding terms mistake and bug, using examples (K2)

Software (i.e. program code) is present everywhere, not just in business applications (e.g. banking, e-commerce) but in cars, electrical appliances, arms, etc.

Failed software may result in:

- Financial loss (e.g. business unable to sell product)
- Wasted time (e.g. time to fix and recover from failure)
- Loss of reputation (e.g. negative market reaction)
- Injury or death (e.g. safety control system failure)

… or may have an insignificant effect.

Testing reduces the risk of software failure.

Errors occur because we are not perfect and, even if we were, we are working under constraints such as delivery deadlines. Factors such as software or infrastructure complexity, or changing technology also contribute.

Causes of Software Defects

Error – A human action that produces an incorrect result

Defect – A flaw in a system that can cause it to fail to perform its required function (aka a fault or bug)

Failure – Deviation of a system from its expected delivery, service or result. Failure may also be caused by data, hardware or environmental conditions (such as radiation, magnetic fields or pollution.)

All defects start with a human error (for example a typo, misunderstanding, lack of capacity planning etc.) The tester's job is to expose the defect by causing a failure before the software gets to production.

The Role of Testing

Rigorous testing of systems and documentation can help to reduce the risk of problems occurring during operation and contribute to the quality of the software system, if defects found are corrected before the system is released for operational use.

Software testing may also be required to meet contractual or legal requirements, or industry-specific standards.

Testing is needed throughout the software life cycle:

Software development – To provide quality checks at each stage and remove defects

Maintenance – To test changes when live software is modified and also to ensure existing system has not been affected (regression)

Operations – To assess system characteristics such as reliability or availability

Quality and Testing

Testing can give confidence in the quality of the software if it finds few or no defects. A properly designed test that passes reduces the overall level of risk in a system. When testing does find defects, the quality of the software system increases when those defects are fixed.

Quality – *The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations* – ISTQB Glossary

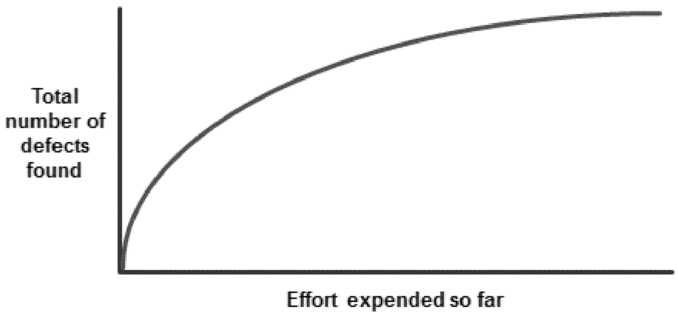
Typical quality measurements are:

- Number of defects found
- Number of failures in a given time period (reliability)
- Usability rating
- Maintainability

Lessons should be learned from previous projects. By understanding the root causes of defects found in other projects, processes can be improved, which in turn should prevent those defects from reoccurring and, as a consequence, improve the quality of future systems. This is an aspect of quality assurance. In short, testing detects, measures and improves software quality.

Testing should be integrated as one of the quality assurance activities (i.e. alongside development standards, training and defect analysis).

As testing progresses the number and severity of defects found will trail off as shown in a typical graph below. So the same amount of effort later in the project is likely to find fewer defects than early in the project – the law of diminishing returns. When to stop is one of the most difficult decisions to make, but risk assessment will help.



How Much Testing is Enough?

Testing is very much context dependent. Testing a simple application to control the soap in a car wash machine will not be the same as testing an application that controls the medication in a life support system!

Deciding how much testing is enough should take account of the level of risk, including technical and business product and project risks, and project constraints such as time and budget. (Risk is discussed further in the Test Management session.)

Testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system being tested, for the next development step or handover to customers.

1.2 What is Testing?

Learning Objectives

- Recall the common objectives of testing (K1)
- Provide examples for the objectives of testing in different phases of the software life cycle (K2)
- Differentiate testing from debugging (K2)

Here are some alternative views on what testing is for. Each has a certain truth.

- "...establishing confidence." Testing helps to provide users with confidence in the delivered software.
- "…the intent of finding errors" This may sound negative, but removing defects is the key to improving software quality.
- "…meets required results" - Testing can assess the extent to which the product serves its intended purpose.

Testing – *The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects* – ISTQB Glossary

Testing is not simply executing software and checking results. Test activities exist before and after test execution, including:

- Planning and control
- Choosing test conditions
- Designing test cases
- Evaluating completion criteria
- Reporting on the testing process
- Closure

These activities in a generic test process will be covered later. Once recognised they have a bearing on estimating, monitoring and control, use of basis, techniques and quality gateways.

As well as dynamic execution of software, testing also includes static tests:

- Reviewing of documents or source code
- Static analysis

The thought process and activities involved in designing tests early in the life cycle (verifying the test basis via test design) can help to prevent defects from being introduced into code. Reviews of documents (e.g. requirements) and the identification and resolution of issues also help to prevent defects appearing in the code.

Both dynamic and static testing help to improve quality by:

- Improving the system under test
- Improving the development process
- Improving the testing process

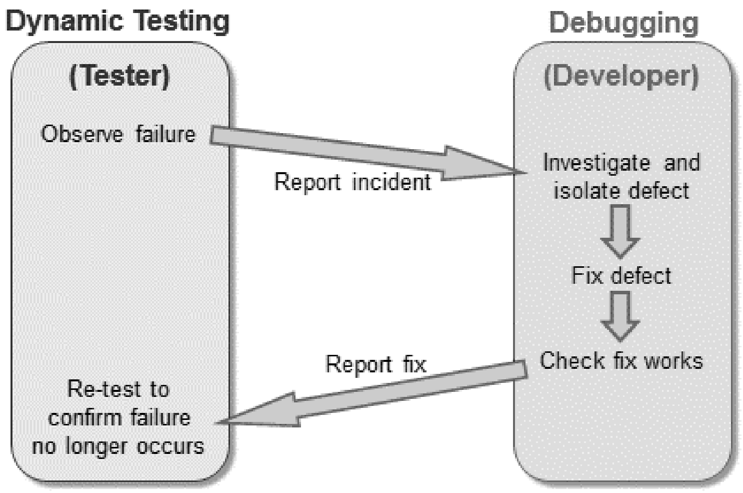
Testing Objectives

Testing has different objectives at different stages, such as:

- Finding defects
- Meeting business requirements
- Gaining confidence about the level of quality
- Providing information for decision-making
- Preventing defects

Testing and Debugging

Debugging and testing are different. Testing can show failures that are caused by defects. Debugging is the development activity that identifies the cause of a defect, repairs the code and checks that the defect has been fixed correctly. Subsequent re-testing by a tester ensures that the fix does indeed resolve the failure.



The responsibility for each activity is very different, i.e. testers test and developers debug.

1.3 Seven Testing Principles

Learning Objectives

- Explain the seven principles in testing (K2)

1. **Testing Shows the Presence of Defects**

   Testing can show defects are present but cannot prove there are no remaining defects. Testing reduces the probability of undiscovered defects remaining in the software.

   No matter how many defects we find and remove, we can never be sure we have found them all. We can increase the reliability of software but can never guarantee 100% quality.

   Even if no defects are found, it is not proof of correctness.

2. **Exhaustive Testing is Impossible**

   Most modern software systems are complex, with various hardware, software and operating systems all interacting. Trying to identify all possible combinations of these, plus all possible combinations of inputs and pre-existing data is practically impossible.

   Therefore testers must prioritise and focus testing where it will do the most good, e.g. the most critical functions, the most commonly-used transactions, the most volatile parts of code, etc.

   Risk analysis is used to identify what to test and what not to test.

3. **Early Testing**

   Testing activities should start as soon as possible in the software development life cycle to help prevent defect multiplication, for example where an incorrect requirement leads to a number of incorrect software components being built.

   This ensures testing is focused on defined objectives, such as high-priority objectives, key business functions, customer needs and reliability.

4. **Defect Clustering**

   A small number of modules tend to contain the most defects and are responsible for operational failures. Testing effort should be focused proportionally to the expected and later observed defect density of modules.

   Testers have always recognised some areas of software are more vulnerable than others. This is key to risk-based testing and to the exploratory/error guessing non-systematic techniques.

   The 80:20 rule applies: 80% of the defects are in 20% of the code.

5. **Pesticide Paradox**

   Rerunning tests is less powerful than devising new tests. The same tests iterated many times will eventually find no more bugs.

Developers, testers and users will have different focus in their test designs and so will find different defects. Also, developers learn from their mistakes and produce software so that it will pass standard tests, so different ones have to be devised.

New tests should be devised by different testers with different viewpoints at different stages of testing, e.g. UAT will focus on different tests from System Testing.

6. **Testing is Context-Dependent**

   Testing is done differently in different contexts, for example an e-commerce website will be tested differently from safety-critical software or a batch payroll system.

   This could mean changes in test objectives, strategy, approach, effort, resources and choice of design techniques.

7. **Absence-of-Errors Fallacy**

   Finding and fixing defects doesn't help if the system is unusable and doesn't fulfil the users' expectations.

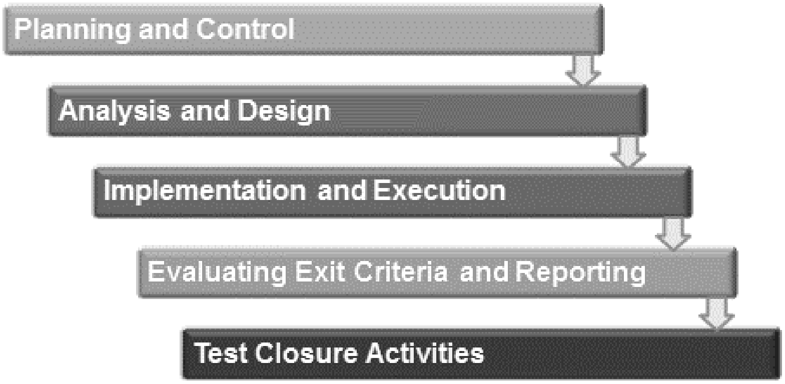
   Removing defects is a different objective from meeting business needs.

1.4 Fundamental Test Process

Learning Objectives

- Recall the five fundamental test activities and respective tasks from planning to closure (K1)

The Fundamental Test Process consists of five main activities, each of which is made up of a number of low-level testing tasks. Although logically sequential, the activities in the process may overlap or run concurrently.


Planning and Control
Analysis and Design
Implementation and Execution
Evaluating Exit Criteria and Reporting
Test Closure Activities

Test Planning and Control

Test planning is the activity of verifying the mission of testing, defining the objectives of testing and the specification of test activities in order to meet the objectives and mission. Without a plan the detailed testing will lack direction.

Test **Planning** has the following major tasks:

- Determine the scope and risks, and identify the objectives of testing
- Implement the test policy and/or the test strategy
- Determine the test approach (e.g. techniques, test items, coverage, interfaces, etc.)

- Determine the required test resources
- Schedule the test activities (e.g. analysis, design, implementation, execution and evaluation)
- Determine the entry and exit criteria

Test control is the on-going 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, the testing activities should be monitored throughout the project and fed back to test planning.

Test **Control** tasks include:

- Monitor testing activities and results throughout a project
- Compare actual progress against the plan
- Take controlling actions as required

Test Analysis and Design

Test analysis and design is the activity where general testing objectives are transformed into tangible test conditions and test designs. This is where systematic test design techniques can be employed.

Tasks include:

- Review the test basis (e.g. requirements, risk analysis reports, architecture, design, interface specifications)
- Evaluate the test basis for testability
- Identify and prioritise test conditions
- Design and prioritise high-level test cases
- Identify test data to support test conditions and cases
- Design the test environment set-up and identify any required infrastructure and tools
- Create bi-directional traceability between test basis and test cases

Test Implementation and Execution

Test implementation and execution is the activity where test procedures or scripts are specified by combining the test cases in a particular order and including any other information needed for test execution, the environment is set up and the tests are run.

Test **Implementation** tasks include:

- Finalise, implement and prioritise test cases
- Develop and prioritise test procedures, or write automated scripts
- Create test suites from the test procedures for efficient execution
- Create test data
- Verify that the test environment has been set up correctly

Test **Execution** tasks include:

- Execute test procedures according to the planned sequence, either manually or using test execution tools
- Log the outcome of test execution
- Compare actual results with expected results
- Report discrepancies as incidents and analyse them in order to establish their cause (e.g., a defect in the code, in specified test data, in the test document, or a mistake in the way the test was executed)
- Repeat test activities as necessary, for example, for example, re-execution of a test that previously failed in order to confirm a fix (confirmation testing), execution of tests in order to ensure that defects have not been introduced in unchanged areas of the software or that defect fixing did not uncover other defects (regression testing)

Evaluating Exit Criteria and Reporting

Evaluating exit criteria is the activity where test execution is assessed against the defined objectives. This should be done for each test level. Major tasks include:

- Check test logs against the exit criteria specified in test planning
- Assess if more tests are needed or if the exit criteria specified should be changed
- Write a test summary report for stakeholders

Test Closure Activities

Test closure activities collect data from completed test activities to consolidate experience, testware, facts and numbers. Test closure activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), a milestone has been achieved, or a maintenance release has been completed.

Test closure activities include the following major tasks:

- Check planned deliverables have been delivered
- Close incident reports or raise change requests for open incidents
- Documenting the acceptance of the system
- Finalise and archive testware and the test environment for later reuse
- Hand over to the maintenance organisation
- Analyse lessons learned to determine changes needed for future releases and projects, and improve test maturity

1.5 The Psychology of Testing

Learning Objectives

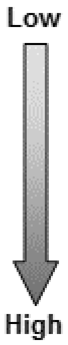
- Recall the psychological factors that influence the success of testing (K1)
- Contrast the mindset of a tester and of a developer (K2)

Test Independence

Test independence means test cases being designed by someone other than the person who wrote the software, such as testers, users or external consultants. A certain degree of independence (avoiding the author bias) often makes the tester more effective at finding defects and failures.

Several levels of independence can be defined as shown below:

- Tests designed by the person(s) who wrote the software under test
- Tests designed by another person(s) within the development team
- Tests designed by a person(s) from a different organisational group (e.g. an independent test team) or test specialists (e.g. usability or performance testers)
- Tests designed by a person(s) from a different organisation or company (i.e. outsourcing or certification by an external body)



Testers and Developers

Developers and testers have different views of software.

Developers want their software to work and therefore may not try too hard to show it doesn't. They see finding defects as a negative process which causes delays, particularly if they are under pressure to meet deadlines.

Testers know that all software contains bugs and so focus their attention on finding those defects (there is an old adage in software development that “if it ain’t got bugs it ain’t software.”)

Looking for defects in a system requires curiosity, professional pessimism, a critical eye, attention to detail and experience.

Mindset of a developer	Mindset of a tester
The software is of acceptable quality	The software contains defects
It will work	It won't work
Finding a defect is bad news	Finding a defect is good news
Short term view	Long-term view
Testers are too negative	Developers are too optimistic
Positive view: look for what works	Negative view: look for what doesn't work
Innocent until proven guilty	Guilty until proven innocent
"Professional optimist"	"Professional pessimist"

Developers often see testing as a negative activity concentrating too much on what doesn’t work resulting in much criticism and little or no praise.

To solve this problem we need to improve communications.

Tester-Developer Communication

Good communication between testers and developers is crucial to producing quality software. Communication of defects and fixes needs to be done in a constructive manner.

There are several ways to improve communication and relationships between testers and others:

- Start with collaboration rather than conflict, as both testers and developers have the common goal of better quality software
- Find fault in the work product, not the developer
- Ensure communication of findings on the product is neutral, objective, and fact-based. For example, write objective and factual incident reports and review findings
- Imagine yourself in their position and try to understand why they react the way they do
- Confirm mutual understanding

There are many ways to formalise the channels for raising, fixing and retesting of bugs: by e-mail, via the incident management tool, or some other means.

1.6 Code of Ethics

Involvement in software testing enables individuals to learn confidential and privileged information. A code of ethics is necessary, among other reasons to ensure that the information is not put to inappropriate use. Recognising the ACM and IEEE code of ethics for engineers, the ISTQB® states the following code of ethics:

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

Judgement - Certified software testers shall maintain integrity and independence in their professional judgement

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

Notes

Module 2 – Testing Throughout the Software Lifecycle

Topics

1. **Software Development Models**
2. **Test Levels**
3. **Test Types**
4. **Maintenance Testing**

2.1 Software Development Models

Learning Objectives

- Explain the relationship between development, test activities and work products in the development life cycle, by giving examples using project and product types (K2)
- Recognise the fact that software development models must be adapted to the context of project and product characteristics (K1)
- Recall characteristics of good testing that are applicable to any life cycle model (K1)

Life Cycle Models

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

There are many system development life cycle models, which fall into two main categories:

Sequential

- Waterfall
- V-Model

Iterative-Incremental (Agile)

- Rapid Application Development (RAD)
- Rational Unified Process (RUP)
- Extreme Programming (XP)
- Scrum
- Dynamic Systems Development Method (DSDM)

Sequential Life Cycle Models

In sequential models the entire system is built in a single sequence of activities that successively define, build, test and implement the software. Examples are waterfall, Structured Systems Analysis and Design Method (SSADM), the V-model and the W-model.

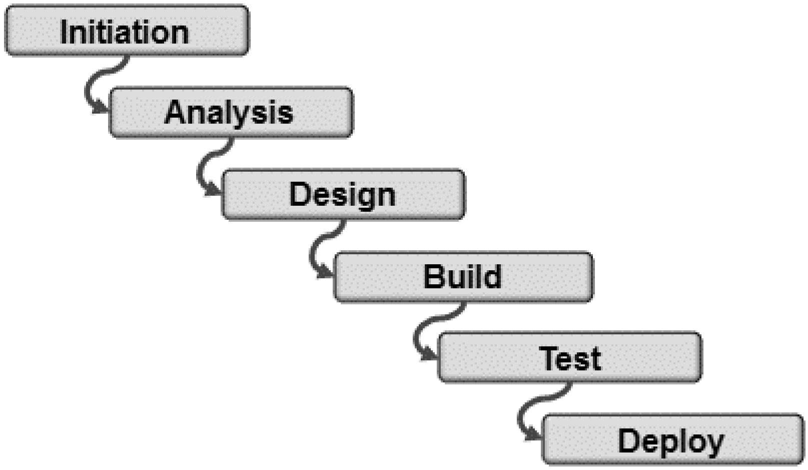
Iterative-incremental (Agile) Life Cycle Models

In iterative-incremental development models, systems are built in a series of short development cycles which deliver working systems in a number of separate increments that can later be integrated together.

"Agile Development" is an umbrella term for several iterative and incremental software development methodologies. The most popular agile methodologies include Scrum, Crystal Clear, Dynamic Systems Development Method (DSDM), Extreme Programming (XP), Lean Development, and Feature-Driven Development (FDD).

While each of these methods has its specific approach, they all share a common vision and core values. They all involve continuous planning, testing, integration, and other forms of continuous evolution of both the project and the software. They are all lightweight (especially compared to traditional waterfall-style processes), and inherently adaptable. Just as important, they all focus on empowering people to collaborate and make decisions together quickly and effectively.

Waterfall Model



This is the traditional approach to systems development. Each stage is quite separate, carried out by specialists, and each must complete before the next begins. Each stage outputs a deliverable (requirements, design, code, etc.) which is input to the next.

All testing is done at the end, after the code is developed. Therefore this model does not allow for early testing, as recommended in the principles earlier.

V-Model

Although essentially sequential in structure (like the Waterfall model), the V model includes a number of different test levels, each corresponding to a development stage. This allows testing activities to be fully integrated with other tasks in the project life cycle.

Mini cycles (shown by broken lines in the diagram below) show the purpose of each test level, and show the importance of catching defects early.

The V model allows test specification to begin during the development stages, meaning that the test function is better prepared when the software is ready, and untestable or ambiguous requirements or designs are resolved at an early stage. This results in large savings and improvements in quality.

The V model also shows how static testing can be carried out during the development stages, before the code is written. Documents are identified which can be reviewed to trap faults as early as possible.

Verification and validation (and early test design) can be carried out during the development of the software work products.

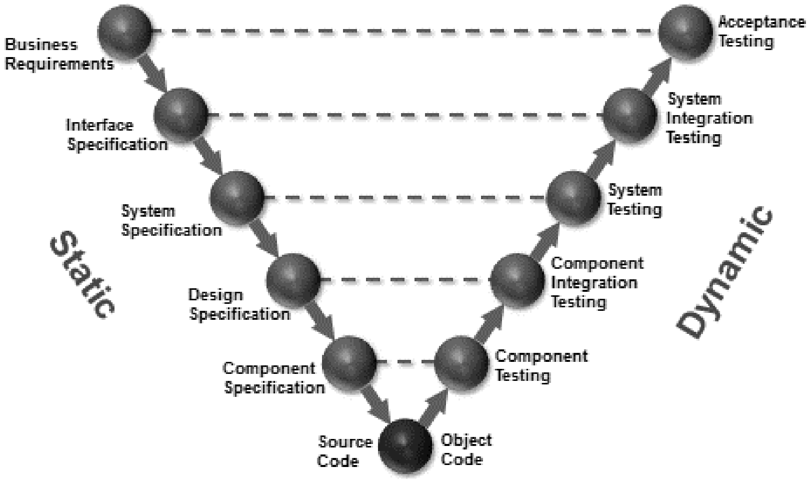
Many variants of the V-model exist, with slightly different number and description of levels. The ISTQB® Foundation exam syllabus refers to a model with 4 test levels as follows:

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

V-model may have more, fewer or different levels of development and testing, depending on the project and the software product.

On this course, we will describe a V-model with 5 levels of testing, (with two very different integration test levels):

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



Iterative-Incremental (Agile) Models

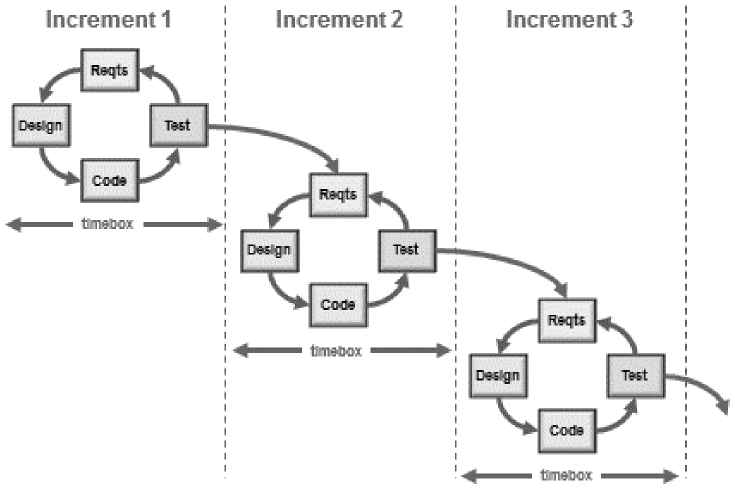
These methods break project requirements into small increments which can be developed separately. This allows development activities to proceed in parallel. Development takes place in rapid cycles or iterations over short time frames (time boxes) that typically last from one to four weeks.

Each iteration involves a team working through a full software development cycle, including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders.

Rapid development exploits the use of technology at all development stages (e.g. prototyping, design modelling tools, debugging tools, etc.) Typically, development teams work closely together with fewer formal documents (requirements, test specs, etc.) and more emphasis on face-to-face communication.

This minimises overall risk and allows the project to adapt to changes quickly.

The more common term used in business to describe any iterative life-cycle is Agile, coined in the *Agile Manifesto* in 2001.



Testing Within an Iterative-Incremental Life Cycle

Each iteration may have several levels of testing, as it is still important to carry out different test levels – component, system, acceptance – in Iterative or Agile models.

Although there is a temptation for developers to do more testing, the principle of independent testing still applies.

The product of each increment, added to others developed previously, forms a growing partial system. Each increment must be integrated with all previous increments, so thorough integration testing and regression testing is increasingly important on all iterations after the first one. Automation can help with this.

This can be helped by using automated test execution tools to repeat tests rapidly.

Although this approach is ultimately flexible to cope with changing requirements, there is no clear transition criteria defined between testing stages. It may also be difficult to write test specs as there may be limited system documentation.

Verification and Validation

Documents and processes can be analysed from two different perspectives: verification and validation.

The terms are commonly used interchangeably in the industry, and it is common to see these two terms incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

Verification: *Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled*. [ISO 9000]

- Is deliverable complete?
- Does it do the job it needs to do?
- Does it adhere to appropriate standards?
- Have we built the product right?

Validation: *Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled*. [ISO 9000]

- Does output deliverable match input specs (predecessor document)?
- Does it meet specified requirements?
- Is it what the customer wants?
- Have we built the right product?

In the V model, verification and validation of software work products take place at each stage during development.

In iterative models, verification and validation will tend to apply to increments rather than testing stages.

Good Testing Within a Life Cycle Model

Testing Characteristic	Sequential Model	Agile Model
For every development activity there is a corresponding testing activity	Clearly defined in V Model	Activities still apply, but less formally
Each test level has test objectives specific to that level	Unit Test - find defects; UAT - meet requirements	Different objectives still important for test design
The analysis and design of tests for a given test level should begin during the corresponding development activity	Clearly defined in V Model	Continuous involvement of testers working with developers in multi-skilled teams
Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle	Formal static testing	Less formal documents to review, but testers can still review prototypes, designs, etc.

In all development life cycles, testing helps to ensure that the work products meet the user expectations (validation) and are being developed in the right way (verification). Testing is conducted in different ways depending on the development life cycle models. However, there are some characteristics of good testing which apply across all development models as shown in the table above.

Although this is easier to achieve with sequential development, it is still essential in iterative models and is possible at any stage.

In any life cycle model, there are several characteristics of good testing:

- For every development activity there is a corresponding testing activity
- Each test level has test objectives specific to that level
- The analysis and design of tests for a given test level should begin during the corresponding development activity
- Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle.

Testing needs to be adapted to circumstances and may need to deviate from the standard models.

For example, test levels may need to be combined or re-organised depending on project type, e.g. Commercial Off The Shelf (COTS) package may have no component test, just integration and acceptance testing

It is important that the test plan describes any such deviation from the standard approach so that people know the scope of testing.

2.2 Test Levels

Learning Objectives

- Compare the different levels of testing: major objectives, typical objects of testing, typical targets of testing (e.g., functional or structural) and related work products, people who test, types of defects and failures to be identified (K2)

Each test level will be described in terms of:

- Objectives
- Test Types involved (functional, non-functional, structural)
- The work product(s) being referenced for deriving the test cases (i.e. test basis)
- The test object (i.e. what is being tested)
- Typical defects and failures found
- Test harness requirements and tool support
- Responsibilities and environment requirements

The V Model will be used to illustrate each test level, although the features of each test level can apply to any life cycle model. Remember, the syllabus refers to just four test levels, but we will separate Component Integration Testing and System Integration Testing as they are very different in practice.

Component (Unit) Testing

Component testing is the first testing stage and is often called unit, module or program testing. It consists of testing and debugging small programming work components that are separately testable.

The aim of component testing is to determine whether an individual function, subroutine, or module works properly in isolation, before we try to integrate it with the rest of the system. Only when we are happy that this component behaves as expected, does it make sense to combine it with other components.

Objectives

- Find and remove defects

Test types

- Verify that code functions according to component specification, design or data model
- Check coverage of code via structural testing
- May check non-functional aspects such as resource management, robustness and maintainability

Test basis

- Component specification
- Detailed design or code

Test objects

- Software modules, programs, objects, classes
- Functions, procedures, subroutines
- Data conversion / migration programs

Typical Defects and Failures

- May be many, but usually simple to fix
- Incorrect logic
- Incorrect definition or use of variables
- Incorrect data types
- Misinterpretation of specifications

Tool Support

- Integrated Development Environment (IDE)
- Debugging tools
- Unit test framework, incorporating stubs and drivers

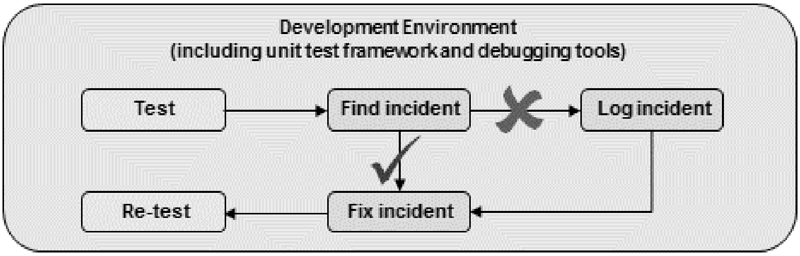
Responsibilities and Environment

- Conducted by developers, not testers

- Carried out in development environment, in isolation from rest of system

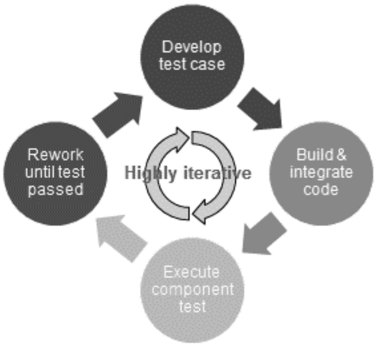
Component Testing is different from all later test stages as it is not done by testers but by the developers of the code, in the developer's environment. This environment may include useful tools for testing and debugging code, such as the ability to step-through code line by line and view data values in variables.

Developers may not record testing activity as thoroughly as independent testers. Defects are typically fixed as soon as they are found, without formally recording incidents.



Component testing may require test drivers, stubs, and automated support. The test driver provides inputs to test the module thoroughly, whereas stubs are written to represent other modules that are invoked. Automated support can aid measurement of coverage, debugging etc.

One approach to component testing is to prepare and automate test cases before coding. This is called a Test-First Approach (TFA) or Test-Driven Development (TDD).



This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, executing the component tests, correcting any issues and iterating until they pass. The aim is to drive out the functionality the software actually needs, rather than what developers think it probably ought to have.

Test-Driven Development has grown out of the Agile software movement and Extreme Programming in particular. Writing the test before the code focuses the mind - and the development process - on delivering only what is absolutely necessary.

Component Integration Testing

The aim of component integration testing is to test all major interfaces and interactions between the individual components, and to uncover communication failures that occur between components. It should not test the functionality within the component (as that has already been done by Component testing), only how the components communicate with each other. It is normally performed by developers in the development environment.

Objectives

- Tests interfaces and interactions between components

Test types

- Involves functional and structural testing of links
- Some non-functional testing (e.g. performance) may be included

Test Basis

- Software and design
- System architecture, workflows

Test Objects

- Internal interfaces

Typical Defects and Failures

- Communication failures between components
- Parameter mismatches

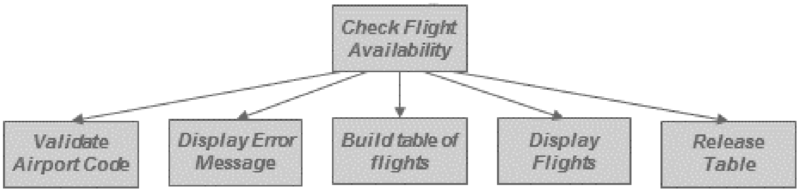
Tool Support

- Unit test framework, incorporating stubs and drivers

Responsibilities and Environment

- Conducted by developers, not testers
- Carried out in development environment, as an extension of component testing

Example:

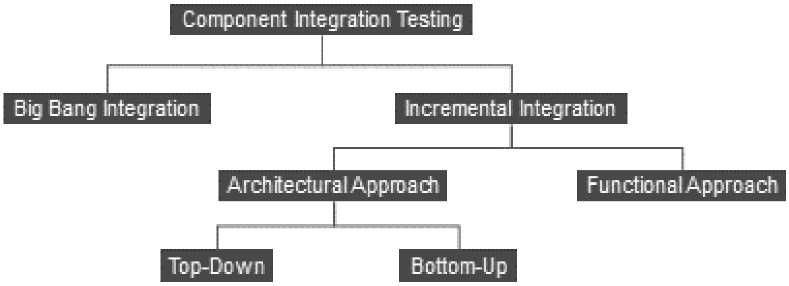


This is from a program to build airport locations in a flight reservation application. There is an error module if a bad location has been input, otherwise it builds a table of available flights, displays that to the GUI then releases it from memory. Typical business applications may need hundreds to be integrated.

Component Integration Strategies

Before integration testing can be planned, an integration strategy is required to determine how the components will be put together for testing. Strategies may be non-incremental ("big-bang") or incremental, i.e. adding components one at a time.

Incremental integration strategies may be based on the system architecture (either top-down or bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system. Incremental strategies are time-consuming but effective as it will be immediately clear which interface (link) has failed.



Big Bang Integration

- Integrate all components at once
- Not recommended, as it may be difficult to identify which link has failed and may result in higher costs in the long run

Top-Down Integration

- Add components one at a time, based on architecture
- Begin testing with components highest in the hierarchy
- Components are added in descending hierarchical order
- Use stubs to simulate lower-level components
- Identifies defects in the architecture

Bottom-Up Integration

- Add components one at a time, based on architecture
- Begin testing with components lowest in the hierarchy
- Components are added in ascending hierarchical order
- Drivers are used to simulate higher-level components
- Higher risk that architecture may be incorrect

Functional Integration

- Select a specific area of functional capability, sequence of transaction processing or other aspect of the system
- Integrate the components needed for that part of the system
- Test these components to determine whether they work correctly together

- A better risk approach, as it tests real business scenarios such as most-used path or critical business function

System Testing

This tests the behaviour of an end-to-end integrated system as defined by the scope of a development project or programme. It is the first test of the whole system or product, and is usually conducted by independent testers.

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

The test environment should be as close to the production environment as possible in order to minimise the risk of environment-specific failures not being found in testing. Testing of Configuration data may be required.

Objectives

- Test the behaviour of a whole system or product

Test types

- Functional and non-functional testing
- Some structural testing, e.g. navigation of web page structure

Test Basis

- System and software requirement specification
- Use cases
- Functional specification
- Risk analysis reports
- Other high level text descriptions or models of:
- System behaviour
- Interactions with operating system and system resources

Test Objects

- End-to-end integrated system (excluding manual processes)
- System, user and operation manuals
- System configuration and configuration data

Typical Defects and Failures

- Normally fewer than component testing
- Likely to include non-functional issues (e.g. load, volume)
- Incorrect system design specification

Tool Support

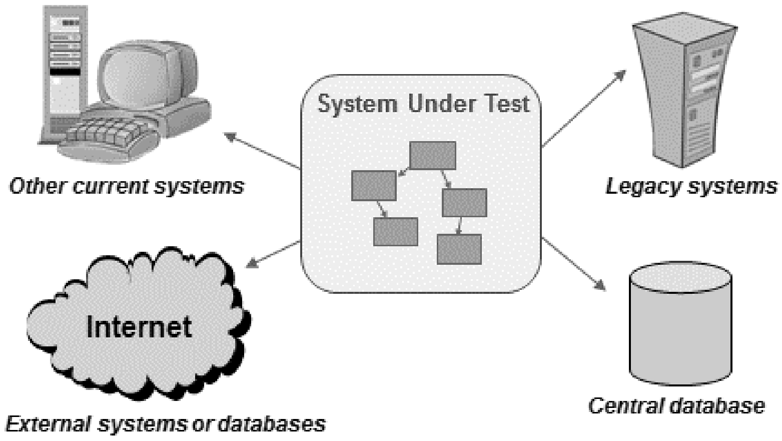
- May use test execution tools for automated testing

Responsibilities and Environment

- Usually performed by independent test team
- Test environment should be as close to target as possible

System Integration Testing

Very few computer systems are truly stand-alone. Most will link with other systems, either internal to the organisation or external.



The purpose of System integration testing is to expose defects in the interfaces and the interactions between one system and another, or between hardware and software. It is normally done after system testing by independent system testers in a near-live environment.

When testing links with external systems, testers may control only their own side of the interface, which might be considered as a risk, as their test system may be linked to an external production system.

Workflow implementation of business processes may involve a series of systems, where cross platform issues may be significant.

Objectives

- Test interfaces between different systems
- Test business processes or workflows which may involve a series of systems
- Test interactions between hardware and software, including data transfer, networks, connections, protocols, security
- Usually done after system testing

Test Basis

- System design
- Business processes, workflows
- Use cases

Test Objects

- Sub-systems database implementation
- Infrastructure
- Interfaces between systems

Typical Defects and Failures

- Communication failures or incompatibility between systems

Tool Support

- May need to simulate functioning of external systems or databases

Responsibilities and Environment

- Independent test team
- Test environment should be close to live system
- Testers may control only their side of the interface to external systems
- Cross-platform issues may be significant

Acceptance Testing

The purpose of acceptance testing is to establish confidence in the system or specific characteristics of the system. The main focus is ensuring the system is fit for purpose, not finding defects. It may assess the system's readiness for deployment and use. The aim should literally be to 'accept' and therefore should be a test of the system under typical usage.

This stage is likely to highlight more non-functional issues, such as usability (screen design or data presentation), accessibility (use by disabled users), performance, load, volume issues, etc.

Acceptance testing can be carried out by real business users or by independent testers representing the users. The test environment will often be a clone of the production system, to make it as realistic as possible. A Model Office environment may be used to test business processes as well as the software system.

Acceptance testing is normally the final stage of testing before software deployment, but there may be other stages to follow, such as large-scale integration or pre-production confidence testing.

Objectives

- Establish confidence in the system or parts of the system
- Ensure the system is fit for purpose
- Assess the system's readiness for deployment and use.
- Note there may be other stages to follow before deployment, such as large-scale integration

Test types

- Functional and non-functional testing
- Some structural testing, e.g. navigation of web page structure

Test Basis

- User requirements
- System requirements
- Use cases
- Business processes
- Risk analysis reports (breakdown of features by risk)

Test Objects

- Business processes on fully integrated system
- Operational and maintenance processes
- User procedures, forms, reports

Typical Defects and Failures

- Non-functional issues (e.g. usability, performance)
- Incorrectly specified requirements

Tool Support

- May use test execution tools

Responsibilities and Environment

- Customers or users of a system
- Other stakeholders as necessary
- Test environment should be as close to target as possible
- Model Office may be used

Acceptance testing may occur as more than just a single test level. For example, acceptance testing of the usability of a component may be done during component testing. Acceptance testing of a new functional enhancement may come before system testing, while a COTS software product may be acceptance tested when it is installed or integrated.

Types of Acceptance Testing

User Acceptance Testing – Verifies the fitness for use of the system from a business perspective.

Operational Acceptance Testing – The acceptance of the system by system administrators, including:

- Testing of backup/restore
- Disaster recovery
- User management
- Maintenance tasks
- Data load and migration tasks
- Periodic checks of security vulnerabilities

Contract Acceptance Testing – Testing that the terms of a contract (e.g. outsourced software development) have been met, such as timescales for delivering software to client.

Regulation Acceptance Testing – Testing compliance with governmental, legal, safety and other regulations, e.g. H&S, DPA, FOI, DDA, etc.

Alpha and Beta Testing – Developers of market, or COTS, software often want to get feedback from potential or existing before the software product is put up for sale commercially.

- **Alpha**: done by customers or potential customers at the developing organisation’s site
- **Beta** (aka Field-Testing): done by customers or potential customers at the customers' site

Some organisations use the terms Factory and Site Acceptance Testing for systems that are tested before and after being moved to a customer’s site.

2.3 Test Types

Learning Objectives

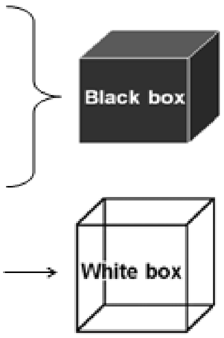
- Compare four software test types (functional, non-functional, structural and change related) by example (K2)
- Recognise that functional and structural tests occur at any test level (K1)
- Identify and describe non-functional test types based on non-functional requirements(K2)
- Identify and describe test types based on the analysis of a software system's structure or architecture (K2)
- Describe the purpose of confirmation testing and regression testing (K2)

There are four software testing types based on particular test objectives:

Functional – Testing what the system does, based on analysis of specified requirements

Non-Functional – Testing how well the system meets the required software characteristics such as performance and usability

Structural – Measures coverage of a system structure (e.g. a control flow model or menu structure model)



Change-Related – Testing a system following modifications or corrections; includes re-testing (aka confirmation testing) and regression testing.

Functional Testing

Functional testing is concerned with WHAT the system must do to achieve its objectives, as defined in business requirements, functional specifications, Use Cases, business process models, etc.

It considers the external behaviour of the software, so is black-box testing. It occurs at all test levels, e.g. the component specification is a source of functional tests for component testing. Functional testing is usually associated with data manipulation, such as input, validation, processing, storage and output.

Note that the ISTQB® syllabus include interoperability and security testing as functional, where other disciplines might consider these non-functional.

Functional Testing Example

“The vehicle’s VIN (Vehicle Identification Number) is keyed in. The system should check that the format of the VIN is valid (2 alphas / 3 numeric / 2 alphas / 10 numeric) - otherwise error message 1 is to be displayed.”

Input validation tests are a typical functional tests, where various specification-based techniques can be used to construct functional tests from the requirement.

In this example, functional tests would include correct format, numerics in alpha fields and vice versa, special characters and null input.

Non-functional Testing

Non-functional testing tests the attributes of a system that do not relate to functionality. It is concerned with HOW WELL a system works. Non-functional requirements are often constraints on the functionality, e.g. constraint on how, how often, who, when or where a function can take place.

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

Non-functional testing may be performed at all test levels, e.g. usability could be tested as soon as a set of linked web pages are available.

It includes the measurement of software characteristics that can be quantified on a varying scale, such as response times for performance testing. These tests may be referenced to a quality model such as the one defined in 'Software Engineering – Software Product Quality' (ISO 9126).

Non-functional testing includes:

- Performance
- Load
- Stress
- Volume
- Usability
- Accessibility
- Maintainability
- Reliability
- Portability

Non-functional requirements are of course just as important to the customer as the functional requirements, sometimes more so!

Non-functional testing is very difficult without precise expression of measurable requirements, often in the form of a range of acceptable values. In order to test them they should be expressed as measurable criteria, probably from a Service Level Agreement for aspects like availability and reliability.

Non-functional Testing Example

The Service Level Agreement (SLA) for a system states: "Provide a 3 second response during peak periods 09:00 – 10:00 and 16:00-17:00"

Performance testing would validate these using specialised tools in a controlled environment to simulate production activity during these times.

Structural Testing

This is testing based on knowledge of the internal structure of software code or system architecture, rather than the specified functionality, so it is considered white-box. Structural test design techniques include coverage measures of what is being tested.

Coverage measurement applies at all levels, as any aspect of a system which can be represented in a structural diagram can be used as the basis for structural testing, e.g.

- Component testing: coverage of statement, decisions, paths
- Component integration testing: coverage of component interfaces
- Acceptance testing: Web page structure with coverage of page navigation

Structural Testing (sometimes called glass box) usually follows specification-based tests to ensure thoroughness of coverage. Full structural testing is very thorough but time-consuming, and is often mandated in safety-critical systems such as missile guidance systems.

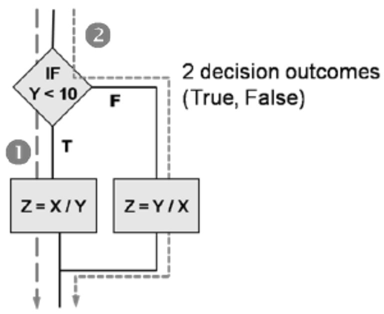
Structural Testing Example

The fragment of program code below is represented by the structure (flowchart) on the right. Structural testing could be used to ensure each decision outcome (e.g. the True and False options of an IF statement) is executed to give 100% decision coverage.

Code Excerpt

```
If Y < 10 Then
     Z = X / Y
Else
     Z = Y / X
End If
```

Flowchart



Change-Related Testing

This is testing following a change or fix to the software or environment. This should involve two types of test:

Re-testing (aka Confirmation Testing) – *Testing that reruns test cases that previously failed in order to verify the success of corrective actions.* ISQTB Glossary.

Re-run test that previously passed to confirm the fix.

Regression Testing – *Testing of a previously tested program following modification, to ensure that defects have not been introduced or uncovered in unchanged areas of the software as a result of the changes made.* ISQTB Glossary

Re-run of test that previously passed

An estimated 20% to 50% of systems changes introduce new defects. Regression testing checks that amendments to software or environment have no adverse effect on unmodified parts of the system.

It may be performed at all test levels and types, including functional, non-functional and structural testing. Regression testing should take place at all stages of testing after a functional improvement or repair.

Scope of Regression Testing

The extent of regression testing is based on the risk of not finding defects in software that worked previously

Full regression testing

- Test all functions
- Minimise risk of failure
- May be very time-consuming and costly
- Probably not realistic except in very small projects

Partial regression testing

- Test selected functions or business areas
- Less time and cost
- More practical approach
- … But increased risk of unforeseen failure

Repeatable Testing via Automation

Tests used for re-testing and regression testing should be repeatable, so they can be re-run every time there is a change or fix.

Automation assists both re-testing and regression testing. Regression testing may be a very large task if it is to re-test all aspects; suites are repeated many times and tend to evolve slowly. A number of these may be combined to form a full test suite which can be rerun following the changes.

One of the major benefits of using tools is that they execute at high speed, and the more sophisticated types are truly hands-off. They normally include break points, and restarts and resetting data may be included in the script.

2.4 Maintenance Testing

Learning Objectives

- Compare maintenance testing (testing an existing system) to testing a new application with respect to test types, triggers for testing and amount of testing (K2)
- Recognise indicators for maintenance testing (modification, migration and retirement) (K1)
- Describe the role of regression testing and impact analysis in maintenance (K2)

Maintenance Testing – *Testing the changes to an operational system or the impact of a changed environment to an operational system.* ISTQB Glossary

Once deployed, a software system is often in service for years or decades. During this time the operational system, its configuration data or its environment are often corrected, changed or extended. The testing of these changes is called maintenance testing, and is triggered by:

- Planned enhancements or upgrades to business systems
- Corrective and emergency changes
- Changes to the operating environment, (e.g. upgrades to the operating system or database)
- Upgrade of commercial-off-the-shelf (COTS) software
- Migration of applications from one platform to another
- Retirement of legacy software systems

Scale of Maintenance Testing

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.

The same testing principles and techniques apply for maintenance testing as for testing big projects, but the scale is likely to be smaller. For example, the timescale might be days or weeks rather than weeks or even months for a new project. And instead of a large requirements catalogue, the test basis might be simply a New Feature Request (NFR), Request for Change (RFC) or live incident report for a production failure.

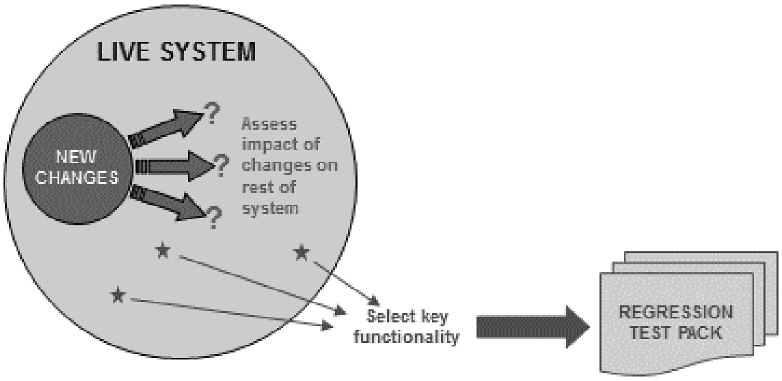
If business functionality is changing (e.g. enhancement or correction), then all test stages and test types could apply. But if business functionality is unchanged (e.g. system migration or operating system upgrade), then regression testing will be key.

Without the discipline of project planning, it can be difficult to incorporate sufficient testing into a maintenance change. So for planned enhancements, maintenance testing can be included in release planning; but for emergency or corrective changes (“hot fixes”) this may not be possible, and maintenance testing may be compromised.

Making changes to old legacy systems can be difficult for both developers and testers if the specifications are out of date (or even non-existent!) or if the original development team have moved to another project or even left the company.

In addition to testing what has been changed, maintenance testing includes extensive regression testing of parts of the system that have not been changed. Determining how the existing system may be affected by changes is called impact analysis, and is used to help decide how much regression testing to do and the selection of the regression test suite. The size and risk of the change is used to decide the scope of regression testing

Every time new functionality is added to a production system, new tests should be added to a standard regression test pack, and old tests removed.


LIVE SYSTEM
NEW CHANGES
?
?
?
Assess impact of changes on rest of system
Select key functionality
REGRESSION TEST PACK

Notes

Notes

Module 3 – Static Techniques

Topics

3.1 Static Techniques and the Test Process

Learning Objectives

- Recognise software work products that can be examined by the different static techniques (K1)
- Describe the importance and value of considering static techniques for the assessment of software work products (K2)
- Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software life cycle (K2)

Static Testing is the examination of program source code or other project documentation, without the execution of the code, to find defects. There are two types of static testing:

Reviews are manual examination of documents - done by humans

Static Analysis is automated analysis of source code and software models - done by tools

In contrast, dynamic testing is performed by executing software under test and comparing actual and expected results.

Reviews and static analysis have the same objective as dynamic testing – identifying defects. They are complementary techniques. Static techniques find causes of failures (defects) directly, while dynamic testing finds failures which may be caused by defects.

Static Testing v Dynamic Testing

Static Testing	**Dynamic Testing**
Performed early in the lifecycle, before software is executed	Performed late in the lifecycle, after software is written
Analyse code or documents without running software	Execute software and analyse results
Find defects in code or documents directly	Observe failures in system which could identify defects in code
Defects are cheap to fix	Defects are expensive to fix
Add quality into software	Check quality of software

Reviews

Any written project document or software product can be reviewed. Typical documents include:

- Business requirements
- Functional specifications
- System and database designs
- Source code
- Test plans, test cases and scripts
- User guides, help text
- Web pages
- etc.

Studies have shown that defects found early in a project are easier and cheaper to fix than those found later. Benefits of conducting reviews include:

- Early defect detection and correction
- Increased development productivity
- Reduced development cost and timescales
- Fewer defects in code
- Reduced time and cost of dynamic testing
- Improved communication between testers, analysts, developers and users
- Conformance to standards leading to improved maintainability of code and documents

Goals of the Review Process

The aim of reviews is to check and improve the quality of project products, including:

- Verification and validation of documents against specifications and standards
- Consistency with predecessor documents
- Completeness and conformance to standards

Reviews should be used by testers to identify defects such as errors, omissions, additions, inconsistencies, ambiguities, readability, terminology, spelling, grammar, layout, and structure.

A good review will also include causal analysis to learn from issues and bring about process improvement. By involving a number of reviewers in a controlled manner consensus can be reached on subjects such as best design practice.

Reviews are better than dynamic testing at finding some types of defect, such as checking standards, identifying inconsistencies in requirements and designs, assessing maintainability and checking interface specifications.

3.2 Review Process

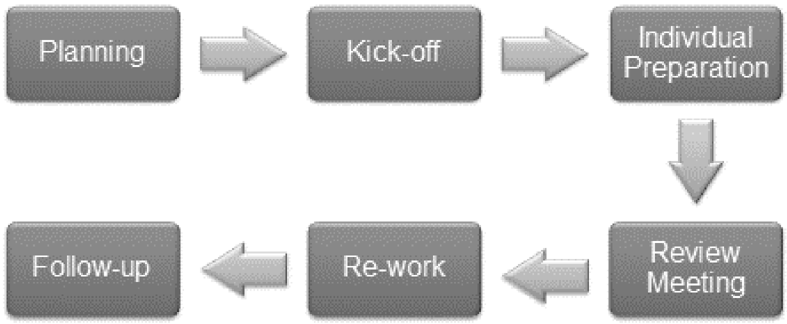
Learning Objectives

- Recall the activities, roles and responsibilities of a typical formal review (K1)
- Explain the differences between different types of reviews: informal review, technical review, walkthrough and inspection (K2)
- Explain the factors for successful performance of reviews (K2)

Different types of reviews vary from informal, characterised by no structure or written instructions, to formal or systematic, characterised by team participation, documented procedures and documented results. The formality of a review process is related to factors such as the maturity of the development process, any legal or regulatory requirements or the need for an audit trail.

The way a review is carried out depends on the agreed objectives of the review (e.g. find defects, gain understanding, educate testers and new team members, or discussion and decision by consensus.)

In this chapter, a 'formal review' is defined as any review with documented procedures and requirements. A typical formal review has the following main activities:



Planning – The moderator and document author define the review criteria and select which parts of documents to review. They select the review team, ensuring the right personnel are selected to add value, and allocate roles. For formal review types, they define the entry and exit criteria.

Kick-off – Documents are distributed to the reviewers, often about 1-3 days in advance; the objectives, process and documents are explained to the participants.

Individual preparation – Reviewers read the documents and note any potential defects, questions and comments. They may compare product to predecessor products, standards, guidelines, and best practices as well as architecture strategies. Reviewers may be required to report readiness to the moderator after reviewing the document(s) or the meeting cannot be held.

Review meeting – The participants discuss the document, note defects, and log findings, with documented results or minutes. The meeting is managed by the moderator, who ensures discussions are kept on track and remain objective. In some review types, only defects are identified and agreed; in others, there may be discussions on alternatives and solutions.

Rework – Following the meeting, the author corrects the document based on the defects found (typically done by the author), and updates the status of defects (in formal reviews).

Follow-up – The moderator ensures that document has been corrected and collects metrics such as time spent by reviewers on checking, defect density in reviewed document, and breakdown of defects by business area or severity. This will be used for causal analysis to find recurring issues.

Roles and Responsibilities

A typical formal review will include the following roles:

Manager decides on the execution of reviews but may not participate directly. The manager allocates time in project schedules (if not scheduled, preparation won't be thorough enough), and determines whether review objectives have been met.

Moderator (or Leader) is responsible for managing all review team activities. The duties include planning, motivating, facilitating meetings, training reviewers and monitoring the whole review process. The moderator should be trained in review process, so they can lead the review meeting, set the agenda and control the pace. The moderator also records summary results for quality management and follow-up action.

Author prepares the material to be reviewed, participates in discussions during the review meeting and incorporates the agreed changes afterwards.

Reviewers (aka Checkers or Inspectors) are responsible for finding defects in the product under review. They should have specific relevant technical or business background and may include stakeholders (predecessor and successor owners.) Reviewers from different areas will have different focus, such as Business users, Developers, Analysts, IT Operations, Testers, Compliance officers

Scribe (or Recorder) documents all the issues, problems and open points that were identified during the meeting.

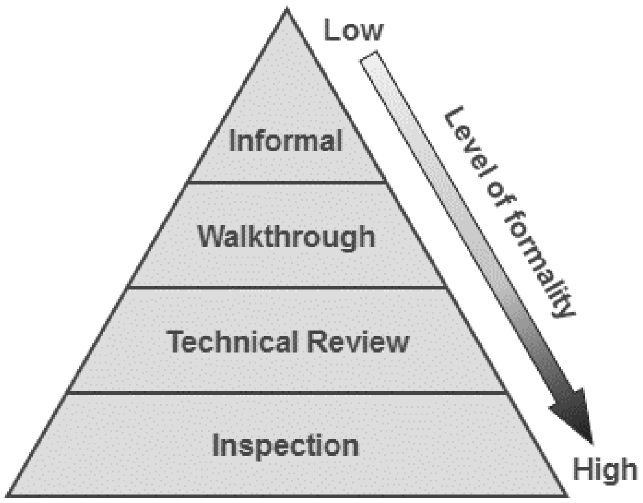
Note that these are roles, not job titles - in some review types, one person could play a number of roles within the process, particularly the leader who may serve as recorder and reviewer too.

Types of Review

Different review types vary in the formality of their procedures and results.

A single document or product may go through different types of reviews. For example, an informal review could be carried out before it is completed to make sure it is in the right format before going through a technical review; or depending on the level of risk, an inspection may be followed by a walkthrough with the customer.

All of these review types may be termed a ‘peer review’ if conducted by reviewers at the same organisational level without management participation; any review with managers present is a ‘management review’.



The main characteristics, options and purposes of common review types are:

Informal Review

Informal reviews are one-to-one discussions with no formal process, roles or documented record of findings. Typical examples include ‘buddy reviews’, desk-checking of program code, pair programming, or a technical team leader reviewing designs and code of team members.

They are an inexpensive way to get some benefit, and are common in most business areas, with senior staff helping and advising more junior staff, or peers assessing each other’s work.

Walkthrough

Walkthroughs are review meeting led by the author of the document being reviewed. They may take the form of scenarios, dry runs and peer group participation. Ideally, reviewers should prepare their comments before the meeting. A scribe (who is not the author but may be a reviewer) should produce a report with a list of findings.

Walkthroughs should allow enough time for discussion, even if that means scheduling another meeting.

Walkthroughs are very useful for highly visual products via storyboarding, workflow tools etc. Walkthroughs should find defects but not propose solutions – that is the author's task after the meeting.

Walkthroughs are probably the most common review type in many organisations. They are often seen primarily as a means of presenting information and learning, e.g. a business analyst presenting a requirements document to other project personnel.

Many attendees will therefore see their role as 'passive' – to learn rather than provide input. However, testers should see them as an opportunity to analyse and query the document, to find defects and improve its quality.

Technical Review

Technical reviews are a documented, defined defect-detection process involving both peers and technical experts. Ideally they are led by a trained moderator (not the author) and may vary from quite informal to very formal, with the use of guidelines and checklists.

The main purpose is to review a document from a particular technical or specialist viewpoint, to discuss, make decisions, evaluate alternatives, find defects, solve technical problems and check conformance to specifications, plans, regulations and standards.

Reviewers will include technical specialists or domain experts (e.g. technical architect, data designer, compliance officer, system administrator) who should prepare comments and recommendations before the meeting.

At the end of the meeting, a review report should be produced with a list of findings, the verdict whether the software product meets its requirements, and recommendations.

Inspection

The inspection process is a very thorough, formal process designed to detect defects in documents and program source code, using rules, checklists and entry/exit criteria. It was developed by Michael Fagan of IBM in the mid-1970s and it has since been extended and modified.

The process should have entry criteria to determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process.

The inspection meeting is led by a trained moderator (not the author) and is usually conducted as a peer review without management involvement. To be most effective, reviewers have defined roles and must submit a list of issues before the meeting.

The scribe should produce a formal inspection report including a list of findings, and there is a formal follow-up process which is conducted by the author and confirmed by the moderator.

Inspections also involve causal analysis which collects metrics on defect information from each inspection, looks at common causes of defects and provides feedback to improve future development and inspection processes.

Success Factors for Reviews

For any type of review to be successful, it must have clear, predefined objectives and must involve the right people for the review objectives. This includes testers who are valued reviewers and can contribute to the review and also learn about the product which enables them to prepare tests earlier.

People issues and psychological aspects must be dealt with and defects found are welcomed and expressed objectively. It is important that reviews are conducted in an atmosphere of trust; the outcome must not be used for the evaluation of the participants.

Review techniques are applied that are suitable to achieve the objectives, and to the type and level of software work products and reviewers. This might include checklists or roles if appropriate to increase effectiveness of defect identification.

Reviews can be improved by the provision of training in review techniques, especially for the more formal techniques such as inspection. There should be an emphasis on learning and process improvement, and management can support a good review process (e.g. by incorporating adequate time in project schedules.)

If the meeting lasts more than two hours people will lose concentration and the review should then be split into two. The meeting should concentrate on logging and classifying issues and should avoid debugging problems there and then - that is the province of the rework stage.

Most people are reasonable especially as they recognise that their work will be undergoing a review process at some time too. It also helps to avoid conflict by referring to problems or queries as “issues” rather than faults or inadequacies.

In summary:

- Each review has clear predefined objectives
- The right people for the review objectives are involved
- Testers are valued reviewers who contribute to the review and also learn about the product which enables them to prepare tests earlier
- Defects found are welcomed and expressed objectively
- People issues and psychological aspects are dealt with (e.g., making it a positive experience for the author)
- The review is conducted in an atmosphere of trust; the outcome will not be used for the evaluation of the participants

- Review techniques are applied that are suitable to achieve the objectives and to the type and level of software work products and reviewers
- Checklists 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 supports a good review process (e.g., by incorporating adequate time for review activities in project schedules)
- There is an emphasis on learning and process improvement

Use of Checklists in Reviews

Using checklists can make reviews more effective and help to find defects. Checklists can be based on various perspectives, e.g.

- User, developer, tester or operations
- List of typical requirements problems

Example checklist for inspecting requirements

Correctness

- Do any requirements conflict with or duplicate other requirements?
- Is each requirement written in clear, concise, unambiguous language?
- Is each requirement verifiable by testing, demonstration, review, or analysis?
- Is any necessary information missing from a requirement? If so, is it identified as TBD?
- Can all of the requirements be implemented within known constraints and in scope for the project?
- Are any specified error messages unique and meaningful?

Quality Attributes

- Are all performance objectives properly specified?
- Are all security and safety considerations properly specified?
- Are other pertinent quality attribute goals explicitly documented

3.3 Static Analysis by Tools

Learning Objectives

- Recall typical defects and errors identified by static analysis and compare them to reviews and dynamic testing (K1)
- Describe, using examples, the typical benefits of static analysis (K2)
- List typical code and design defects that may be identified by static analysis tools (K1)

The objective of static analysis is to find defects in software source code and software models, but not written documents. It requires the use of tools and is performed without actually executing the software being examined by the tool (whereas dynamic testing does execute the software code.) As with reviews, static analysis finds defects rather than failures.

Static analysis does not test functionality but can locate structural defects that are hard to find in dynamic testing. Subtle coding errors and fault-prone code can often be found far more quickly using these tools than by visual inspection or testing.

The value of static analysis is:

- Early detection of defects prior to test execution
- Early warning about suspicious aspects of the code or design by the calculation of metrics,such as a high complexity measure
- Identification of defects not easily found by dynamic testing
- Detecting dependencies and inconsistencies in software models such as links
- Improved maintainability of code and design
- Prevention of defects, if lessons are learned in development

Benefits of Static Analysis

Static analysis can provide early detection of defects prior to test execution, and early warning about complex code. It can identify structural defects not easily found by dynamic testing, as well as dependencies and inconsistencies in software models, such as links between modules.

It can also help to prevent further defects, if lessons are learned in development. Static analysers are usually customisable, so they can check coding standards and improve maintainability of code and design.

Generally, it is much quicker to find and fix these types of defects before setting up data and running dynamic tests. For example, discovering high complexity might result in a major redesign.

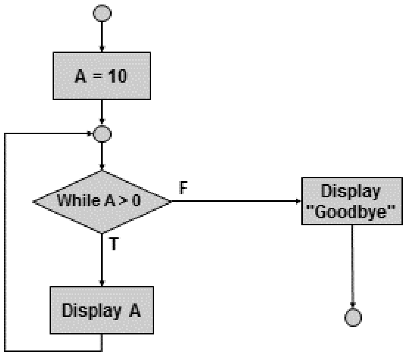
Typical defects discovered by static analysis tools include:

- Referencing a variable with an undefined value
- Variables that are not used or improperly declared
- Inconsistent component interfaces
- Unreachable (dead) code, including uncalled functions and procedures
- Missing and erroneous logic, e.g. infinite loops
- Overly complicated constructs
- Syntax violations
- Programming standards violations
- Security vulnerabilities

Many of these faults, such as syntax checking and undeclared variables, can be detected by compilers. Procedure and variable cross-references produced by the compiler will show up uncalled functions and procedures and where variables are used. Compilers can also calculate metrics and complexity measures.

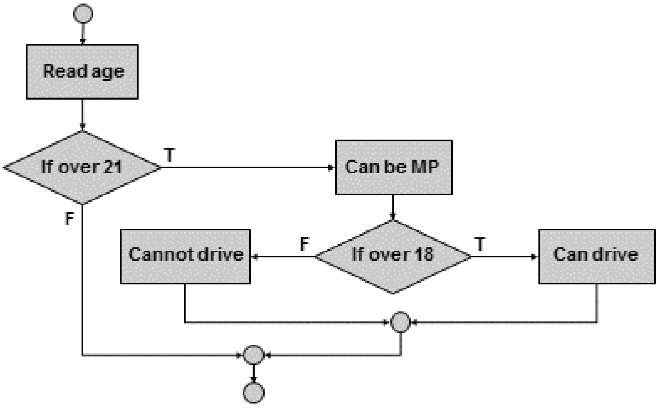
Examples of Defects Found by Static Analysis

Looping Code



This will loop forever as the condition ‘While A > 0’ will always be True.

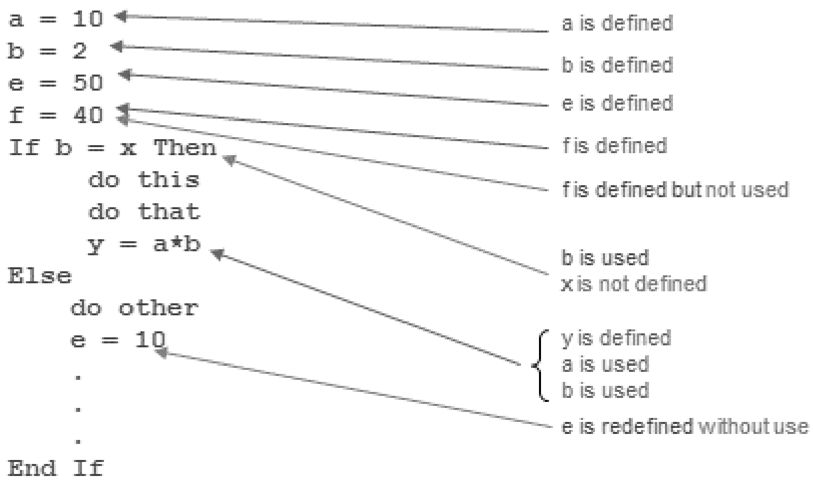
Unreachable Code



The statement ‘Cannot drive’ will never be executed as the condition ‘If over 18’ will always be True.

Data Flow Analysis

Examples of the typical problems that a data flow analyser would detect are shown below (assume a standalone section).



```
a = 10
b = 2
e = 50
f = 40
If b = x Then
      do this
      do that
      y = a*b
Else
     do other
     e = 10
     .
     .
     .
End If
```

Use of Static Analysis Tools

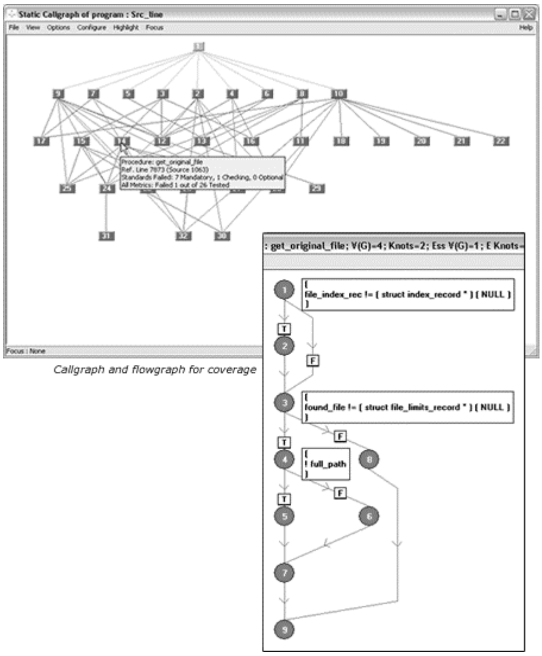
Static analysis tools are typically used by developers, checking against predefined rules or programming standards, before and during component and integration testing, and by designers during software modelling.

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

Many of these tools provide a graphical view of the layout of the code, along with various measures of code complexity such as LOC and Cyclomatic Complexity. This can be useful for testers to:

- Locate complex source code, which might indicate possible defect clustering and a focus for dynamic tests
- Produce graphical output (e.g. control flow graphs)

- Design structural (white box) tests based on control flow, data flow or other criteria
- Examine source code from a number of perspectives based on industry accepted measures.


Static Callgraph of program : Src_line
File View Options Configure Highlight Focus Help
Focus : None
T
F

```
Procedure: get_original_file
Ref. Line 7873 (Source 1063)
Standards Failed: 7 Mandatory, 1 Checking, 0 Optional
All Metrics: Failed 1 out of 26 Tested
```

```
: get_original_file; V(G)=4; Knots=2; Ess V(G)=1; E Knots=
(
file_index_rec != ( struct index_record * ) ( NULL )
)
(
found_file != ( struct file_limits_record * ) ( NULL )
)
(
! full_path
)
```

Callgraph and flowgraph for coverage

Notes

Notes

Module 4 – Test Design Techniques

Topics

1. **The Test Development Process**
2. **Categories of Test Design Techniques**
3. **Specification-based or Black-box Techniques**
4. **Structure-based or White-box Techniques**
5. **Experience-based Techniques**
6. **Choosing Test Techniques**

4.1 The Test Development Process

Learning Objectives

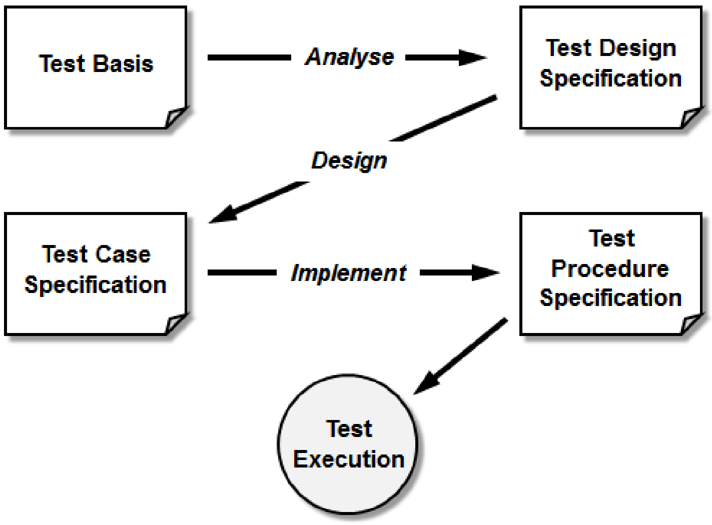
- Differentiate between a test design specification, test case specification and test procedure specification (K2)
- Compare the terms test condition, test case and test procedure (K2)
- Evaluate the quality of test cases in terms of clear traceability to the requirements and expected results (K2)
- Translate test cases into a well-structured test procedure specification at a level of detail relevant to the knowledge of the testers (K3)

The Test Development Process is a breakdown of the Analysis and Design phase of the Fundamental Test Process discussed in Chapter 1.

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

The formal test development process comprises three main steps:

1. **Analyse** the test basis, identify the test conditions and document them in the Test Design Specification
2. **Design** the test cases required to test the test conditions and document them in the Test Case Specification
3. **Implement** the test procedures for executing the test cases and document them in the Test Procedure Specification



IEEE829 – Standard for Software Test Documentation

The ‘Standard for Software Test Documentation’ (IEEE Std 829-1998) describes the content of eight documents to be produced during software testing.

1. Test Plan
2. Test Design Specification
3. Test Case Specification
4. Test Procedure Specification
5. Test Item Transmittal Report
6. Test Log
7. Test Incident Report
8. Test Summary Report

This chapter describes the three specification documents. The other documents in this standard will be covered later.

Test Design Specification – documents the test conditions (coverage items) for a test item, the detailed test approach and the associated high level test cases.

Test Case Specification – documents a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item.

Test Procedure Specification – documents the sequence of actions for the execution of a test. If the test is to be executed manually, it is called a test procedure; if executed by automated tools it is called a test script.

Test Condition

Test Condition – *An item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, quality attribute, or structural element* – ISTQB Glossary

The test conditions are Identified from the test basis, and documented in Test Design Specification

The Test Design Specification contains:

- Test conditions, and
- High-level test cases

Test conditions are the lowest-level testable elements that can be derived from the test basis. These are then combined into meaningful business combinations, called high-level test cases.

For example, suppose a requirement states: "*User enters customer code which must be in valid format AA999. If customer is registered on database, customer details are displayed.*" This would give test conditions such as:

- Valid customer code, invalid customer code, registered customer, non-registered customer

These are then combined into high-level test cases, such as:

- The input of a valid registered customer code
- The input of an valid format but not registered customer code

Test Case

Test Case – *A set of input values, execution pre-conditions, expected results and execution post-conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement* – ISTQB Glossary

Test Cases are documented in the Test Case Specification and can be thought of as a combination of elements:

- **Input values** – one or more test conditions (already grouped into high-level test cases)
- **Execution pre-conditions** – state of the system prior to test execution
- **Data requirements** – both input and already on system
- **Expected results** – such as screen display, error message, data storage
- **Execution post-conditions** – state of system after test execution

A test case based on the example above might contain:

Test condition:	The input of a valid registered customer code
Input value:	AD015
Pre-condition:	Customer AD015 on database
Expected result:	Customer details displayed
Post-condition:	No change

Traceability

Test conditions and cases should be linked directly to the test basis (e.g. requirements) to provide traceability.

A Dewey numbering system may be used (e.g. 1.1, 2.6.3, etc.) However, when conditions are spread across multiple cases and vice versa traceability becomes much more difficult. A Traceability Matrix can be created in Excel or using test tools, linking test cases directly to requirements, or via test conditions. This ensures that:

- Each requirement is adequately covered by tests
- Each test has a purpose related to requirements

TRACEABILITY MATRIX					
Requirements	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**
SR1: The modem port shall be initialised on system	✓	✓	✓		
SR2: Protocol shall be no parity, 8 data bits, 1 stop bit	✓				
SR3: Upon command from the front end, the system …		✓		✓	✓
SR4: The format for the data frame is as described in …			✓		
SR5: Display output will be controlled via a dedicated …				✓	
SR6: The refresh rate for the waveform data display …				✓	
SR7: ECG input shall be acquired by the software from …					✓
SR8: After the sample ECG is acquired, it is displayed …					✓

Test Procedure

Test Procedure – *A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script* – ISTQB Glossary.

The test procedure (Documented in the Test Procedure Specification) contains all instructions for running the test cases, such as:

- Where the test data is and how to load it
- Login to test system
- Step by step instructions to run the test cases
- Where and how to record and check the test results
- Action to take to finish test, e.g. backup data

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

Example test procedure steps:

Step	Action	Expected Result
1	Login with user id "A1234" password "P@ssw0rd"	Main menu displayed
2	Double click “Customer Inquiry"	Customer Inquiry screen displayed
3	Enter Customer ID “AD015"	Customer AD015 details displayed
4	Click “Account" button	Customer Account screen displayed

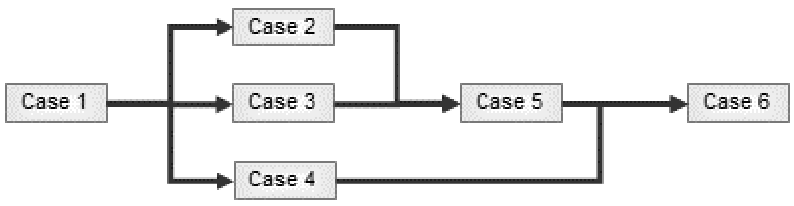
Test Execution Schedule

The various test procedures and automated test scripts are subsequently formed into a Test Execution Schedule that defines the order in which they are executed.

The order of execution may be based on:

- Logical or technical dependencies of test cases (matching pre and post conditions)
- Order of business functions or related activities
- Prioritisation due to risk analysis
- Test cycles or phases (e.g. regression testing)
- Availability of test environment or hardware

The Test Execution Schedule is not part of the IEEE 829 standard, but is widely used by testers.



4.2 Categories of Test Design Techniques

Learning Objectives

- Recall reasons that both specification-based (black-box) and structure-based (white box) test design techniques are useful and list the common techniques for each (K1)
- Explain the characteristics, commonalities, and differences between specification based testing, structure-based testing and experience-based testing (K2)

The purpose of a test design technique is to identify test conditions, test cases and test data. It is a classic distinction to denote test techniques as black-box (also called specification-based techniques) or white-box (also called structure-based techniques.)

Specification-Based / Black-Box Test Design Techniques

Black-box test design techniques are a way to derive and select test conditions, test cases, or test data based on an analysis of the test basis documentation. This includes both functional and non-functional testing.

Black-box testing takes an external view of the software and does not use any information regarding the internal structure of the component or system to be tested.

Specification-based tests will show if the software meets specified requirements, but will not discover if the specification is incorrect (i.e. does not match user needs.)

Structure-Based / White-Box Test Design Techniques

White-box test design techniques are based on an analysis of the internal structure of the component or system.

Tests are designed to assess coverage of the structure of the program code or system. 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 Test Design Techniques

Experience-based test design techniques use the knowledge, skill and expertise of testers to derive the test cases.

The experience of testers, developers, users and other stakeholders about the software, its usage and its environment can be used to help design appropriate tests for known systems.

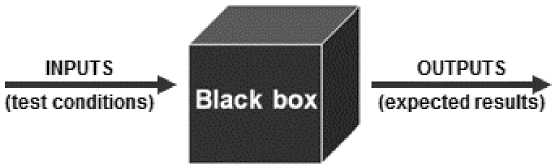
Testers can use their knowledge about likely defects and their distribution to focus and prioritise tests.

4.3 Specification-based or Black-box Techniques

Learning Objectives

- Write test cases from given software models using equivalence partitioning, boundary value analysis, decision tables and state transition diagrams/tables (K3)
- Explain the main purpose of each of the four testing techniques, what level and type of testing could use the technique, and how coverage may be measured (K2)
- Explain the concept of use case testing and its benefits (K2)

All dynamic test levels will include black-box testing - functional and non-functional testing based on analysis of business requirements, designs or specifications.



Five specification-based test design techniques will be described:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Tables
- State Transition Diagrams/Tables
- Use Case Testing

Equivalence Partitioning

In equivalence partitioning, inputs to the software or system are divided into groups or ranges of values that are expected to exhibit similar behaviour, so they are likely to be processed in the same way, i.e. have the same output.

The groups are called equivalence partitions or classes. Instead of testing every value in the partition, one value can be chosen to represent all, and used to test the whole partition.

Partitions can be identified for ranges or sets of values, outputs, time-related values, status values, interface parameters, etc.

Equivalence partitions can be for valid data, i.e. values that should be accepted; and invalid data, i.e. values that should be rejected. Tests can be designed to cover all valid and invalid partitions.

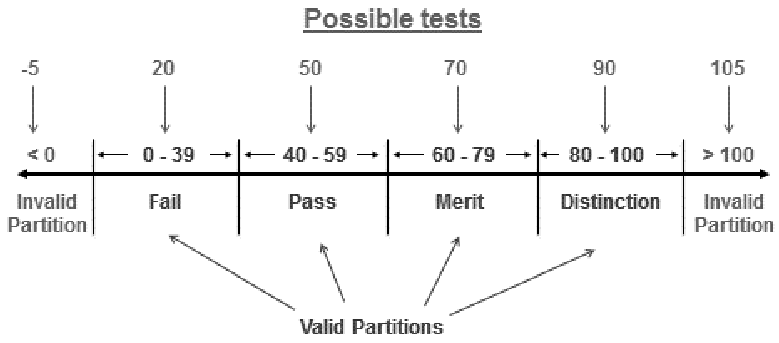
Equivalence partitioning is applicable at all levels of testing and can be used to achieve input and output coverage goals.

Equivalence Partitioning Example

An application is used to assign grades to students taking exams, as follows:

- To pass a student must score at least 40
- To gain a merit a student must score at least 60
- To gain distinction a student must score at least 80
- Any score less than 0 or greater than 100 is invalid

At a minimum we should test one fail, one pass, one merit and one distinction, rather than every possible score. Drawing a simple line chart can help identify the partitions, as shown below.



In this example, there are 4 valid partitions and 2 invalid partitions, giving a total of 6 tests to cover the whole specification.

The diagram also helps to clarify the specification, and identify gaps, e.g. upper/lower boundaries. In this case, the specification did not explicitly state what the output value is for an input less than 40 – we have assumed it is ‘Fail’.

Boundary Value Analysis

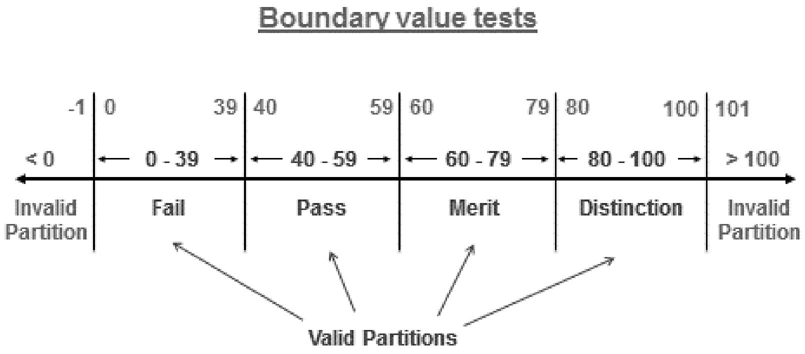
Boundaries are the dividing lines between equivalence partitions. Behaviour at the edge of a partition is more likely to be incorrect than behaviour within the partition, so boundaries are an area where testing is likely to yield defects.

The minimum and maximum values of a partition are its boundary values.

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

Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect finding capability is high.

This technique (aka Boundary Analysis) is often considered as an extension of equivalence partitioning or other black-box test design techniques.



In this example, there are 5 boundaries, giving a total of 10 tests to cover the whole specification. The diagram also helps to clarify the minimum increment between values, e.g. should we test 39 or 39.9 or 39.99?

EP and BVA Exercise

An application will calculate tax automatically for employees in the organisation.

The application should only allow input values below £200,000. If a value of this magnitude or greater is attempted, an error message is returned. Salaries are recorded in whole pounds (not pence).

Employees begin paying tax at a £5,000 threshold.

This first band is charged at 10% and this rate applies to the next £10,000 earned. 20% applies above this until at £40,000 a rate of 40% begins.

Employees may earn zero but there is no concept of negative salary.

Draw a line diagram and mark the boundaries

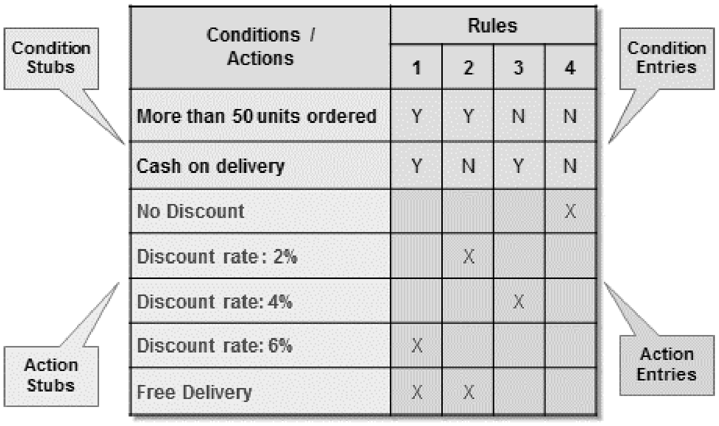
How many valid and invalid partitions are there?

What values would you use for boundary tests?

Decision Tables

Decision Tables represent the behaviour of a system that depends on the outcome of multiple decisions. They may be used to record complex business rules that a system is to implement.

The table specifies the possible combination of conditions and the resulting actions, according the business rules. Each column of the table corresponds to a business rule that defines a unique combination of conditions and which result in the execution of the actions associated with that rule. Conditions are usually stated in Boolean form (True/False or Yes/No).


Condition Stubs
Condition Entries
Action Stubs
Action Entries

Conditions / Actions	Rules			
	1	2	3	4
More than 50 units ordered	Y	Y	N	N
Cash on delivery	Y	N	Y	N
No Discount				X
Discount rate: 2%		X		
Discount rate: 4%			X	
Discount rate: 6%	X			
Free Delivery	X	X		

The process for creating a decision table is as follows:

1. Identify the condition stubs (descriptions) from the test basis.
2. Identify the action stubs (descriptions) from the test basis.
3. Establish the condition entries – all the possible combination of conditions.
4. List the rules from the test basis and work out the action entries for each rule – the outcomes associated with each combination.

Each column corresponds to a test case, in which the condition entries are the test conditions, and the action entries are the expected results.

If there are two conditions, then there are a maximum 4 possible combinations; and for 3 conditions, there can be up to 8 possible combinations. If n is the number of conditions, then:

Number of possible combinations = 2^n

Decision Table Example

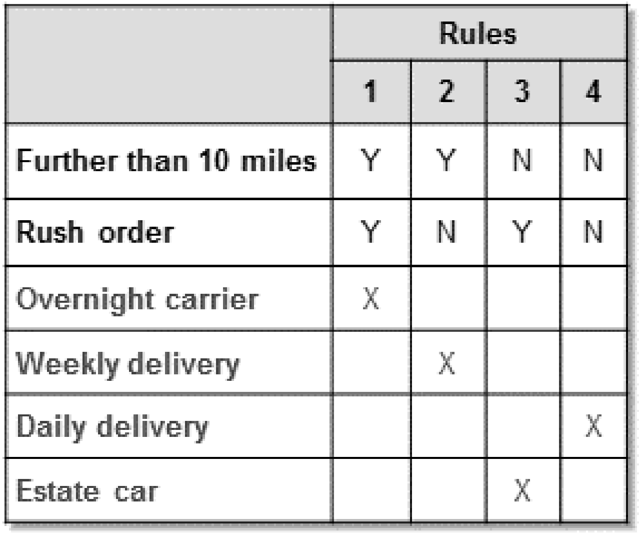
Requirement: Delivering goods from warehouse to shops

For each shop's delivery schedule we look to see how far away the shop is from the warehouse then we go through each item on the schedule.

If the shop is further than 10 miles away we add the item to the next weekly delivery for that area, unless it's a rush order in which case we send it by overnight carrier.

When we have an item to be delivered locally (i.e. less than or equal to 10 miles) we add it to the next day's delivery unless it's a rush order in which case Joe delivers it with an estate car immediately.

This can be represented as a decision table like this:



	Rules			
	1	2	3	4
Further than 10 miles	Y	Y	N	N
Rush order	Y	N	Y	N
Overnight carrier	X			
Weekly delivery		X		
Daily delivery				X
Estate car			X	

In this example each combination of conditions has a specified action and every combination is tested. Often however, there will be combinations of conditions for which there is no specified action, or an exception (error) generated. Then the tester will have to decide how many to test based on risk.

When faced with complex requirements, it is often easier to understand the logic using a diagrammatic technique, in this case a decision table. Like EP and BVA, it also helps to clarify the rules and establish any omissions or ambiguities.

The strength of decision table testing is that it creates combinations of conditions that might not otherwise have been exercised during testing.

If the example is extended to include a third condition, then the number of action columns should expand to eight. However, there are shortcuts which allow columns to be combined if they have the same outcome.

Further Decision Table Example

Requirement: Delivering goods from warehouse to shops

For each shop’s delivery schedule we look to see how far away the shop is from the warehouse then we go through each item on the schedule.

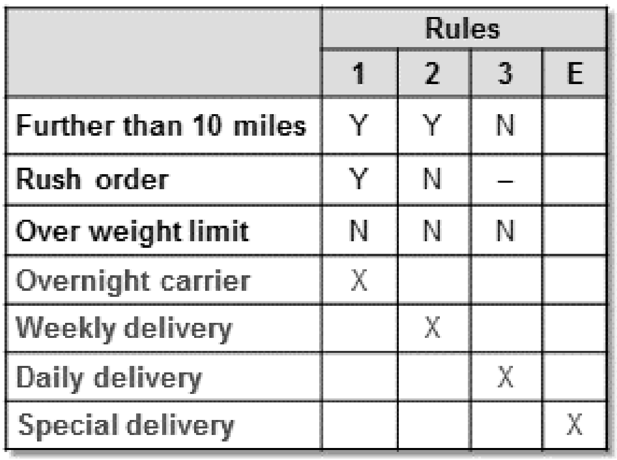
If the shop is further than 10 miles away we add the item to the next weekly delivery for that area, unless it’s a rush order in which case we send it by overnight carrier.

When we have an item to be delivered locally (i.e. less than or equal to 10 miles) we add it to the next day’s delivery.

However, all items that are over the weight limit will need to go by special delivery.

- The **indifference symbol** (–) means either Y or N
- The **else** symbol (E) indicates any other combination not explicitly specified in the business rules

Use of indifference and ‘else’ reduces condition entries but increases the risk of missing something.



	Rules			
	1	2	3	E
Further than 10 miles	Y	Y	N	
Rush order	Y	N	–	
Over weight limit	N	N	N	
Overnight carrier	X			
Weekly delivery		X		
Daily delivery			X	
Special delivery				X

Decision Table Exercise

Cheques are assessed and processed by the bank as follows:

In order for the account to be debited, the account number is checked first. It must be valid with the correct matching signature and sufficient funds for the cheque to be processed.

If the account number is invalid then the cheque is returned to the depositor with an error message.

If the account number is valid but the signature does not match then we will initiate a security check.

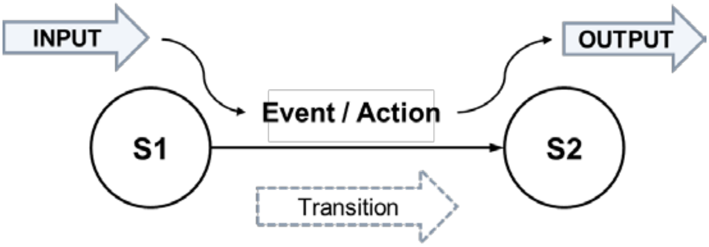
If the account number is valid, with matching signature, the account balance is then checked. If there are not sufficient funds then a letter will be sent to the customer.

Draw a decision table to show the minimum tests which would be required to generate each action.

State Transition Diagrams/Tables

Some systems can be described in terms of ‘states’ or modes that the system can assume, the transition from one state to another because of some input condition, which leads to an event and results in an action which will finally give an output. These systems can be represented by a state diagram showing:

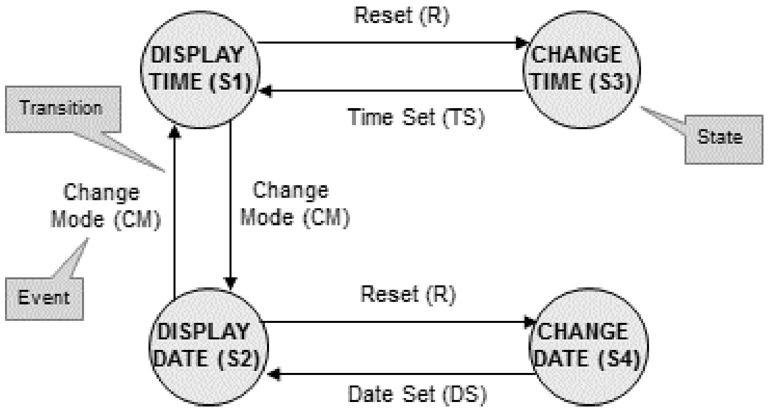
- The **states** a component or system may be in
- The **transitions** or changes form one state to another
- The **events** or circumstances that cause transitions
- The **actions** as a result from the transition, for example an error message



The states of the system or object under test are separate, identifiable and finite in number.

State diagrams are often used for digital systems, embedded software or technical automation in general. However, the technique is also suitable for modelling a business object having specific states or testing screen-dialogue flows (e.g. for internet applications or business scenarios.)

State Transition Testing is a test design technique in which test cases are designed to execute valid and invalid state transitions, by creating a state table showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions.



The above state transition diagram shows the operation of a digital clock.

- States are shown by a box or circle with the name of the state
- Transitions are shown by arrows indicating the change of state
- Events which cause transitions are shown by text on the arrows

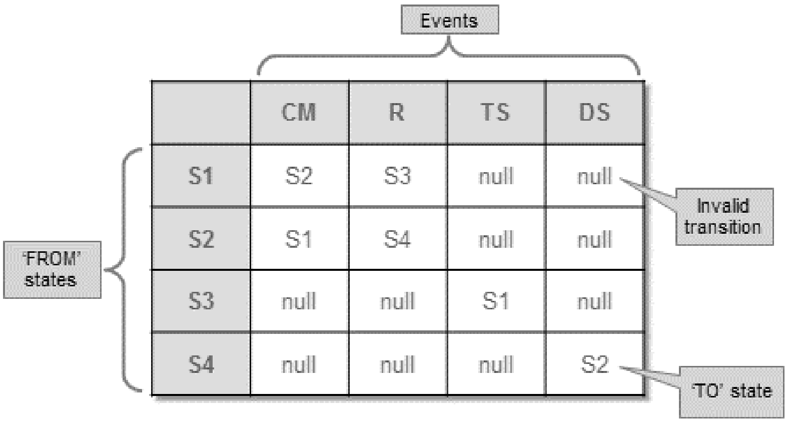
Test cases can be derived from the State Transition Diagram to exercise each of the possible transitions, by creating a State Matrix for every valid transition (or arrow.) The start state and event represent test conditions, and the finish state is the expected result.

Test case	1	2	3	4	5	6
Start state	S1	S1	S3	S2	S2	S4
Event/input	CM	R	TS	CM	R	DS
Finish state	S2	S3	S1	S1	S4	S2

Creating a test case to cover each possible valid single transition in the model is known as 0-switch coverage. It provides the ability to detect the most obvious faults but will not detect more subtle ones which would only be detectable by exercising sequences of transitions.

N-switch (or Chow-switch) testing is a form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. Therefore 0-switch testing means covering 0+1 (or single) transitions.

The State Matrix above shows only valid transitions. However, if we want to show both valid and invalid transitions, we will need to create the State Table:



	CM	R	TS	DS
S1	S2	S3	null	null
S2	S1	S4	null	null
S3	null	null	S1	null
S4	null	null	null	S2

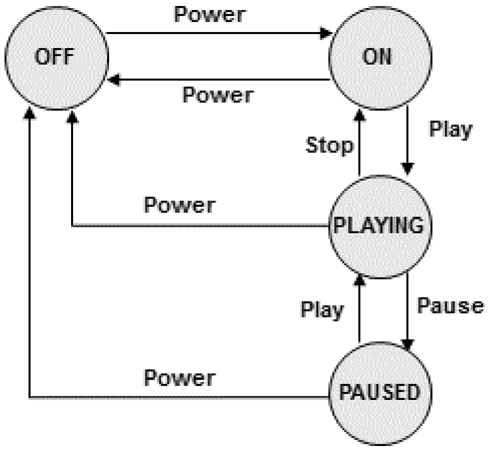
Events
'FROM' states
Invalid transition
'TO' state

The table is filled in with the outcome of performing each transition from each start state. If this result is unknown or not shown on the state diagram, the corresponding cell is left blank or marked ‘null’ to indicate an invalid transition.

State tables are therefore a useful testing aid to derive the invalid “null” transitions directly. The decision whether to test invalid as well as valid transitions will depend on context and risk.

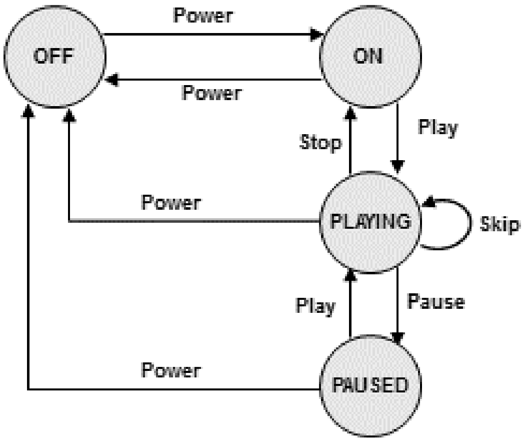
State Transition Exercise

Draw a state table using this State Transition Diagram which represents the states an MP3 player can assume:



Same State Transition

Events that cause a system to remain in the same state are shown with a recursive arrow. These state transitions must also be included when defining test cases. For example, in the diagram below, if you press the “skip” button on the MP3 player while it is playing it will start playing the next track.



When testing valid transitions (0-switch transitions) on a state transition diagram, a same-state transitions counts as one, just like any other arrow.

This illustrates that State diagrams do NOT show all system functionality, only transitions between states. Sometimes, outcomes are shown on diagrams under the arrows. Or there may be a separate text description accompanying the state diagram, which should also be used as a test basis.

Use Case Testing

Use Case Testing is a black-box test design technique in which test cases are designed to execute scenarios of use cases.

Use Case: *A sequence of transactions in a dialogue between an actor and a component or system with a tangible result, where an actor can be a user or anything that can exchange information with the system* – ISTQB® Glossary

Use cases are a way of defining high-level requirements or business processes viewed from a user perspective. They may be described at the abstract level (business use case, technology-free, business process level) or at the system level (system use case on the system functionality level).

Contents of a Use Case

- Actor List
- Pre conditions
- Main process flow (i.e. most likely path)
- Alternate flow(s)
- Exception flow(s)
- Post conditions

Use Case Example Main Flow

'Create Policy' Use Case

1. User selects ‘Create Policy’ from the ‘Maintain Policies’ menu.
2. The system displays a Quote ID input field.
3. User enters a valid Quote ID.
4. The system displays the quote details (name, address, customer ID, policy type, term, cost).
5. User checks the details, enters the start date and clicks on ‘Setup DD’ button.
6. The System displays the ‘Create Direct Debit’ screen.
7. User enters the customer’s bank details and premium collection amount and clicks on ‘Setup’.
8. The system generates the policy and displays ‘Policy No xxxxxx created’.

This structure is very similar to a test procedure (script), so is ideal for designing test cases, particularly UAT with customer or user participation.

Process flows in a Use Case are written based on intended use which makes them very useful for designing acceptance testing by end users. The descriptions could act as the test procedure thus minimising test development costs. However, they are detailed enough to test properly and therefore uncover errors.

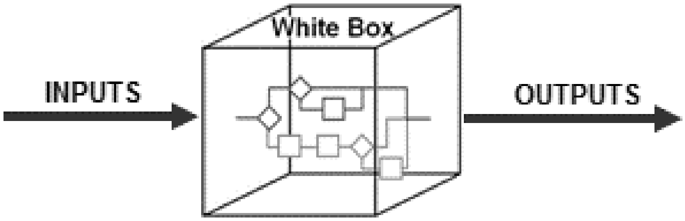
Use cases can also uncover integration defects which individual component testing would miss (similar to functional integration testing).

4.4 Structure-Based or White-Box Techniques

Learning Objectives

- Describe the concept and value of code coverage (K2)
- Explain the concepts of statement and decision coverage, and give reasons why these concepts can also be used at test levels other than component testing (e.g., on business procedures at system level) (K2)
- Write test cases from given control flows using statement and decision test design techniques (K3)
- Assess statement and decision coverage for completeness with respect to defined exit criteria. (K4)

Structure-based or white-box testing is based on an identified structure of the software or system. It does not require an analysis of business requirements or specifications and does not assess the functionality of software. It ensures that a structure, or specific parts of a structure, is adequately covered by tests.



Structural testing can apply at all test levels, wherever the system can be represented in a structural or architectural diagram, e.g.

- **Component** level: the structure of a software component, i.e. statements, decisions, branches or even distinct paths
- **Integration** level: the structure may be a call tree (a diagram in which modules call other modules)
- **System** level: the structure may be a menu structure, business process or web page structure

This chapter will focus on designing white-box tests to measure internal structure of software code. This would normally be done by developers or designers as part of component testing.

White-box testing, particularly measurement of code structures, is a very comprehensive form of testing but can be complex, time-consuming and resource-intensive, and may be too detailed for many businesses. It is often carried out using specialist tools.

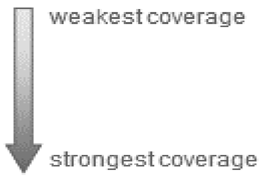
It is most likely to be conducted where extremely thorough testing is required, such as safety-critical systems, defence control systems, medical equipment software, etc.

Typically, white-box testing would be done after black-box testing, to ensure adequate coverage. Dynamic coverage measurement tools would be used to measure actual coverage achieved during functional and non-functional black-box testing, then further white-box tests would be designed to complete the coverage required.

Coverage

The key concept in structural testing is coverage. Coverage is expressed as a percentage of a structure covered by tests, and all white-box testing can be used to measure coverage of structure. There are different coverage measures of software code structure:

- Statement coverage
- Decision coverage
- Condition coverage
- Multiple condition coverage
- All Paths coverage



We only need to know techniques for statement and decision coverage at Foundation level. But we need to be aware that stronger techniques exist.

Two code-related structural test design techniques for code coverage, based on statements and decisions, are discussed. For decision testing, a control flow diagram or the simplified control flow graphs (CFGs), may be used to visualize the alternatives for each decision.

Statement Testing and Coverage

A statement is an atomic action, i.e. a single instruction to the computer, for example:

```
Gross_Amount = Net_Amount + Tax
```

Test cases are devised to execute specific executable statements at least once. This approach does not ensure complete coverage of functionality, but it does provide a level of confidence that simple coding errors have been detected.

$$\text{Statement Coverage} = \frac{\text{no. of statements executed}}{\text{total no. of statements}} \times 100$$

Decision Testing and Coverage

A decision has two possible outcomes: True and False, for example:

```
If Value between 50 and 60 Then
  Do Action A
Else
  Do Action B
End If
```

Test cases are devised to execute specific decision outcomes or branches.

$$\text{Decision Coverage} = \frac{\text{no. of decision outcomes executed}}{\text{total no. of decision outcomes}} \times 100$$

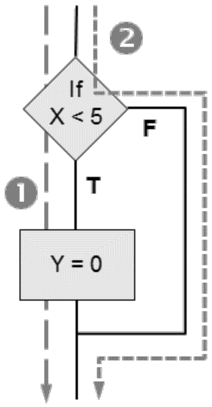
All decisions are statements, so 100% decision coverage guarantees 100% statement coverage (but not vice versa.) Decision testing is a form of control flow testing as it generates a specific flow of control through the decision points.

Note that in the above example, it is not necessary to test equivalence partitions or boundaries for this level of structural testing. Just one value in the range (True) and one outside (False) would be enough to achieve 100% decision coverage, so 100% coverage doesn't necessarily mean exhaustive testing!

Statement and Decision Coverage Example 1

Software source code and flowcharts can both be used to assess coverage:

```
If X < 5 Then
  Y = 0
End If
```



Test 1: Using a value of 3 for X will cause the condition '`If X < 5`' to evaluate to True, so executing the statement '`Y = 0`'. This test executes all the statements, so has achieved 100% statement coverage.

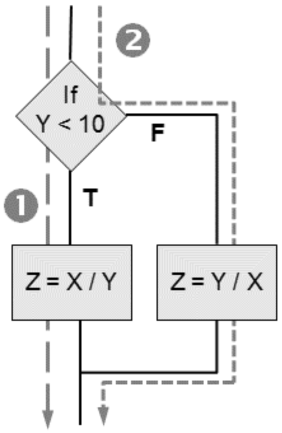
Test 2: Using a value of 8 for X will cause the condition to evaluate to False, so the system will jump directly to the '`End If`' statement, skipping the line '`Y = 0`'. Both tests are required to achieve 100% decision coverage.

So full statement coverage can be achieved without exercising all the software structure. This is why statement coverage is considered the weakest coverage measure, and decision coverage is a stronger measure.

Statement and Decision Coverage Example 2

Consider the following code segment:

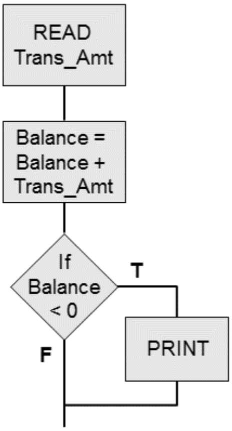
```
If Y < 10 Then
  Z = X / Y
Else
  Z = Y / X
End If
```



In this example, there are statements on both the True and False branches. So two tests are required for 100% statement coverage as well as for 100% decision coverage.

White-Box Exercise 1: One Decision

```
READ Trans_Amt
Balance = Balance + Trans_Amt
If Balance < 0 Then
  PRINT "Account in Debit"
End If
```

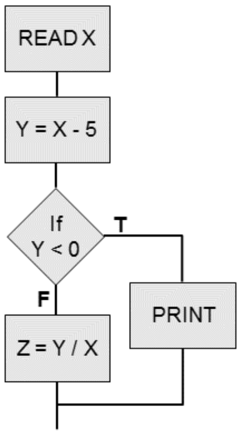


What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

White-Box Exercise 2: One Decision With **ELSE**

```
READ X
Y = X – 5
If Y < 0 Then
   PRINT "Y negative"
Else
   Z = Y / X
End If
```



What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

What is the actual % statement and decision coverage achieved by a test case where X = 10?

White-Box Exercise 3: Create Your Own Flowchart

```
READ No_Items
READ Item_Price
Sales_Total = No_Items * Item_Price
If Sales_Total > 100 Then
  Discount = Sales_Total * 0.05
  Net_Total = Sales_Total - Discount
Else
  Net_Total = Sales_Total
End If
PRINT Net_Total
```

What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

White-Box Exercise 4: Two Sequential Decisions

```
READ X
READ Y
If X > 0 Then
  PRINT "X positive"
Else
  PRINT "X negative"
End If
If X > Y Then
  PRINT "X more than Y"
Else
  PRINT "X not more than Y"
End If
```

What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

White-Box Exercise 5: Two Nested Decisions

```
READ New_Customer_Count
If New_Customer_Count = 0 Then
  DISPLAY "EMPTY FILE"
Else
  If New_Customer_Count > 100 Then
    DISPLAY "TARGET REACHED"
  Else
    DISPLAY "TARGET NOT REACHED"
  End If
End If
```

What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

What is the actual % statement and decision coverage achieved by a test case where New_Customer_Count = 0?

White-Box Exercise 6: In English

Requirements

A customer is ordering goods online.

If the customer spends more than £100, then the postage is free. Otherwise postage costs £10 if the order weighs more than 2kg, or £5 for any other weight.

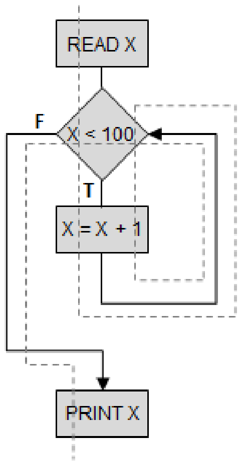
What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

Loops

Loops are an important software construct, but counting paths for white-box testing doesn't work quite the same way for loops as for simple IF decisions. For example:

```
READ X
While X < 100 Then
  X = X + 1
End Loop
PRINT X
```



If the WHILE decision outcome is True when it is evaluated the first time, then control passes through the True branch and loops back to the decision, where it is re-evaluated. As long as it remains True, the loop will continue.

When the decision outcome becomes False, control passes out of the loop to the statements after it.

This means that both the True and False branches can be covered by a single test, which can never happen with an IF decision.

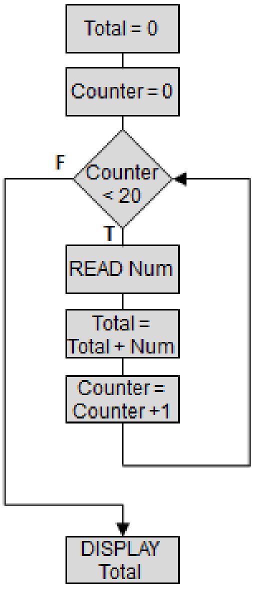
Note that this assumes the loop is not infinite, i.e. if the decision is True the first time, it will eventually become False.

If the WHILE decision outcome is False when it is evaluated the first time, then the loop is never entered.

The flowchart for loops looks slightly different from IF decisions as the decision diamond has two control lines flowing into it, one from the preceding statement and one from the end of the loop.

White-Box Exercise 7: Loop

```
Total = 0
Counter = 0
While Counter < 20 Then
  READ Num
  Total = Total + Num
  Counter = Counter + 1
End Loop
DISPLAY Total
```



What is the minimum number of tests needed to achieve

- ***100% statement coverage?***
- ***100% decision coverage?***

4.5 Experience-based Techniques

Learning Objectives

- Recall reasons for writing test cases based on intuition, experience and knowledge about common defects (K1)
- Compare experience-based techniques with specification-based testing techniques (K2)

Experience-based testing is where tests are derived from the tester's skill or intuition, and their experience of similar applications and technologies. It is not based on analysis of specifications or structure.

It is generally less thorough than systematic techniques (black-box and white-box), but it can be more effective if there is a lack of testing time or resources, and it can identify special tests not easily captured by formal techniques.

Testers use their expertise to identify key areas to test, such as:

- Critical or most-used functions
- Areas most likely to fail (e.g. defect clusters or complex code)

Experienced test designers are best suited to this approach, which if possible, should be used to augment the systematic techniques. It may be much less effective for inexperienced testers.

Error Guessing

In error guessing, testers use their knowledge of systems and testing to anticipate possible defects. They design tests to target known or suspected weaknesses, or address aspects of the system that have caused problems in the past. Testers should think laterally and consider how users *could* use the system, not just how they *should* use it (which is the aim in specification-based testing.)

Error guessing can often be used after black-box testing to cover specific problem areas more thoroughly.

The success of error guessing depends on the knowledge and skill of the tester. But the effectiveness can be improved if several testers, users or developers contribute to identifying possible errors.

Fault Attack

A structured approach to error guessing, called a Fault Attack, is to enumerate a list of possible defects and then design tests to force each one in turn. These defect and failure lists can be based on:

- Experience
- Available defect and failure data
- Common knowledge about why software fails
- List of error messages from program specification
- Ideas from groups of testers, users and developers

Exploratory Testing

Exploratory testing combines tester experience with a structured approach. Testers explore the software and learn about it by testing. The technique involves concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, scope and tasks, and carried out within time-boxes.

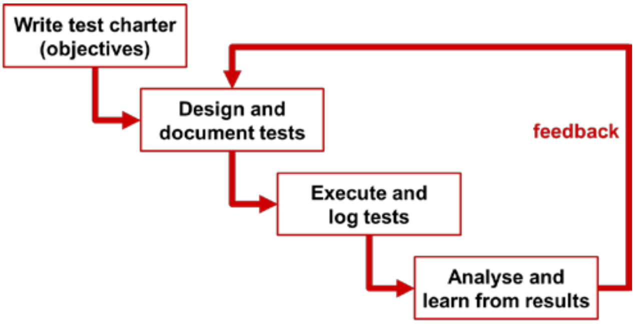
It is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing. It can serve as a check on the test process, to help ensure that the most serious defects are found.

Exploratory testing is not purely random or ad-hoc, as it involves a high-level plan, and all test activities (design, execution log, incidents) must be documented. But unlike specification-based testing, exploratory testing deviates from the plan to ‘explore’ areas of interest and learn more about the system.

It may be used on its own, or in conjunction with other techniques to ensure that testing is focused on the most important areas and identify possible problems.

Exploratory testing is common within agile methodologies, which often incorporate limited specification documents with short development timescales.

The process is:


Write test charter (objectives)
Design and document tests
Execute and log tests
Analyse and learn from results
feedback

1. Start with a high-level test design. As there is limited or no specification, this may be just to explore basic functions (such as user input) or to follow the ‘happy path’ processing, where no error conditions are met.
2. Execute the tests and log the outcomes.
3. Investigate any unexpected results. In standard black-box testing, these would be recorded as incidents, and the rest of the scripted tests would be executed as planned. But in exploratory testing, the plan is suspended and new tests are designed to explore the anomalies and learn about their cause. Deviating from the plan to explore areas of interest is sometimes called the ‘tour bus principle’
4. Document just enough to ensure that all tests are repeatable, typically in an Exploratory Test Session Sheet. This also increases the documented knowledge about the system.

4.6 Choosing Test Techniques

Learning Objectives

- Classify test design techniques according to their fitness to a given context, for the test basis, respective models and software characteristics (K2)

Studies have been unable to prove conclusively that one type of testing is better than another, so it is important to select the most appropriate depending on circumstances.

The choice of which test techniques to use depends on a number of factors, such as:

- 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 life cycle and test level
- Use case models
- Previous experience of types of defects found

For example:

Factor	Appropriate test technique
Web-based system	State transition testing
Rules-based requirements	Decision tables
High-risk or safety-critical	White-box testing
Detailed specifications	Black-box testing
Unavailable specifications	Exploratory testing
Availability of source code	White-box testing
Agile methodology or limited time	Experience-based techniques

Choosing the Best Technique

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

Specification-based techniques are used

- In sequential development models
- At all levels of testing
- For business-critical systems

Structure-based techniques are used

- Mostly at lower test levels (component testing, component integration testing)
- For safety-critical systems

Experience-based techniques are used

- To augment systematic techniques
- In iterative / agile development models

When creating test cases, testers generally use a combination of test techniques to ensure adequate coverage of the object under test.

Notes

Notes

Module 5 – Test Management

Topics

1. **Test Organisation**
2. **Test Planning and Estimation**
3. **Test Progress Monitoring and Control**
4. **Configuration Management**
5. **Risk and Testing**
6. **Incident Management**

5.1 Test Organisation

Learning Objectives

- Recognise the importance of independent testing (K1)
- Explain the benefits and drawbacks of independent testing within an organisation (K2)
- Recognise the different team members to be considered for the creation of a test team (K1)
- Recall the tasks of typical test leader and tester (K1)

Test Organisation and Independence

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

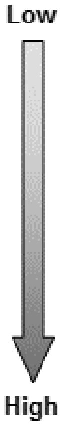
Development staff may participate in testing, especially at the lower levels, but their lack of objectivity often limits their effectiveness.

For large, complex or safety critical projects, it is usually best to have multiple levels of testing, with some or all of the levels done by independent testers.

Testing tasks may be done by people in a specific testing role, or may be done by someone in another role, such as a project manager, quality manager or business and domain expert.

Options for independence in a test team include:

- No independent testers (developers testing their own code)
- Independent testers within the development teams (other developers doing the testing)
- Independent test team or group within the organisation
- Independent testers from the business organisation or user community
- Independent test specialists for specific test types such as usability testers, security testers or certification (regulatory) testers
- Independent testers outsourced or external to the organisation



The **benefits** of independent testers include:

- They will see other and different defects, and are unbiased.
- They verify assumptions made during specification and implementation of the system
- They bring experience, skills, quality and standards

Drawbacks include:

- Isolation from the development team, leading to possible communication problems
- They may be seen as bottleneck or blamed for delays in release
- They may not be familiar with the business, project or systems
- Developers may lose a sense of responsibility for quality

Tasks of the Test Leader and Tester

Most test teams consist of two distinct roles, test leader and tester. The activities and tasks performed by people in these two roles depend on the project and product context, the people in the roles, and the organisation.

Tasks of the Test Leader

The role of the test leader (or test manager / test coordinator) may be performed by a project manager, a development manager, a quality assurance manager or the manager of a test group. Typically the test leader plans, monitors and controls the testing activities and tasks described later.

Typical test leader tasks may include:

- Write or review the test policy and strategy for the organisation
- Coordinate the plan with project managers and others
- Contribute the testing perspective to other project activities, such as development, integration and implementation
- Plan the tests, including selecting test approaches, estimating the time, effort and cost of testing, acquiring resources, defining test levels, cycles, and planning incident management
- Assess the test objectives, context and risks
- Initiate the specification, preparation, implementation and execution of tests
- 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
- 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
- Select tools to support testing and organise any training in tool use for testers
- Supervise the implementation of the test environment
- Write test summary reports based on the information gathered during testing

Tasks of the Tester

Testers (or test analysts) who work on test analysis, test design, specific test types or test automation may be specialists in these roles. Depending on the test level and the risks related to the product and the project, different people may take over the role of tester, keeping some degree of independence.

Typically testers at the component and integration level would be developers, testers at the acceptance test level would include business experts and users, and testers for operational acceptance testing might be operators or system administrators.

Typical tester tasks may include:

- Review and contribute to test plans
- Analyse and review user requirements, specifications and models for testability
- Create test specifications
- Set up the test environment with appropriate technical support
- Prepare and acquire test data
- Implement tests on all test levels, execute and log tests
- Evaluate results and record incidents
- Use test tools as necessary and automate tests
- Measure performance of systems as necessary
- Review tests developed by others

5.2 Test Planning and Estimation

Learning Objectives

- Recognise the different levels and objectives of test planning (K1)
- Summarise the purpose and content of the test plan, test design specification and test procedure documents according to the ‘Standard for Software Test Documentation’ (IEEE Std 829-1998) (K2)
- Differentiate between conceptually different test approaches, such as analytical, model based, methodical, process/standard compliant, dynamic/heuristic, consultative and regression-averse (K2)
- Differentiate between the subject of test planning for a system and scheduling test execution (K2)
- Write a test execution schedule for a given set of test cases, considering prioritisation, and technical and logical dependencies (K3)
- List test preparation and execution activities that should be considered during test planning (K1)
- Recall typical factors that influence the effort related to testing (K1)
- Differentiate between two conceptually different estimation approaches: the metrics based approach and the expert-based approach (K2)
- Recognise/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 and estimation is a breakdown of the Planning activity of the Planning and Control phase of the Fundamental Test Process discussed in Chapter 1.

It outlines the purpose of test planning within development and implementation projects, and for maintenance activities. Test planning is a continuous activity and is performed in all life cycle processes and activities.

Planning Activities

Planning is the first test activity to be carried out at each test level, but it is a continuous process and is performed throughout the life cycle. Feedback from test activities is used to recognise changing risks so that planning can be adjusted.

Planning is a lot more than scheduling; it will also include:

- Determining the scope and risks and identifying the objectives of testing
- Defining the overall approach to testing, including definition of the test levels, entry and exit criteria and how test results will be evaluated
- Integrate and coordinate testing activities into the software life cycle activities (such as acquisition, supply, development, operation and maintenance)
- Making decisions about testing tasks, roles, schedule and evaluation of test results
- Schedule test analysis, design, implementation, execution and evaluation
- Assigning resources for the different activities defined
- Defining the amount, level of detail, structure and templates for the test documentation
- Selecting metrics for monitoring and controlling testing
- Setting the level of detail for test procedures

Levels of Planning

Test planning may be documented in a hierarchy of documents, each one refining the approach taken to testing projects within an organisation.

Test Policy: *A high-level document describing the principles, approach and major objectives of the organization regarding testing* – ISTQB Glossary.

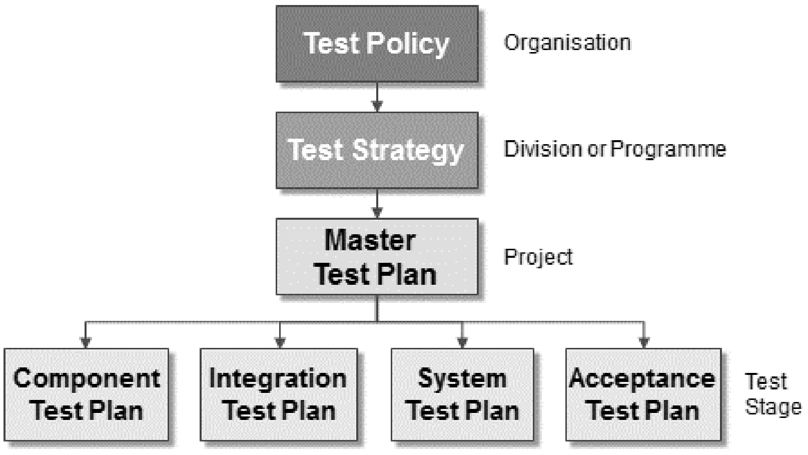
It is set at the highest level of the organisation and is the overall approach to quality assurance giving general guidelines for all project testing. It may be the IT department's philosophy if there is no organisational document.

Test Strategy: *A high-level description of the test levels to be performed and the testing within those levels for an organisation or programme* – ISTQB Glossary. A division or programme sets out their testing strategy based on the Test Policy and identified risks.

There are two main aspects:

- Risks to be resolved by testing the software
- Testing approach used to address the risk

Test Plan: *A document describing the scope, approach, resources and schedule of intended test activities* – ISTQB Glossary. Each project adapts the strategy to create a test plan for that particular project.



There may be further plans for test levels, maintenance testing, regression testing, static testing etc. not all shown here.

Test Plan Outline

Most businesses adopt the IEEE Std 829-1998 Test Plan outline for documenting test planning, but even if the standard is not being used, the approach to test planning should be the same. IEEE 829 defines the following Test Plan contents:

Test Plan Content	Description
Test plan identifier	Unique ID, revision number, author
Introduction	Executive summary and background
Test items	List of software modules
Features to be tested	Application functionality, in user terms
Features not to be tested	Together with Features to be tested, define scope of testing
Approach	From Test Strategy, modified for project: analytical, preventative, reactive, etc.
Item pass/fail criteria	Strictly, this is the criteria for deciding if a test passes or fails, but is usually used for entry and exit criteria
Suspension criteria and resumption requirements	Circumstances that would prevent testing from continuing, such as extremely poor quality software
Test deliverables	Standards for test specifications, logs, incident reports, etc.
Testing tasks	Any specialised tasks not covered in approach
Environmental needs	Software, data, hardware, network, software licences, etc. for each test level

Test Plan Content	Description
Responsibilities	Who manages testing, risks, incidents, reporting, training, etc.
Staffing and training needs	Training in test skills or tools for testers and users, involved in testing
Schedule	Timetable, milestones, delivery dates based on estimates and strategy for control of slippage
Risks and contingencies	Project and product risks and workarounds in case of failure
Approvals	Who has authority to sign off this and other testing documents

Entry Criteria

Entry criteria define when to start testing such as at the beginning of a test level or when a set of tests is ready for execution.

Typically entry criteria might include:

- Test environment available and ready
- Test tool configured in the test environment
- Testable code available
- Test data available, including configuration data, logins, etc.
- Test summary report or evidence available from previous testing, including quality and coverage measures
- Third-party software delivered and software licences bought
- Other project dependencies in place

Entry criteria for one test level might match the exit criteria of the previous level. Examples of entry criteria:

- 80% of system tests completed (for entry to UAT)
- Component testing is completed with decision coverage reports (for entry to system testing)
- All documentation is present and has been signed off

Exit Criteria

Exit criteria define when to stop testing at the end of a test level or when a set of tests has a specific goal. This decision should be based on a measure of software quality achieved as a result of testing, rather than just on time spent on testing.

Typically exit criteria may consist of:

- Measures of testing thoroughness, i.e. coverage of code, requirements, functionality or risk
- Estimates of defect density or reliability
- Cost
- Residual risks such as number of defects outstanding or requirements not tested
- Schedules such as those based on time to market

Typical examples might be:

- All paths coverage of code
- High and medium risk areas completed
- No outstanding high-severity incidents
- All security testing completed successfully

Exit criteria may differ greatly between different levels, for example:

- Coverage of code for component testing
- Coverage of requirements or risk for system testing
- Non-functional measures such as usability in acceptance testing

Test Approaches

The test approach should be set out in the Test Strategy, then refined and implemented in test plans and test designs for a specific project. It is the starting point for planning test activities, choosing test design techniques and defining entry and exit criteria.

Test approaches may be:

- **Preventative**, where tests are designed as early as possible
- **Reactive**, where test design comes after the software has been produced

Typical test approaches include:

- **Analytical** approaches, such as risk-based testing where testing is directed to areas of greatest risk.
- **Model-based** approaches, such as stochastic testing using statistical information about failure rates (such as reliability growth models) or usage (such as operational profiles).
- **Methodical** approaches, such as failure based (including error guessing and fault-attacks), check-list based, and quality characteristic based.
- **Process- or standard-compliant** approaches, such as those specified by industry-specific standards or the various agile methodologies.
- **Dynamic and heuristic** approaches, such as exploratory testing where testing is more reactive to events than pre-planned, and where execution and evaluation are concurrent tasks.
- **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.
- **Regression-averse** approaches, such as those that include reuse of existing test material, extensive automation of functional regression tests, and standard test suites.

Choosing an approach

The selection of an approach depends on:

- Context
- Testing objectives and risks
- Hazards and safety
- Technology
- Available resources and skills
- Regulations
- Nature of the system

Different approaches may be combined, for example, a risk-based dynamic approach.

Test Estimating

Test Estimation – *A calculated approximation of the cost or effort required to complete a task.* ISTQB Glossary

Test managers need to estimate the time, effort and cost of testing tasks in order to plan test activities, identify resource requirements and draw up a schedule. Two possible approaches are:

- The metrics-based approach: based on analysis of similar projects or on typical values.
- The expert-based approach: based on assessment by the owner of the tasks or domain experts.

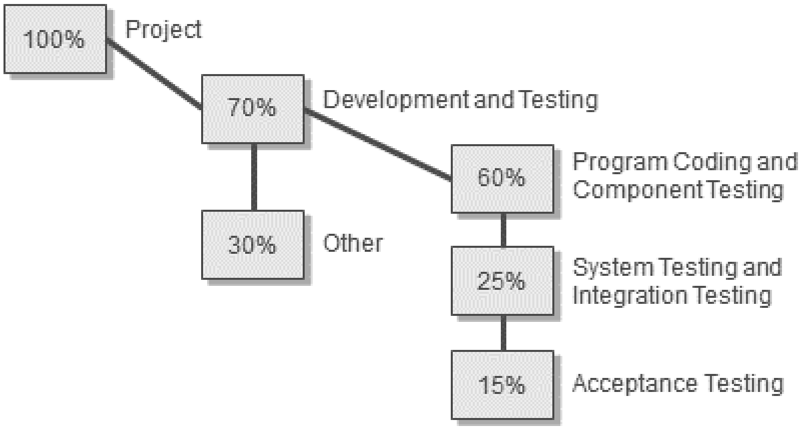
Metrics-based Estimation Example:

Standard Percentages Using Work Distribution Model

Testing is notoriously more difficult to estimate than other project activities such as requirements analysis and development.

Standard Percentages uses industry or company-based historical data to calculate the proportion of time the testing stages take for different project types.

Then if you can size the overall project, you can work out how much testing effort is required, without estimating the testing tasks directly. Alternatively, if you can obtain more accurate estimates for a non-testing part of the project, you can apply the appropriate proportions.


100%
Project
70%
Development and Testing
60%
Program Coding and Component Testing
30%
Other
25%
System Testing and Integration Testing
15%
Acceptance Testing

For example, using the figures above (which are not official) – if the development manager provides an estimate of 40 days for coding and unit testing, then that is equivalent to 60% of the total development and testing time, so system and integration testing (25%) would take 17 days and acceptance testing (15%) 10 days.

Obviously not all projects are the same e.g. a package purchase would require integration and acceptance rather than unit testing.

Expert-based Estimation Example:

Micro-estimates Using Work Breakdown Structure

Experience is still the most widely used approach. Asking the task owner means they are more likely to commit to a deadline and also think through every detail of the job e.g. remembering about setting up test data etc.

Breaking a large task into bite-sized chunks makes it easier to understand and leads to more accurate estimates.

Task	Days
Read functional specification	2
Identify functions to test	1
Attend walkthrough	1
Define test conditions	3
Write test cases and scripts	5
Determine test data	2
Set up test environment	1
TOTAL	**15**

Factors to Consider When Estimating

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

- Product factors, such as quality of the specification , size and complexity of the product and requirements for reliability, security and documentation
- Development process factors and the stability of the organisation, tools used, test process, skills of the people involved, and time pressure
- Quality factors, for example the number of defects and the amount of rework required

Functionality forms the core approach to estimating but all sorts of environmental factors come into play too. Rework is a major consideration: filing incident reports, retesting, changing documents can take an equivalent time to the initial tests.

5.3 Test Progress Monitoring and Control

Learning Objectives

- Recall common metrics used for monitoring test preparation and execution (K1)
- Explain and compare test metrics for test reporting and test control (e.g., defects found and fixed, and tests passed and failed) related to purpose and use (K2)
- Summarise the purpose and content of the test summary report document according to the ‘Standard for Software Test Documentation’ (IEEE Std 829-1998) (K2)

Test Progress Monitoring

Throughout a project, the progress of test activities should be monitored and checked frequently in order to:

- Provide feedback and visibility about testing to stakeholders and managers
- Assess progress against estimated schedule and budget in the test plan
- Measure testing quality, such as number of defects or coverage achieved, against exit criteria
- Assess the effectiveness of the test approach with respect to objectives
- Collect data for future project estimation

The metrics used to measure progress should, if possible, be based on objective data (i.e. numerical analysis of test activity) rather than subjective opinion. These metrics may be collected manually or automatically using test tools such as test management tools, execution tools or defect trackers.

The test manager may have to report on deviations from the test plans such as running out of time before completion criteria achieved.

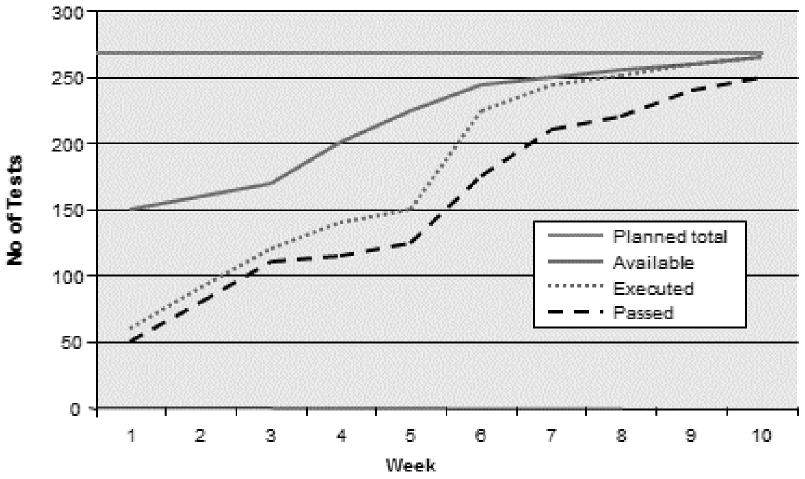
Common test metrics include:

- Percentage of work done in test case and environment preparation
- Test case execution - number of test cases run/not run, test cases passed/failed
- Defect information - Defect density, defects found and fixed, failure rate and retest results
- Coverage of requirements, risks or code
- Dates of test milestones
- Testing costs, Including the cost compared to the benefit of finding the next defect or to run the next test

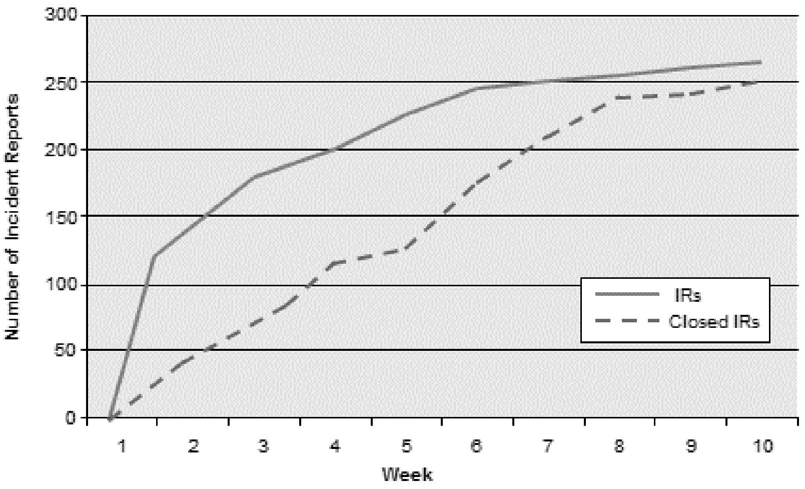
The best metrics are those that match the exit criteria (e.g. coverage, risk and defect data) and estimates (e.g. time and cost) defined in the test plan.

Typical Progress Metrics

Plotting a graph (using a test planning tool or spreadsheet) is a great way to monitor test activity against the plan. The graph below is typical, and allows test managers to check that the number of tests passed does not diverge too far from the number executed.



The second graph shows how fast incidents are dealt with as they are raised. If the lines do not start to converge towards the end of the project, test managers can take appropriate action such as adding more resources to fixes or addressing quality problems.



Test Control

If test monitoring shows that progress is slipping from planned targets, or that exit criteria are not being met, then test managers may have to take some guiding or control actions to get back on track. Options for actions may often be limited, but might include:

- Re-prioritise testing activities (e.g. focus testing on high-risk objectives)
- Change test schedule (e.g. allocate more time to testing)
- Re-assign resources (e.g. assign more testers, or more developers working on fixes)
- Set entry criteria for deliverables from developers (e.g. set minimum quality levels at start of testing)
- Adjust exit criteria (e.g. re-assess acceptable quality measures)

Test Reporting

Test managers will report regularly on test progress to project managers, project sponsors and other stakeholders. The report should be a summary of test endeavour, including what testing actually occurred, statistics on key metrics, and dates on which milestones were met.

Reports will be based on metrics collected from testers and may be consolidated using spreadsheets or test management tools. It is then analysed by project stakeholders to support recommendations and decisions about future actions:

- Assessment of defects remaining
- Economic benefit of continued testing
- Outstanding risks
- Level of confidence in tested software
- Effectiveness of objectives, approach and tests

Test Summary Report

The Test Summary Report is part of the IEEE Std 829-1998, and is used to summarise test activity at the end of a test level. The outline of the standard report is as follows:

Report Item	Description
Summary	What was tested, including software items and versions, test environment, and references to other documents such as test plans, logs and incident reports
Variances	Changes from plan or test specifications, such as tests not followed as per plan, and tests performed in addition to the plan
Comprehensiveness assessment	Assessment of the test process, describing features not tested and reasons
Summary of results	High-level description of incidents, fixes, outstanding issues and state of resolution
Evaluation	Assessment of software quality and estimate of software reliability and failure risk
Summary of activities	Details of the test process, such as elapsed time, cost, resources and personnel used

5.4 Configuration Management

Learning Objectives

- Summarise how configuration management supports testing (K2)

The purpose of configuration management is to establish and maintain the integrity of system products throughout the project life cycle.

It applies to all products connected with software development (such as software components, data and documentation) and testing (such as test plans, test specs, test procedures and test environments.)

Configuration management is therefore more than just version control, it's also about knowing what versions are deployed in different environments at different times.

It ensures that every item of testware (software and test documentation) is uniquely identified, version controlled, tracked for changes, related to each other and related to development items.

Although these items may be physically stored in many locations, configuration management describes and links them in a common controlled library. This ensures that all identified documents and software items are referenced unambiguously in test documentation, which avoids tests being run against the wrong software version.

5.5 Risk and Testing

Learning Objectives

- Describe a risk as a possible problem that would threaten the achievement of one or more stakeholders' project objectives (K2)
- Remember that the level of risk is determined by likelihood (of happening) and impact (harm resulting if it does happen) (K1)
- Distinguish between the project and product risks (K2)
- Recognise typical product and project risks (K1)
- Describe, using examples, how risk analysis and risk management may be used for test planning (K2)

Risk-based testing is seen as the best approach to enabling the best testing to be done in the time available. This section covers measuring, categorising and managing risk in relation to testing. Risk is defined as:

Risk: *A factor that could result in future negative consequences; usually expressed as impact and likelihood* – ISTQB Glossary.

A risk is a specific event which would cause problems if it occurred. Individual risks can be assessed in terms of:

- **Likelihood** – the probability of the event, hazard or situation occurring
- **Impact** – the undesirable consequences if it happens, for example financial, rework, embarrassment, legal, safety or company image

There are two types of risk: **project risk** and **product risk**.

Project Risks

Project risks are the risks that affect the project's (or test team's) capability to deliver its objectives, such as:

Technical Issues

- Problems in defining the right requirements
- The extent that requirements can be met given existing constraints
- Low quality of design, code or test data and tests
- Test environment not ready on time
- Late data conversion or migration planning

Organisational Factors

- Skill, training and staff shortages
- Personnel issues
- Political issues, such as problems with testers communicating their needs and test results, or failure to follow up on information found in testing and reviews
- Unrealistic expectations of testing

Supplier Issues

- Failure of a third party
- Contractual issues

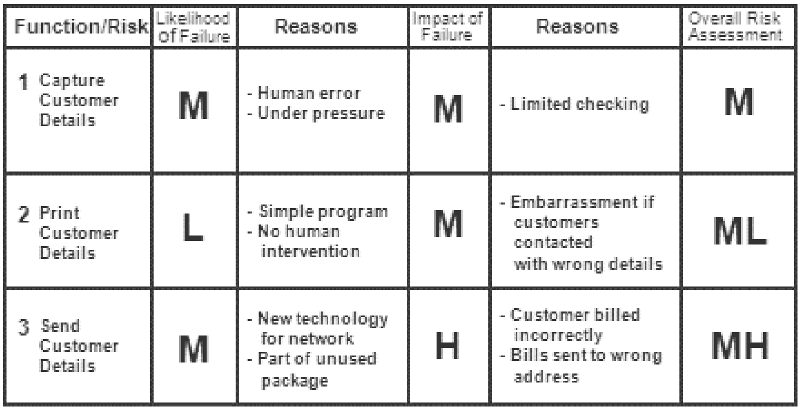
Product Risks

"Product" means the software or system. Potential failures in the software are known as product risks, as they are a risk to the quality of the system. Product risks include:

- Failure-prone software delivered
- Poor software characteristics (e.g. functionality, reliability, usability, performance)
- Poor data integrity and quality (e.g. data migration issues, data conversion problems, data standards violation)
- Software does not perform its intended functions
- Potential for software or hardware to cause harm to an individual or company

Risk Assessment Matrix

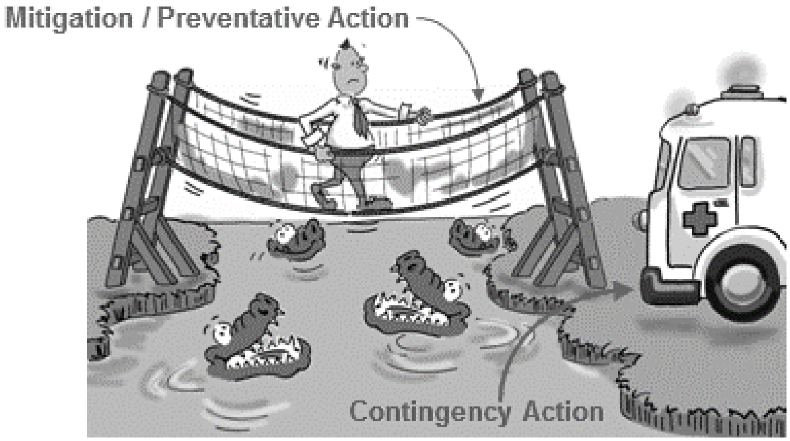
For each risk identified, its likelihood and impact must be assessed. These may be numerical measures (e.g. 1-10) or simply Low, Medium and High. Combining likelihood and impact measures into an overall risk measurement allows risks from different areas to be compared, assessed and prioritised. Risks and their assessments should be recorded in the test plan, perhaps using a matrix like this:



Function/Risk	Likelihood of Failure	Reasons	Impact of Failure	Reasons	Overall Risk Assessment
1 Capture Customer Details	M	- Human error - Under pressure	M	- Limited checking	M
2 Print Customer Details	L	- Simple program - No human intervention	M	- Embarrassment if customers contacted with wrong details	ML
3 Send Customer Details	M	- New technology for network - Part of unused package	H	- Customer billed incorrectly - Bills sent to wrong address	MH

Once a risk has been assessed, there are two possible actions which can be put in place to reduce the risk:

- **Mitigation** – preventative or proactive action to reduce the likelihood of the risk happening
- **Contingency** – emergency or reactive approach to reduce the impact if the risk happens



Risk-based Testing

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.

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. For example, developers can identify the most complex code, users may describe the most used functions, managers may highlight the areas with the biggest financial impact, etc.

In a risk-based approach, the risks identified may be used to:

- Determine the test techniques to be employed, e.g.
  - White-box testing for safety-critical system
  - Usability testing for customer input screens
  - Security testing for e-commerce system
- Determine the extent of testing to be carried out, e.g.
  - What to test … and not to test
  - What to test first … and last
  - What to test most … and least

- Prioritise testing in an attempt to find the critical defects as early as possible, e.g.
  - Use traceability matrix to link test design to priority requirements
  - Create test execution schedule to run tests in the most efficient order
  - Focus on defect clusters
- Determine whether any non-testing activities could be employed to reduce risk. e.g.
  - Provide training to inexperienced designers
  - Improve documentation

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

5.6 Incident Management

Learning Objectives

- Recognise the content of an incident report according to the 'Standard for Software Test Documentation' (IEEE Std 829-1998) (K1)
- Write an incident report covering the observation of a failure during testing (K3)

Incident – *Any event occurring that requires investigation* – ISTQB Glossary. During test execution, discrepancies between actual and expected results are logged as incidents and must be investigated.

Incidents may be raised at any time throughout the software development life cycle, against documentation during static testing as well as code or a system under test.

It is important that businesses should have an Incident Management process to track incidents from discovery and classification to correction and confirmation of solution and finally closure. Systems Management standards often reserve the term 'incident' only for production faults.

Causes of Incidents

Testers tend to assume that all failures are caused by a defect in the software, but there are many possible causes.

- Software defect
- Requirement or specification defect
- Environmental problem, e.g. hardware, operating system or network
- Test procedure or script fault, e.g. incorrect, ambiguous or missing step
- Incorrect test data
- Incorrect expected results
- Tester error, e.g. not following the procedure correctly

Test Incident Reports

Incidents must be recorded in incident reports, either manually or using an incident management tool. There are many reasons for reporting incidents including:

- Provide feedback to enable developers and other parties to identify, isolate and correct defects
- Enable test leaders to track the quality of the system and the progress of the testing
- Provide ideas for test process improvement
- Identify defect clusters
- Create a history of defects and their resolutions
- Supply metrics for assessing exit criteria

The IEEE Std 829-1998 Test Incident Report contains the following sections:

Report Item	Description
Report Identifier	Unique reference for each incident
Summary	Summary of the circumstances in which the incident occurred, referring to software item and version, test case, test procedure and, test log
Description	Description the incident in detail, including • Input • Expected and actual results • Anomalies • Time and date • Procedure step • Environment • Attempts to repeat • Testers and observers comments
Impact	Indication of what impact this incident will have on the progress of testing

Most organisations find the IEEE 829 standard is not detailed enough for their needs, and find it useful to include further details, such as:

- Severity – Level of importance to the business or project requirements
- Priority – Impact on the testing process (and so urgency to fix)
- Complexity – Impact on development (i.e. scale of fix)
- Status – Progress of incident resolution (e.g. open, awaiting fix, fixed, deferred, awaiting retest, closed)
- Other areas that may be affected by a change resulting from the incident
- Actions taken by project team members to isolate, repair and confirm the incident as fixed
- System life cycle process in which the incident was observed
- Change history

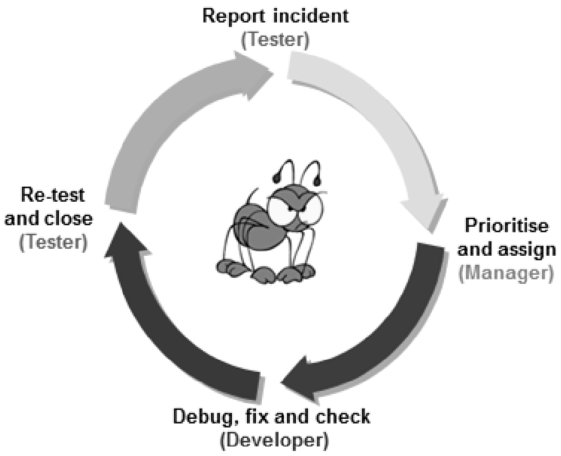
There are different views on the use of the priority and severity measures, e.g. severity = impact on functioning of system; priority = how quickly the business requires fix. Or severity = impact of failure; priority = likelihood of failure. Whatever definitions are used, every test organisation should have clear guidelines on their use. Some organisations have incident measures independently assessed by a test manager or defect manager.

Test Incident Life Cycle

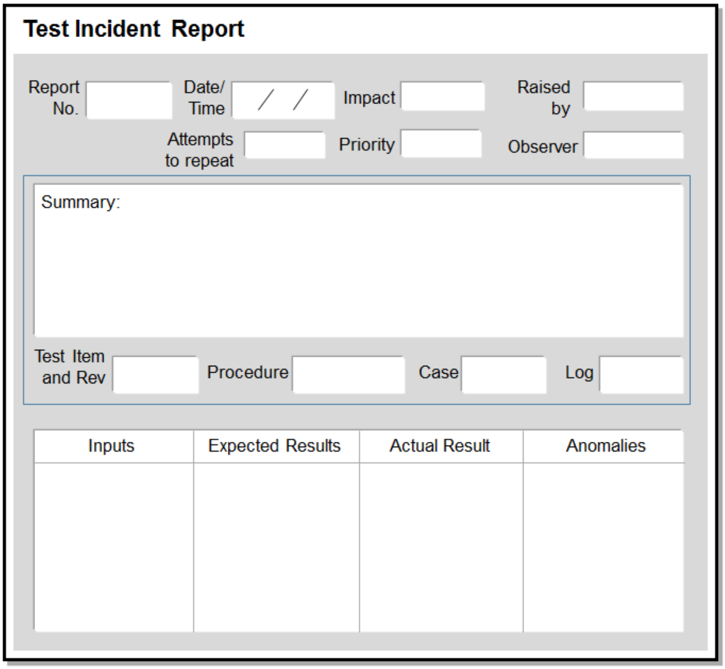
The incident or bug life cycle is managed using the 'status' measure on the incident report. Possible status values are: open, assigned, deferred, duplicate, waiting to be fixed, fixed awaiting re-test, closed, re-opened, etc.

Other status values might indicate that the bug is a pre-existing production defect, that the system works as specified, or that the defect has already been recorded.

If re-testing shows that the defect has not been successfully removed by the developer, the incident report may be re-opened, rather than being closed.



It is vital to keep control of defects, particularly in big projects. The use of forms (like the one below) and incident management tools help to ensure consistent recording of incidents.


Test Incident Report

Report No.		Date/ Time	/ /	Impact		Raised by	
		Attempts to repeat		Priority		Observer	

Summary:

Test Item and Rev		Procedure		Case		Log	

Inputs	Expected Results	Actual Result	Anomalies

Incident Report Exercise

<u>Email to: C. Coder, Programming Dept.</u>

We are on the fifth revision of software item CQ127 and test case 78 still isn't fixed! The discount of 2.5% should apply from an order total of £200 but this value still gives 2.0% and the changeover actually happens at £200.01. It's not just me, Jenny Sims pair tested this with me, over three iterations of the test. A matter of a penny may seem unimportant to you but customers often order to exactly the threshold value. Do you give any thought to our business reputation? I can continue with the remainder of the tests but you need to get your act together and correct this by the end of next week when acceptance testing starts. Also, please reload the base data at 1600 today.

Yours T. Tester, Testing Dept.

Use this information to fill out the sample incident report template above

What useful information is missing from the email?

Which parts of the email should be omitted from an incident report?

Notes

Module 6 – Tool Support for Testing

Topics

1. **Types of Test Tools**
2. **Effective Use of Tools: Potential Benefits and Risks**
3. **Introducing a Tool into an Organisation**

6.1 Types of Test Tools

Learning Objectives

- Classify different types of test tools according to their purpose and to the activities of the fundamental test process and the software life cycle (K2)
- Explain the term test tool and the purpose of tool support for testing (K2)

Test automation or Computer-Aided Software Testing (CAST) is widely used in test organisations to support the test process.

Test tool – *A software product that supports one or more test activities, such as planning and control, specification, building initial files and data, test execution and test analysis* – ISTQB Glossary.

Test tools can be used for a number of activities that can support testing, including:

- Tools used directly in testing tasks (e.g. test execution, data generation, result comparison)
- Tools that help to manage the test process (e.g. management of tests, requirements, data, incidents)

- Tools used to monitor and report activities during testing (e.g. performance, memory, file activity)
- Generic tools that can be used to help testing (e.g. spreadsheets, SQL, debugging tools)

Testers may use tool support for a number of reasons, depending on the context:

- Improve the efficiency of testing (e.g. scripted test execution)
- Support manual activities (e.g. test planning, design or reporting)
- Automate repetitive tasks (e.g. regression testing)
- Automate activities that are difficult to do manually (e.g. static analysis or large scale performance testing)
- Increase reliability of testing (e.g. automating large data comparisons or simulating behaviour)

Test Tools Classification

There are a number of tools that support different aspects of testing. Tools can be classified based on several criteria such as purpose, commercial / free / open-source / shareware, technology used and so forth. Tools are classified in the ISTQB® Foundation syllabus according to the testing activities that they support.

Some tools clearly support one activity; others may support more than one activity, but are classified under the activity with which they are most closely associated. For example, a test execution tool may incorporate a built-in test comparator, but it is classified as a test execution tool rather than a comparator tool.

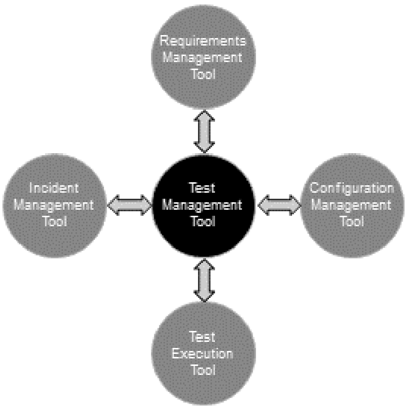
One of the most important features of tools is their ability to interact and integrate with other tools. Tools from a single provider, especially those that have been designed to work together, may be bundled into one package.

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

Tool Support for Management of Testing and Tests

Management tools apply to all test activities over the entire software life cycle.

Test Management Tools



These are 'umbrella' tools which provide interfaces for executing tests, tracking defects and managing requirements. They also support traceability of test objects to requirements specifications, and might have an independent version control capability or an interface to an external configuration management tool.

They typically support quantitative analysis and reporting of test activity in a wide variety of formats.

Requirements Management Tools

These tools store business requirements in a central repository which incorporates

- Unique identifiers
- Descriptions of functional and non-functional requirements
- Attributes such as source, priority, rationale and status
- Links between requirements

These tools may also help with identifying inconsistent, overlapping or missing requirements, and can support tracing the requirements to individual tests.

ARM (Automated Requirement Measurement) is a free tool provided by the NASA Goddard Space Flight Centre to scan requirements specifications for specific words and phrases indicative of quality problems, such as weak, fuzzy, and ambiguous terms that will inevitably lead to miscommunications.

Incident Management (Defect Tracking) Tools

These tools store and manage incident reports, such as defects, failures, change requests or perceived problems and anomalies, and manage the life cycle of incidents, optionally with support for statistical analysis.

They support the logging of incident characteristics (e.g. type, priority, severity) and status (e.g. rejected, ready, deferred.)

These may be bundled with other test management tools or available separately, including some free tools. Many tools can produce graphs based on different parameters from the mass of incident data that is typically logged during testing.

Configuration Management (CM) Tools

Although not strictly test tools, these can support the test process. They are necessary for storage and version management of testware and related software especially when configuring more than one hardware/software environment in terms of operating system versions, compilers, browsers, etc.

They store version/build/configuration information, support tracking and control of testing and development products, and provide direct integration with many testing tools

CM tools may have to cater for many platforms and artefacts across a large organisation, so a specialist tool is likely to be better than one that is bundled with a CASE or test tool.

Tool Support for Static Testing

Static testing tools provide a cost effective way of finding more defects at an earlier stage in the development process.

Review Tools

These tools (also known as Review Process Support tools) assist with review processes, checklists, review guidelines and defects. They are used to store and communicate reviewers' comments, report on defects and effort, and store metrics for causal analysis and process improvement. They can be of further help by providing aid for online reviews for large or geographically dispersed teams.

Review tools are most useful for formal inspections which utilise databases, forms, checklists and causal analysis.

Static Analysis Tools (D)

These tools help developers and testers find defects in source code prior to dynamic testing. Static analysis tools can provide complexity feedback for developers via various industry-defined metrics. They can analyse structures and dependences and help in planning and risk analysis.

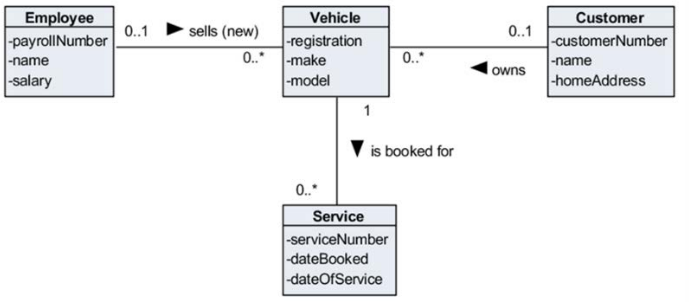
Subtle coding errors and fault-prone code can often be found far more quickly using these tools than by visual inspection or testing. Code may also be reviewed and sorted using user-defined criteria including metrics, often generating automatic reports and blocking code when a threshold is exceeded until the problem is resolved.

Static analysers may also be customised to check that organisational standards are maintained as well as industry-wide measures. Static analysers should allow message filtering otherwise the output can be overwhelming.

Modelling Tools (D)

These tools are used to validate software models (e.g., physical data model, activity model, process model), by enumerating inconsistencies and finding defects. These tools can often aid in generating some test cases based on the model.

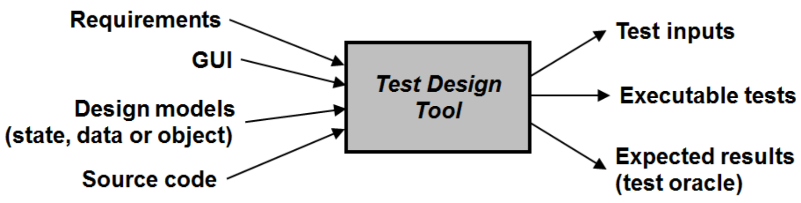
Although mainly used by designers and developers, modelling tools that record state transition diagrams or use cases can also be suitable for testers. Even if they don't auto-generate tests, the way that sequence or state diagrams show transitions is ideally suited to deriving tests. Similarly a use case that describes pre and post conditions and a series of steps simplifies the test design process.



Tool Support for Test Specification

Test Design Tools

These tools are used to generate executable tests, test inputs (data) or test oracles (expected results) automatically from requirements, graphical user interfaces, design models (such as business rules, state diagrams or data models) or code.



Code-based test generators are similar to static analysers and are language-specific. Typically they identify insertion points for exception testing or identify paths and path conditions, especially useful when there is undocumented software.

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

Test Design tools can save valuable time and provide increased thoroughness of testing because of the completeness of the tests that the tool can generate.

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. Test data preparation tools enable data to be selected from existing databases or created afresh. If tests are using copies of production data with sensitive information, these tools can ensure security through data anonymity.

Large-scale random data can be generated according to defined business rules.

Base data records will need to be set up in a certain state which will enable test cases to work. This may require a complex series of transactions to achieve, but can more easily achieved by artificial means.

Tool Support for Test Execution and Logging

Test Execution Tools

These tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language. They can also be used to record tests, and usually support scripting language or GUI-based configuration for parameterisation of data and other customisation in the tests.

Test execution tools usually provide a test log for each test run and can compare actual and expected results.

A GUI capture/playback tool forms the heart of most execution tools. Keystrokes and mouse pointer actions are intercepted by the tool and stored in a test script which will enable replay. The state of GUI objects such as windows, fields, buttons and other controls as well as display output is also recognised and recorded.

Scripts can then be repeated or amended to vary the tests run. Data, test cases and expected results may be held in separate test repositories and varied as required.

Test Harness / Unit Test Frameworks Tools (D)

A unit test harness or framework facilitates the testing of components or parts of a system by simulating the environment in which that test object will run, through the provision of mock objects as stubs or drivers. This is particularly useful for component integration testing, such as top-down or bottom-up testing.

Test harnesses and drivers are used to execute software under test which may not have a user interface or to run groups of existing automated test scripts which can be controlled by the tester.

Stubs are used to simulate routines which have not yet been written, and can print or display the values passed to them. Drivers are used to pass variables to a routine under test, and print or display variables returned from them.

Simulators are used to support tests where code or other systems are either unavailable or impracticable to use (e.g. testing software to cope with nuclear meltdowns).

Test Comparator Tools

Test comparator tools are used to compare actual results and expected results by analysing differences between files, databases or test results.

Standalone comparison tools normally deal with a range of file or database formats. But many test execution tools have built-in comparators that deal with character screens, GUI objects or bitmap images. These tools often have filtering or masking capabilities, so they can ignore rows or columns of data, or areas screens.

Comparator tools which check printed output are more specialised, as they need to avoid throwing up differences on variable print data like page numbers, line numbers, run dates and times. They do need to check whether the same messages and values are present on the print files for the same identifier (e.g. master file record code).

Coverage Measurement Tools (D)

These tools measure the percentage of specific types of code structures that have been exercised by a set of dynamic tests. They may measure coverage of statements, branches or decisions, and calls to modules or functions. They are used during white-box testing to measure actual coverage achieved by a set of structural tests.

Such tools may be intrusive - i.e. they add extra code to the software under test to measure the coverage. This is known as the probe effect.

Coverage measurement tools typically produce detailed coverage reports or graphs showing what code has been executed, the number of times particular statements or decision outcomes have been executed, and non-executable statements such as declarations and comments.

Error handlers can be specifically included or excluded from the reported results and results can be tailored to include all test runs or just the current run. It may be possible to view and monitor coverage increasing on-line while the software is running, and results may be exported to a spreadsheet for further analysis.

Security Tools

These tools are used to evaluate security characteristics of software such as the ability of the software to protect data confidentiality, integrity, authentication, authorisation, availability and non-repudiation.

They can also check for viruses and denial-of-service attacks. Due to the nature of these tools, they are mostly focused on a particular technology, platform and purpose.

These are essential in order to protect confidential business systems and sensitive data from attacks on web-based or distributed systems. They are often very sophisticated tools and are more likely to be used by a security specialist than by a typical tester.

Tool Support for Performance and Monitoring

Dynamic Analysis Tools (D)

Dynamic analysis tools provide run-time information on the state of software under test. They find defects that are evident only when software is executing, such as time dependencies or memory leaks. For example, the memory on a workstation may continuously decline leading ultimately to an untidy application failure.

They are typically used by developers in component and component integration testing, and when testing middleware (application software that connects software components.)

Performance, Load and Stress Testing Tools

Performance testing tools monitor and report on how a system behaves under a variety of simulated usage conditions in terms of the number of concurrent users, their ramp-up pattern, frequency and relative percentage of transactions.

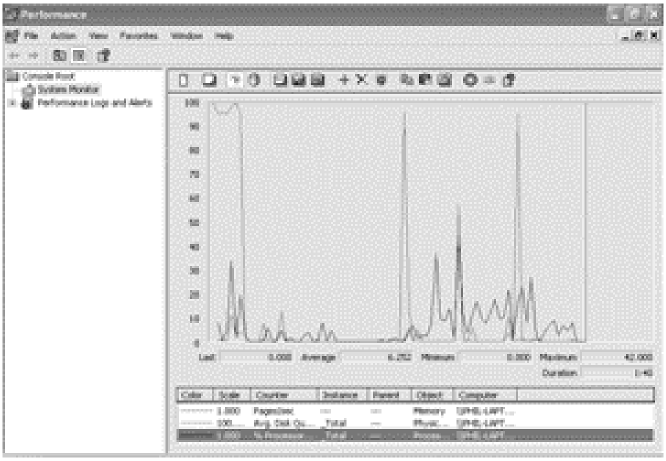
The purpose of performance testing is to show that the system has the capacity to handle large numbers of users and transactions during normal operation and peak periods to comply with any Service Level Agreement on response times.

Load testing assesses the capability of the infrastructure to handle the expected loads, while stress testing evaluates the system beyond its expected limits. The simulation of 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 testing tools normally provide reports based on test logs, and graphs of load against response times. Larger companies may have specialist performance test teams to design tests and analyse the results.

Monitoring Tools

Monitoring tools continuously analyse, verify and report on usage of specific system resources (such as CPU, memory, disc capacity, network), and give warnings of possible service problems.



These are really part of Systems Management tools rather than being specialised tester tools, but can be used during dynamic testing to highlight system resource issues which might have an impact on availability, performance or other SLAs.

Data Quality Assessment Tools

These tools assess data quality and integrity for data-centric projects such as data conversion/migration projects and data warehouse applications. They can review and verify data conversion and migration rules, and ensure that processed data is correct, complete, accessible and compliant with pre-defined standards.

6.2 Effective Use of Tools: Potential Benefits and Risks

Learning Objectives

- Summarise the potential benefits and risks of test automation and tool support for testing (K2)
- Remember special considerations for test execution tools, static analysis, and test management tools (K1)

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

Possible benefits of using tools include:

- Reduce repetitive work, e.g. regression testing, re-entering the same test data, checking against coding standards
- Ensure greater consistency and repeatability, e.g. tests executed in the same order with the same frequency
- Improve accuracy by removing human error
- Provide objective assessment and reporting of static measures, coverage and system behaviour
- Maintain history of test documents e.g. test plans, specifications and logs
- Provide easy access to information about testing, e.g. statistics and graphs about test progress, incident rates

Potential Risks

Risks of using tools include:

- Unrealistic expectations for the tool, including functionality and ease of use
- Underestimating the costs, time and effort required to introduce a tool and to achieve significant and continuing benefits, including training, the need for changes in the testing process and continuous improvement
- Over-reliance on the tool, such as using automated testing when manual testing would be better
- Neglecting version control of test assets within the tool
- Neglecting relationships and interoperability issues between tools, such as requirements management tools, version control tools, incident management tools, and tools from multiple vendors
- Tool supplier issues such as poor response or support, or supplier going out of business
- Suspension of open-source or free tools
- Inability to support a new platform

Special consideration for some types of tools

Some tools have particular issues which need careful consideration before being implemented within a testing organisation.

Test Execution Tools

These tools execute tests using automated test scripts, generated by recording the actions of a manual testers using capture-replay technology. This type of tool often requires significant effort in order to achieve significant benefits.

Such scripts contain both the actions and data from the recorded test, which limits their flexibility. To obtain the greatest benefit, it is necessary to de-couple the actions and data, which can then be modified to create a more flexible set of potential tests.

This process, and the on-going maintenance of automated scripts, requires technical expertise in the scripting language, either by testers or by specialists in test automation.

A **data-driven** testing approach separates out the test inputs (data), usually into a spreadsheet, which can then be edited without knowledge of a scripting language. The generic test script, containing the original actions, can then be run many times against different data in the spreadsheet, or against data generated using algorithms based on configurable parameters at run time.

A **keyword-driven** testing approach separates out the actions, in the form of standard keywords, into a spreadsheet. These can then be modified by testers (even if they are not familiar with the scripting language) to define new tests, tailored to the application being tested.

Static Analysis Tools

Static analysis tools can give useful information about source code, including defects, complexity measures and conformance to coding standards. But they may generate a very large quantity of warning messages, especially if applied to existing code. These messages will not stop the code from being compiled into an executable program, but may obscure more important structural defects and should be addressed to ease future maintenance of the code.

The most effective approach is to apply filters to exclude some messages initially, allowing testers to focus on key areas of concern.

Test Management Tools

Test management tools need to interface with other tools (such as requirements management and defect management tools) or spreadsheets in order to produce useful information in a format that fits the needs of the organisation. It is therefore essential to ensure that these tools can communicate successfully, particularly if they are from different suppliers.

6.3 Introducing a Tool into an Organisation

Learning Objectives

- State the main principles of introducing a tool into an organisation (K1)
- State the goals of a proof-of-concept for tool evaluation and a piloting phase for tool implementation (K1)
- Recognise that factors other than simply acquiring a tool are required for good tool support (K1)

Selection Considerations

The introduction of a testing tool is costly, so it must be treated like any other business project, with clear business objectives and an estimate of cost-benefit ratio based on a concrete business case to justify the expenditure.

Before introducing a tool into an organisation, it is important to assess the testing maturity, strengths and weaknesses of the testing process and team, and to identify opportunities for process improvement using tools. Any prospective tool must be evaluated against clear requirements and objective criteria.

A feasibility study (proof-of-concept) can be established to demonstrate that a tool performs effectively with the software under test and with the current infrastructure, and to identify any changes needed for effective use of the tool.

The tool supplier must be evaluated to clarify what provision they make for training, service support, upgrades and commercial aspects. The test team's requirements for coaching and mentoring should also be considered, especially the team’s test automation skills.

Introducing the Selected tool

Before making a commitment to implementing a tool within an organisation, a pilot project is usually undertaken to ensure the benefits of using the tool can actually be achieved. The objectives of the pilot are to learn more detail about the tool, evaluate how it fits with existing processes and practices, identify changes required in the test process and assess whether the benefits will be achieved at reasonable cost.

It is also an opportunity to decide on standard ways of using, managing, storing and maintaining the tool and the test assets (e.g. naming conventions for files and tests, creating libraries, defining modularity of test suites and common conventions for use.)

Deployment Success Factors

Full deployment of the tool should be based on a successful result from the evaluation of the pilot. This should be assessed along with implementation plans to ensure it is right to proceed and that everything is ready for a successful roll-out. Success factors for deployment of a tool within an organisation include:

- Roll out incrementally across the organisation
- Adapt and improve processes to fit with use
- Provide training and coaching/mentoring for new users
- Define usage guidelines
- Gather usage information from actual use
- Monitor use and benefits
- Provide support for the test team for a given tool
- Gather lessons learnt from all teams

Notes

Notes

Examination Hints and Tips

The examination lasts one hour and is closed book, with 40 multiple choice questions, 4 options per question, each question carrying equal weight. You need 26/40 to pass (65%).

It is administered by the BCS to the ISTQB® syllabus. Questions are set from the Learning Objectives stated at the beginning of each section. The number of questions is weighted:

- By topic according to time allocation
  - Chapter 1: 7 question
  - Chapter 2: 6 questions
  - Chapter 3: 3 questions
  - Chapter 4: 12 questions
  - Chapter 5: 8 questions
  - Chapter 6: 4 questions

- By cognitive level (K1, K2, K3/K4)
  - K1: 20 question (50%)
  - K2: 12 questions (30%)
  - K3 and K4: 8 questions (20%)
    - K3 subjects may be examined at K1, K2 and K3 etc.

You can't take in pens, paper, etc. Question paper, answer paper and pencil are provided. Write notes/calculations/diagrams on the question paper.

Remember the syllabus is about the theory of best practice; it may not be what you typically do at work. Read each question carefully and understand what is needed; watch for negative questions. Answer all questions; there is no penalty for wrong answers, so if in doubt, guess.

There is one correct answer per question; others may look plausible. Choose the one closest to the exam syllabus.

Pace yourself, e.g. 1 minute per question first pass leaves 20 minutes to come back to any you’ve left blank and to double-check the answer sheet. Some questions (K1) will only take a few seconds, so there is more time for more complex questions.

QA Practice Exam Questions

Chapter 1 – Fundamentals of Testing

Q1. A human action that produces an incorrect result is called:

A. An error
B. A fault
C. A failure
D. A defect

Q2. A deviation of the software from its expected delivery or service is:

A. A failure
B. A fault
C. A defect
D. An error

Q3. Which of the following statements are true?

i. A defect is produced as a result of system failure
ii. Failures can be caused by environmental conditions such as radiation
iii. Defects, bugs and faults are the same thing
iv. Failures can cause bugs
v. Faults always cause system failures

A. i, ii and v are true; iii and iv are false
B. ii, iii and iv are true; i and v are false
C. ii and iii are true; i, iv and v are false
D. iv and v are true; i, ii and iii are false

Q4. Which of the following statements is true?

A. Testing can show failures that are caused by defects
B. Developers use debugging instead of testing
C. Regression testing checks that the defect has been fixed
D. Negative testing is not to be recommended

Q5. Static tests are:

A. Reviews of documents or code
B. Tests of electrical equipment
C. Onsite tests
D. Used in steady-state systems

Q6. Which of the following statements apply to testing and which to debugging?

1. Investigates the cause of a fault
2. Identifies failures
3. Confirms whether a failure has been fixed
4. Fixes software if necessary
5. Checks the defect has been fixed
6. Steps through program code

A. 1, 2 and 3 are testing; 4, 5 and 6 are debugging
B. 1, 2 and 5 are testing; 3, 4 and 6 are debugging
C. 2 and 3 are testing; 1, 4, 5 and 6 are debugging
D. 2, 3 and 4 are testing; 1, 5 and 6 are debugging

Q7. Who would usually perform debugging activities?

A. Incident Managers
B. Developers
C. Analysts
D. Testers

Q8. An unlikely objective for a test is

A. To prove there are zero defects
B. To detect faults
C. To assess regulatory compliance
D. To assess software quality

Q9. Which is **NOT** a testing principle?

A. Early testing
B. Defect clustering
C. Pesticide paradox
D. Functional testing

Q10. The Test Closure phase of a testing project begins when

A. The exit criteria have been met
B. All faults have been fixed
C. The implementation date is reached
D. All requirements have been tested

Q11. Which activity in the fundamental test process creates test suites for efficient test execution?
A. Analysis and design
B. Implementation and execution
C. Planning and control
D. Test closure

Q12. During which fundamental test process activity do we determine if more tests are needed?

A. Test implementation and execution
B. Evaluating exit criteria and reporting
C. Test analysis and design
D. None of the above

Q13. Which activities form part of test planning?

- i. Developing test cases
- ii. Defining the overall approach to testing
- iii. Determining the required resources
- iv. Building the test environment
- v. Writing test conditions
- vi. Implementing the test strategy

A. i, ii and iv are true; iii, v and vi are false
B. ii, iii and vi are true; i, iv and v are false
C. iv, v and vi are true; i, ii and iii are false
D. i, ii and iii are true; iv, v and vi are false

Q14. Independence in testing means

A. The programmer being left to get on with their own tests
B. Someone other than the developer designing the tests
C. Each tester working on their own without interference
D. Testing being planned independently from the rest of the project

Q15. Meetings between developers and testers

A. Are unnecessary thanks to email
B. Are necessary so that blame can be apportioned
C. Are for constructive communication of defects
D. Are bad as they compromise independence

Q16. One key reason why developers have difficulty testing their own work is:

A. Lack of technical documentation
B. Lack of test tools on the market for developers
C. Lack of training
D. Lack of objectivity

Q17. Which of the following is a benefit of independent testing?

A. Independent testers are much more qualified than developers
B. Independent testers see other and different defects and are unbiased
C. Independent testers cannot identify defects
D. Independent testers can test better than developers

Q18. Which of the following has the highest level of independence? Test cases which are designed by:

A. The person who wrote the software under test
B. A person from a different section
C. A person from a different organisation
D. Another person within the development team

Q19. Which of the following is **NOT** part of the Code of Ethics?

A. Public
B. Product
C. Financial
D. Self

Q20. Which, in general, is the skill least required of a good tester?

A. Being diplomatic
B. Able to write software
C. Having good attention to detail
D. Able to be relied on

Chapter 2 – Testing throughout the Software Life Cycle

Q1. Which of the following describes how testing activities can occur in parallel with development activities?

A. Test specification
B. Test execution
C. Quality standards
D. V-model

Q2. In iterative development models which of the following is true?

A. Regression testing is not important on all iterations after the first one
B. Regression testing is less important on all iterations after the first one
C. Regression testing is increasingly important on all iterations after the first one
D. Regression testing is not done on iterations after the first one

Q3. Validation involves which of the following?

i. Using prototypes to speed up development
ii. Checking that the right product has been built
iii. Checking that standards have been met
iv. Starting test specification during development stages

A. i, ii, iii and iv are true
B. ii is true; i, iii and iv are false
C. i, ii and iii are true; iv is false
D. iii is true; i, ii, and iv are false

Q4. Testing interfaces to external organisations is **MOST LIKELY** to occur in which part of the life cycle?

A. System Integration Testing
B. System Testing
C. Acceptance Testing
D. Configuration Testing

Q5. Navigation of a web site might be tested as part of:

A. Performance testing
B. Volume testing
C. Stress testing
D. Usability testing

Q6. In Test-Driven Development, when should a set of test scripts be written for a component?

A. When the equivalence classes have been specified
B. At least four weeks before testing commences
C. Before the code is written
D. When there are zero compile defects

Q7. When conducting Component Integration testing, functional integration

A. Tests the application built in an incremental life cycle
B. Tests multi-threaded program modules
C. Selects a specific area of functional capability in the system
D. Is a type of test automation

Q8. Integration testing, where no incremental testing takes place prior to combining all the system's components, is called

A. System integration testing
B. Business process testing
C. Big bang testing
D. Thread testing

Q9. A Use Case makes a good

A. Test basis
B. Configuration specification
C. Component design
D. Flowchart

Q10. Specially-written software that replaces a called component during component integration testing is called what?

A. Driver
B. Simulator
C. Stub
D. Harness

Q11. Which of the following is untrue of user acceptance testing?

A. The aim is to establish confidence in the system
B. The main purpose is to find defects in the system
C. It is the responsibility of users
D. It may assess the systems readiness for deployment and use

Q12. Testing following modifications, migration, or retirement of production software is

A. Maintenance testing
B. Change control
C. Configuration management
D. Performance test

Q13. Which of the following statements about testing are true?

i. White box testing is only useful during system and acceptance testing
ii. Alpha testing takes place at a customer site
iii. Structural testing is equivalent to white box testing
iv. Interoperability testing is a non-functional testing type
v. Stress tests are part of functional testing

A. i, iii and iv are true; ii and v are false
B. iii is true; i, ii, iv and v are false
C. iii and iv are true; i, ii and v are false
D. i, iii, iv and v are true; ii is false

Q14. Functional testing would test which of the following?

A. The system will recover from any failure within 20 minutes
B. The system will provide an average response time of 3 seconds
C. The system will support 1000 simultaneous online users
D. The system will produce a weekly order report

Q15. A regression test:

A. Is the same as re-testing
B. Checks unchanged areas of software for unexpected side effects
C. Determines if changed areas of software contain defects
D. Is only used during acceptance testing

Q16. The difference between re-testing and regression testing is

A. Re-testing is running a test again; regression testing looks for unexpected side effects
B. Re-testing looks for unexpected side effects; regression testing is repeating those tests
C. Re-testing is done after faults are fixed; regression testing is done earlier
D. Re-testing uses different environments, regression testing uses the same environment

Q17. Regression testing should be performed:

i. Every week
ii. After the software has changed
iii. As often as possible
iv. When the environment has changed
v. When the project manager says

A. i and ii are true; iii, iv and v are false
B. ii, iii and iv are true; i and v are false
C. ii and iv are true; i, iii and v are false
D. ii is true; i, iii, iv and v are false

Q18. Which of the following defines the scope of maintenance testing?

A. The coverage of the current regression pack
B. The size and risk of any changes to the system
C. The time since the last change to the system
D. The number of defects found at the last regression tests

Chapter 3 – Static Testing

Q1. Typical defects that are easier to find in reviews than in dynamic testing are:

A. Deviations from standards
B. Requirement and design defects
C. Insufficient maintainability and incorrect interface specifications
D. All of the above

Q2. What common objective is shared by static testing and dynamic testing?

A. Identifying defects
B. Improving development productivity
C. Checking the quality of software
D. None of the above

Q3. In a walkthrough the scribe should **NOT** also be the

A. Reviewer
B. Author
C. Moderator
D. Manager

Q4. A developer who desk-checks his or her own code by visually stepping through its execution is performing

A. An informal review
B. An inspection
C. A walkthrough
D. Dynamic analysis

Q5. The following characteristics define a technical review:

- i. Have a documented, defined fault-detection process
- ii. Are never led by a trained moderator
- iii. Include peers and technical experts
- iv. May have no management participation
- v. May correct issues as they arise

A. i, iii and iv are true; ii and v are false
B. iii is true; i, ii, iv and v are false
C. iii and iv are true; i, ii and v are false
D. i, iii, iv and v are true; ii is false

Q6. A moderator who is **NOT** also the author is mandatory in

A. An informal review
B. An inspection
C. A walkthrough
D. Dynamic analysis

Q7. What type of review requires formal entry and exit criteria, and keeps metrics?

A. Informal review
B. Inspection
C. Walkthrough
D. Technical review

Q8. Reviews are worthwhile because (choose the **BEST** answer):

A. They reduce costs by removing defects before test execution
B. They help build good relationships between development team and users
C. It means that less system documentation is needed
D. It means that less test documentation is needed

Q9. One of the defining characteristics of walkthroughs is that they are led by

A. The manager
B. The chief reviewer
C. The author
D. A trained moderator

Q10. In a formal review, reviewers are each assigned a focus so that

i. We exploit individuals' expertise
ii. Fewer reviewers are necessary
iii. Material need not be distributed in advance
iv. Responsibilities are defined clearly
v. Each has a smaller task
vi. A moderator is unnecessary

A. i, ii, iii and vi are true
B. i, iv, v and vi are true
C. i, iv and v are true
D. ii, iii and v are true

Q11. Static analysis can detect which of the following more easily than dynamic testing?

A. Viruses
B. Trojans
C. Database concurrency
D. Parameter type mismatches

Q12. Which of the following statements about compilers is true in testing?

A. Are the same as static analysers
B. Have no role in static analysis
C. Are used to support document inspections
D. May provide some similar functionality to a static analyser

Q13. Which of the following is true of static analysis?

A. Finds errors rather than failures
B. Finds failures rather than defects
C. Finds failures rather than errors
D. Finds defects rather than failures

Q14. What can static analysis **NOT** find?

A. The use of a variable before it has been defined
B. Unreachable (“dead”) code
C. Whether the value stored in a variable is correct
D. The re-definition of a variable before it has been used

Q15. Static analysis tools are:

A. An alternative to dynamic testing tools
B. Able to provide quality information about source code
C. Able to monitor and report on executing code
D. Able to reproduce faults and investigate the state of programs

Chapter 4 – Test Design Techniques

Q1. In IEEE Std. 829-1998, Test Design Specifications define

A. Test input, test data and expected results
B. Test conditions and the associated high level test cases
C. Execution order and procedures/scripts
D. Relevant test levels

Q2. In IEEE Std. 829-1998, which document specifies the sequence of actions for executing tests?

A. The test case specification
B. The test procedure specification
C. The test design specification
D. The test plan

Q3. A range of data from which a test value is selected is termed

A. An equivalence partition
B. A boundary value
C. A condition group
D. A valid class

Q4. Customer numbers in the Birmingham database can range between 1000 and 6999 inclusive. Which of the following sets of inputs is the **minimum** to execute each valid and invalid equivalence partition?

A. 1000, 3500, 6999
B. 500, 5000, 8500
C. 999, 1000, 6999, 7000
D. 1000, 1001, 6998, 6999

Q5. Customer numbers in the Birmingham database can range between 1000 and 6999 inclusive. Which of the following sets of inputs tests all valid and invalid boundary values?

A. 1000, 3500, 6999
B. 500, 5000, 8500
C. 999, 1000, 6999, 7000
D. 1000, 1001, 6998, 6999

Q6. Which of the following is a useful technique when testing combinations of conditions?

A. Decision coverage
B. Decision table
C. Decision outcome
D. Equivalence partitions

Q7. Use cases are beneficial to testers because

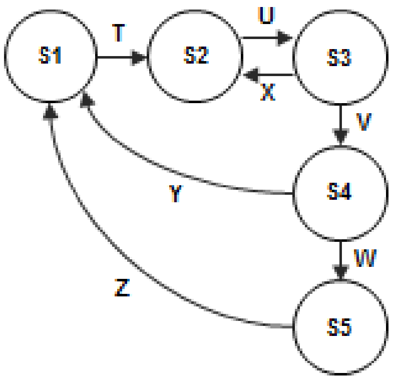
A. Process flows are based on typical business usage
B. They are created independently by testers
C. They divide inputs into equivalence partitions
D. They describe allowable transitions

Q8. What is a valid boundary value?

A. Any valid input value entered by the user
B. Any test value inside a valid partition
C. A value identified by the boundary value analysis that falls within a valid partition
D. A boundary test that gives a valid result

Q9. In the State Transition Diagram below, how many 0-switch test cases are required?

A. 7
B. 10
C. 4
D. 5



Q10. In the State Diagram in Q9 above, which of the following represents an **invalid** sequence of transitions?

A. T–U–X–U–V–W–Z
B. V–W–Z–T–U–V
C. T–U–V–W–Z–Y
D. V–Y–T–U–X–U–V

Q11. Given the specification below, which of the following values for age are in different equivalence partitions?

If you are aged 21 or under, you are too young to be insured. Between 22 and 40 inclusive, you will receive a 20% discount. Anyone over 40 is not eligible for a discount.

A. 22, 40, 61
B. 21, 41, 62
C. 21, 40, 60
D. 22, 41, 63

Q12. Using the insurance decision table below, what is the expected result of the following test cases?

A. Smoker who plays dangerous sports
B. Non-smoker who plays dangerous sports

	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**
Conditions				
Dangerous sports	Y	Y	N	N
Smoker	Y	N	Y	N
Actions				
Refuse cover	X	-	-	-
Add 10% to premium	-	-	X	-
Refer to underwriter	-	X	-	-

A. A - Referred to underwriter; B - Cover refused
B. A - Cover refused; B - Referred to underwriter
C. A - 10% added to premium; B - Cover given
D. A - Cover refused; B - 10% added to premium

Q13. The evaluation of a decision to TRUE or FALSE is a

A. Decision outcome
B. Condition code
C. Boolean twist
D. Mandelbrot set

Q14. Which of the following approaches provides the weakest level of coverage?

A. Decision coverage
B. Statement coverage
C. Multiple condition coverage
D. Path coverage

Q15. Consider the following pseudo code:

```
Read (points)
If points = 0 Then
    Display “Failed”
Else
    If points < 10 Then
        Print “You need more points”
    Else
        Print “Congratulations, you won”
    End If
End If
```

What is the **minimum** number of tests which would be required for 100% statement and for 100% decision coverage?

A. 1 for statement coverage, 2 for decision coverage
B. 2 for both statement and decision coverage
C. 2 for statement coverage, 3 for decision coverage
D. 3 for both statement and decision coverage

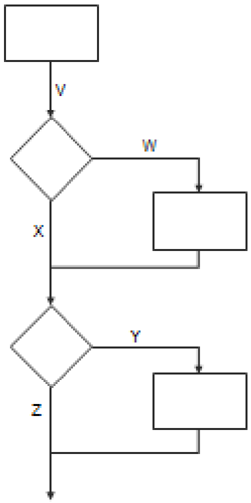
Q16. If a program is tested and 100% decision coverage is achieved, which of the following coverage criteria is then guaranteed to be achieved?

A. 100% equivalence class coverage
B. 100% path coverage and 100% statement coverage
C. 100% statement coverage
D. 100% multiple condition coverage

Q17. The following flowchart has been tested with two test cases.

TC1: followed the path V X Z
TC2: followed the path V W Z

What level of coverage has been achieved?



A. Statement coverage 100%; Decision coverage 100%
B. Statement coverage 80%; Decision coverage 75%
C. Statement coverage 75%; Decision coverage 75%
D. Statement coverage 66%; Decision coverage 50%

Q18. Which of the following statements is true of experience-based techniques?

A. Are used to assess software structure
B. Are only used in component testing
C. Cannot be used with customised software
D. Augment other techniques

Q19. Error guessing is **BEST** used

A. As the first approach to deriving test cases
B. After more formal techniques have been applied
C. By inexperienced testers
D. After the system has gone live

Q20. The choice of test techniques to use can depend on

A. The age of the tester
B. The level of risk
C. The size of the system
D. The number of developers working on the system

Q21. The following is a list of test design techniques. Which would be termed "white box" and which "black box"?

1. Decision Testing
2. Statement testing
3. Boundary Value Analysis
4. Use Case Testing
5. Decision Table Testing
6. State Transition Testing

A. 1, 2 and 5 are white box; 3, 4 and 6 are black box
B. 1, 2 and 6 are white box; 3, 4 and 5 are black box
C. 1, 2 and 3 white box; 4, 5 and 6 are black box
D. 1 and 2 are white box; 3, 4, 5 and 6 are black box

Q22. Which of these statements about test techniques is incorrect?

A. Condition testing is a structural technique and has a measure attached
B. Equivalence partitioning is not the same category of technique as branch/decision testing
C. State transition testing is the same category of technique as statement testing
D. Decision table testing is the same category of technique as state transition testing

Q23. Consider the following test cases:

Test Case	Description	Pre-conditions
TC1	Add new customer	
TC2	Add new customer with errors	
TC3	Add new product	
TC4	Add sales transaction	Customer and product details must exist on database
TC5	Print customer invoice	Customer and product details must exist on database; customer must have sales transactions
TC6	Print monthly customer statement	Part of end-of-month process; customer and product details must exist on database. The customer must have sales transactions
TC7	Delete product	Product must exist on database

Which of the following sequences would make an appropriate test execution schedule for the above set of test cases?

A. TC1, TC3, TC6, TC4, TC5, TC7, TC2
B. TC3, TC7, TC1, TC2, TC6, TC4, TC5
C. TC2, TC1, TC3, TC4, TC5, TC6, TC7
D. TC2, TC3, TC4, TC1, TC6, TC7, TC5

Chapter 5 – Test Management

Q1. A usability tester is an example of which of the following?

A. Information psychologist
B. Test specialist
C. Test tool
D. Beta tester

Q2. Which of the following is **NOT** a task undertaken by the test leader?

A. Co-ordinate test strategy
B. Adapt planning based on test results
C. Prepare and acquire test data
D. Write test summary reports

Q3. Which is **NOT** a potential drawback of independent testing?

A. The independent testers may become a bottleneck
B. Developers get less opportunity to gain testing experience
C. Testers may become isolated from the developers
D. Developers may lose a sense of responsibility for quality

Q4. Which of the following is **NOT** a task typically undertaken by the tester?

A. Create test specifications
B. Prepare and acquire test data
C. Review test strategy
D. Review tests written by others

Q5. Which of these is **NOT** part of the IEEE Std. 829 test plan?

A. Item pass/fail criteria
B. Suspension criteria and resumption requirements
C. Test deliverables
D. Condition list

Q6. In the IEEE Std. 829 test plan, what are test items?

A. Software to be tested
B. Business features
C. Deliverable documentation
D. Risk concerns

Q7. Which of these is a reactive approach to testing?

A. Model-based
B. Standards-based
C. Consultative
D. Exploratory

Q8. Test planning is influenced by:

A. Known constraints
B. Test cases
C. The test environment
D. The development team

Q9. Exit criteria may be defined in terms of:

A. The size of the system
B. The number of testers available
C. Coverage criteria and defects outstanding
D. Entry criteria and the risks involved

Q10. Which of the following would not normally be present in a test plan conforming to the IEEE Std. 829 software test documentation standard?

A. Schedule
B. Staffing and training needs
C. Features to be tested
D. Test cases

Q11. An estimate of delivered software reliability should be contained in which of the following documents?

A. Test summary report
B. Master test plan
C. Test incident report
D. Test strategy

Q12. Which set of metrics would be **BEST** for monitoring test execution?

A. Number of detected defects and testing cost
B. Number of residual defects in the software
C. Percentage of completed tasks in the preparation of test environment, and test cases prepared
D. Number of test cases run / not run, and number of test cases passed / failed

Q13. Match the following test activities:

1. Test estimation
2. Test control
3. Test monitoring

with the following objectives:

m) Track progress
n) Assess effort required to perform activities
o) Reallocate resources

A. 1-n, 2-o, 3-m
B. 1-n, 2-m, 3-o
C. 1-o, 2-m, 3-n
D. 1-m, 2-n, 3-o

Q14. Which of the following is used to maintain integrity of software and related products throughout the life cycle?

A. Change control
B. Version control
C. Test management
D. Configuration management

Q15. Which is **NOT** a function of configuration management?

A. Version control of testware
B. Tracking changes to testware
C. Ensuring unambiguous references to software items in test documentation
D. Definition of standards for software test documentation

Q16. Which of the following is true of testing?

A. Provides contingency
B. Mandates life cycle
C. Provides use cases
D. Mitigates risk

Q17. In a risk-based approach, the risks identified may be used to:

i. Determine the test techniques to be employed
ii. Determine the extent of testing to be carried out
iii. Prioritise testing in an attempt to find critical defects as early as possible
iv. Determine the cost of the project

A. ii is true; i, iii and iv are false
B. i, ii and iii are true; iv is false
C. ii and iii are true; i and iv are false
D. ii, iii and iv are true; i is false

Q18. What is the difference between a project risk and a product risk?

A. Project risks are potential failure areas in the software or system; product risks are risks that surround the project's capability to deliver its objectives
B. Project risks are the risks that surround the project's capability to deliver its objectives; product risks are potential failure areas in the software or system
C. Project risks are typically related to supplier issues, organisational factors and technical issues; product risks are typically related to skill and staff shortages
D. Project risks are risks that delivered software will not work; product risks are typically related to supplier issues, organisational factors and technical issues

Q19. Which of the following is determined by the level of product risk identified?

A. Extent of testing
B. Scope for the use of test automation
C. Size of the test team
D. Requirement for regression testing

Q20. Which of the following are project risks, and which are product risks?

1. Failure of a third party
2. Lacking functionality
3. Skill shortage
4. Problems in requirements definition
5. Organisation politics
6. Poor performance of software

A. 1, 3, 4 and 5 are project risks; 2 and 6 are product risks
B. 1, 2, 3, 4 and 5 are project risks; 6 is product risk
C. 1, 3, 4, 5 and 6 are project risks; 2 is product risk
D. 1, 3 and 5 are project risks; 2, 4 and 6 are product risks

Q21. Test incident reports would be raised against which of the following?

A. Product risk identification
B. Progress slippage
C. Informal reviews
D. Expected and actual results mismatch

Q22. Which of these are valid objectives for Test Incident Reports?

1. Provide developers with sufficient information to reproduce a failure
2. Remove the need for verbal communication with developers
3. To measure tester efficiency
4. To show the quality of the system under test
5. To show the importance of an incident
6. To log live system failures

A. 1, 2 and 5
B. 1, 3, 4, 5 and 6
C. 1, 4, and 5
D. 1, 4, 5 and 6

Q23. Which of these would be listed in an IEEE Std. 829 Test Incident Report?

1. Report Identifier
2. Description of incident
3. Developer name
4. Estimate
5. Risk analysis
6. Impact of incident

A. 1, 2 and 6
B. 1, 3, 4 and 6
C. 1, 2, 4 and 6
D. 1, 4, 5 and 6

Q24. What information need **NOT** be included in a test incident report?

A. How to fix the fault
B. How to reproduce the fault
C. Test environment details
D. The actual and expected outcomes

Chapter 6 – Tool Support for Testing

Q1. Static analysis tools can provide the following

- i. Verification against coding standards
- ii. Defect analysis graphs
- iii. Graphics while program executes
- iv. Location of complex code
- v. Control flow graphs

A. ii and v are true; i, iii and iv are false
B. i and v are true; ii, iii and iv are false
C. i, iv and v are true; ii and iii are false
D. i and ii are true; iii, iv and v are false

Q2. What type of tool is used to speed up regression testing?

A. Data generation tool
B. Test harness
C. Test execution tool
D. Dynamic analysis tool

Q3. Coverage measurement tools may be

A. Intrusive or non-intrusive
B. Data generators
C. Control conduits
D. Black box evaluators

Q4. What type of tool may generate tests from state diagrams?

A. Performance tools
B. Modelling tools
C. Data preparation tools
D. Static analysers

Q5. Which tools may analyse test results?

A. Test comparator tools
B. Data preparation tools
C. Test execution tools
D. a and c

Q6. Which of these tools are **MORE SUITABLE** for developers?

1. Incident management tool
2. Static analysis tool
3. Test framework
4. Stress testing tool
5. Coverage measurement tool
6. Security tool
7. Dynamic analyser
8. Monitoring tool

A. 2, 4, 5 and 7
B. 2, 3, 5 and 7
C. 1, 2, 5, 7 and 8
D. 2, 3, 5, 6 and 7

Q7. Completion of component coverage test exit criteria can be confirmed by

A. Monitoring tools
B. Static analysers
C. Coverage measurement tools
D. Security tools

Q8. Test execution tools

A. Are only suitable for developers
B. Dispose of bad tests
C. Log test results
D. Trap memory errors

Q9. Test design tools can generate test inputs from all except:

A. Requirements
B. GUI
C. Code
D. Expected results

Q10. A drawback of creating a script, by recording the actions of a manual tester is that

A. Data is distributed throughout the script
B. Tests cannot be rerun without an operator
C. Differences between versions of developed software cannot be compared
D. Web applications cannot be tested

Q11. What facility is offered by a test management tool but **NOT** a spreadsheet?

A. Can produce graphs
B. Allows columns to be filtered
C. Has built-in test repository
D. Can store transactional data

Q12. Which of these would probably result in the immediate rejection of a prospective test tool?

A. Platform not supported
B. Results analysis must be done separately
C. Doesn’t integrate with other test tools
D. No built-in repository

Q13. Which of the following is an objective of a pilot project for the introduction of a testing tool?

A. Evaluate testers' competence to use the tool
B. Complete the testing of a key project
C. Assess whether the benefits will be achieved at reasonable cost
D. Discover what the requirements for the tool are

Q14. The most important benefit of a proof-of-concept pilot is to

A. Compare a number of tools simultaneously
B. Train users of the tool
C. Check fit with existing processes
D. Check it will work with a complex project

Q15. A new tool should be initially introduced to the organisation:

A. In a small pilot project
B. As part of a business critical project
C. To every on-going project immediately
D. In projects with experienced project managers

Answers to Questions

Chapter 1 – Fundamentals of Testing

Q1.	A	Q11.	B
Q2.	A	Q12.	B
Q3.	C	Q13.	B
Q4.	A	Q14.	B
Q5.	A	Q15.	C
Q6.	C	Q16.	D
Q7.	B	Q17.	B
Q8.	A	Q18.	C
Q9.	D	Q19.	C
Q10.	A	Q20.	B

Chapter 2 – Testing throughout the Software Life Cycle

Q1.	D	Q10.	C
Q2.	C	Q11.	B
Q3.	B	Q12.	A
Q4.	A	Q13.	B
Q5.	D	Q14.	D
Q6.	C	Q15.	B
Q7.	C	Q16.	A
Q8.	C	Q17.	C
Q9.	A	Q18.	B

Chapter 3 – Static Testing

Q1.	D	Q9.	C
Q2.	A	Q10.	C
Q3.	B	Q11.	D
Q4.	A	Q12.	D
Q5.	D	Q13.	D
Q6.	B	Q14.	C
Q7.	B	Q15.	B
Q8.	A		

Chapter 4 – Test Design Techniques

Q1.	B	Q13.	A
Q2.	B	Q14.	B
Q3.	A	Q15.	D
Q4.	B	Q16.	C
Q5.	C	Q17.	B
Q6.	B	Q18.	D
Q7.	A	Q19.	B
Q8.	C	Q20.	B
Q9.	A	Q21.	D
Q10.	C	Q22.	C
Q11.	C	Q23.	C
Q12.	B		

Chapter 5 – Test Management

Q1.	B	Q13.	A
Q2.	C	Q14.	D
Q3.	B	Q15.	D
Q4.	C	Q16.	D
Q5.	D	Q17.	B
Q6.	A	Q18.	B
Q7.	D	Q19.	A
Q8.	A	Q20.	A
Q9.	C	Q21.	D
Q10.	D	Q22.	C
Q11.	A	Q23.	A
Q12.	D	Q24.	A

Chapter 6 – Tool Support for Testing

Q1.	C	Q9.	D
Q2.	C	Q10.	A
Q3.	A	Q11.	C
Q4.	B	Q12.	A
Q5.	D	Q13.	C
Q6.	B	Q14.	C
Q7.	C	Q15.	A
Q8.	C		

Index